XSIENA: The Content-Based Publish/Subscribe System

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
TECHNISCHEN UNIVERSITÄT DRESDEN
FAKULTÄT INFORMATIK

eingerichtet von

Dipl.-Ing. Zbigniew Jerzak

geboren am 31.08.1979 in Zabrze, Polen

Betreuender Hochschullehrer:	Prof. Christof Fetzer, PhD
Referent:	PD Dr.-Ing. Gero Mühl
Datum der Einreichung:	16. April 2009
Datum der Verteidigung:	28. September 2009

Declaration

Herewith I declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher education, except where due acknowledgment has been made in the text.

Dresden, September 29, 2009

Zbigniew Jerzak

Preface

Just as packet switched networks constituted a major breakthrough in our perception of the information exchange in computer networks so have the decoupling properties of publish/subscribe systems revolutionized the way we look at networking in the context of large scale distributed systems. The decoupling of the components of publish/subscribe systems in time, space and synchronization has created an appealing platform for the asynchronous information exchange among anonymous information producers and consumers. Moreover, the content-based nature of publish/subscribe systems provides a great degree of flexibility and expressiveness as far as construction of data flows is considered.

However, a number of challenges and not yet addressed issued still exists in the area of the publish/subscribe systems. One active area of research is directed toward the problem of the efficient content delivery in the content-based publish/subscribe networks. Routing of the information based on the information itself, instead of the explicit source and destination addresses poses challenges as far as efficiency and processing times are concerned. Simultaneously, due to their decoupled nature, publish/subscribe systems introduce new challenges with respect to issues related to dependability and fail-awareness.

This thesis seeks to advance the field of research in both directions. First it shows the design and implementation of routing algorithms based on the end-to-end systems design principle. Proposed routing algorithms obsolete the need to perform content-based routing within the publish/subscribe network, pushing this task to the edge of the system. Moreover, this thesis presents a fail-aware approach towards construction of the content-based publish/subscribe system along with its application to the creation of the soft state publish/subscribe system. A soft state publish/subscribe system exposes the self stabilizing behavior as far as transient timing, link and node failures are concerned. The result of this thesis is a family of the XSiena content-based publish/subscribe systems, implementing the proposed concepts and algorithms. The family of the XSiena content-based publish/subscribe systems has been a subject to rigorous evaluation, which confirms the claims made in this thesis.

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Christof Fetzer, for his support and advice during the whole duration of my Ph.D. program at the Dresden University of Technology. I would also like to thank Dr.-Ing. Gero Mühl for accepting the role of the reviewer of this thesis.

I had the pleasure to be the member of the Systems Engineering group. I would like to thank my colleagues: Ute, Andrey, Gert, Martin, Torvald, et al.; who created a friendly and inspiring work environment. I really enjoyed my work with Robert Fach with whom I have had the pleasure to author two papers and a book chapter related to this thesis.

I would also like to thank Grzegorz Hryń for the proof reading of this thesis and his constructive criticism. My work on this thesis has been partially supported by the Polish Ministry of Science and Higher Education grant number N N516 375034.

Finally, I would like to thank my parents and my wife for their support and encouragement. Thank you.

Publications

[Jer05] Zbigniew Jerzak. Highly Available Publish/Subscribe. *HASE '05: Supplement Proceedings of the Ninth IEEE International Symposium on High Assurance Systems Engineering*, Heidelberg, Germany, October 2005, pp. 11–12. IEEE Computer Society.

[JF06] Zbigniew Jerzak and Christof Fetzer. Handling Overload in Publish/Subscribe Systems. *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, Lisbon, Portugal, June 2006, pp. 32–37. IEEE Computer Society.

[JF07] Zbigniew Jerzak and Christof Fetzer. Prefix Forwarding for Publish/Subscribe. *DEBS '07: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*, Toronto, Canada, June 2007, pp. 238–249. ACM.

[JFF07] Zbigniew Jerzak, Robert Fach, and Christof Fetzer. Fail-Aware Publish/Subscribe. *NCA '07: Proceedings of the Sixth IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, July 2007, pp. 113–125. IEEE Computer Society.

[JF08b] Zbigniew Jerzak and Christof Fetzer. Bloom Filter Based Routing for Content-Based Publish/Subscribe. *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, Rome, Italy, July 2008, pp. 71–81. ACM.

[JF08a] Zbigniew Jerzak and Christof Fetzer. BFSiena: a Communication Substrate for StreamMine. *DEBS '08: Proceedings of the Second International Conference on Distributed Event-Based Systems*, Rome, Italy, July 2008, pp. 321–324. ACM.

[FBFJ] Christof Fetzer, Andrey Brito, Robert Fach, and Zbigniew Jerzak. StreamMine. To appear in: Alex Buchmann and Annika Hinze, editors, *Handbook of Research on Advanced Distributed Event-Based Systems, Publish/Subscribe and Message Filtering Technologies*. IGI Global.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Packet switched networks were a major breakthrough in our perception of the information exchange. Prior to their introduction, scientists were considering the adoption of the standard telephone systems to the needs of data transmission. One of the major issues which they were faced with was the need to amortize the connection setup cost, which could easily be as high as 50% for an intercontinental data exchange. Moreover, a set up connection was failure-prone as the fixed wiring between the sender and the receiver implied that the probability of failure was exponentially proportional to the number of components involved in the forwarding of the data. The above problems seemed to be the major obstacles which were almost impossible to circumvent using the telephone system design approach. It was only until the Paul Baran at Rand Corporation and Donald Davis at National Physical Laboratory introduced the new concepts of *transitivity* [Bar64] and *packet* that the above problems could be finally solved.

The idea of the packet assumed that in order to provide fairness and prevent link monopolization by one of the sources, the data should be split up into packets. Packets, being small pieces of information, can naturally commute on the links solving the problem of link sharing. The transitivity concept assumed that every node which receives a packet, and is not a destination node, tries to forward this packet using its routing configuration. In order for the transitivity approach to work another fundamental change had to be made. Instead of implicit source and destination addresses used by the telephone systems, Paul Baran suggested using *explicit* addresses embedded into the meta-data of every packet. This approach allowed to avoid the connection setup costs, as no explicit connections needed to be created prior to data transmission. This approach also created a fault-tolerant network as the reliability of the data transfer *increased* with the number of components, providing there was on average more than one outgoing edge per system node.

The ideas of Paul Baran and Donald Davies were implemented in 1969 with the support of the Advanced Research Projects Agency (ARPA) director James Licklider. In 1974 they were further refined by Vinton Cerf and

Robert Kahn [CK74] (for a recent edition see [CK05]) to form the modern TCP/IP [CDS74] protocol. However, the widespread of the Internet and its massive growth in recent years has caused the TCP/IP networking to become a „success disaste" [Jac06]. At the time when TCP/IP was developed the amount of information being transmitted was low. There existed a few machines and the IP address space was relatively stable. Moreover, the TCP/IP protocol has been designed so as to naturally support the point-to-point conversation model, making broadcast a not feasible approach.
Contemporary communication networks are different from those of over thirty years ago. The focus is not put on conversation of two entities anymore. Nowadays, an N-to-M (or many-to-many) data dissemination model has become the prevalent type of communication in the Internet. In the N-to-M model, multiple clients are interested in the same data items. Simultaneously, a single client collects data from multiple sources. The TCP/IP communication model has proved not to be well suited for such scenarios. An example might be 2006 Winter Olympics when an NBC router became severely congested by holding 6000 copies of the same video data requested by different clients [Jac06]. This implies 6000 users making a TCP/IP connection and downloading the same file. The contemporary networking infrastructures try to overcome such problems on the global scale [DPW04, Lei07] however, even such solutions are vulnerable [SK08] due to the their underlying design principles inherited from the TCP/IP.

1.1 Decoupling

The popularity of the TCP/IP model has changed the domain in which it was designed to operate. The solutions provided by the TCP/IP are still valid, however its success introduced new challenges which need to be addressed. The new challenges include a large amount of users with multiple machines, each storing large amount of data which needs to be synchronized. Recent studies have shown that the amount of data we are able to collect and store has been increasing exponentially [TB00] for the past 20 years. On the other hand, the prices for the long distance network bandwidth have remained relatively stable [Gra03, Jac07]. The increasing gap between the amount of information we are able to store and process and the amount of information we are able to transmit is leading to the shift towards the Content-Centric Networking (CCN) [CPB$^+$05, Jac06, Jac07].
The CCN paradigm states that users are no longer interested in the data sources, instead, the data itself has become the focal point. The CCN paradigm changes the semantics of the data exchange over computer networks. The explicit addresses put on transmitted data are obsoleted and the data itself is exposed so as to become the address. Removing explicit source and destination addresses, Content-Centric Networking effectively *decouples* the producer and consumer

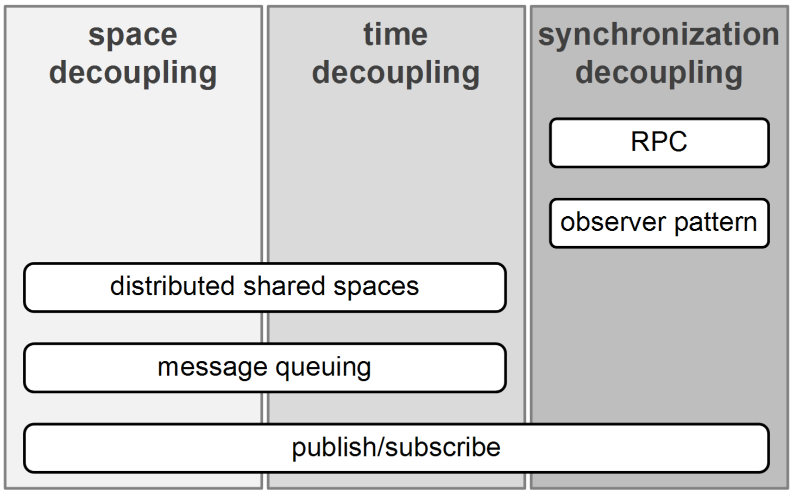


Figure 1.1: Comparison of communication paradigms

of data.

Decoupling of content producers and consumers is the basic approach towards construction of asynchronous, distributed systems. Decoupling removes the dependencies between the communication components reducing the amount of synchronization needed. The process of decoupling has started with the introduction of the Remote Procedure Calls (RPC) [BN84, TA90, CHY+98] which allowed to partially hide the complexity of explicit network communication and introduced the notion of *synchronization decoupling* [YSTH87, Car93].

Synchronization decoupling assumes that data consumers are not blocked when awaiting for the data to arrive. Moreover, content producers do not block when waiting for the consumers to receive the content they have created.

The idea of synchronization decoupling has been strengthened by the formalization of the observer design pattern [GHJV95], which allows data consumers to register their interest directly with producers and become notified about objects of interest whenever those are created or changed. Simultaneously to the introduction of synchronization decoupling the concepts of shared virtual memory [LH89] and tuple spaces [CG89] have been introduced. These new concepts allowed to decouple participants of the distributed system in terms of *time* and *space*.

Space decoupling allows the content producers and consumers to remain anonymous to each other. Space decoupling implies that neither content producers know the identity of the consumers, nor content consumers are aware of the source of the data.

Time decoupling frees the content producers and consumers from the need to be active at the same time in order to exchange data. Specifically, content producers can create data whilst there are no consumers for it. Simultaneously, consumers can consume data for which producers have already left the system.

Both space and time decoupling and the concepts behind the tuple spaces have been subsequently employed in message queuing systems [BHL95] and Message Oriented Middleware (MOM) [BCSS99]. Figure 1.1 outlines the decoupling properties and their adoption in different systems.

1.2 Publish/Subscribe Paradigm

Decoupling in space, time and synchronization has been unified for the first time in the publish/subscribe paradigm [EFGK03] – see Figure 1.1. Publish/subscribe paradigm allows for strictly decoupled communication between publishers (content producers) and subscribers (content consumers).
The first system to adopt the publish/subscribe paradigm was the Information Bus [OPSS93]. The Information Bus system, similar in concept to the generative communication model of tuple spaces, implemented a topic-based publish/subscribe paradigm. The Information Bus and other topic-based publish/subscribe systems [SB89, AEM99, SM02, BEG04, BBQ+07] provide self-describing objects combined with anonymous communication. Shortly after the introduction of the topic-based publish/subscribe systems the increasing need for heterogeneity and expressiveness has lead to the introduction of the content-based [RW97] and type-based [Eug07] publish/subscribe systems.
The publish/subscribe interaction paradigm naturally implements the concepts behind the Content Centric Networking. The decoupling properties unified within the publish/subscribe paradigm provide a loosely coupled form of interaction required in large scale dynamic distributed systems [Que07]. Consequently, the publish/subscribe paradigm has experienced a widespread adoption in the distributed systems community, becoming an active field of research. The decoupled communication and resulting reconfigurability and scalability have given rise to multiple implementations of the publish/subscribe paradigm, which are used by many applications [Jac03, ZSB04, PC05b, MSSB07, Tar08b] and standards [SM02, Gro04].

1.3 Contribution

In this thesis it is argued that design of the content-based publish/subscribe systems should follow the principles of the end-to-end argument [SRC84] in that the content-based matching functionality should be moved from the content-based network into the publish/subscribe application layer.
The case for the above argument is made by describing the design, implementation and evaluation of the family of content-based systems called XSiena. The XSiena publish/subscribe system is the first system to provide a unified solution to the content-based routing at the edge of the content-based network. The approach proposed in this thesis substantially differs from the previous work

in that domain by allowing for semantically unrestricted content-based routing with the support for decoupling of the content-based network components in the domains of time, space and synchronization.
The main contributions of this thesis with respect to the development of the XSiena content-based systems family include:

Prefix-based Forwarding The development and implementation of algorithms allowing to abstract the result of the content-based matching at the application level. The result can be subsequently encoded within prefixes attached to messages. The prefix is the only part of the message which needs to be evaluated by a content-based system for the routing purposes. It is also shown how to achieve consistency between routing structures in a distributed system.

Bloom filter-based Routing It is shown how to use Bloom filters [Blo70] to develop a fixed size digest of a content-based event matching result. Bloom filter-based routing, similarly to prefix-based forwarding performs the content-based matching only once per message, in the application layer. The proposed matching algorithm is linear with the number of filters matching a given event, regardless of the total number of filters in the system.

Moreover, within this thesis it also shown how to solve practical problems in the domain of large scale distributed systems using content-based XSiena system as their communication backbone. These problems are especially challenging, due to the decoupled nature of the content-based systems:

Fail-Awareness The fail-aware XSiena system presents, based on the Timed Asynchronous Distributed System Model [CF99], a practical approach to detection of timely message delivery in content-based publish/subscribe systems. The detection of timely message delivery is performed without the need to introduce expensive clock synchronization mechanisms. To the best of our knowledge, the fail-aware XSiena system is the first system to offer a lightweight and fully decoupled content-based routing substrate which can be used in soft real-time applications.

Soft State It is shown how the fail-aware XSiena system can be successfully used to cope with the timing and causality issues often found in large scale distributed systems. Specifically, soft state XSiena system demonstrates how to remove hard state from the content-based network, allowing it to easily recover from transient links and node failures.

1.4 Outline

The remainder of this thesis is organized as follows: Chapter 2 combines the presentation of the related work in the area of the content-based systems with the introduction of the basic terminology used in this thesis. The discussion focuses on the types of architectures and associated routing strategies used in content-based publish/subscribe systems along with the semantics and representation of the information being exchanged.

Chapter 3 (based on [JFF07, JFF08]) presents the system model for the XSiena family of content-based systems. The system model describes the assumptions being made about the properties of processes, communication channels and possible failures. Chapter 3 presents also the semantics of the XSiena family of content-based systems to form a foundation upon which the following chapters are built.

Chapter 4 (based on [JFF07]) presents the algorithms which encode the results of the application-level content-based matching within the prefix trees. Chapter 4 discusses the benefits of the prefix trees when compared to traditional content-based publish/subscribe systems and their influence on the content-based information forwarding. Moreover, the discussion focuses on the challenges related to the prefix trees, especially regarding the consistency of the distributed prefix trees in the context of the asynchronous updates.

Chapter 5 (based on [JF08b, JF08a, FBFJ]) presents algorithms allowing to construct fixed-size digests of the results of the content-based matching. Specifically, chapter 5 addresses the problem of the usage of Bloom-filters as a compact representation of the results of the content-based matching. Chapter 5 presents also routing algorithms taking advantage of the Bloom filter-based content representation allowing for significant improvement in the routing speed.

Chapter 6 (based on [JFF07]) presents the design and implementation of the fail-aware content-based system. The fail-awareness property allows the system to calculate an upper bound on the propagation delay of messages. The upper bound can be used to determine whether a message received at the destination node is late or not. Chapter 6 argues that the fail-aware extension to the XSiena content-based system allows to use content-based networks in lightweight, soft real-time applications.

Chapter 7 applies the results presented in Chapter 6 to construct a soft state XSiena system. It is shown how the construction of the soft state content-based network allows to eliminate the need to store hard state in the nodes which have not created it. Specifically, it is shown how the soft state design allows the system to cope with the timing issues and transient node and link failures.

Chapter 9 presents the evaluation of the algorithms and architectures presented in Chapters 4, 5, 6 and 7. Chapter 9 specifically focuses (Chapters 4 and 5) on the micro-benchmarks for the performance of the routing algorithms. Moreover, results obtained in the PlanetLab [Ros05] environment (Wide Area Networks), which highlight the specific issues discussed in Chapters 6 and 7 are presented.

Chapter 10 concludes this thesis and provides the outlook for the XSiena family of content-based publish/subscribe systems.

Chapter 2

Background

This chapter presents an overview of recent research activities related to the publish/subscribe systems. It describes the basic components and concepts related to the publish/subscribe systems, along with the types and semantics of messages, followed by the description of common publish/subscribe architectures and corresponding data routing models.
This chapter introduces the basic concepts and terms related to the publish/subscribe systems and content-centric networking so as to provide the necessary background for the remainder of this thesis. The discussion within this chapter is not limited to content-based publish/subscribe systems, however it primarily focuses on the architectures which are relevant to the XSIENA system.

2.1 Components

This section discusses the basic publish/subscribe components and their interactions. The component model is to a large extent independent of the specific variant of the publish/subscribe system.

2.1.1 Applications

The goal of publish/subscribe systems is to allow for a decoupled communication between the users of the system. In a typical scenario [Pie04, MFP06, FBFJ], the publish/subscribe system itself is a layer used by other applications to exchange data – see Figure 2.1. Applications use publish/subscribe system to both consume and produce (publish) content. Applications consuming content are called subscribers, while applications producing content are called publishers. Application are not limited to either one or another role – it is common that a single application plays the role of both publisher and subscriber.
The content production and consumption is asynchronous in that an application can immediately continue the execution after dispatching the data to the

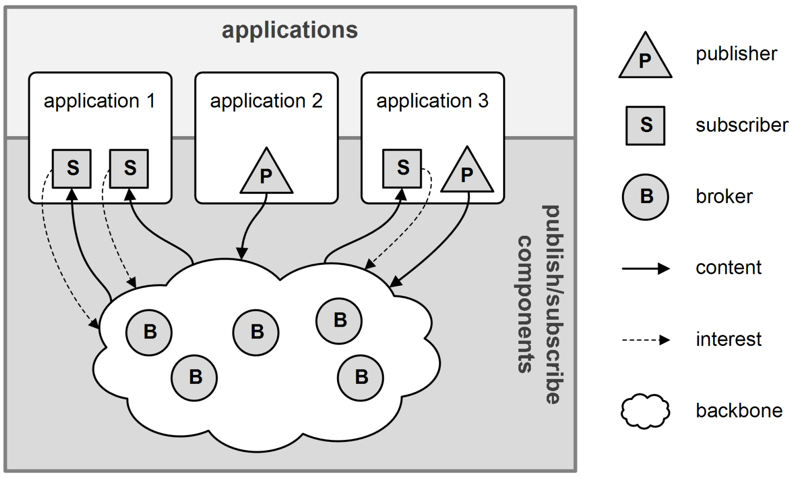


Figure 2.1: Components of the publish/subscribe system

publish/subscribe system. Similarly, for the content consumption, application is informed about new data via asynchronous upcall from the publish/subscribe layer. The asynchronous content production and consumption allows the applications to remain decoupled with respect to time – applications can produce content for which there are no consumers yet. Similarly, applications consuming data can receive content for which the producing applications have already left the system or have crashed.

Applications, unless explicitly instrumented, remain anonymous to each other. This implies that content consumers are not aware about the origin of the content they receive via the asynchronous upcall. Similarly, the content producers are not aware of the potential consumers of the content they publish. The underlying publish/subscribe system transparently connects content producers and consumers without revealing their identities. The decoupled properties of the publish/subscribe system imply that most applications are message-based [EFGK03, Pie04].

The above principles allow publish/subscribe systems to host multiple distributed application types, including financial services [OPSS93], mobile computing [Tar08b], real-time systems [Bar07, DXGE07], industrial automation [Jac03], scientific computation [SL07], content distribution [CLS03] and many more [ZSB04, PC05b, MSSB07].

2.1.2 Subscribers

Applications willing to receive content play the role of subscribers. A subscribing application usually plays the role of the observer (observer design pattern [GHJV95]) by implementing an upcall provided by the publish/subscribe layer. The main task of the upcall is to asynchronously deliver messages matching the interest of the subscriber. Upcalls are usually provided in form

```
//subscriber interface
interface Subscriber extends Deliverable {
   void subscribe(Interest i);
   void unsubscribe(Interest i);
   void notify(Message m);
}

//upcall interface
interface Deliverable {
   void deliver(Message m);
}
```

Listing 2.1: An abstract subscriber API

```
//publisher interface
interface Publisher {
   void publish(Message m);
}
```

Listing 2.2: An abstract publisher API

of interfaces or abstract classes – Listing 2.1 shows the `Deliverable` upcall interface.

In order to be able to receive data every subscribing application must be able to express its interest. Application express their interest using a method provided by the publish/subscribe layer. Listing 2.1 presents the `subscribe` method which can be used by the application to specify the content of interest. It is common for the `subscribe` method to take the `Deliverable` upcall interface as parameter [CRW01, EFGK03, PEKS07].

Subscribers are stateful in that they need to manage the interest they use to describe the content of interest. Therefore, subscribers not interested in a given content anymore can use the `unsubscribe` method to express the removal of their interest – the parameter to the `unsubscribe` method. Subscribers leaving the publish/subscribe system should unsubscribe to all previously issued interest.

2.1.3 Publishers

Content producers in publish/subscribe systems take the role of publishers. Similarly to subscribes the content is produced in form of messages, which are published into the publish/subscribe layer. Publishers are typically lightweight, stateless entities as they usually do not receive any messages from the publish/subscribe layer. Listing 2.2 shows a typical publisher interface [CRW01, MÖ2, PEKS07]. Content, in form of messages, is published using the `publish` method call. Stateless publishers, in contrast to stateful subscribers, imply cheap departures and arrivals as no additional actions need to be taken upon occurrence of such event.

2.1.4 Brokers

Brokers form the backbone of the publish/subscribe system. Depending on the system type and architecture their number and interconnections can vary. Applications are decoupled from the actual brokers and their structure by the publisher and subscriber interfaces. The number of brokers can range from a single one in case of centralized systems [SA97] to multiple in case of distributed infrastructures [CRW01, Pie04]. Multiple brokers can be arranged into different architectures, most common being: bus [OPSS93], hierarchical [CRW01, CDNF01], acyclic [MÖ2, FJLM05, Tar08b], cyclic [LMJ08] and peer-to-peer [CDKR02, Pie04, VRKvS06].

2.2 Messages

This section discusses how subscribers express their interest and how publishers disseminate the content in the publish/subscribe networks. Specifically, in the remainder of this thesis it is assumed that all communication between the components of the publish/subscribe system is carried out using messages.

2.2.1 Filters

In order to receive content applications have to express their interest. In publish/subscribe systems interest is described using filters. A filter f is a function which evaluated for its argument (content) returns the value of either true (one) or false (zero):

$$f : \mathbb{C} \rightarrow \{0, 1\} \tag{2.1}$$

where $\mathbb{C}$ is the domain of the filter function f representing all content for which f can be evaluated. A filter returning true implies that the content for which it was evaluated matches the interest of the filter issuer. Zero is returned otherwise. The exact definition of the matching process depends on the type of filters and content used in the publish/subscribe system. The specific discussion regarding the matching process is deferred to the later part of this thesis – see Section 2.3.
The selectiveness of filter functions is bounded by two extremities: a filter $f_\emptyset$ which never selects any published content and filter f_∞ which always selects all published content. It is worth noting that some publish/subscribe systems, e.g. [CRW01], do not provide the possibility to express f_∞. Subscriber issuing multiple filters $f_1, f_2, \ldots, f_n$ creates a union between the interests described by the filters: $f_1 \cup f_2 \cup \ldots \cup f_n$.

In publish/subscribe systems filters describing the subscriber's interest are propagated in form of subscription messages. Subscription messages are issued by the subscribers using the `subscribe` method. Subscription messages encapsulate filters and contain additional implementation specific meta-data. A subscriber wishing to revoke his interest issues an unsubscription message using the `unsubscribe` method (see Section 2.1.2) which contains a filter describing his disinterest.

2.2.2 Events

Content in publish/subscribe systems is published in form of events. Despite recent efforts [CABB04, WJLS04, WJL04, FCMB06], in most of the publish/subscribe systems [CRW01, Pie04, Tar07] filters must match events as far as syntax and semantics is concerned for the data exchange to take place. Using Equation 2.1 it can be said that events must form the domain $\mathbb{C}$ of the corresponding filter function f.
The goal of the publish/subscribe backbone is to deliver content to the interested applications. This implies that subscribers must be notified about events matching their filters. Event e is delivered ($\text{deliver}(e, \mathbf{S}) = \mathit{true}$) to subscriber $\mathbf{S}$ if and only if any of the filters $\mathbb{F}_\mathbf{S} = \{f_1, \ldots, f_n\}$ issued by the subscriber $\mathbf{S}$ matches event e:

$$\text{deliver}(e, \mathbf{S}) = \mathit{true} \quad \Leftrightarrow \quad \exists_{f \in \mathbb{F}_\mathbf{S}} : f(e) = 1 \tag{2.2}$$

In publish/subscribe systems events are propagated in form of publication messages. Publication messages are issued using the `publish` method – see Section 2.1.3.

2.2.3 Advertisements

Advertisements are filters which can be optionally [PEKS07] used by publishers to summarize events they are going to produce in the future. Advertisements play the role of additional constraints which need to be satisfied in order to deliver events to the applications. An advertisement a is a function which evaluated for its argument returns the value of either true (one) or false (zero):

$$a : \mathbb{C} \to \{0, 1\} \tag{2.3}$$

where $\mathbb{C}$ is the domain of the advertisement a representing all content for which a can be evaluated. Similarly to filters, the selectiveness of the advertisement function is bounded by two extremities: an advertisement $a_\emptyset$ which never matches any published events and advertisement a_∞ which always matches all published events. Publisher issuing multiple advertisements $a_1, a_2, \ldots, a_n$ creates a union between the events described by the advertisements: $a_1 \cup a_2 \cup \ldots \cup a_n$.

```
//extended publisher interface
interface Publisher {
   void publish(Event e);
   void advertise(Advertisement a);
   void unadvertise(Advertisement a);
}

//subscriber interface
interface Subscriber extends Deliverable {
   void subscribe(Filter f);
   void unsubscribe(Filter f);
   void notify(Event e);
}

//upcall interface
interface Deliverable {
   void deliver(Event e);
}
```

Listing 2.3: An extended publish/subscribe API

In publish/subscribe system which uses advertisements an event e can be issued (publish$(e, \mathbf{P}) = \mathit{true}$) by the publisher $\mathbf{P}$ if and only if any of the advertisements $\mathbb{A}_\mathbf{P} = \{a_1, \ldots, a_n\}$ issued prior to the publication of the event e by the publisher $\mathbf{P}$ matches event e:

$$\text{publish}(e, \mathbf{P}) = \mathit{true} \quad \Leftrightarrow \quad \exists_{a \in \mathbb{A}_\mathbf{P}} : a(e) = 1 \tag{2.4}$$

Based on Equation 2.4 the Equation 2.2 can be rewritten as:

$$\text{deliver}(e, \mathbf{S}) = \mathit{true} \quad \Leftrightarrow \quad (\exists_{a \in \mathbb{A}_\mathbf{P}} : a(e) = 1) \wedge \left(\exists_{f \in \mathbb{F}_\mathbf{S}} : f(e) = 1\right) \tag{2.5}$$

which can be read as: an event e, published by the publisher $\mathbf{P}$, is delivered to the subscriber $\mathbf{S}$ if and only if there exists an advertisement a issued by the publisher $\mathbf{P}$ and a filter f issued by the subscriber $\mathbf{S}$, such that both match e. Listing 2.3 shows the extended publisher API (with respect to Listing 2.2), which utilizes advertisements. Publisher can summarize the content it is going to publish using the `advertise` method. Advertisements are propagated in form of advertisement messages. Analogously as in the case of `unsubscribe` method, publisher not willing to publish given content anymore can use the `unadvertise` method to announce the change in the content it is going to produce. The change will be propagated in form of unadvertisement messages. The introduction of advertisement messages changes the publishers into stateful components – cf. Section 2.1.3. Stateful publishers must manage the advertisement messages for events they have published and unlike stateless publishers cannot leave the network without issuing corresponding unadvertisement messages.

2.3 Syntax and Semantics

The actual semantics of events, filters, and advertisements depends on the specific implementation of the publish/subscribe system. One of the aspects with the strongest influence on the syntax and semantics of events and filters in publish/subscribe systems is the type of the publish/subscribe system itself – whether it is topic-based [OPSS93], content-based [CRW01] or type-based [Eug07]. Moreover, the type of the broker architecture and the domain in which the publish/subscribe system should operate determine the semantics and syntax of messages. This section outlines the most common variants of the semantics of the publish/subscribe systems.

2.3.1 Topic-Based

Topic-based semantics (also known as subject-based) has its origins in the group communication systems [Pow96]. It structures the event space by dividing it into flat [OPSS93, CDKR02, BBQ+07, MZV07] or hierarchical [BEG04] topics.

A flat topic structure implies that subscriptions must specify an identifier describing a given topic [CDKR02]. In most cases such identifier would be a string literal identifying, e.g., the name of the software package to which updates user wants to subscribe [MZV07].

Hierarchical topics, on the other hand, naturally represent the tree-like arrangements of nested objects, terms or concepts. An example hierarchical topic might have the following structure: `/Hardware/Memory Structures/Design Styles/Cache Memories` – which would indicate the `B.3.2` Cache Memories term in the ACM Computing Classification System[1]. In a topic based publish/subscribe system, every publication dealing with Cache Memories would be published under this topic. Publishing under the given topic usually implies attachment of the topic or its identifier to every publication – for examples see [CDKR02] or [Que07].

Topic-based semantics, has obvious limitations, especially with respect to the selectiveness of the topics. A publication matching multiple topics needs to be published multiple times, which for a broad spectrum of publications might cause severe link congestions, as multiple copies of the same message need to be sent over the same link – see also Section 1.

The limitations of the topic-based representation has led to the development of the content-based filters and events. The following sections present the two most common content-based semantics: predicate-based and XML-based.

[1]See: `http://www.acm.org/about/class/1998/`

2.3.2 Predicate-Based

Predicate-based semantic is one of the most popular among the content-based publish/subscribe systems [CRW01, MÖ2, Pie04, ZSB04, FJLM05, TK06, CH06, TB07]. The predicate-based semantics states that a filter f is a conjunction of n predicates:

$$f = \{p_1 \wedge \ldots \wedge p_n\} \tag{2.6}$$

where predicate p_i is a function which evaluates to either true (one) or false (zero) for a given argument.
In predicate-based semantics, an event e is defined as a set (disjunction) of n values:

$$e = \{v_1 \vee \ldots \vee v_n\} \tag{2.7}$$

where a value v is an argument to the predicate function p.
Using predicate-based semantics filter f matches event e if and only if:

$$f(e) = 1 \quad \Leftrightarrow \quad \forall_{p_i \in f} \exists_{v_i \in e} : p_i(v_i) = 1 \tag{2.8}$$

In most predicate-based semantics filters (with notable exception of [CRW01]) are identical to advertisements. Following this convention advertisement a can be defined as a conjunction of n predicates:

$$a = \{p_1 \wedge \ldots \wedge p_n\} \tag{2.9}$$

An advertisement a matches event e if and only if:

$$a(e) = 1 \quad \Leftrightarrow \quad \forall_{p_i \in a} \exists_{v_i \in e} : p_i(v_i) = 1 \tag{2.10}$$

A predicate p is usually defined [CRW01, MFP06] as being composed of attribute name an and attribute constraint ac. Attribute constraint ac itself is composed of operator op and attribute value av:

$$p = \left(an,\ \underbrace{(op, av)}_{ac} \right) \tag{2.11}$$

An example predicate p might have the following syntax $p = (temp, >, 25)$. In general, different content-based publish/subscribe systems use different possible attribute names and attribute constraints when using predicate-based message semantics. Those systems vary therefore in terms of expressiveness and complexity of the content-based filters and events they use.
Similarly to predicates, event values v are defined as attribute name an and attribute value av pairs:

$$v = (an,\ av) \tag{2.12}$$

An example attribute value v matching the above example predicate $p = (temp, >, 25)$ would have the following syntax $v = (temp,\ 30)$. It is important to

Filter f	Event e	$f(e) = 1$
$\{y < 8\}$	$\{y = 8\}$	false
$\{x = *\}$	$\{x = 7\}$	true
$\{x > 5\}$	$\{x = 7\}$	true
$\{x > 5\}$	$\{x = 7;\ y = 10\}$	true
$\{x > 5;\ y < 8\}$	$\{x = 7\}$	false
$\{x > 5;\ y < 8\}$	$\{x = 7;\ y = 5\}$	true

Table 2.1: Filters and events in predicate-based semantics

note that attribute names and attribute values used in events must semantically match the attribute names and attribute constraints used in filters.

Table 2.1 shows example filters and events in a predicate-based semantics. The notation used in Table 2.1 and throughout the remainder of this thesis shorthands the predicate notation for the sake of brevity. Moreover, the conjunction between the predicates is implicit and denoted via the semicolon sign. Predicate $x = *$ matches every attribute name and attribute value pair where attribute name equals x.

In case of the events, the predicate syntax is omitted and implicit disjunction denoted by the semicolon sign is used instead. Table 2.1 illustrates also the result of the content-based match between the filter f and event e. The result of the content-based match $f(e)$ remains unchanged if advertisements identical to filters in Table 2.1 are used instead. The reader is referred to [MFP06] for a more thorough overview of most commonly used attribute names and attribute constraints types.

2.3.3 XML-Based

Another popular content-based semantics is based on the eXtended Markup Language (XML) [BPSMY06]. Content-based systems using XML-based semantics [SCG01, CF04, DRF04] construct events as XML documents. Such systems usually rely on the subset of XPath [DeR99] or XQuery [BFS07] query processor syntax in order to specify filters.

```
msg/nitf
      [head/pubdata
         [@edition.area = "SF"]]
```

Listing 2.4: Example XQuery-based filter [DRF04]

Listing 2.4 shows an example of the XQuery filter which selects documents having a root element `nitf` with a child element `head`, which itself contains child element `pubdata` whose attribute `@edition.area` has the value `SF`. Listing 2.5 shows an example XML document matching the above query.

```xml
<?xml version="1.0"?>
<!DOCTYPE nitf SYSTEM "nitf-3-2.dtd">
<nitf>
<head>
   <pubdata type="web" edition.area="SF" />
</head>
<body>
   <body.content>
      <p>Content!</p>
   </body.content>
</body>
</nitf>
```

Listing 2.5: Example XML event

Due to potentially large size of documents and very high complexity of the query expressions the XML-based matching of events and filters is resource intensive and there exists substantial body of work which aims at optimizing this process, including, but not limited to: XFilter [AF00], XTrie [CFGR02] and YFilter [DF03] XML query processors.

2.3.4 Other Approaches

Previous sections have presented publish/subscribe message semantics which were implicitly stateless. A stateless semantics implies that filters are evaluated for each event separately and there exists no dependency between subsequent events. Such approach, however, is not always sufficient especially if more complex filtering expressions are required.

One of the first approaches to use complex expressions in content-based publish/subscribe systems were *patterns* introduced in the SIENA system [CRW01]. A pattern in the SIENA system is a sequence of filters which are applied to the ordered sequence of events.

A significantly more complex form of filtering expressions was proposed in the Cayuga system [DGH+06, DGP+07], where filters (similarly to the YFilter [DF03] approach) are based on nondeterministic finite state automata with additional possibility of parametrization and aggregation.

A sample Cayuga filter might read: For any stock (*stk*) there is a monotonic decrease in price for at least 10 minutes ($dur = 10$), which starts at the volume $vol > 10000$. The following quote on *stk* should have a price 5% higher than the last seen one ($prc > 1.05 \cdot minPrc$). Figure 2.2 shows the automaton which implements the above filter.

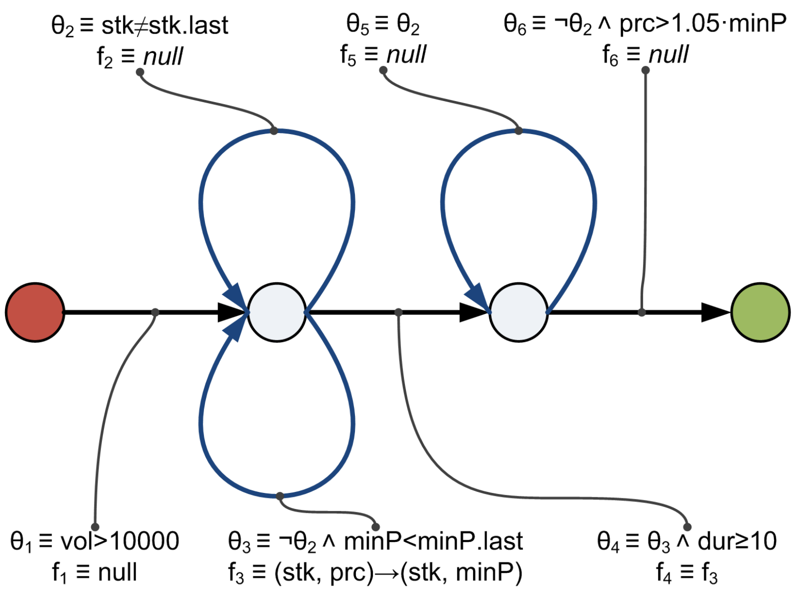


Figure 2.2: Nondeterministic finite state automaton [DF03]

2.4 Architectures

Previous sections have considered only single components of content-based systems. Applications (Section 2.1.1) play the roles of publishers (Section 2.1.3) and subscribers (Section 2.1.2) in order to publish content in form of publication messages (Section 2.2.2) and subscribe to content using subscription messages (Section 2.2.1). The fabric which connects the publishers and subscribers are the brokers (Section 2.1.4), which themselves can be interconnected creating various architectural patterns. In this section the most popular broker patterns and their influence on the way the information is being exchanged in publish/subscribe systems are described. Architectures described in the following sections are usually constructed as an overlay on top of the physical connections.

2.4.1 Bus

The bus architecture (see Figure 2.3) was one of the first ones to be proposed in context of the publish/subscribe systems [OPSS93]. Bus architecture assumes that all brokers share the same communication medium, implying that all messages are seen by all brokers. Bus architecture effectively implements a broadcast system for filters, events, and advertisements. Bus architecture assumes that brokers are connected so that there exist no cycles in the network. Bus architecture, although very simple, is simultaneously the most infrequent used one, as the broadcasting of messages to all brokers implies a very high overhead on the system and is usually not practical in its straightforward

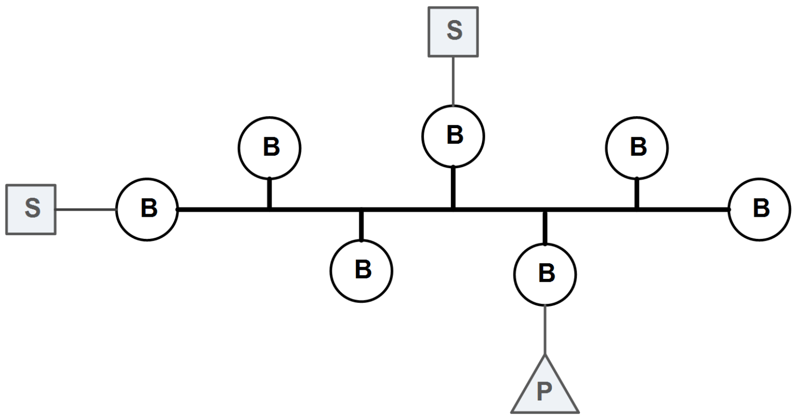


Figure 2.3: Brokers connected into a bus architecture

implementation.

2.4.2 Hierarchical

A hierarchical approach proposed in Siena [CRW01] and JEDI [CDNF01] publish/subscribe systems assumes the existence of broker network connected so as to form a *k*-ary tree – see Figure 2.4. The broker being root of the tree is responsible for forwarding all traffic from the left-hand side of the tree to the right-hand side of the tree. Every broker (except for the root) in hierarchical architecture is connected to *k* child brokers and one parent broker.
One of the main drawbacks of the hierarchical architecture is the fact that the higher the given broker is placed in the hierarchy the more load it has to cope with. This implies that the root node is prone to become overloaded, which is especially crucial if one considers that it is a single point of failure which can partition the whole system in two halves.

2.4.3 Acyclic

Acyclic architecture is one of the most commonly used abstraction for building of publish/subscribe systems [OAA+00, CRW01, MÖ2, CF04, CMPC04, FJLM05, Jer05, CP06, BH06, TK06, Eug07, JF08a]. Acyclic architecture differs from the hierarchical architecture (see Section 2.4.2) in that there is no explicit hierarchy maintained by the brokers. Specifically, every broker maintains a list of *k* peers without a distinct parent node – see Figure 2.5. This, in turn, implies that there is no root broker which has to handle most of the traffic. Acyclic architecture is therefore preferred to the hierarchical one, as the message flow is generally better (more fair) distributed across the nodes. Acyclic architecture, similarly to the hierarchical architecture, partitions if any single broker or link fails.

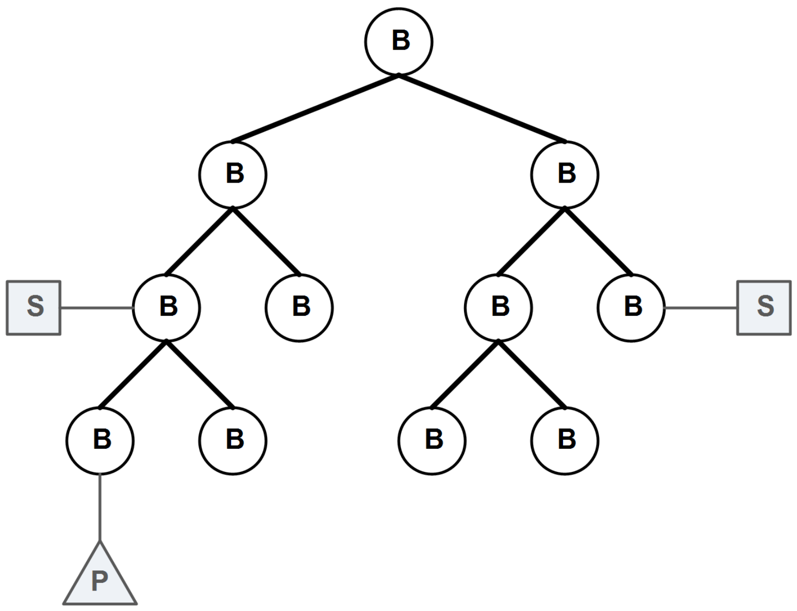


Figure 2.4: Brokers connected into a hierarchical architecture

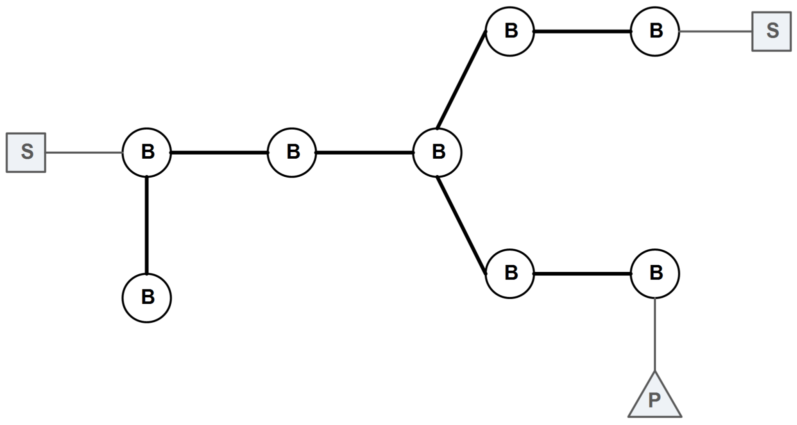


Figure 2.5: Brokers connected into an acyclic architecture

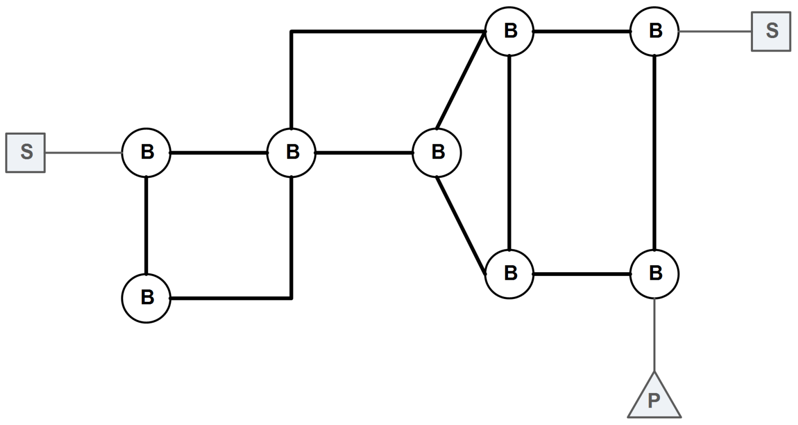


Figure 2.6: Brokers connected into a general cyclic architecture

2.4.4 Cyclic

The cyclic architecture (also known as mesh) is the most general interconnection type for brokers in content-based systems. Cyclic architecture is conceptually most similar to the abstraction provided by the TCP/IP networking where most of the nodes can talk directly to most of the other nodes in the system – see Figure 2.6. Cyclic networks introduce link redundancy, which implies that link or node failures do not necessarily result in the partitioning of the network, as alternative routes can be used to reach the same destination. However, the cyclic architecture also poses challenges as far avoidance of infinite message loops is concerned. This is especially difficult in publish/subscribe systems, since brokers are usually aware only of the immediate neighbor nodes – see Section 1.1.

In general, systems which implicitly or explicitly work at the level of cyclic architectures [PCM03, LMJ08] dynamically construct a spanning tree, e.g., using the reverse path forwarding method [DM78], which prevents the occurrence of cycles [PCM03] by reducing the cyclic case to the acyclic one.

2.5 Routing

The goal of the routing is to deliver publication messages produced at the publishers to the interested subscribers via a network of brokers. This section discusses the most popular routing strategies used in publish/subscribe systems and their variants depending on the types and semantics of messages used.

The discussion will focus on the acyclic broker architecture – it is assumed that in the case of a cyclic architecture an acyclic overlay is constructed (see Section 2.4.4) – an assumption frequently made in the context of the publish/subscribe systems [CRW01, MFP06, TK06]. For the discussion of algorithms dealing with the hierarchical architectures the reader is referred

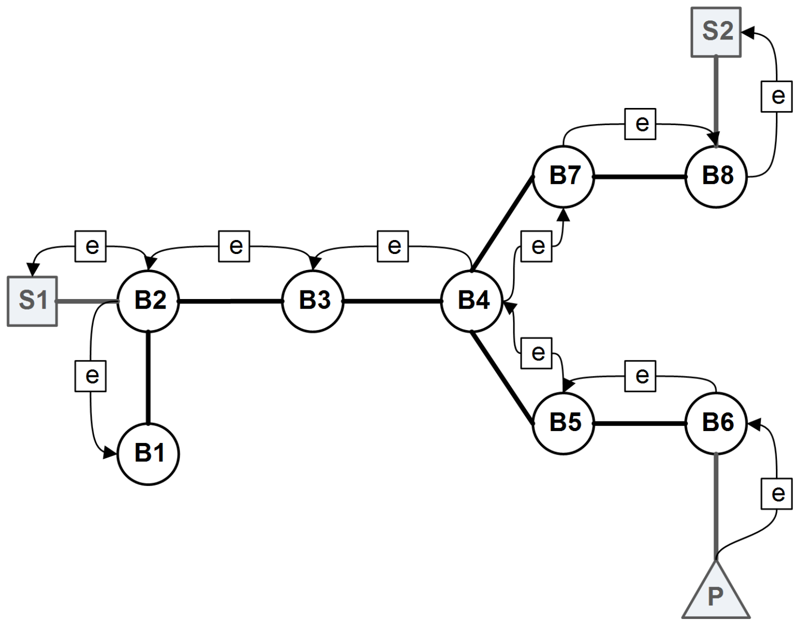


Figure 2.7: Event flooding in a broker overlay

to [CRW01] and [CDNF01].

2.5.1 Event Flooding

The event flooding strategy [MÖ2] (also known as broadcast) implies that every event message produced by every publisher is delivered to every subscriber, and thus effectively to every application. It is then up to the applications to filter out the events which are of no relevance to them. The event flooding approach is the simplest routing strategy (it is stateless in that it uses neither advertisements nor filters) among all routing strategies and can be trivially implemented in all architectures except the acyclic one.

When event flooding routing strategy is used broker **B** receiving an event *e* forwards it on all its outgoing interfaces with the exception of the interface on which the event has arrived – see Figures 2.7 and 2.9(a). It can be formally said that the set of interfaces $\mathbb{I}_{e\rightarrow}$ on which event *e* needs to be forwarded is given as:

$$\mathbb{I}_{e\rightarrow} = \mathbb{I}_{\mathbf{B}} \setminus \{I_{\leftarrow e}\} \tag{2.13}$$

where $\mathbb{I}_{\mathbf{B}}$ is the set of all interfaces of the broker **B** and $I_{\leftarrow e}$ is the interface on which event *e* arrived.

Event flooding algorithm, although very simple, is not practical as in large deployments the cost of delivery of every message to every node, event not interested in the specific kind of information is prohibitive.

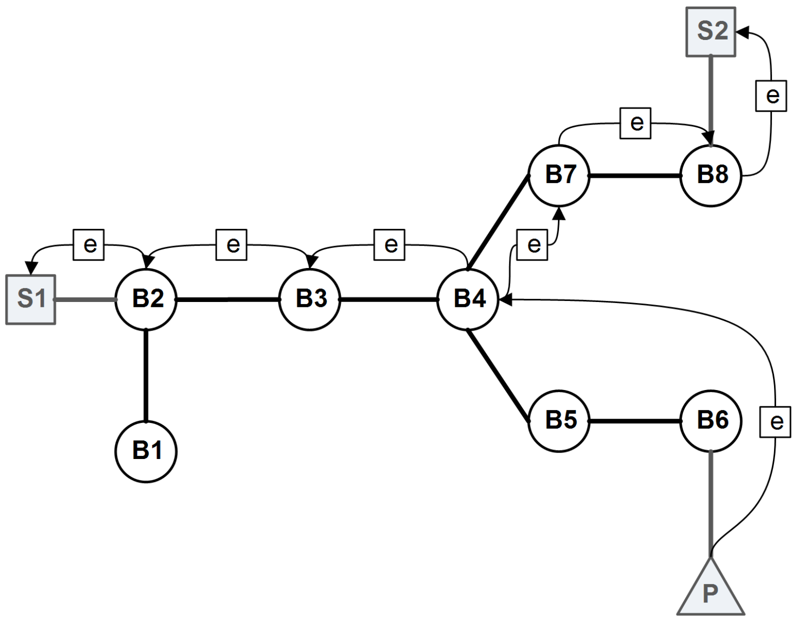


Figure 2.8: Group-based routing in a broker overlay

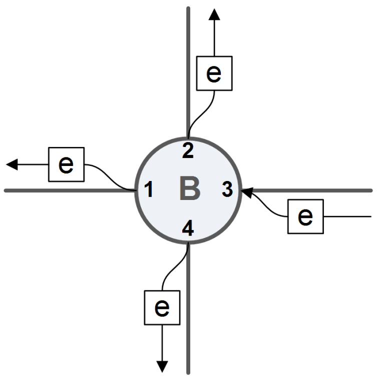
2.5.2 Group-Based Routing

Group-based routing [RHKS01, BEG04, Gro04] is an approach which can be used to limit the flooding of events used in the event flooding routing scheme. Group-based routing collects all subscribers sharing the same interest into groups for which multicast techniques are used to disseminate events. Figure 2.8 shows the group routing strategy in a network of brokers. It can be observed that publisher **P** publishes events directly to the broker **B4** which is the head of the group which distributes the event to both interested subscribers **S1** and **S2**

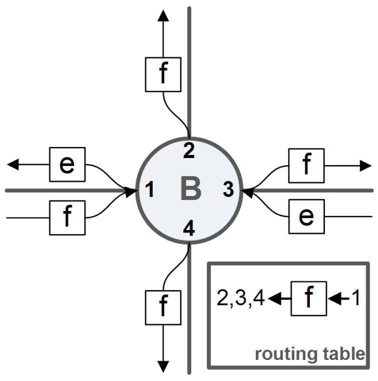
One of the multicast approaches often considered in context of the group-based routing is the IP multicast [DC90]. The group based approach has the advantage (in comparison to the event flooding strategy) of reducing the number of false positives in the publish/subscribe system. A false positive occurs when subscriber receives event which is of no interest to him. By grouping subscribers and delivering related events only to the interested groups the false positives ratio is reduced to zero.

The group-based approach is however difficult to implement and maintain, especially in the environments with large number of subscribers. Having a set of $\mathbb{S}$ subscribers group-based approach implies the need to construct $2^{|\mathbb{S}|}$ groups (the worst case) to satisfy the interests of all subscribers. Moreover, an algorithm which would try to limit the number of groups by maximizing they overlap and reducing the number of messages needed is known to be NP-complete [AGK+01].

Therefore several approaches have appeared which try to overcome the above



(a) event flooding



(b) simple routing

Figure 2.9: Event flooding and simple routing strategies

issue by using either probabilistic methods [EG02] or heuristics for the construction of groups [OAA+00, RLW+02]. However, most of the recent algorithms substitute the notion of groups with dynamic filtering in overlay topologies.

2.5.3 Simple Routing

In simple routing (see, e.g., [BCM+99]) the propagation of events is restricted by the propagation of filters. Subscribers issue subscription messages containing filters which are broadcast into the network of brokers. Upon reception of a filter f every broker: (1) forwards it on all outgoing interfaces, with the exception of the interface on which the filter f arrived and (2) stores the filter f in its routing state. Figure 2.9(b) shows the propagation of the filter f, arriving at the interface number 1 of the broker **B**. The simple routing algorithm results in creation of the filter spanning trees rooted at the subscriber issuing the given filter.

Filter Routing

In simple routing strategy every broker maintains the routing state associated with the filters. The routing state in the brokers is usually called the routing table and contains all filters issued by subscribers which were not unsubscribed. Figure 2.9(b) illustrates the routing table of the broker **B** which contains one entry, stating that the filter f has arrived on the interface number 1 and was forwarded on the interfaces number 2, 3 and 4. By analogy to the Equation 2.13 one can say that the set of interfaces $\mathbb{I}_{f\rightarrow}$ on which filter f needs to be forwarded is given as:

$$\mathbb{I}_{f\rightarrow} = \mathbb{I}_{\mathbf{B}} \setminus \left\{ I_{\leftarrow f} \right\} \tag{2.14}$$

where $I_{\leftarrow f}$ is the interface on which filter f arrived.

Event Forwarding

The routing tables stored in the brokers are used for the forwarding of events. An event e arriving at the broker **B** and matching filter f (the exact definition of the match depends on the semantics used – see, e.g., the Equation 2.8) is forwarded on the reverse path of the filter f. Figure 2.9(b) shows event e being forwarded on the interface 1 on which previously a matching filter f arrived. Formally speaking, it can said that the set of interfaces $\mathbb{I}_{e\rightarrow}$ on which event e has to be forwarded by a given broker **B** is given as a sum of all interfaces on which filters matching event e arrived minus the source interface $I_{\leftarrow e}$ of the event e:

$$\mathbb{I}_{e\rightarrow} = \left\{ I_{\leftarrow f} \quad | \quad f \in \mathbb{F}_{\mathbf{B}} \wedge f(e) = 1 \wedge I_{\leftarrow f} \not\equiv I_{\leftarrow e} \right\} \tag{2.15}$$

where $\mathbb{F}_{\mathbf{B}}$ represents all unsubscribed filters seen by the broker **B**. $\mathbb{F}_{\mathbf{B}}$ can be also seen as a set of all filters stored in the routing table of the broker **B**.

Intuitively, the Equation 2.15 implies that events are forwarded along the branches of the filter spanning tree until they reach the subscriber at its root. The propagation of filters to every broker in the publish/subscribe networks ensures that there exists at least one broker in the whole publish/subscribe network which can correlate the interest of subscribers with the events of the publisher, regardless of the publisher's location in the publish/subscribe network.

Figure 2.10 shows the routing of subscription messages and events in the simple routing scheme. It can be observed that both filters issued by the subscribers **S1** and **S2** and disseminated into the whole network – Figure 2.10(a). A subsequently issued event, matching both filters, is delivered on the reverse paths of the subscription messages to both subscribers – Figure 2.10(b).

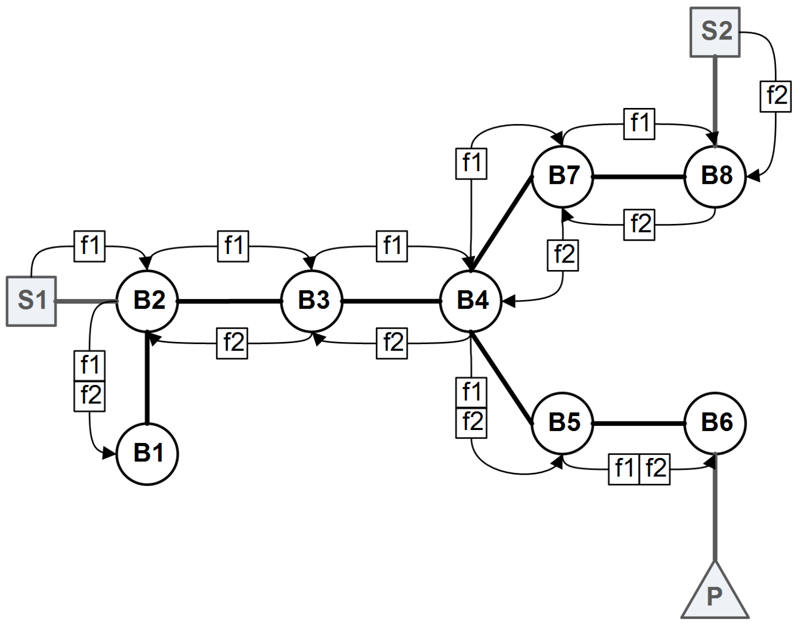
Unsubscribing

Subscribers which are not interested in a given content anymore issue unsubscription messages to inform the publish/subscribe system about that fact. The process of the unsubscription of the filter f by the broker **B** in case of simple routing consists of two phases: (1) the forwarding of the unsubscription message containing the filter f and (2) the removal of the filter f from the routing table of the broker **B**.

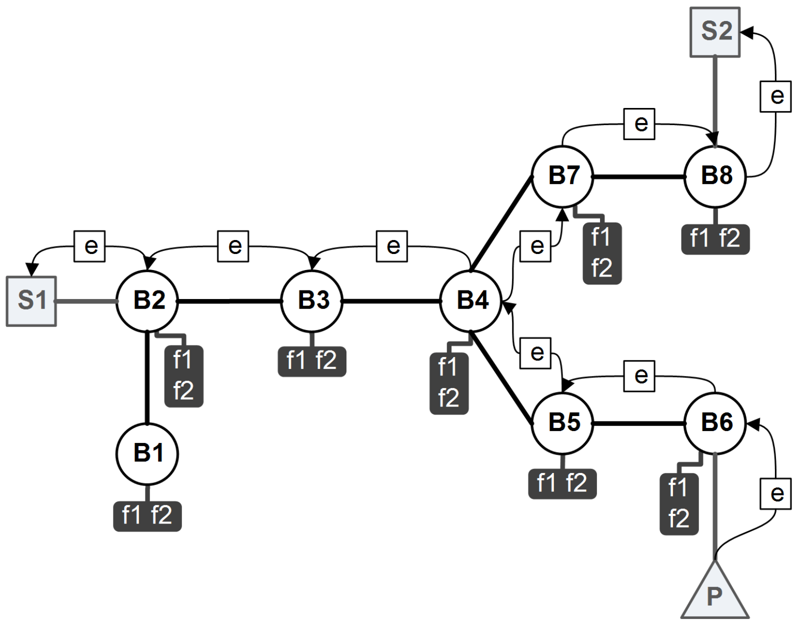
The process of forwarding of the unsubscription message containing the filter f is identical to that of subscription messages:

$$\mathbb{I}^{\text{unsub}}_{f\rightarrow} = \mathbb{I}_{\mathbf{B}} \setminus \left\{ I_{\leftarrow f} \right\} \tag{2.16}$$

The removal of the filter f from the routing table of the broker **B**, requires broker **B** to be able to determine which filter in its routing table is identical with the filter f.



(a) routing of filters



(b) forwarding of events

Figure 2.10: Simple routing in a broker overlay

The checking if two filters are identical is specific to the assumed filter semantics. A general definition says that two filters f and g are identical if they select the same sets of events [MÖ2]:

$$f \equiv g \quad \Leftrightarrow \quad \mathbb{E}_f \equiv \mathbb{E}_g \tag{2.17}$$

where $\mathbb{E}_f$ and $\mathbb{E}_g$ are the sets of events selected by filters f and g, respectively. In the context of the predicate-based semantics it is said that two filters f and g are identical $f \equiv g$ if and only if:

$$f \equiv g \quad \Leftrightarrow \quad \forall_{p_f \in f} \exists_{p_g \in g} : p_f \equiv p_g \wedge \forall_{p_g \in g} \exists_{p_f \in f} : p_f \equiv p_g \tag{2.18}$$

where p_f and p_g are predicates belonging to filters f and g, respectively. Two predicates p_f and p_g are identical $p_f \equiv p_g$ if and only if:

$$p_f \equiv p_g \quad \Leftrightarrow \quad an_f \equiv an_g \wedge op_f \equiv op_g \wedge av_f \equiv av_g \tag{2.19}$$

where an_f, op_f and av_f are attribute name, operator and attribute value of the filter f – see Section 2.3.2.
The filter g which needs to be removed (using the remove(g) operation) from the routing table of the broker **B** upon the reception of the unsubscription message containing filter f is therefore given as:

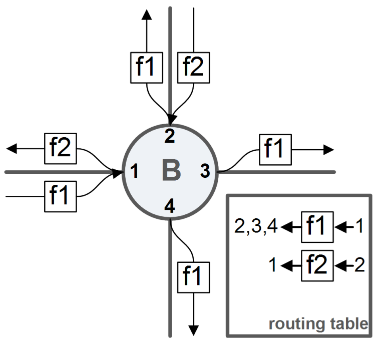
$$\text{remove}(g) \quad \Leftrightarrow \quad g \in \mathbb{F}_{\mathbf{B}} \wedge g \equiv f \wedge I_{\leftarrow g} \equiv I_{\leftarrow f} \tag{2.20}$$

where the term $I_{\leftarrow g} \equiv I_{\leftarrow f}$ guarantees that in case of multiple filters equal to filter f stored at the broker **B** only the one which arrived on the same interface as the unsubscription message containing filter f ($I_{\leftarrow f}$) is erased.

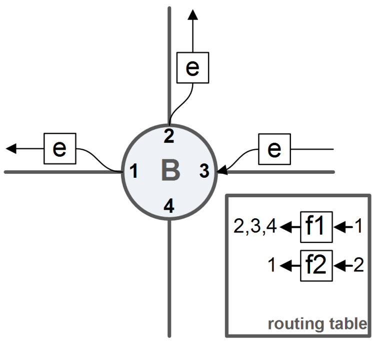
Dynamic Overlay Management

So far only cases when the set of the interfaces $\mathbb{I}_{\mathbf{B}}$ of the broker **B** was stable have been considered. However in publish/subscribe systems brokers or subscribers often depart and new brokers and subscribers often join the overlay network. A departure of a broker implies that the interface on which it was connected to its neighbors is removed. The arrival of a new broker results in a new interface being added to the set of interfaces $\mathbb{I}$ on all of its new neighbors.
When one of the interfaces I_i is removed from the set of interfaces of the broker **B**, the routing table of the broker **B** needs to be updated accordingly. The update process requires that every filter f, stored in the routing table of the broker **B**, which arrived on the removed interface I_i is usubscribed (and therefore removed – see Equations 2.16 and 2.20) from the routing table of the broker **B**.
The addition of a new interface I_i requires the broker **B** to propagate all filters in its routing table on the new interface:

$$\forall_{f \in \mathbb{F}_{\mathbf{B}}} : \mathbb{I}_{f \rightarrow} \leftarrow I_i \tag{2.21}$$



(a) advertisement and subscription routing



(b) event forwarding

Figure 2.11: Identity-based routing strategy

2.5.4 Identity-Based Routing

Using Equation 2.18 the routing of filters can be optimized in that identical filters are not propagated on the same interface. Such routing strategy is called identity-based routing [MÖ2, MFP06]. The basic idea of the identity-based routing relies on the fact that identical filters forwarded on the same interface are idempotent with respect to the routing tables of the downstream brokers.
Figure 2.11(a) shows the process of the routing of two filters $f1$ and $f2$. Filter $f1$ was the first one to arrive at the broker **B** and since it was the first filter to be seen by **B** it was forwarded on all outgoing interfaces. Subsequent arrival of the filter $f2$ identical to $f1$ ($f1 \equiv f2$) results in filter $f2$ being propagated only on the interface 1 as this is the only interface on which the identical filter $f1$ was not forwarded yet. It can be observed that identity-based routing has prevented the propagation of the filter $f2$ on two interfaces: 3 and 4. It is also worth noting that the third filter $f3$ identical to $f1$ and $f2$ ($f1 \equiv f2 \equiv f3$) arriving at interface 4 or 3 will not be propagated further by the broker **B** at all, as there is no interface on broker **B** which would not have either the filter $f1$ or $f2$ already propagated on it. Such filter would only be stored in the routing table of the broker **B**.

Filter Routing

It can be formally said that in identity-based routing filter f arriving at broker **B** is propagated on all interfaces with the exception of its source interface $I_{\leftarrow f}$ and interfaces on which filters identical to f were already propagated:

$$\mathbb{I}_{f\rightarrow} = \left(\mathbb{I}_{\mathbf{B}} \setminus \left\{I_{\leftarrow f}\right\}\right) \setminus \left\{\mathbb{I}_{g\rightarrow} \quad | \quad g \in \mathbb{F}_{\mathbf{B}} \wedge g \equiv f\right\} \tag{2.22}$$

where $\mathbb{I}_{g\rightarrow}$ is a set of all interfaces on which filter g was forwarded.
A filter f received by the broker **B** is dropped (using the drop(f) operation) if and only if a filter g identical to f was previously delivered to the broker **B** on

the same interface on which filter f arrived:

$$\text{drop}(f) \quad \Leftrightarrow \quad \exists_{g \in \mathbb{F}_{\mathbf{B}}} : g \equiv f \wedge I_{\leftarrow g} \equiv I_{\leftarrow f} \tag{2.23}$$

A filter f being dropped is neither forwarded nor stored in the routing table of the broker **B**.

Event Forwarding

Figure 2.11(b) illustrates the process of event forwarding when identity-based routing is used. It can be observed that event e which matches filter $f1$ (and thus filter $f2$ as well, since $f1 \equiv f2$) is forwarded on the reverse path of both matching filters. Therefore, one can conclude that the event forwarding process when the identity-based routing strategy is used is identical with the one presented in Equation 2.15.
Figure 2.12 shows the process of identity-based routing in a broker overlay. It can be observed that subscribers **S1** and **S2** subscribe using two identical filters – Figure 2.12(a). Therefore, the propagation of the filter issued by the subscriber **S1** (first filter) reaches all brokers in the publish/subscribe network. The subsequent filter issued by the **S2** is propagated only on the directed paths not traversed by the identical filter previously issued by the subscriber **S1**. The propagation of events (Figure 2.12(b)) is analogous to that of the simple routing – see Figure 2.10(b).

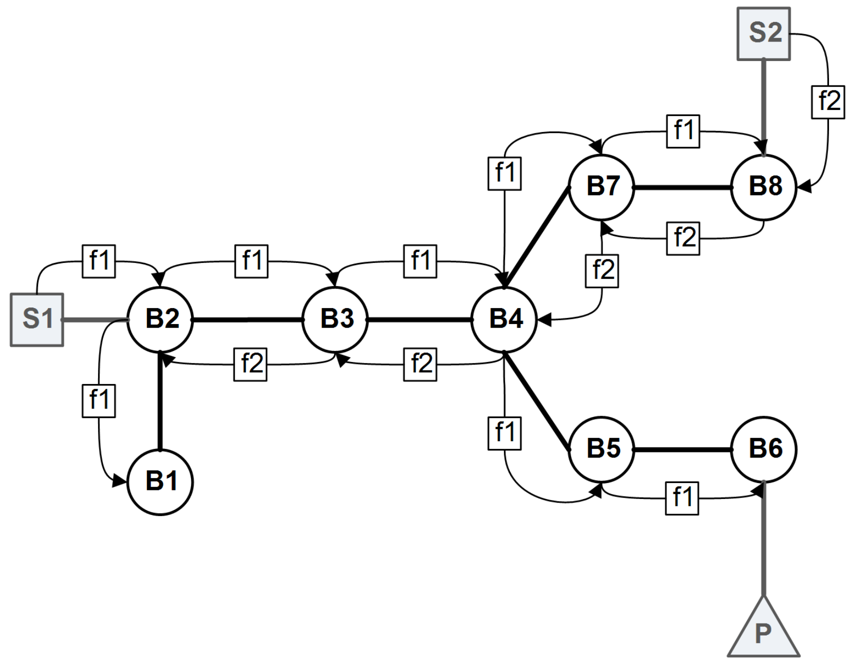
Unsubscribing

The process of unsubscribing a filter is more complex than in the case of simple routing. One needs to consider that the propagation of the filter $f2$ on interfaces 3 and 4 (see Figure 2.11(a)) was prevented by the previous propagation of the filter $f1$ on those interfaces. However, upon the unsubscription of the filter $f1$, filter $f2$ should be propagated on the interfaces 3 and 4.
Therefore, the process of the unsubscription of a given filter f performed by the broker **B** consists of three phases: (1) propagation of the unsubscription message containing the filter f, (2) propagation of the subscription messages containing filters identical to f which were not propagated due to the presence of f and (3) removal of f from the routing table of the broker **B**.
The propagation of the unsubscription message containing the filter f is performed for a set of interfaces containing all interfaces on which identical filters arriving from the same source interface were forwarded:

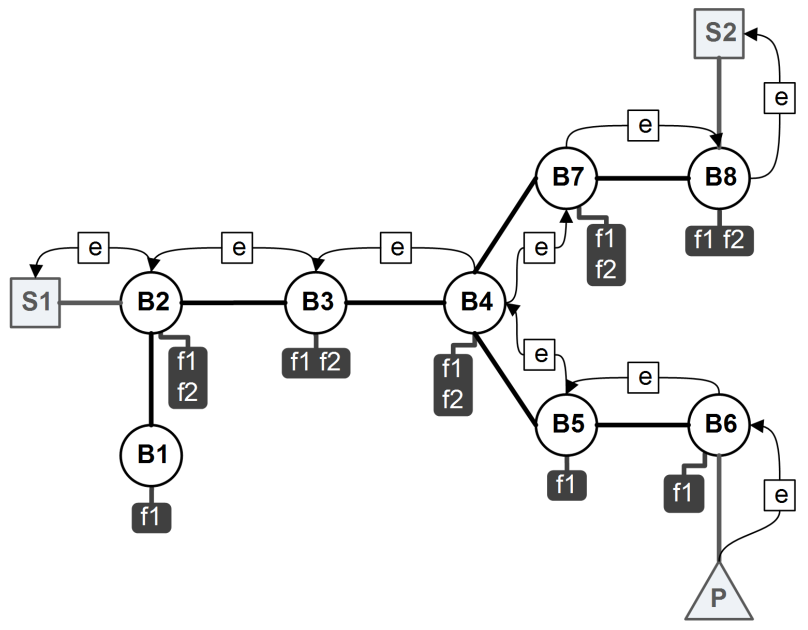
$$\mathbb{I}^{\text{unsub}}_{f\rightarrow} = \left\{ \mathbb{I}_{g\rightarrow} \quad | \quad g \in \mathbb{F}_{\mathbf{B}} \wedge g \equiv f \wedge I_{\leftarrow g} \equiv I_{\leftarrow f} \right\} \tag{2.24}$$

The propagation of the subscription messages containing filters identical to f which were not propagated due to the presence of f requires the broker **B** to remove all filters equal to the unsubscribed filter f from its routing table:

$$\text{remove}(g) \quad \Leftrightarrow \quad g \in \mathbb{F}_{\mathbf{B}} \wedge g \equiv f \wedge I_{\leftarrow g} \not\equiv I_{\leftarrow f} \tag{2.25}$$



(a) routing of filters



(b) forwarding of events

Figure 2.12: Identity-based routing in a broker overlay

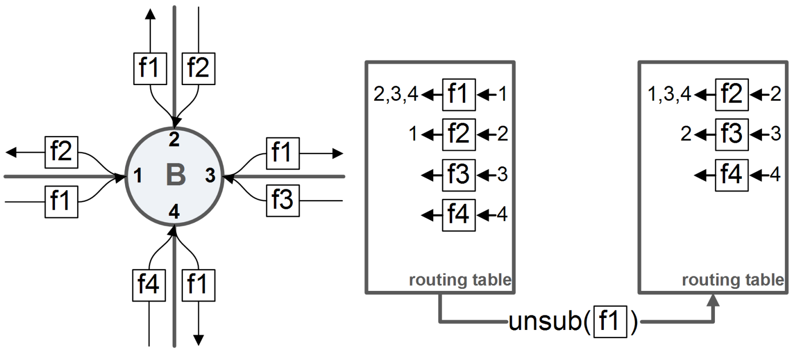


Figure 2.13: Unsubscription using identity based routing strategy

and resubscribe the removed filters again. Filters need to be removed from the routing table of the broker **B** in order not to get dropped at the resubscription attempt – see Equation 2.23.

As an example let us consider the Figure 2.13, which shows broker **B** after four subscription messages with identical filters $f1$, $f2$, $f3$ and $f4$ ($f1 \equiv f2 \equiv f3 \equiv f4$) have been received. Through inspection of the routing table it can be concluded that the filter $f1$ was subscribed to as the first one, subsequently filter $f2$ was subscribed with filters $f3$ and $f4$ subscribed to as the last ones. The second routing table shows the state of the broker **B** after the unsubscription message for the filter $f1$ arrived at the interface 1. It can be observed that filter $f1$ was removed from the broker's **B** routing table. Subsequently, filters $f2$, $f3$ and $f4$ were resubscribed to by the broker **B**. The filter $f2$ was the first one to be resubscribed to, followed by the filters $f3$ and $f4$. It is worth noting that the order of the filter resubscription is irrelevant.

Dynamic Overlay Management

When one of the interfaces I_i is removed from the set of interfaces of the broker **B**, the routing table of the broker **B** needs to be updated accordingly – every filter f stored in the routing table of the broker **B** which arrived on the removed interface I_i needs to be usubscribed (see Equations 2.24 and 2.25) from the routing table of the broker **B**.

The addition of a new interface I_i requires the broker **B** to propagate all filters in its routing table on the new interface, under the assumption that no filter identical to the one to be propagated on the interface I_i has already been propagated on the added interface I_i:

$$\forall_{f \in \mathbb{F}_\mathbf{B}} \left(\text{forward}(f) = I_i \Leftrightarrow \nexists_{g \in \mathbb{F}_\mathbf{B}} : g \equiv f \wedge I_i \in \mathbb{I}_{g\rightarrow} \right) \tag{2.26}$$

2.5.5 Coverage-Based Routing

Many existing publish/subscribe systems [MÖ2, BCV03, CRW04, LHJ05, TK06, BFG07, BBQV07, JF08a, LMJ08] use the notion of coverage to optimize the routing of subscription messages. Coverage (first introduced in context of publish/subscribe systems in [CRW01]) is a binary relation between filters. Intuitively, it can be said that filter f covers filter g ($f \succ g$) if filter f matches the superset of events matched by the filter g:

$$f \succ g \quad \Leftrightarrow \quad \mathbb{E}_f \supseteq \mathbb{E}_g \tag{2.27}$$

According to the above definition if filter f is identical to filter g, than filter f covers filter g and vice versa:

$$f \equiv g \quad \Rightarrow \quad f \succ g \wedge f \prec g \tag{2.28}$$

The coverage relation is:

- reflexive: $g \prec f$ always holds
- antisymmetric: $f \succ g \wedge f \prec g \;\Rightarrow\; f \equiv g$
- transitive: $h \succ g \wedge g \succ f \;\Rightarrow\; h \succ f$

However, the coverage relation is not total ($\exists_{f,g} \;:\; f \nprec g \wedge f \nsucc g$), which implies that coverage relation creates a partial order for all filters.
Coverage relation is often used in the predicate-based semantics – however, geometrical applications are also common [RLW+02, WQV+04, BFG07]. Using predicate-based semantics one can say that filter f covers filter g if and only if:

$$f \succ g \quad \Leftrightarrow \quad \forall_{p_f \in f} \exists_{p_g \in g} \;:\; p_f \succ p_g \tag{2.29}$$

which can be read as: for every predicate p_f in covering filter f there exists a predicate p_g in covered filter g such that predicate p_f covers p_g. Predicate p_f covers predicate p_g if and only if:

$$p_f \succ p_g \quad \Leftrightarrow \quad an_f \equiv an_g \wedge ac_f \succ ac_g \tag{2.30}$$

The exact semantics of the attribute constraint coverage ($ac_f \succ ac_g$) depend on the operators and values used for the specific implementation. The reader is referred to [MFP06] for an overview of simple attribute constraints and their coverage relations. Intuitively, an attribute constraint ac_f covers attribute constraint ac_g if attribute constraint ac_f selects a superset of attribute values selected by attribute constraint ac_g.
Table 2.2 illustrates example coverage relations in predicate-based semantics (cf. Table 2.1) between two filters f and g.

Filter f	Filter g	$f \succ g$	$f \prec g$
$\{y < 8\}$	$\{y < 8\}$	true	true
$\{x > 5\}$	$\{y > 5\}$	false	false
$\{x > 5\}$	$\{x > 8\}$	true	false
$\{x > 5\}$	$\{x > 8;\ y > 5\}$	true	false
$\{x > 5;\ y > 5\}$	$\{x > 8\}$	false	false
$\{x \neq 5\}$	$\{x > 8\}$	true	false
$\{x \neq 5\}$	$\{x = 5\}$	false	false
$\{x > 5\}$	$\{x < 8\}$	false	false
$\{x \geq 5\}$	$\{x > 5\}$	true	false

Table 2.2: Coverage relation between filters in predicate-based semantics

Filter Routing

Following the intuition behind the coverage relation of two filters, it can be stated (similarly as in the case of filter identity) that the propagation of a filter g on the interface I is not necessary if there exists filter f previously propagated on the interface I, which covers g. Filter f selects a superset of events selected by filter g (see Equation 2.27) and therefore the propagation of filter g will not influence the behavior of the downstream brokers in any way.
The usage of coverage to limit the propagation of filters allows to reduce the load on both the network and the brokers in the publish/subscribe system. It can be formally said that in coverage-based routing a filter f arriving at the broker **B** on interface $I_{\leftarrow f}$ is propagated on all interfaces with the exception of its source interface and interfaces on which filters covering filter f were already propagated:

$$\text{forward}(f) = \left(\mathbb{I}_{\mathbf{B}} \setminus \left\{I_{\leftarrow f}\right\}\right) \setminus \left\{\mathbb{I}_{g\rightarrow} \quad | \quad g \in \mathbb{F}_{\mathbf{B}} \wedge g \succ f\right\} \tag{2.31}$$

A filter f received by the broker **B** is dropped (neither forwarded nor stored in the routing table of the broker **B**) if and only if a filter g covering filter f was previously delivered to the broker **B** on the same interface on which filter f arrived:

$$\text{drop}(f) \quad \Leftrightarrow \quad \exists_{g \in \mathbb{F}_{\mathbf{B}}} : g \succ f \wedge I_{\leftarrow g} \equiv I_{\leftarrow f} \tag{2.32}$$

Event Forwarding

Event forwarding in coverage-based routing is similar to that of the identity-based routing – see Section 2.5.4. Formally speaking, the set of interfaces on which event e has to be forwarded by a given broker **B** is given as a sum of all interfaces on which filters matching event e arrived minus the source interface $I_{\leftarrow e}$ of the event e:

$$\mathbb{I}_{e\rightarrow} = \left\{ I_{\leftarrow f} \quad | \quad f \in \mathbb{F}_{\mathbf{B}} \land f(e) = 1 \land I_{\leftarrow f} \neq I_{\leftarrow e} \right\} \tag{2.33}$$

The process of the coverage-based routing in a broker overlay is identical with the one demonstrated in the Figure 2.12. The only difference is that instead of strict filter identity between the filters issued by the subscriber **S1** and **S2** a coverage relation will now suffice.

Unsubscribing

Similarly as in the case of the identity-based routing the process of the unsubscription of a given filter f performed by the broker **B** consists of three phases: (1) propagation of the unsubscription message containing the filter f, (2) propagation of the subscription messages containing filters covered by the filter f which were not propagated due to the presence of f and (3) removal of f from the routing table of the broker **B**.
The propagation of the unsubscription message containing filter f is performed for a set of interfaces on which identical filters arriving from the same source interface were forwarded:

$$\mathbb{I}^{\text{unsub}}_{f\rightarrow} = \left\{ \mathbb{I}_{g\rightarrow} \quad | \quad g \in \mathbb{F}_{\mathbf{B}} \land g \equiv f \land I_{\leftarrow g} \equiv I_{\leftarrow f} \right\} \tag{2.34}$$

The propagation of the subscription messages containing filters covered by f which were not propagated due to the presence of f requires the broker **B** to remove all filters covered by the unsubscribed filter f from the broker **B**'s routing table:

$$\text{remove}(g) \quad \Leftrightarrow \quad g \in \mathbb{F}_{\mathbf{B}} \land g \prec f \land I_{\leftarrow g} \not\equiv I_{\leftarrow f} \tag{2.35}$$

and resubscribe the removed filters again. Filters need to be removed from the routing table of the broker **B** in order not to get dropped at the resubscription attempt – see Equation 2.32.

Dynamic Overlay Management

The removal of the interface I_i from the routing table of the broker **B** results in every filter f stored in the routing table of the broker **B** which arrived on the removed interface I_i being usubscribed – see Equations 2.34 and 2.35.
The addition of a new interface I_i results in the propagation of all filters in the routing table of the broker **B** on the new interface. The filters are propagated stepwise, under the assumption that no filter covering the one to be currently propagated on the interface I_i has already been propagated there:

$$\forall_{f \in \mathbb{F}_{\mathbf{B}}} \left(\text{forward}(f) = I_i \Leftrightarrow \nexists_{g \in \mathbb{F}_{\mathbf{B}}} : g \succ f \land I_i \in \mathbb{I}_{g\rightarrow} \right) \tag{2.36}$$

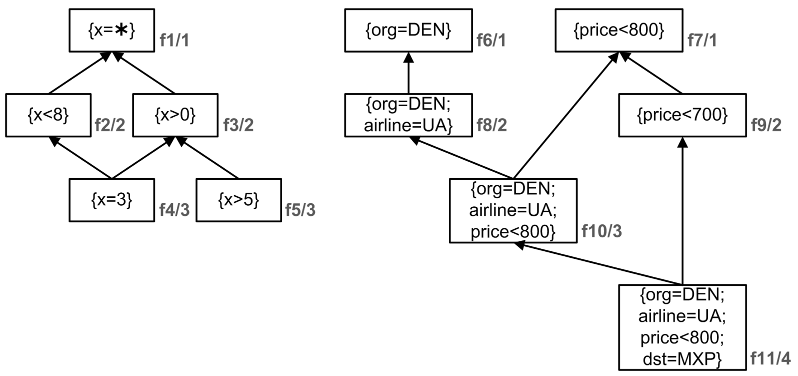

{x=∗} f1/1
{x<8} f2/2
{x>0} f3/2
{x=3} f4/3
{x>5} f5/3
{org=DEN} f6/1
{price<800} f7/1
{org=DEN; airline=UA} f8/2
{price<700} f9/2
{org=DEN; airline=UA; price<800} f10/3
{org=DEN; airline=UA; price<800; dst=MXP} f11/4

Figure 2.14: Partially ordered set (poset) [CRW01]

Routing Table Implementations

Coverage-based routing algorithms imply that the implementation of the routing tables can exploit the coverage relation in order to increase the speed of event forwarding, especially in comparison to naive approaches. The first such design of the routing table was proposed in the context of the SIENA system [CRW01].

The SIENA routing table is called *poset* – a name derived from the partial order property of the coverage relation. Poset contains all filters which arrive at the broker and are not dropped – cf. Equation 2.32. In poset filters are arranged according to their coverage relation – the most general filters, i.e., not covered by any other filter, are placed as the root of the structure. Figure 2.14 shows a poset structure holding eleven filters ($f1, \ldots, f11$). Each filter is denoted its name and the interface number on which it arrived. It can be observed that the most general filter with attribute name x ($\{x = *\}$) which arrived on the interface 1 ($f1/1$) covers a more specific filter $\{x < 8\}$ which arrived on the interface 2 ($f2/2$). Entries in the poset structure which have no parent are called root filters.

The matching of events is excelled in that every event needs only to be checked against the set of the root filters in order to determine whether any of the filters stored in the poset matches a given event. In case of a match the matching procedure uses breadth first search to traverse the children of the root filter in order to find all, more specific filters matching the given event.

The insertion of a new filter f is performed by computing a set of predecessors and a set of successors of the filter f. The set of predecessors contains all filters in the poset which cover the filter f, while the set of successors contains all filters in the poset which are covered by the filter f. In the last step of the insertion process filter f is inserted into poset as a parent of all filters in the successor set and a child of all filters in the predecessor set. The removal process is analogous.

Authors in [TK06] proposed an optimized version of the poset, called poset-

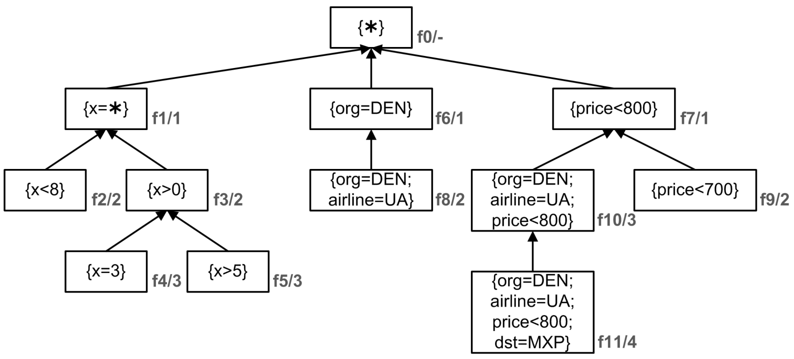


Figure 2.15: Poset-derived forest [TK06]

derived forest. The development of the poset derived forest was motivated by a set of observations regarding the properties of the poset structure:

1. for any filter f in the poset $\mathcal{P}$ there exists no such filter g, such that g is covered by f and filter g arrived on the same interface as f:

$$\forall_{f\in\mathcal{P}}\nexists_{g\in\mathcal{P}} \quad : \quad f \succ g \land I_{\leftarrow f} \equiv I_{\leftarrow g} \tag{2.37}$$

2. every poset has the maximum depth of $|\mathbb{I}_{\mathbf{B}}|$, where $|\mathbb{I}_{\mathbf{B}}|$ is the amount of interfaces on the poset managing broker **B**

3. only root elements and their direct successors (children) have a non-empty set of interfaces on which they were forwarded $\mathbb{I}_{f\rightarrow} \neq \emptyset$

The above properties allow to reduce the computational cost of the maintenance of the routing table structure as well as increase the speed of event matching. The resulting data structure, containing all filters from the poset in Figure 2.14, is illustrated in Figure 2.15. It can be observed that the poset-derived forest maintains the tree property in that every filter has at most one covering filter. This implies that the poset-derived forest layout is influenced by the temporal order of the filter arrival. Moreover, the implementation may contain a virtual root node which connects all root filters – a convenience element. The poset-derived forest data structure was further refined in [Tar07], in that links between different poset-derived forests were proposed to accelerate the process of the calculation of a set of covered and covering filters.
Another data structure, based on the counting algorithm [YGM94, PFLS00, FJL+01] was proposed in [CW03]. Authors concentrated on improving the speed of event forwarding as the cost of the filter routing speed. The proposed algorithm partitions filters into separate predicates and subsequently into attribute names and attribute constraints.
Figure 2.16 illustrates the data structure based on the counting algorithm, which contains filters identical to those stored in the poset (Figure 2.14) and

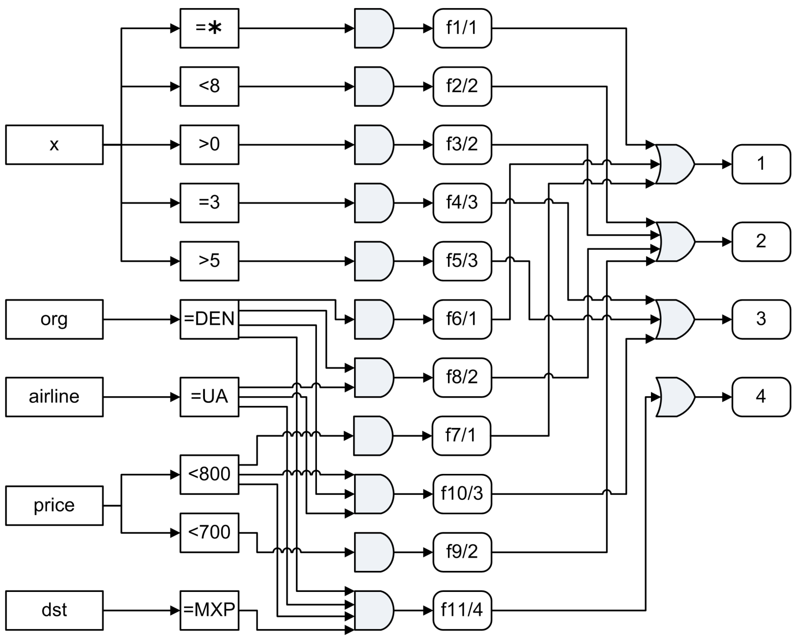


Figure 2.16: Forwarding table [CW03]

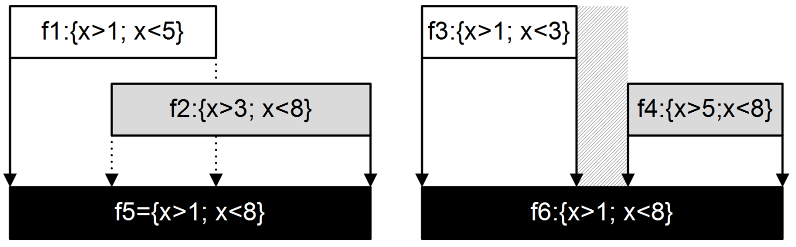
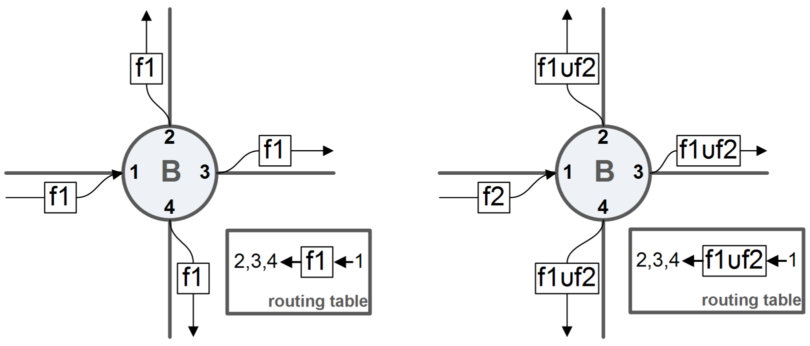


Figure 2.17: Perfect and imperfect filter merging

poset-derived forest (Figure 2.15) data structures. Predicates are arranged as inputs to the logical AND gates which control the triggering of the filters. Filters, in turn, are inputs to the logical OR gates which control the triggering of the interfaces. An incoming event *e* is decomposed into attribute name and attribute value pairs which are subsequently used to trigger the predicates. A predicate is triggered if it matches the attribute name and attribute value pair. Triggered predicates propagate to filters and filters propagate to interfaces on which a given event should be forwarded.

The above counting (the number of inputs to trigger the AND gate) algorithm outperforms both poset and poset-derived forest data structures. However, the insertion of new filters as well as removal of the old ones from the forwarding table data structure is more expensive than in case of the former data structures. Another drawback of the counting algorithm is the fact that it is stateful with respect to subsequent event matches – the counters used for the logical AND gates need to be reset prior to the matching of a new event.



(a) first filter (b) second filter merging with the first one

Figure 2.18: Filter routing using filter merging

Authors in [MFB02, CBGM03, TK05, Tar08a] propose the notion of filter merging to reduce the number of filters transmitted and stored in the routing tables of the brokers. The idea of filter merging is based on the fact that two or more different filters can be combined so as to be represented by only one filter. Figure 2.17 shows examples of filter merging: filters $f1$ and $f2$ form a perfect merger $f5$, whilst filters $f3$ and $f4$ form an imperfect merger $f6$.

Imperfect merging is an alternative to perfect merging, especially in situations where perfect merging is either too complex or difficult to compute [MFP06] – optimal merging of filters has been shown to be NP-hard [CBGM03]. The main drawback of the imperfect filter merging is the increased possibility of false positives. A false positive can be defined in this context as an event which has been delivered to the subscriber **S** or the broker **B**, for which there exists no active subscription in that subscriber or broker. Occurrence of false positives is especially troublesome as filter merging is often used to reduce the number of the root filters in the routing tables of the brokers. The reader is referred to [MFP06, Tar08a] for an overview of algorithms for filter merging and routing using filter merging.

Figure 2.18 shows the routing of filters if filter merging is enabled at the routing broker **B**. The arrival of the first filter $f1$ (Figure 2.18(a)) and its propagation is identical as in case of identity-based routing (Section 2.5.4), or coverage-based routing (Section 2.5.5). However, the arrival of the second filter $f2$ which can be merged with the filter $f1$ results in the propagation of the merger $f1 \cup f2$ of both filters $f1$ and $f2$. The merger of both filters will remove the previously propagated filter $f1$ from the routing tables of the downstream brokers (interfaces 2, 3 and 4) – see Equation 2.37.

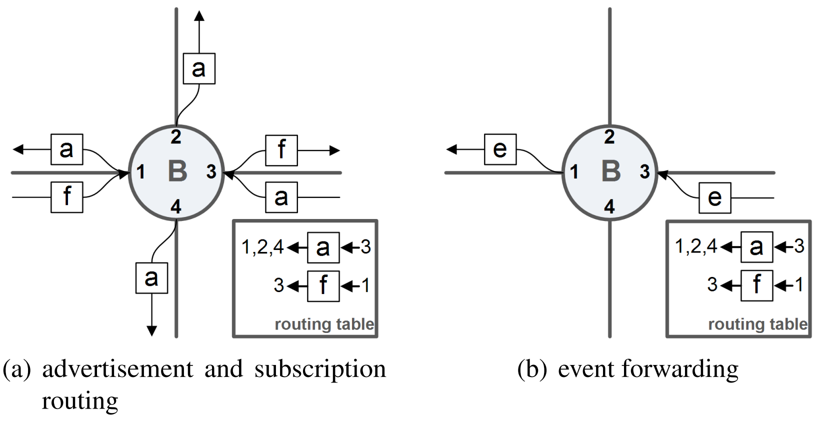


Figure 2.19: Advertisement-based routing strategy

2.5.6 Advertisement-Based Routing

Section 2.2.3 discusses the role of the advertisements from the perspective of the content publishers. Authors in [CRW01] propose to use advertisements to optimize the routing of subscriptions. The intuition behind the advertisement-based approach is that advertisements can influence the propagation of filters so that they are routed by the brokers of the publish/subscribe system only in the direction of the publishers which will produce content matching the filters. Figure 2.19 shows the basic principles behind the advertisement-based routing. Broker **B** receives an advertisement filter a (contained in advertisement message) on the interface 3 – see Figure 2.19(a). After forwarding of the advertisement a and storing it in the advertisement routing table, a filter f arrives on the interface 1. Upon the arrival of the filter f broker **B** determines whether events matching advertisement a will also match filter f. If broker **B** determines that such events might exist, it forwards the filter f on the reverse path of the advertisement a and stores the filter f in the filter routing table. Otherwise, the filter f is dropped.

Subsequent events (see Figure 2.19(a)) matching filter f and advertisement a are (analogously to the coverage-based routing) forwarded on the reverse path of the matching filter f. In order to determine whether events matching advertisement a will also match filter f broker **B** calculates a binary intersection (overlap [MFP06]) relation between advertisement a and filter f.

Intuitively, advertisement a intersects filter f if there exists at least one event which is selected by both advertisement a and filter f:

$$f \bowtie a \quad \Leftrightarrow \mathbb{E}_f \cap \mathbb{E}_a \neq \emptyset \tag{2.38}$$

where $\mathbb{E}_a$ is a set of events selected by advertisement a.

Formally it can be said that filter f intersects advertisement a if and only if:

$$\exists_{p_f \in f} \exists_{p_a \in a} \quad : \quad p_f \bowtie p_a \tag{2.39}$$

Filter f	Advertisement a	$f \bowtie a$
$\{y < 8\}$	$\{y < 8\}$	true
$\{x > 5\}$	$\{y > 5\}$	false
$\{x > 5\}$	$\{x < 8\}$	true
$\{x > 5\}$	$\{x < 8;\ y > 5\}$	true
$\{x \neq 5\}$	$\{x < 8\}$	true
$\{x \neq 5\}$	$\{x = 5\}$	false
$\{x \geq 5\}$	$\{x > 5\}$	true

Table 2.3: Intersection relation between filters in advertisement-based semantics

which can be read as: there exists a predicate p_f in filter f such that there exists a predicate p_a in advertisement a such that predicate p_f intersects p_a. Predicate p_f intersects predicate p_a if and only if:

$$p_f \bowtie p_a \quad \Leftrightarrow \quad an_f \equiv an_a \wedge ac_f \bowtie ac_a \tag{2.40}$$

The exact semantics of the attribute constraint intersection ($ac_f \bowtie ac_a$) depend on the operators and values used for the specific implementation. The reader is referred to [MFP06, JF07] for an overview of simple attribute constraints in context of the intersection relation. Intuitively, an attribute constraint ac_f intersects attribute constraint ac_a if attribute constraint ac_f selects at least one attribute value selected by attribute constraint ac_a. Table 2.3 gives an overview of filter and advertisement intersection relation.

Advertisement Routing

The advertisement routing in advertisement-based semantics is identical to the routing of filters in coverage-based semantics. The intuition behind this reasoning is the fact that advertisements and filters are semantically identical. Advertisements, similarly to filters use coverage relation to limit their propagation. An advertisement a arriving at the broker $\mathbf{B}$ on interface $I_{\leftarrow a}$ is propagated on all interfaces with the exception of its source interface and interfaces on which advertisements covering advertisement a were already propagated:

$$\mathbb{I}_{a\rightarrow} = (\mathbb{I}_{\mathbf{B}} \setminus \{I_{\leftarrow a}\}) \setminus \{\mathbb{I}_{b\rightarrow} \mid b \in \mathbb{F}_{\mathbf{B}} \wedge b \succ a\} \tag{2.41}$$

An advertisement a received by the broker $\mathbf{B}$ is dropped (neither forwarded nor stored in the advertisement routing table of the broker $\mathbf{B}$) if and only if an advertisement b covering advertisement a was previously delivered to the broker $\mathbf{B}$ on the same interface on which advertisement a arrived:

$$\text{drop}(a) \quad \Leftrightarrow \quad \exists_{b \in \mathbb{F}_{\mathbf{B}}} : b \succ a \wedge I_{\leftarrow b} \equiv I_{\leftarrow a} \tag{2.42}$$

The arrival of the new advertisement a at the broker $\mathbb{B}$ results in the forwarding of filters which intersect the advertisement a on the source interface $I_{\leftarrow a}$ of the advertisement a. The filters are forwarded stepwise, under the assumption that no filter covering the one to be currently propagated on the interface $I_{\leftarrow a}$ has already been propagated on the interface $I_{\leftarrow a}$:

$$\forall_{f \in \mathbb{F}_\mathbf{B}} \left(\text{forward}(f) = I_{\leftarrow a} \Leftrightarrow \left(\left(\nexists_{g \in \mathbb{F}_\mathbf{B}} : g \succ f \land I_{\leftarrow a} \in \mathbb{I}_{b \rightarrow} \right) \land a \bowtie f \right) \right) \tag{2.43}$$

Filter Routing

Filter routing in advertisement-based routing is influenced by both advertisements and covering filters. A filter f arriving at the broker **B** on interface $I_{\leftarrow f}$ is propagated only on interfaces on which an intersecting advertisement has arrived (with the exception of the source interface of filter f) minus interfaces on which filters covering filter f were already propagated:

$$\mathbb{I}_{f \rightarrow} = \left(\{ \mathbb{I}_{\leftarrow a} \mid a \bowtie f \} \setminus \left\{ I_{\leftarrow f} \right\} \right) \setminus \left\{ \mathbb{I}_{g \rightarrow} \mid g \in \mathbb{F}_\mathbf{B} \land g \succ f \right\} \tag{2.44}$$

A filter f received by the broker **B** is dropped if and only if a filter g covering filter f was previously delivered to the broker **B** on the same interface on which filter f arrived or there exists no advertisement matching filter f which arrived on interface other than the source interface of the filter f:

$$\text{drop}(f) \quad \Leftrightarrow \quad \left(\exists_{g \in \mathbb{F}_\mathbf{B}} : g \succ f \land I_{\leftarrow g} \equiv I_{\leftarrow f} \right) \lor \left(\nexists_{a \in \mathbb{A}_\mathbf{B}} : a \bowtie f \land I_{\leftarrow a} \not\equiv I_{\leftarrow f} \right) \tag{2.45}$$

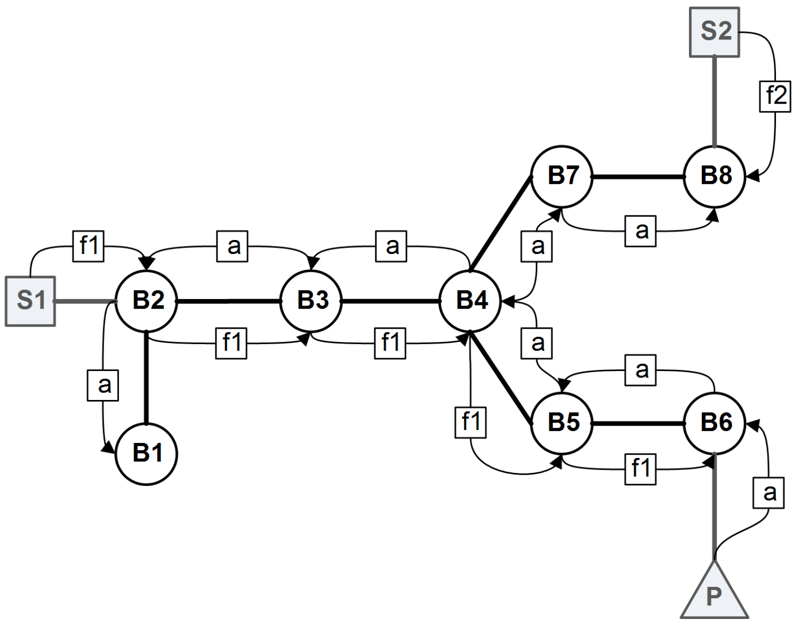
where $\mathbb{A}_\mathbf{B}$ is a set of valid advertisements received by the broker **B**.

Event Forwarding

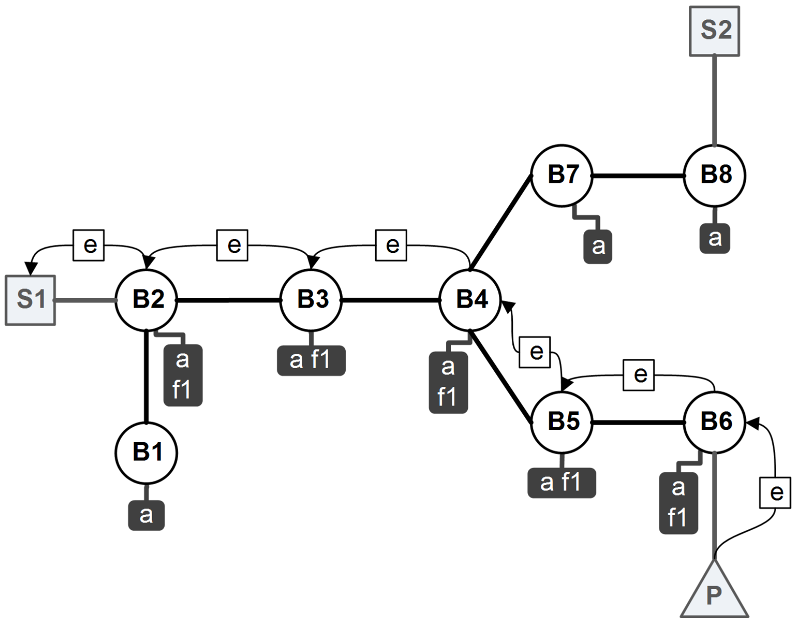
Event forwarding in advertisement-based routing is similar to that of the identity-based routing – see Equation 2.33. The only difference being that brokers might enforce the advertisements with respect to the published event – an event e can be dropped by a broker **B** if there exists no advertisement matching event e and received by the broker on the same interface as event e (regardless of the filters stored in the broker **B**):

$$\nexists_{a \in \mathbb{A}_\mathbf{B}} : a(e) = 1 \land I_{\leftarrow a} \equiv I_{\leftarrow e} \quad \Rightarrow \quad \text{drop}(e) \tag{2.46}$$

Figure 2.20 shows the process of advertisement-based routing in a broker overlay. First, an advertisement is issued by the publisher **P** and propagated into the network of brokers. Subsequently, subscriber **S1** issues a matching filter which is propagated on the reverse path of the advertisement towards the publisher **P**. Specifically, unlike in case of the simple-, identity-, and coverage-based routing (cf. Sections: 2.5.3, 2.5.4 and 2.5.5), filter is not broadcast into the network. The subscription message issued by the subscriber **S2** does not contain a



(a) routing of advertisements and filters



(b) forwarding of events

Figure 2.20: Advertisement-based routing in a broker overlay

filter matching the previously issued advertisement and is therefore dropped by the first broker (**B8**) it encounters – see Figure 2.20(a). Publication message issued by the publisher **P** is forwarded towards the interested subscriber **S1** on the reverse path of the matching filter.

Unadvertising

Publishers willing to stop publishing content must cancel the advertisement message containing the filter which describes the content for which the publication process is being stopped. The process of unadvertising an advertisement a by the broker **B** consists of five steps: (1) propagation of the unadvertisement message containing the advertisement a, (2) propagation of the advertisement messages containing advertisements covered by the advertisement a which were not propagated due to the presence of a, (3) removal of the advertisement a from the routing table of the broker **B**, (4) removal of the $I_{\leftarrow a}$ interface from the set of interfaces $\mathbb{I}_{f\rightarrow}$ for filters for which no intersecting advertisement (after removal of a) on interface $I_{\leftarrow a}$ exists and (5) unsubscription of filters for which no intersecting advertisement exists in the advertisement routing table of the broker **B**.
The propagation of the unadvertisement message containing advertisement a is performed for a set of interfaces containing all interfaces on which identical advertisements arriving from the same source interface were forwarded:

$$\text{forward}_{\text{unadv}}(a) = \{\mathbb{I}_{b\rightarrow} \quad | \quad b \in \mathbb{A}_{\mathbf{B}} \wedge b \equiv a \wedge I_{\leftarrow b} \equiv I_{\leftarrow a}\} \tag{2.47}$$

The propagation of the advertisement messages containing filters covered by a which were not propagated due to the presence of a requires the broker **B** to remove all advertisements covered by the unadvertised advertisement a from its routing table:

$$\text{remove}(b) \quad \Leftrightarrow \quad b \in \mathbb{A}_{\mathbf{B}} \wedge b \prec a \wedge I_{\leftarrow b} \not\equiv I_{\leftarrow a} \tag{2.48}$$

and readvertise the removed advertisements again. Advertisements need to be removed from the routing table of the broker **B** in order not to get dropped at the readvertisement attempt – see Equation 2.42.
The removal of the advertisement a source interface $I_{\leftarrow a}$ from the set of interfaces $\mathbb{I}_{f\rightarrow}$ for filters for which no intersecting advertisement (after removal of a) on interface $I_{\leftarrow a}$ exists allows broker **B** to avoid propagation of events for which the matching filter has been removed in a downstream broker (due to step 5):

$$\forall_{f \in \mathbb{F}_{\mathbf{B}}} \left((\nexists_{b \in \mathbb{A}_{\mathbf{b}}} : f \bowtie b \wedge I_{\leftarrow b} \equiv I_{\leftarrow a}) \quad \Rightarrow \quad \mathbb{I}_{f\rightarrow} \setminus \{I_{\leftarrow a}\} \right) \tag{2.49}$$

Unsubscription of filters for which no intersecting advertisement, after the removal of the advertisement a, exists in the advertisement routing table of

```java
public class RoutingTable {
   //data containers
   private FilterContainer filters;
   private FilterContainer advertisements;

   //public API
   void publish(Event e, SocketAddress src);
   void subscribe(Filter f, SocketAddress src);
   void unsubscribe(Filter f, SocketAddress src);
   void advertise(Filter f, SocketAddress src);
   void unadvertise(Filter f, SocketAddress src);
}
```

Listing 2.6: Example routing table API [JF08a]

the broker **B** can be directly derived from Equation 2.45. It is important to note that the intersecting advertisement needs to originate from the different interface than the filter in question:

$$\forall_{f \in \mathbb{F}_{\mathbf{B}}} \left(\nexists_{b \in \mathbb{A}_{\mathbf{B}}} : b \bowtie f \wedge I_{\leftarrow b} \not\equiv I_{\leftarrow f} \quad \Rightarrow \quad \text{unsub}(f) \right) \tag{2.50}$$

Unsubscribing

The process of the unsubscription of the filter f by the broker **B** is identical as in the case of the coverage-based routing and consists of three phases: (1) propagation of the unsubscription message containing the filter f – see Equation 2.34, (2) propagation of the subscription messages containing filters covered by the filter f which were not propagated due to the presence of f – see Equation 2.35 and (3) removal of f from the routing table of the broker **B** – see Equation 2.32.

Routing Table Implementations

The implementation of the advertisement routing tables is usually identical to that of the filter routing tables, due to the fact that both filters and advertisements share the same semantics – with the exception of the model introduced in [CRW01]. The functionality of the advertisement routing tables needs however to be extended in order to support the routing of filters based on the intersection relation. Typically, implementations [JF08a, Tar08b] decouple the intersection and coverage relation calculation from the routing process providing separate binary methods which can be easily integrated into the routing table implementation.
Listing 2.6 shows a generic routing table API. It can be observed that both filters and advertisements share the same data model represented by the class `Filter`. Similarly, the actual containers storing the filters and advertisements share the same interface. The routing table presented in the Listing 2.6 is

coupled with the network layer of the publish/subscribe system implementation. The methods presented in Listing 2.6 directly invoke the network layer methods to route advertisements and filters and to forward events.

Dynamic Overlay Management

The removal of the interface I_i from the routing table of the broker **B** results in every advertisement a stored in the routing table of the broker **B** which arrived on the removed interface I_i being unadvertised – see Equations 2.47 and 2.48 and discussion regarding the following unsubscription of filters.
The addition of a new interface I_i results in the propagation of all advertisements in the routing table of the broker **B** on the new interface. The advertisements are propagated stepwise, under the assumption that no advertisement covering the one to be currently propagated on the interface I_i has already been propagated on the interface I_i:

$$\forall_{a \in \mathbb{A}_\mathbf{B}} (\text{forward}(a) = I_i \Leftrightarrow \nexists_{b \in \mathbb{A}_\mathbf{B}} : b \succ a \wedge I_i \in \mathbb{I}_{b \rightarrow}) \tag{2.51}$$

No filters are propagated on the newly added I_i interface, as there are no new advertisements which arrived on that interface yet.

Subscription-Based versus Advertisement-Based Routing

Advertisement-based routing and subscription-based routing are both correct and efficient routing strategies for publish/subscribe systems. It is up to the developer of the publish/subscribe system to choose which routing strategy to use. The choice of the routing strategy is determined by two basic factors: the amount of different filters issued by the subscribers of the system and the scope of publishers' publications. The amount of different filters issued by the subscribers translates directly to the amount of different subscription messages propagated into the system. This in turn is driven by the: (1) scope of the interest of the subscribers, (2) the frequency with which subscribers change their scope of interest and (optionally, depending on the routing strategy used) (3) coverage, identity or merging possibilities for different filters issued by the subscribers.
The number of subscription messages can be reduced when advertisement messages are propagated into the system and thus drive the routing of subscription messages. However, the number of the advertisement messages depends, similarly as in the case of the subscription messages, on: (1) the scope of the events published by every publisher, (2) the frequency with which every publishers changes the scope of published events (unadvertisement of the old scope and advertisement of the new one) and (3) the possibility to limit the propagation of advertisements using identity, coverage or merging routing strategies. Moreover, the propagation of subscription messages in

advertisement-based semantics depends on the extent of intersection between the published advertisements and filters.
Using a very simple model based on the simple routing with advertisements, and assuming that the interest of subscribers and content published by publishers remain static over time, allows us to make a basic comparison between the number of subscription messages in the simple routing model and simple routing model with advertisements.
Using only subscriptions one can calculate the total number of subscription messages sent in the system composed of N brokers and $|\mathbb{S}|$ subscribers as:

$$\left(N \sum_{i=1}^{i=|\mathbb{S}|} |\mathbb{F}_{\mathbf{S}_i}| \right) - 1 \tag{2.52}$$

where the term $\sum_{i=1}^{i=|\mathbb{S}|} |\mathbb{F}_{\mathbf{S}_i}|$ denotes the total number of filters issued by all subscribers in the system. It is assumed that subscribers and publishers are physically collocated with the brokers and that the brokers form a spanning tree – acyclic architecture.
The worst case number of subscription messages issued when simple routing strategy with advertisement is used is given by:

$$\mathrm{diam}(N) \sum_{i=1}^{i=|\mathbb{S}|} \left| \left\{ f \in \mathbb{F}_{\mathbf{S}_i} \mid \exists_{a \in \bigcup_{i=1}^{i=|\mathbb{P}|} \mathbb{A}_{\mathbf{P}_i}} : a \bowtie f \right\} \right| \tag{2.53}$$

where $\mathrm{diam}(N)$ is the diameter of the graph formed by the acyclic overlay of the brokers, $|\mathbb{P}|$ is the total number of publishers in the system and $\bigcup_{i=1}^{i=|\mathbb{P}|} \mathbb{A}_{\mathbf{P}_i}$ is the set of all advertisements in the system. For a fair comparison with the subscription-based simple routing strategy the above value must be incremented with the total number of advertisement messages sent in the system:

$$\left(N \sum_{i=1}^{i=|\mathbb{P}|} |\mathbb{A}_{\mathbf{P}_i}| \right) - 1 \tag{2.54}$$

From the above, very simple case, it can be concluded that the comparison of the routing strategies is very difficult as there exist many factors which have a great influence on the message overhead and which can be only evaluated during the run time of the system. This is one of the motivations for systems like [SMRP08] to propose hybrid routing algorithms which can be adapted during runtime to dynamically optimize the cost of the routing in the publish/subscribe systems.
Another issue which needs to be considered is the delay [BBPV05] introduced by the advertisement messages. In subscription-based routing as soon as a subscription message driven path from the subscriber to the publisher is established the subscriber can receive publication messages. In advertisement-based

routing an additional delay has to be considered, as first advertisement messages need to be propagated into the network (similarly to the subscription messages in the subscription-based routing) and only subsequently the propagation of subscription messages which establish the publisher–subscriber path can commence. Depending on the advertisements and filters present in the network this delay can range from one hop delay (intersecting filters are already present at the broker connecting the publisher) to the diam(N) hops delay if the subscription messages could not be propagated into the network at all due to the lack of advertisements intersecting the filters carried by the subscription messages.

2.5.7 Peer-to-Peer Routing

All subscription-based routing strategies are based on the assumption that dissemination of subscription messages to all nodes in the publish/subscribe network ensures that a publication message encounters a node which contains branches of all filter trees of interested subscribers. In other words – the meeting of a publication and subscription message is guaranteed by propagating subscription messages to all nodes in the system. Similarly, in case of advertisement-based routing schemes advertisements take the role of subscriptions which are disseminated to all nodes of the network.

The peer-to-peer routing strategy takes a different approach towards the publication and subscription messages meeting problem. In peer-to-peer routing the address of the meeting node is calculated *before* sending of publication and subscription messages. The calculation of addresses ensures that events and filters which match given events arrive at the same meeting node, also called *rendez-vous* node – the concept first introduced in [BFC93]. The rendez-vous node stores all filters matching a given set of events. Upon the arrival of the event from the matching set the rendez-vous node disseminates it to all interested subscribers.

Peer-to-peer systems can be built using either structured [RD01, RHKS01, SMK$^+$01, ZHS$^+$04], or unstructured [CRB$^+$03, GSGM03, LRS02] overlays. Unstructured overlays use flooding or probabilistic algorithms to locate data held by the overlay nodes. One of the major drawbacks of the unstructured overlays is the low probability of finding rare data items – as this requires visiting the large portion of nodes in the system [CCR04].

Structured overlays, on the other hand, introduce determinism in that they constrain the overlay structure and the placement of data. Data objects are assigned keys and the queries are routed based on the key of the object the query is interested in. The strict overlay management and data placement allows to provide efficient data discovery – usually in the range of $O(log(N))$ steps, where N is the number of nodes in the overlay [QB06].

Structured Peer-to-Peer Systems

One of the first peer-to-peer publish/subscribe systems to be based on the structured overlay is Scribe [CDKR02, CDKR03]. Scribe is based on a structured peer-to-peer location and routing substrate Pastry [RD01]. Pastry, in turn, is an implementation of a Distributed Hash Table (DHT).
Distributed Hash Tables [RD01, RHKS01, SMK+01], first introduced in 2001, similarly to normal hash tables provide a lookup service where every object is associated with a key. In DHTs key-value pairs are stored in a distributed fashion on a set of nodes which are responsible for maintaining the mappings between the keys and the values. Nodes cooperate with each other and provide the ability to store and retrieve objects based on the object's key. Specifically, in Pastry, each node is assigned a globally unique 128-bit identifier. It is assumed that node identifiers are uniformly distributed in the $\left[0, \ldots, 2^{128} - 1\right]$ identifier space. Node identifiers can be obtained by, e.g., calculating a cryptographic hash function [Riv92, DEEJ01] of the IP address of the node. In Pastry values are assigned keys from the same space as the node identifiers. An object is placed on the rendez-vous node which identifier is numerically closest to the object's key. In a N-node network, under normal operation, Pastry can locate an object in a less than $\lceil log_{2^b} N \rceil$ steps, where b is a configuration parameter with a typical value of 4.
It is important to note that the actual number of steps performed (nodes traversed) at the IP level might be significantly different. The difference between the amount of node traversals in the overlay and physical infrastructure is called stretch [CDHR03]. Stretch can be also defined [SMK+01] as a lookup stretch: a ratio between the overlay and optimal routing latencies. The average stretch takes different values, depending on the data access pattern and the specific routing strategy used for the given system. It can be, however, assumed that the lookup stretch values are in the range between two to eight [PFH07].
Pastry provides a following routing primitive:

$$\text{route}(value, key) \tag{2.55}$$

where *value* is the message which is routed to a node whose identifier is numerically closest to the *key*. In each routing step a Pastry node forwarding a key-value pair sends it to a node whose identifier shares at least one more digit with the value's key than the current node identifier does. If no such node can be found than the key-value pair is forwarded to a node which is numerically closer to the key than the current one. This way the routing converges.
Scribe uses the Pastry mechanisms to implement a topic-based publish/subscribe system based on the *rendez-vous* approach. The rendez-vous node, in a topic-based publish/subscribe system, is responsible for managing the subscribers of the given topic T. The rendez-vous node is usually selected by obtaining its identifier via hashing of the textual topic representation:

$$\text{identifier} = \text{SHA1}(T) \tag{2.56}$$

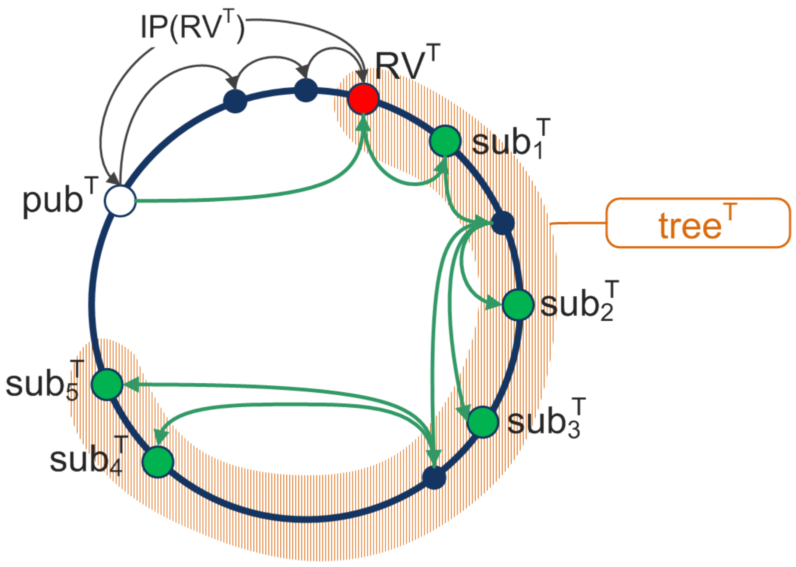


Figure 2.21: Information dissemination in Scribe [Que08]

The rendez-vous node for the topic T is a root of the multicast tree which contains all subscribers to T. The multicast tree is constructed using a scheme based on the reverse path forwarding [DM78]. Specifically, the multicast tree might contain nodes which are not explicitly interested in T. Figure 2.21 shows how an event belonging to the topic T is routed in Scribe. The shaded area indicates the multicast tree for the topic T – tree^T. The publisher pub^T first acquires the IP address of the rendez-vous node RV^T for the topic T – grey lines. Subsequently, it caches the obtained IP address of the rendez-vous node $IP(RV^T)$, and sends events directly to it – green lines. Rendez-vous node, in turn, disseminates events along the branches of the multicast tree for the topic T.

A similar approach to that of Pastry was proposed in CAN [RHKS01] – a content addressable network. CAN, instead of ordering nodes on a ring structure, divides a d-dimensional Cartesian coordinate space into adjacent subspaces assigned to each of the CAN nodes. The routing process in CAN is characterized by a low per-node state: $O(d)$, and short routing path length: $O(dN^{\frac{1}{d}})$, where N is the number of nodes in the system. The size of the CAN routing table (unlike that of Pastry) is independent of the number of nodes in the system. For each topic CAN builds and maintains a dedicated overlay – with the topic being deterministically mapped (hashed) onto a point in the d-dimensional CAN space. Node, owing the given point serves as the bootstrap node for the construction of the topic multicast group – in case of overload possibility multiple hash functions and thus multiple nodes can be used. The event routing in CAN is performed via directed flooding within the multicast group to which a given event belongs – see Figure 2.22.

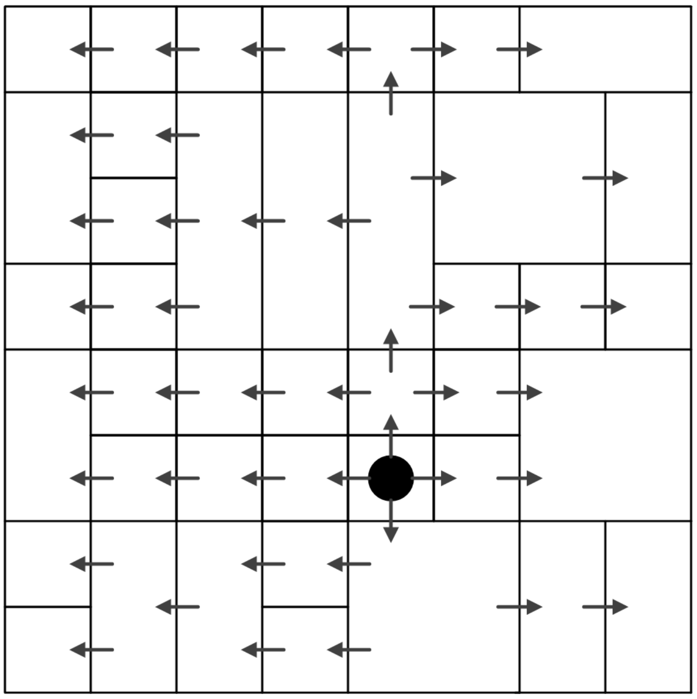


Figure 2.22: Directed event routing in CAN [RHKS01]

Another approach, taking advantage of the content-based routing has been presented in form of the Hermes peer-to-peer middleware [PB03b, Pie04]. Hermes is a publish/subscribe middleware based on Pastry peer-to-peer routing substrate. Hermes provides a content-based publish/subscribe routing in that routing of advertisements and filters is performed using the rendez-vous nodes and routing of events is based solely on the filter routing tables. Specifically, advertisements are routed towards the rendez-vous node which is obtained by calculation of a deterministic function over the type of advertised events. Similarly, subscription messages are routed towards the rendez-vous node, however filters in subscription messages are also content-matched against the advertisements and routed on their reverse paths. The routing of events follows only the filter routing tables, specifically, events are not routed towards the rendez-vous node for their event type.

Moreover, Hermes uses congestion control algorithms [PB03a] to detect congestion in the overlay and tries to fix the cause for it. Similarly to [DGP+07] Hermes provides the ability to detect composite events patterns using an extended version of the finite state automata [PSB03].

SPICE [CQL08] is an implementation of the Implicit Group Messaging (IGM) where publishers deliver the content to the anonymous subscribers forming groups of interest. SPICE can be based on any structured peer-to-peer routing substrate, e.g., Tapestry [ZHS+04].

Unstructured Peer-to-Peer Systems

Data-Aware Multicast [BEG04] is a topic-based publish/subscribe system based on a peer-to-peer substrate. Using probabilistic algorithms topic are organized into hierarchy, which in turn maps to a dynamic hierarchy of groups of processes. However, due to the lack of the general overlay, every publisher must itself become the member of the topic dissemination group for which it wishes to publish events. This, in turn, results in every publisher receiving events it did not subscribe to.
Authors of Sub-2-Sub system [VRKvS06] propose a peer-to-peer-based event notification mechanism which clusters subscribers according to their interest similarity. Sub-2-Sub is a content based system, where similar filters are clustered into topics. Participants are not organized into a structured overlay, rather custom clustering based on topics of interest is used. Every topic forms a separate ring structure, which results in the degree of the overlay growing linearly with the number of topics created from filters. An event matching a given topic is first routed towards the correct ring and subsequently efficiently disseminated within the ring itself. Information about subscriptions is periodically exchanged between the nodes using the CYCLONE [SVvS05] protocol.
TERA [BBQ+07, Que07] is a topic-based publish/subscribe system based on a peer-to-peer architecture. Nodes in TERA (in contrast to [BEG04]) are organized into two layer infrastructure – the global overlay network which connects all nodes and topic layer overlays which connect all nodes interested in the same topic. TERA utilizes random walks and access point lookup tables to deliver events to topic layers containing all nodes with matching filters. Event e is first delivered to the access point node for the given topic, which subsequently disseminates the event within the topic specific overlay. The peer-to-peer overlay (similarly as in the case of [VRKvS06]) is managed based on gossiping and the view exchange technique [SVvS05].

Chapter 3

System Model

This chapter summarizes the system model which is used for the implementation of the XSiena publish/subscribe system. The system model describes the assumptions about the processes, communication links and failures (Section 3.1), as well as clocks (Section 3.2) in the context of the XSiena system. Section 3.3 defines the architecture, syntax, semantics and routing strategies used in the XSiena system and throughout the remainder of this thesis.
The underlying system model is based on the Timed Asynchronous Distributed System Model [CF99]. It has been shown [JFF07, JFF08] that the Timed Asynchronous Distributed System Model is suitably weak to be used for building loosely coupled, distributed systems and simultaneously it is sufficiently strong to allow for building of adaptive systems which react to violation of the real-time properties caused by the lower level services.

3.1 Processes

It is assumed that all processes in the XSiena system are timed, i.e., there exists a time interval $\sigma_{\max}$ within which every process is supposed to respond to a request sent to it. However, there are no guarantees that a request is indeed processed and answered within $\sigma_{\max}$. Figure 3.1 shows a cumulative distribution function for one million processing times of a UDP ping request. The distribution functions are plotted for two hosts – `Lab PC` is a host in our laboratory and `PlanetLab node` is the `planet1.inf.tu-dresden.de` node of PlanetLab consortium. It can be observed that for both hosts the response times are characterized by a significant long tail and that it is not possible to set a reasonable upper bound for the processing time $\sigma_{\max}$.
Processes composing the XSiena system may suffer performance failures. Specifically, a process might not respond within $\sigma_{\max}$ seconds because it is either slow or it has crashed. It is assumed that processes do not suffer Byzantine failures. Such assumption can be fulfilled by converting Byzantine failures into crash failures using different software [WF07] or hardware [BBV+05]

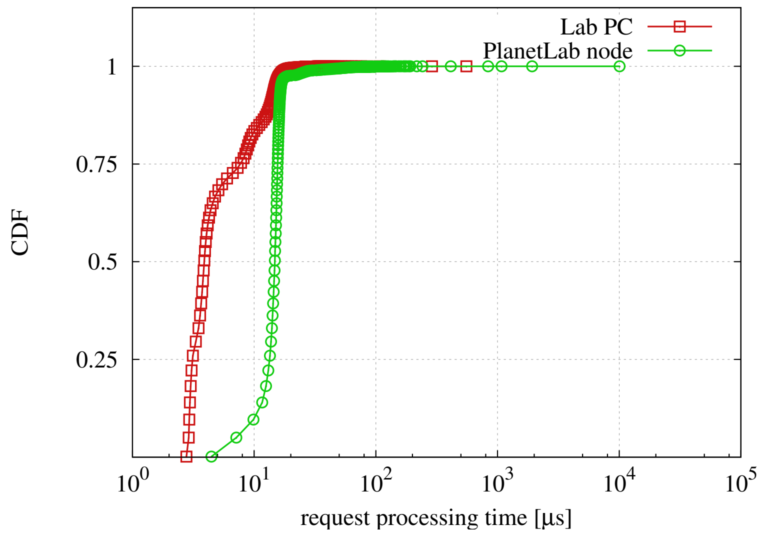


Figure 3.1: UDP-based ping message processing time

techniques.

Processes communicate using unreliable transport protocol with omission/performance (non-Byzantine) failure semantics. Messages sent between processes may be arbitrarily delayed or might get dropped. Specifically, no *a priori* assumptions regarding upper bound $\delta_{\max}$ nor lower bound $\delta_{\min}$ on the message transmission delay are made. It can be illustrated (see Figure 3.2) that independently of the network environment it is not possible to determine a practical upper bound for message transmission delays.

Figure 3.2 shows the long-tail cumulative distribution functions of the round trip times for one million UDP ping messages between two hosts in different network environments. The experiment has been conducted using three network setups: (1) the Local Area Network (`LAN`) – between two computers in our laboratory, (2) the Metropolitan Area Network (`MAN`) – between two computers at the Dresden University of Technology campus and (3) the Wide Area Network (`WAN`) – between two computers at the Dresden and Berlin Universities of Technology. Based on the above and other related work [MPHD06, WSH08] it is assumed that there exists no upper bound on the frequency of communication and process failures.

3.2 Clocks

Every process in the XSiena content-based publish/subscribe system has access to a local hardware clock. The term $H(t)$ denotes the value of the hardware clock at the real-time t. It is assumed that every hardware clock $H_i(t)$ has a drift rate which is bounded by ρ_i. Specifically, it is assumed that all hardware clocks

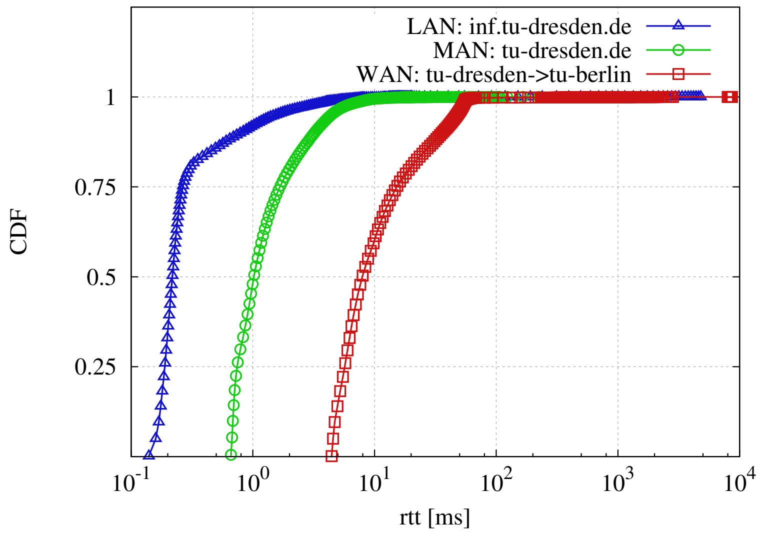


Figure 3.2: UDP-based ping message latencies (RTT)

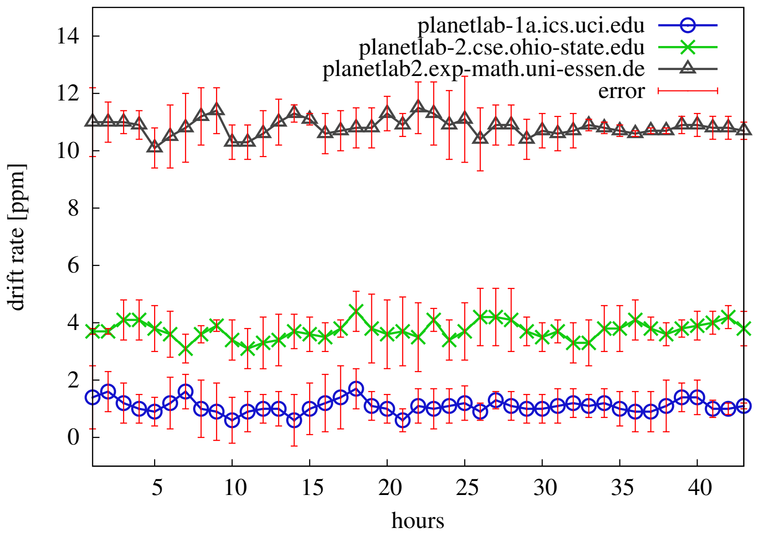


Figure 3.3: Drift rate of the time stamp counter (`rdtsc`) for three PlanetLab hosts

have a drift rate bounded by the *a priori* known constant $\rho_{\max}$. It has been shown in [JFF07] and [JFF08] that for commercial, off-the-shelf components the value of $\rho_{\max}$ usually stays within $[0ppm, 100ppm]$.
Figure 3.3 shows the drift rate of the time stamp counter for three different PlanetLab hosts. Time stamp counter is a typical COTS component embedded in most of CPUs. The drift rate has been measured using a modified SNTP client. The SNTP client code has been instrumented so as to substitute the calls to the `gettimeofday` with the invocations of the `rdtsc` assembly command returning the current value of the time stamp counter. It can be observed that the drift for all hosts is stable and that it is feasible to determine an upper bound $\rho_{\max}$ for all clocks.
A hardware clock is said to be correct if:

$$H(t) - H(s) \leqslant (t - s)(1 + \rho_{\max}) \tag{3.1}$$

and

$$H(t) - H(s) \geqslant (t - s)(1 - \rho_{\max}) \tag{3.2}$$

holds.
Specifically, it is assumed that hardware clocks have crash-stop failure semantics. Authors in [FC99a] show how to achieve this property despite possible arbitrary value failures.

3.3 System Architecture

XSiena family of publish/subscribe systems is composed of brokers, publishers and subscribers. Publishers and subscribers are used by the publish/subscribe applications to publish and receive content, respectively.
The communication in the XSiena system is performed using advertisement, publication and subscription messages. Advertisement and subscription messages contain filters. Advertisement and subscription filters share the same syntax and semantics. Advertisement messages describe the content published by publishers and subscription messages describe the interest of subscribers. Publications messages contain events which encapsulate the content. The entire family of the XSiena publish/subscribe systems is content-based with all messages using the predicate-based semantics.
Brokers in the XSiena publish/subscribe system are connected forming an acyclic graph overlay. The routing of advertisement and subscription messages, unless explicitly stated, exploits the coverage and intersection relations between the respective filters. The communication between all components of the XSiena system is asynchronous and neither subscribers nor publishers nor brokers know about the identity of the components other than their immediate neighbors in the acyclic overlay.

Chapter 4

Prefix Forwarding

It has been shown in [STA05] that content-based matching problem can be reduced to the range-searching problem, for which no worst case computationally efficient solution exists. This implies that content-based matching also does not have a worst case efficient solution. In [KHCS05] it has been formally proved that the hardness of content matching is equivalent to the Partial Match [JKKR04] problem. Currently in most content-based publish/subscribe systems content-based matching has to be performed at every node. Therefore, it is desired to limit the number of necessary content-based matches to the minimum in order to excel the operations of the content-based publish/subscribe system.

This chapter describes the prefix forwarding algorithm which limits the number of content-based matches in the XSIENA publish/subscribe system. The intuition behind the prefix forwarding idea stems from the end-to-end argument [SRC84]. The content-based matching functionality is shifted from the brokers residing inside of the content-based publish/subscribe system into the edge of the network.

The shift towards the edge of the network allows to limit the number of content-based matches for event forwarding to only one per delivered event. In a traditional content-based publish/subscribe systems events are matched with filters stored at every broker they pass on their way from the publisher to the interested subscriber. The amount of content-based matches grows with the number of nodes in the network, as well as the number of subscribers and filters issued.

Moreover, one can easily think of systems in which the size of events is in the range of multiple megabytes and potential evaluation of possibly comparably large filters is very expensive already at the level of a single event. Despite the recent efforts to provide hardware solution for efficient content-based matching [MLC08] it still remains a challenge to provide a flexible solution to that problem, which would not be restricted to a specific domain. Therefore, it is desirable to limit the number of content-based matches performed in the network, thus allowing for fast and flexible event forwarding.

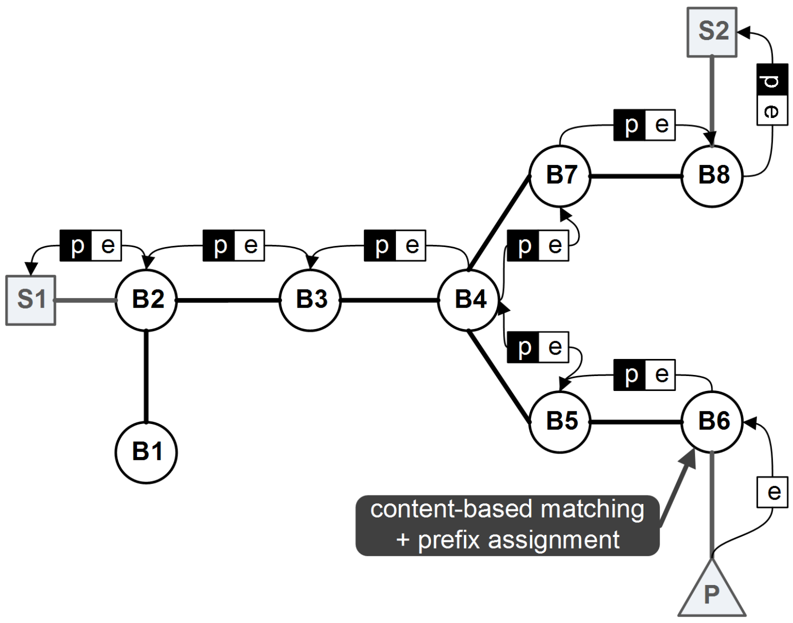


Figure 4.1: Overview of prefix event forwarding

Simultaneously, it is desirable that the proposed solution does not sacrifice the decoupling properties [EFGK03] (see also Section 1.1) of the publish/subscribe system in order to limit the number of content-based matches. With the upholding of the decoupling properties one has to exclude the trivial solutions based on the centralized servers.
The remainder of this chapter is structured as follows: Section 4.1 presents the basic ideas behind the prefix forwarding of events in publish/subscribe systems. Section 4.2 describes the construction and routing of filters, followed by the detailed description of the event forwarding in Section 4.3. Section 4.4 describes the consistency issues which arise due to the application of the prefix-based event forwarding. This chapter is conclude with the brief overview of the related work in Section 4.5.

4.1 Outline

Figure 4.1 presents the outline of the event forwarding in the prefix-based routing. Publisher **P** publishes event *e* which is processed at the publisher connecting broker **B6**. The processing of the event *e* consists of two phases: (1) the content-based phase and (2) the prefix forwarding phase.
During the content-based phase event *e* is matched with the filters stored in the broker **B6** using the content-based matching algorithm. Specifically, the content of the event *e* is compared with the content of filters using the Equation 2.8. The result of the content-based match is however, not a set of interfaces on which the event *e* should be forwarded (as in the case of

the traditional routing – see Section 2.5), rather a prefix p which is attached onto the event e. The prefix p encodes in a compact form the result of the content-based match performed by the broker **B6**.
The prefix forwarding phase performed by the broker **B6** relies only on the prefix p attached to the event e. Specifically, neither the content of the event e nor the content of the filters stored in the routing table of the **B6** is examined by the broker. The prefix p encodes the set of interfaces on which the event e needs to be forwarded by the broker **B6**. The encoding of the prefix p allows broker **B6** to effectively calculate the set of interfaces $\mathbb{I}_{e\rightarrow}$ for the event e. The evaluation in Section 9.1 shows that the calculation of the forwarding set $\mathbb{I}_{e\rightarrow}$ for event e is orders of magnitude faster than the same operation performed using traditional [CRW01] content-based algorithms.

4.2 Filter Routing

The prerequisite for the prefix-based forwarding of events is the presence of filters. In this aspect the prefix forwarding version of the XSiena system is similar to other publish/subscribe systems. Subscription messages containing filters are propagated throughout the whole broker network.
The main difference between the prefix forwarding version of XSiena and other content-based systems, is the construction and the representation of the routing tables. The routing table in the prefix forwarding version of XSiena is called routing tree and is a hierarchical data structure storing filter predicates.

4.2.1 Filter Construction

In the generic predicate based semantics it is possible to construct filters which are logically inconsistent. An example of a logically inconsistent filter is: $\{x > 3;\ x < 2\}$. It is also possible to construct filters which are unnecessarily complicated, e.g., $\{x > 3;\ x > 4;\ x < 6;\ x > 5\}$. Authors of [MFP06] propose to restrict the filters in that within one filter f no two predicates p_i and p_j can exist such that their attribute names are identical:

$$\forall_{p_i \in f} \nexists_{p_j \in f} \quad : \quad an_i \equiv an_j \tag{4.1}$$

Authors in [MFP06, Tar08a] propose to use ranges, e.g., $\{x \in [5, 8]\}$ vs. $\{x > 5;\ x < 8\}$, as attribute constraints to satisfy this property. In prefix forwarding version of the XSiena system, a different approach it taken. The prefix forwarding version of the XSiena system allows for multiple predicates with identical attribute names within the same filter, however upon the creation of the filter the predicate expressions within filters are evaluated and thus compact and logically coherent filters are obtained.
The evaluation of the predicates is performed at the application layer of the publish/subscribe network, so that no such operation is performed with the

Filter	Input predicates	Output filter
$f1$	$x < 5;\ x > 2;\ y > 3$	$\{x < 5;\ x > 2;\ y > 3\}$
$f2$	$z = 9;\ z < 10;\ z > 2$	$\{z = 9\}$
$f3$	$x = *;\ x < 4$	$\{x < 4\}$
$f4$	$x > 5;\ x < 3$	$\emptyset$ (invalid filter)

Table 4.1: Evaluation of filter predicates

network itself. As filters are constructed on a per predicate basis, i.e., one predicate is added to the filter after another, the XSiena system relies for the predicate evaluation on the fact that adding a new predicate to the filter can only narrow its selectiveness. Table 4.1 presents a series of input predicates and the resulting filters. It can be observed that, e.g., in the case of the filter $f2$ the first predicate inserted into the filter $z = 9$ was the most selective one and prevented the insertion of other less selective predicates for the same attribute name. On the other hand, the selectiveness of the filter $f3$ was initially very broad, due to the presence of the predicate $x = *$. The selectiveness of the filter $f3$ was subsequently narrowed by the addition of the second more selective predicate $x < 4$. The attempt to create a logically inconsistent filter $f4$ fails with an exception being generated by the system.

4.2.2 Routing Tree

The routing tree holds complete routing information which is used to calculate the event prefixes and to route filters. Routing tree is maintained by every broker in the publish/subscribe network. This might seem contrary to the intuition behind the edge concept, as one might expect that only brokers which are at the edge of the publish/subscribe network should maintain the routing tree. However, one has to recall that in the XSiena system every broker **B** might become an edge broker, when a publisher **P** dynamically connects to the XSiena system via the broker **B**.

Therefore the core task of every broker in the system (apart from the forwarding of the events) is the maintenance of the routing tree. The maintenance task is limited to the updates to the routing tree due to the arrival of new subscription and unsubscription messages.

The routing tree stores filters in a disjoint form – see Figure 4.2. Each node of the routing tree represents a single predicate. Upon arrival of a new filter it is inserted into the routing tree on a per predicate basis. Within a filter, predicates are deterministically sorted in alphabetical order by the attribute names. In case of two identical attribute names, sorting considers the operators in those predicates. In case of two equal attribute names and operators predicates are

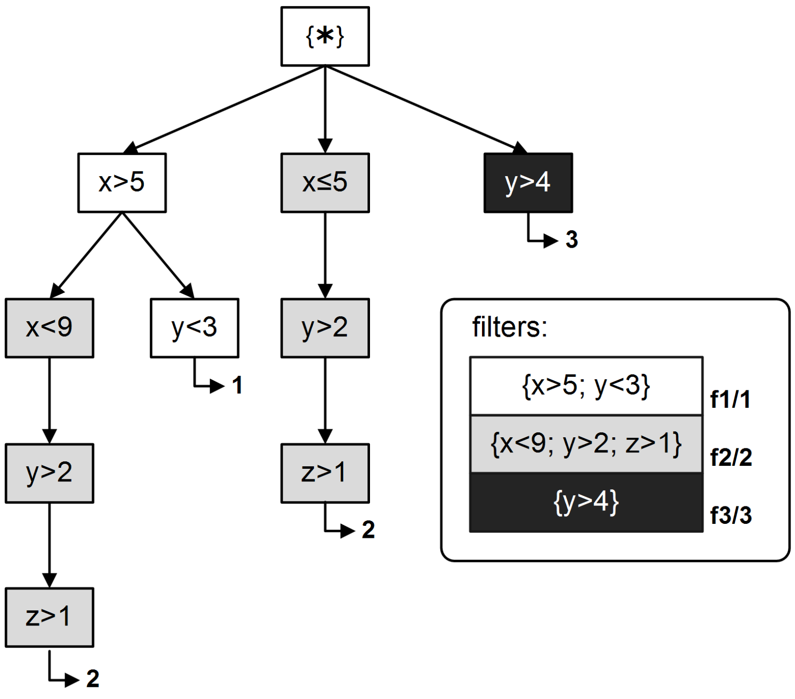
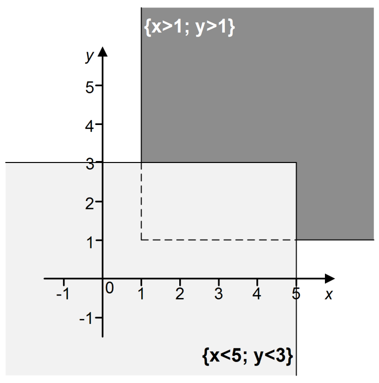


Figure 4.2: Routing Tree

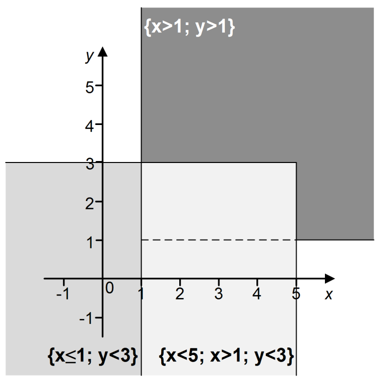
sorted according to the attribute values.
The intuition behind the filter storage in the routing tree is to partition the filter space into disjoint interest regions on every level of the routing tree. This in turn allows for a faster matching of events as a single event attribute name and value pair always selects a single path in the routing tree.

Filter Splitting

In order to be able to partition the filter space on every level into disjoint subspaces a split operation on the filters which are inserted into the routing tree must be performed. The intuition behind the split operation is that having two intersecting filters f_1 and f_2 (see Equation 2.38) an event e might match both of them. However, splitting one of the filters along the intersection axis one can make sure that the event e matches either one or the other filter – with respect to the split predicate.
The split operation relies on the fact that predicates in the filters are logically consistent, thanks to their evaluation upon the creation of the filters – see Table 4.1. The split operation takes as an input a d-dimensional filter f and according to the split axis it splits the given filter into two filters: f_{left} and f_{right}. The split axis itself is a one dimensional predicate. The resulting split filters f_{left} and f_{right} are guaranteed to be disjoint – the property required by the routing tree.



(a) filters before split



(b) filters after split

Figure 4.3: Filter splitting along the x axis

		Output filters	
Filter f	Split axis	f_{left}	f_{right}
$\{x < 9;\ y > 2\}$	$x > 5$	$\{x \leqslant 5;\ y > 2\}$	$\{x > 5;\ x < 9;\ y > 2\}$
$\{x < 9;\ y > 2\}$	$x = 5$	$\{x < 5;\ y > 2\}$	$\{x > 5;\ x < 9;\ y > 2\}$
$\{x > 1;\ z < 7\}$	$x \leqslant 6$	$\{x > 6;\ z < 7\}$	$\{x \leqslant 6;\ x > 1;\ z < 7\}$

Table 4.2: Filter splitting examples

Figure 4.3 visualizes an example split operation. One can observe two filters $\{x > 1;\ y > 1\}$ and $\{x < 5;\ y < 3\}$. Filter $\{x < 5;\ y < 3\}$ is being split along the x axis. The split axis is the $x > 1$ predicate of the first filter. The resulting filter f_{left} takes the form of $\{x \leqslant 1;\ y < 3\}$, whilst the f_{right} takes the form: $\{x < 5;\ x > 1;\ y < 3\}$. Table 4.2 illustrates other splitting examples.

Filter Insertion

Figure 4.2 illustrates a routing tree which will be used to demonstrate the process of the filter insertion. The presented routing tree contains three filters f_1, f_2 and f_3 which arrived at the routing tree hosting broker on the interfaces 1, 2 and 3, respectively. Filter $f1$ arrived as the first one and was inserted on a per predicate basis creating two new tree nodes $x > 5$ and $y < 3$. First the node $x > 5$ was inserted directly as a child of the root node. In a following step algorithm used the inserted node as the current node and inserted the $y < 3$ predicate as the child node of the $x > 5$ node. The node $y < 3$ as the last filter predicate is marked with the source interface of the filter f_1.
The arrival of the second filter f_2 triggers the splitting process. So far the root node of the routing tree had only one child node – the $x > 5$. The insertion of the filter $f_2 \equiv \{x < 9;\ y > 2;\ z > 1\}$ starts at the first predicate

$x < 9$. It can be observed that both predicates intersect. However, it can be recalled that the routing tree maintains disjoint filter regions at each level. Therefore, the insertion algorithm will use the predicate already present in the tree ($x > 5$) to split the arriving predicate $x < 9$ (an effectively the whole arriving filter f_2) into two disjoint predicates: $x > 5$ and $x \leqslant 5$. The insertion algorithm will fork at this place and continue with the insertion of two new filters: $f_{\text{left}} \equiv \{x < 9;\ x > 5;\ y > 2;\ z > 1\}$ and $f_{\text{right}} \equiv \{x \leqslant 5;\ y > 2;\ z > 1\}$. The f_{left} shares the predicate $x > 5$ with the routing tree node, therefore as covered it will be inserted as a child of the $x > 5$ routing tree node. On the other hand $x \leqslant 5$ predicate from the f_{right} filter is disjoint with respect to the $x > 5$ routing tree node, therefore it will be inserted as its sibling, directly under the root node. It can be observed that both filters f_{left} and f_{right} share the same source interface attached to the last node of the respective filter representation in the routing tree. The insertion process from now on is similar to that of the first filter. The insertion of the last filter f_3 is also straightforward, as there are no intersecting nodes.

```
void insert(Filter f, Node n){
   if(f.first==null) break; //end of recursion

   Node cr=null; // covered by f.first
   Node cs=null; // covers f.first
   Node in=null; // intersects f.first
   Node eq=null; // equals f.first
   for(child c : n.children){
      if(intersects(f.first, c)) {in = c; break;}
      if(f.first == c) {eq = c; break;}
      if(covers(c, f.first)) {cs = c; break;}
      if(covers(f.first, c)) {cr = c;}
   }

   if(in){ //split and reinsert
      Filter (fleft, fright) = f.split(in);
      insert(fleft, n);
      insert(fright, n);
   }
   else if(eq){ //step down
      f.removeFirst();
      insert(f, eq);
   }
   else if(cr){ //store as parent
      n.removeChild(cr);
      Node st = n.store(f.first);
      st.addChild(cr);
      f.removeFirst();
      insert(f, st);
   }
   else if(cs){ //step down
      insert(f, cs);
   }
34    else { //store as child of n
35       Node st = n.store(f.first);
36       f.removeFirst();
37       insert(f, st);
38    }
39 }
```

Listing 4.1: Insertion of a new filter into the routing tree

The filter insertion process is described in detail by the recursive algorithm presented in Listing 4.1. The first predicate of the currently inserted filter is stored in the `Filter.first`. It is tested against the current node of the routing for: (a) intersection (line 9), (b) equality (line 10), and (c) coverage – whether the predicate to be inserted is covered (line 11) or covers (line 12) the current routing tree node. This reflects all possible relations two predicates (p_1 and p_2) can be in: (1) p_1 can cover p_2, (2) p_1 can be covered by p_2, (3) p_1 can be equal to p_2 and (4) p_1 can intersect p_2. The equality relation between two predicates is already contained within the coverage relation (see Equation 2.28), however, the algorithm needs to distinguish this special case.

After the determination of the relation between the currently inserted predicate and the set of children nodes of the current node appropriate part of the recursive algorithm is executed. In case of the intersection ($intersects(f.first, c)$) algorithm performs a split of the inserted filter – Listing 4.1, lines 19–23. The `split()` method (line 16) returns the two new filters f_{left} and f_{right} and forks the insertion algorithm. Intuitively, split operation allows to make a deterministic decision as to which branch of the RT to follow when inserting new filter or matching a new event by not allowing for overlapping predicates to be placed at the same level of the routing tree.

In case of predicate equality, the given predicate is removed from the filter to be inserted and the insert procedure replaces the current node with the node equal to the removed predicate (lines 21–22). If the current predicate to be inserted covers the current node in the routing tree it is installed as a new node which is the parent of the covered node and the insertion process continues with the newly inserted node as the current node (lines 25–29). If the predicate to be inserted is covered by the child node the algorithm replaces the current node with the covering one and continues the insertion process (line 32). This steps walks down the routing tree until the last covering node for the predicate to be inserted is found. If the predicate to be inserted is disjoint with all the nodes at the current tree level it is inserted as a sibling node (lines 35–37).

When inserting the last predicate into the routing tree it is tagged with the source interface of the whole filter and the set of interfaces on which the given filter is forwarded – see Figure 4.2. This allows to distinguish the last attribute constraint of the filter and is used in the forwarding process to obtain the reverse path which should be followed by the events. Please note that whenever there are two identical subscriptions arriving at the given node from two different

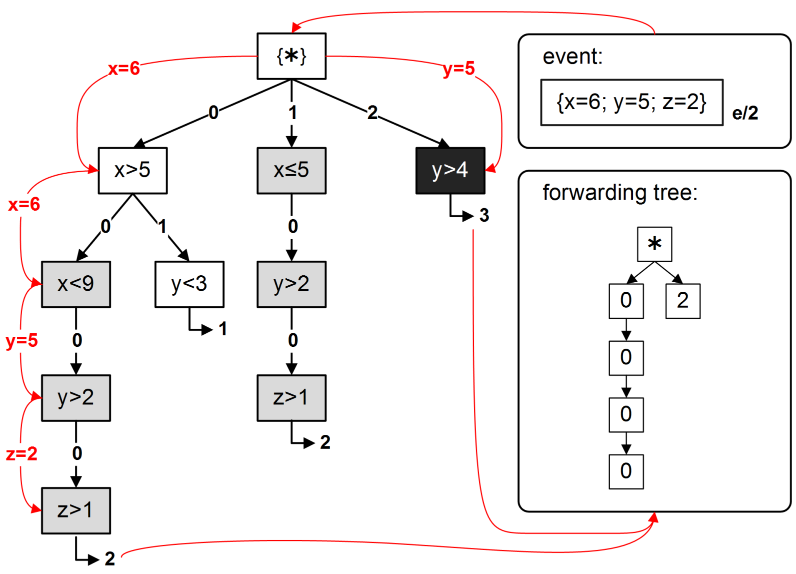


Figure 4.4: Event forwarding using the routing tree

sources the last predicate for both of them will be represented by the same node in the routing tree. Such node will hold a list of two source interfaces for both filters. These lists can grow arbitrarily large with the number of unique connections.
The operation of the routing tree is limited to the forwarding of events. As far as routing of filters is concerned a structure matching the desired filter routing strategy, e.g., a poset [CRW01] or a forest [TK06] can be applied. It is however, worth noting that for simple routing and identity-based forwarding the routing tree itself is a sufficient structure. As far as event forwarding is concerned the routing tree supports most of the routing strategies, including simple routing (Section 2.5.3), identity-based routing (Section 2.5.4) and coverage-based routing (Section 2.5.5).

4.3 Event Forwarding

When a new event *e* is issued by the publisher, the first broker **B**, which connects the publisher to the publish/subscribe network matches the event *e* against its routing tree. The result of this match is a forwarding tree prefix which is subsequently assigned to the event. The event *e* piggybacks the attached forwarding tree prefix on its way towards the interested subscribers. After creation of the forwarding tree prefix, broker **B** and all subsequent brokers on the event's path towards interested subscribers perform the event forwarding based only on the contents of the prefix tree.
Figure 4.4 shows the process of the derivation of the forwarding tree from the routing tree for the event *e*. It can be observed that similarly to the process of

the filter insertion, the event matching is performed on a per attribute name and attribute value pair basis. In case of Figure 4.4 the first attribute name and value pair of the event e is matched with the direct descendants of the root node. One can observe that the node $x > 5$ matches the $x = 6$ attribute name and value pair, which allows the algorithm to follow the path along that node. The following node $x > 9$ also matches the first attribute name value pair, and similarly it is followed on the way towards the leaves of the tree. The next node $y > 2$ does not match the first attribute name and value pair, however, upon advancing to the second attribute name and value pair in the event, the match will be obtained. Again the path is followed and the last node $z > 1$ in the routing tree is reached. The last node matches the last event attribute name and value pair. It stores an indicator which marks the end of the filter which was received by the broker on the second interface.

This implies that the event e matches the filter which arrived on the given interface. At this point the forwarding tree extraction algorithm created a path from the node $z > 1$ till the root of the routing tree and added it to the forwarding tree. The path $0 \rightarrow 0 \rightarrow 0 \rightarrow 0$ indicates the branch in the routing tree which matches the event e. The process of the forwarding tree extraction continues with the second attribute name and value pair $y = 5$ of event e being matched starting from the root of the routing tree. This pair results in the addition of the second branch to the forwarding tree. The so constructed forwarding tree describes all predicates and thus filters which match the event e.

The above algorithm is based on the observation that predicate (node) placement in the routing tree reflects logical conjunctions and disjunctions between filter predicates and filters themselves. Traversing the routing tree from the root towards its leaves along any of the paths equals to creating a filter consisting of the conjunction of all encountered predicates (nodes). Traversal of two paths in parallel results in a logical disjunction between the filters created by the conjunction of the predicates along both paths.

Listing 4.2 shows the forwarding tree extraction algorithm in more detail. Attribute name and attribute value pairs in events are deterministically sorted by their names and values. This, similarly as in the case of the filters, is performed at the publisher side by the publish/subscribe layer. Upon arrival of the new event e broker matches it with the routing tree, starting with the first (root) routing tree node n. The result of the matching – the forwarding tree is initialized with an empty root node and passed as parameter $FPTNode$. The matching starts by iterating over the child nodes (*n.children*) of the current routing tree node and comparing them against every attribute name and value pair (*e.getPredicate*() – line 8) in the event. If the event attribute name and value pair matches the filter predicate ($covers$() – line 8) this path is added to the forwarding tree ($ftN.add$()) and the matching continues using the matching child node of the routing tree and the new node of the forwarding tree as

```
1  match(Event e, Node n, FTNode ftN, int idx){
2        int i=0;
3  //for every child node of the current routing tree node
4        for(Node c : n.children) {
5           int myidx = idx;
6  //for every attr name and value pair in the event
7           while(myidx < e.size()) {
8              if(covers(c, e.getPredicate(myidx))) {
9  //step down the FPT and RT
10                FTNode newFTN = ftN.add(i);
11                match(e, c, newFTN, myidx);
12             }
13 //move to next attr name and value pair in the event
14             ++myidx;
15          }
16 //move to the next child node of the current node
17          ++i;
18       }
19    }
```

Listing 4.2: Event forwarding – creation of the forwarding tree

parameters – line 11.
An important observation with regard to the forwarding tree is that it preserves the coverage relation between the predicates of filters matching an event. Let us assume that event e_1 has been assigned a forwarding tree $0 \rightarrow 0 \rightarrow 1 \rightarrow 0$ and event e_2 has been assigned a forwarding tree $0 \rightarrow 0 \rightarrow 1$. From the above it can be reasoned that every subscriber interested in the event e_1 is also interested in the event e_2. Moreover, in order to draw this conclusion one needs only to look at the forwarding trees of both events and do not need to consider the events' content.

4.3.1 Using Forwarding Tree

Once the event e has been matched by the broker **B** against the routing tree and assigned the forwarding tree, the same broker executes the forward operation. During the forward operation forwarding tree is used to traverse the routing tree. The traversal starts at the root of the forwarding tree and simply follows the branches indicated by the nodes of the forwarding tree. Whenever during the traversal of the routing tree a node containing a list of source interfaces is encountered (Listing 4.3, lines 4–9), those interfaces are added to the set of the interfaces $\mathbb{I}_{e\rightarrow}$ the event e should be forwarded to. When the set of the forward interfaces equals all known interfaces of the broker **B** or the last nodes of the forwarding tree have been reached the forwarding process stops. Subsequently the event e is on all interfaces in the $\mathbb{I}_{e\rightarrow}$ set.
Listing 4.3 shows the algorithm for the forwarding of the events based on

```
 1 forward(FTNode ftN, Node n, Set forwardI, Set allI) {
 2 //for each child node of the current forwarding tree node
 3    for(CFPTNode currFTN: ftN.children()) {
 4 //get corresponding routing tree node
 5       Node currN = n.getChild(currFTN.value());
 6       if(currN.hasInterfaces()) {
 7 //add the source interfaces to the forwardI set
 8          forwardI.addAll(currN.getInterfaces());
 9          if(forwardI.equals(allI)) {
10 //broadcast the event
11             break;
12          }
13       }
14       forward(currFTN, currN, forwardI, allI);
15    }
16 }
```

Listing 4.3: Event forwarding – using the forwarding tree

the forwarding tree. The recursive algorithm takes as parameters the root node of the forwarding tree (`ftN`), the root node of the routing tree (`n`), the initially empty set of interfaces on which a given event should be forwarded (`forwardI`) and the set of all interfaces (`allI`) of the broker running the algorithm. The algorithm starts by searching for branches in the routing tree which would correspond to the nodes of the forwarding tree – line 5. If such node is found and it marks an end of a filter (line 6), the source interface of the filter is added to the `forwardI` set of the event (line 8) and it is checked whether the `forwardI` set contains already all interfaces known to the broker running the algorithm – line 9. If this is the case, the algorithm stops, otherwise it descends down both routing and forwarding trees.

4.3.2 Forwarding Tree Size

Using forwarding trees for event dissemination allows to perform only one content-based match per event. It can bee seen however, that with the increasing number of different subscriptions stored in the routing tree the average forwarding tree size will also increase. In order to limit the overhead one can exploit the property mentioned in the Section 4.3.1. Instead of matching the event until the whole routing tree has been completely explored one can limit this process to a given level (height) of the routing tree.
This idea is based on the fact that the forwarding tree is constructed exploiting the coverage relation of predicates. Specifically, for every predicate (node) p in the routing tree it can be said that all predicates which are its descendants represent filters being a specialization of a filter constructed by traversing the path from the root of the routing tree to the predicate (node) p.
As an example let us again consider Figure 4.4. Instead of creating the complete

forwarding tree for the event e one can abort the matching algorithm at the level 1. This will result in the forwarding tree of the event e having just two entries (apart from the root) – the 0 and the 2. The forwarding algorithm (a variant of the algorithm in Listing 4.3) using such constructed forwarding tree can be outlined as follows:

1. perform the normal forwarding (Listing 4.3) algorithm until the last nodes of the forwarding tree have been reached
2. if the corresponding routing tree node has any subtrees (is not the leaf node) extract all filter source interfaces from these subtrees and add them to the $\mathbb{I}_{e\rightarrow}$
3. forward the event e on all interfaces in the $\mathbb{I}_{e\rightarrow}$ set

The presented approach allows to trade off the size of the forwarding tree for the number of false positives. Intuitively, it is possible that some of the subtrees of the routing tree node corresponding to the last forwarding tree node do not contain filters that match the event. This can be demonstrated for the event $e \equiv \{x = 6;\ y = 5;\ z = 2\}$ depicted in Figure 4.4. Limiting the depth of the forwarding tree to one, the matching of the forwarding tree of the event e will stop at the node $x > 5$ and $y > 4$ of the routing tree. According to the above algorithms all filter interfaces from the subtrees of those nodes will be added to the $\mathbb{I}_{e\rightarrow}$ set. Which will result in the $\mathbb{I}_{e\rightarrow}$ set containing interfaces 1, 2 and 3. This, in turn, implies that event e will be additionally routed on the interface 1 towards the subscriber of the $\{x > 5;\ y < 3\}$ filter, although event e obviously does not match this filter. The forwarding of event e on interface 1 will result in a false positive. However, limiting the height of the forwarding tree will never result in a false negative.

4.4 Consistency Issues

Correct forwarding of an event with attached forwarding tree is only possible if the paths in the routing tree selected by the forwarding tree preserve the predicate conjunctions from the first broker which created the forwarding tree and attached it to the event. Hence, all brokers on the event's path from the publisher to the subscriber must forward the event using its forwarding tree and identical routing trees. In what follows, the set of conditions that must be met by the routing trees in order to achieve identical distribution of predicates is formulated: (1) all routing trees must be composed of the same set of filters, (2) filters must be inserted into all routing trees in the same order and (3) filter insertion must be deterministic, i.e., inserting two identical filters f_1 and f_2 into two identical routing trees RT_1 and RT_2 must result in two new routing trees RT'_1 and RT'_2 which are also identical.

The last condition is already satisfied by the insertion algorithm presented in Listing 4.1. The second condition ensures that split operations in Listing 4.1 are executed in the same order. If filter f_1 is split with respect to f_2 the resulting routing tree might be different from the one created when the filter f_2 is split with respect to the filter f_1. This problem might be circumvented by performing splits with respect to some predefined value.

Let us assume that 0 would be such a predefined value. If the routing tree already contains the $x < 6$ predicate and a split should be performed due to arrival of the $\{x > 3\}$ filter, then instead of splitting predicate $x > 3$ with respect to the predicate $x < 6$ a split of the predicate $x < 6$ with respect to the predicate $x > 3$ should be performed, as the attribute constraint > 3 is closer the predefined 0 value than the attribute constraint < 6. This method might be however impractical as it involves a potentially large overhead resulting from the reordering of the routing caused by the child nodes of the $x < 6$ node.

The first condition (all routing trees must be composed of the same set of filters) is almost never satisfied in a dynamic loosely coupled publish/subscribe system. Let us consider two brokers **B1** and **B2** with two routing trees RT_1 and RT_2. If an event e has entered the publish/subscribe network via the **B1** broker, the **B1** broker will assign a forwarding tree FT to the event e. The assigned forwarding tree FT has been created by matching the event e against the routing tree RT_1. When event e is subsequently forwarded to the broker **B2**, it uses the forwarding tree FT to traverse its Routing Tree RT_2. If routing trees RT_1 and RT_2 are different than branch identifiers encoded in the forwarding tree FT when used with the routing tree RT_2 might map to different conjunctions of predicates than in case of the routing tree RT_1. Specifically, some of the branch identifiers might not be matched at all – resulting in false negatives. Above problems might occur whenever two brokers have a different set of filters due to coverage between filters, packet drops or message processing delays.

4.4.1 Tree Optimizer

Therefore, in order to solve the above issues, the concept of the tree optimizer is introduced. The tree optimizer is a node in the publish/subscribe system which handles the task of the construction of the routing tree. Tree optimizer takes over the routers' responsibility to create and manage the routing trees and thus delivers the necessary determinism for the routing tree construction. It is important to note that the tree optimizer itself obeys and uses the semantics of the content-based publish/subscribe system. The idea of the tree optimizer is parallel to the idea of the lock service [Bur06], where a central instance in a distributed system is responsible for the consensus among all other participants of the system.

The tree optimizer creates the routing trees by first subscribing to all subscriptions of the subscribers. This way it collects the data using the decoupled and

asynchronous semantics of the publish/subscribe system. Another important fact is that the tree optimizer can be distributed in that each distributed tree optimizer subscribes to a part of the routing tree for which it will be responsible – effectively partitioning the filter and event space, so that every broker maintains a number of the routing trees. However, for the purpose of this thesis a single tree optimizer instance is assumed.
The tree optimizer is, by definition, the only entity in the whole publish/subscribe network which is allowed to add or remove nodes to/from the routing tree. This simple fact ensures the necessary determinism for the operations regarding the routing tree. This in turn requires the introduction of the following changes to the routing tree management (cf. Section 4.2): (1) brokers, upon reception of a new filter always forward it to the tree optimizer (2) filters are still inserted into the routing tree at every broker, however the insertion is performed using new `insertR()` function and (3) brokers receive routing tree updates from the tree optimizer and map their own view of the network on the received RT.

4.4.2 Routing Tree Management

In a typical scenario with the management of the routing tree delegated to the tree optimizer one might expect routers to send new filters to the tree optimizer and wait for the routing tree updates from the tree optimizer containing new filters incorporated into the new routing tree. The main problem with such approach is the fact that events which might be matched by the new filters will not be forwarded to subscribers until the new routing tree is disseminated by the tree optimizer. One can however cope with that problem by making use of the routing tree properties described in the Section 4.3.2.
Instead of waiting for the updates to arrive, brokers insert new filters immediately into the routing tree using the modified `insert()` algorithm from the Listing 4.1. The new `insertR()` algorithm is analogous to the old `insert()` algorithm with only difference being that the `insertR()` algorithm does not create new routing tree nodes.
Figure 4.5 shows the intuition behind the `insertR()` algorithm. Let us assume a broker **B** holding a routing tree from Figure 4.2. A set of three filters: $f_1 \equiv \{x > 5;\ y < 7\}$, $f_2 \equiv \{x < 7;\ y > 4\}$ and $f_3 \equiv \{y > 8;\ z < 6\}$ is delivered in the indicated order to the broker **B**. According to the outline of the `insertR()` algorithm broker **B** inserts the filters into the routing tree, however whenever it should create a new node, either due to the coverage relation or due to the split operation it stops the insertion process and adds the filter source interface to the current routing tree node. It can be observed that in the case of the filter f_1 the first predicate $x > 5$ is equal to the one already present in the routing tree. However the second predicate of the filter f_1 covers the $y < 3$ node which according to the original `insert()` algorithm (Listing 4.1) would result in

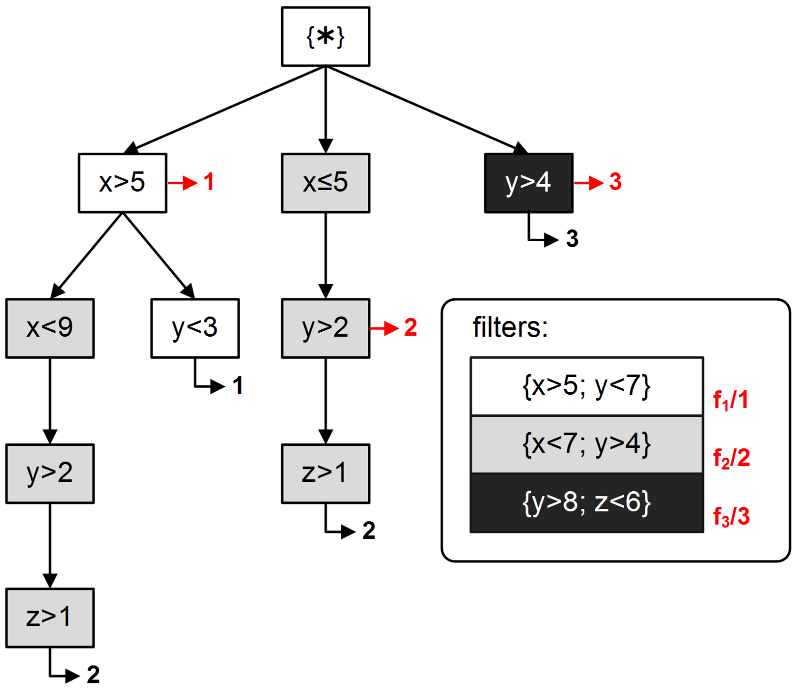


Figure 4.5: Updating routing tree between epochs

the creation of the new node which would than be assigned the existing $x > 5$ node as a child node. Therefore, according to the new algorithm `insertR()` the process stops and inserts the filter source interface at the $x > 5$ node.

The so inserted source interface allows to match a superset of events matched by the inserted filter without the modifications to the predicate (node) structure of the routing tree. Moreover, it is possible to route all events independently of the frequency of updates from the tree optimizer, the only trade-off being false positives. Whenever a routing tree update from the tree optimizer arrives, the number of false positives is automatically reduced as the routing tree becomes more precise again.

In order to maintain the space decoupling in the publish/subscribe system the tree optimizer is responsible for handling only of the contents of the filters in the routing tree. Specifically, the tree optimizer does not consider the source interfaces of filters it receives. This leads to an observation that the routing tree management is decoupled between the tree optimizer and publish/subscribe brokers. The tree optimizer manages the filters and the predicates in the routing tree, while brokers manage the filter source interfaces.

The tree optimizer creates the routing tree without inclusion of the filter source interfaces, inserting only predicates. When the difference between the current routing tree maintained in the tree optimizer and the last routing tree sent by the tree optimizer to the brokers exceeds certain threshold, the tree optimizer sends an updated routing tree version to all brokers.

In order to calculate the difference between the current and last disseminated routing tree the tree optimizer uses a weighted function wdiff(). The weighted function wdiff() calculates the difference between two routing trees RT_1 and RT_2 as:

$$\text{wdiff}(RT_1, RT_2) = \frac{\text{diff}(RT_1, RT_2)}{\text{wsize}(RT_1) + \text{wsize}(RT_2)} \tag{4.2}$$

where diff(RT_1, RT_2) is given by:

$$\text{diff}(RT_1, RT_2) = \sum_{n:(n \in RT_1 \wedge n \notin RT_2)} \frac{1}{\text{height}(n)} + \sum_{m:(m \in RT_2 \wedge m \notin RT_1)} \frac{1}{\text{height}(m)} \tag{4.3}$$

and wsize(RT_1) is given by:

$$\text{wsize}(RT) = \sum_{n \in RT} \frac{1}{\text{height}(n)} \tag{4.4}$$

where height(n) is the distance of the node n from the root of the routing tree. The weighted algorithm emphasizes changes closer to the root as those will have more significant impact on the potential number of false positives. Brokers upon reception of the new routing tree updates insert outstanding filters (since last update) using the `insertR()` algorithm. This way they create a local mapping of the filter return addresses onto the routing tree preserving the decoupling principles.

Another issue arises when one considers that it is possible for some events to be still underway in the publish/subscribe network when a routing tree update is processed by the brokers. This might result in a situation when the forwarding tree of an event e was created with the routing tree RT_1, however the subsequent forwarding process based on the forwarding tree of the event e would have to be performed with a different routing tree RT_1', due to the arrival of the routing tree update $RT_1 \longrightarrow RT_2$ from the tree optimizer.

This problem is solved by implementing a monotonically increasing virtual clock in the tree optimizer. This virtual clock is incremented with every publication of the routing tree update. Upon the update publication, the tree optimizer attaches the clock's value to all new nodes of the tree. This results in a construction of a routing tree whose nodes are tagged with the virtual creation time. When event is being matched based on its content (in order to extract the forwarding tree for the event) with the routing tree of a broker it is assigned a time stamp equal to the virtual clock time stamp of the youngest (highest virtual clock time stamp) routing tree node that event's forwarding tree stores a reference to. Subsequent event forward operations consider only the routing tree nodes which are tagged with virtual clock time stamps being less or equal to the time stamp assigned to the event's forwarding tree.

Figure 4.6 shows the interaction between the tree optimizer and the brokers in the publish/subscribe system. The tree optimizer **TO** first subscribes to

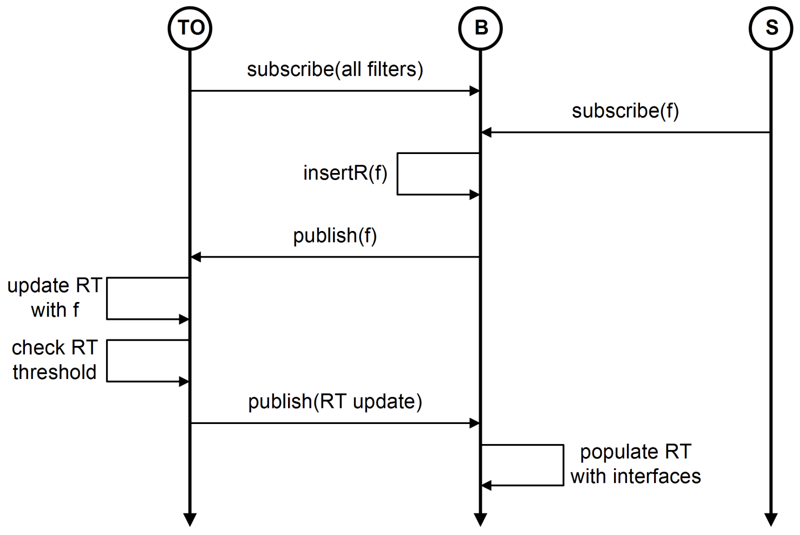


Figure 4.6: Broker and tree optimizer interaction

all filters (`subscribe(all filters)`) issued by the subscribers. All filters subscribed to by the subscribers are inserted by the brokers **B** into their routing tables using the `insertR` algorithm – see Section 4.4.2. Simultaneously, due to the tree optimizer subscription, filters carried by the subscription messages are published (`publish(f)`) by the brokers towards the tree optimizer. Tree optimizer updates its version of the routing tree with the new filters. Whenever the new routing tree managed by the tree optimizer exceeds the difference threshold to the last sent version (see Equation 4.2) the tree optimizer publishes (`publish(RT update)`) the routing tree update. Brokers upon the reception of the routing tree updates populate the new routing trees with the interfaces of the subscription messages, reinserting the corresponding filters using the `insertR` algorithm.

4.5 Related Work

The first work to explicitly suggest following of the end-to-end principle in the design of the content dissemination systems was proposed in [CS05]. Authors propose to combine content-based addressing with a multicast-based approach. Presented solution requires that (1) every broker in the network receives all subscriptions and (2) every broker is aware of all other brokers in the network and (3) every broker maintains an all broker pairs latency matrix. Events in [CS05] are matched against all filters and a list of all matching brokers is extracted. Subsequently, the dynamic multicast protocol, exploiting the brokers distance in terms of latency is used to forward events to the interested brokers. The prefix forwarding approach differs in that no global knowledge regarding the publish/subscribe nodes is required. Moreover, the result of the

event matching reflects only the filters matching the event, thus preserving the decoupling between the components of the publish/subscribe system.
Another approach to excel the content-based matching in publish/subscribe systems has been proposed in [ST06]. Authors combine the content-based addressing with hash-based matching so that downstream brokers can reuse the matching results from the upstream brokers. The proposed solution assumes that broker matching an event attaches hashes of n (out of N) filters matched by the event to the event and forwards the event towards the downstream broker. The downstream broker uses the hashes attached to the event to find interfaces on which to forward the event and performs a content-based matching for interfaces not excluded by the forwarded hash values. The prefix forwarding approach is similar in that the matching performed in the application level is reused, however, the prefix forwarding performs matching on the predicate level and does not rely on the calculation of the hash values for filters – instead deterministic forwarding tree is used. Moreover, the prefix routing algorithm offers significantly higher gains regarding the decrease of event matching times. The prefix forwarding offers over two orders of magnitude speed increase as opposed to 60% offered by the approach presented in [ST06].
Authors in [WQA+02] propose to use Event Distribution Networks (EDN) – a self configuring overlay network of servers which optimizes the event and filter routing performance. An EDN is composed of a set of nodes, part of which, called edge servers are an interface to geographically distributed publishers and subscribers. The remaining nodes reside within the network and are used for event and filter routing. This results in decoupling of the subscribers and publishers from the event distribution network, which in turn allows to freely manage filters within the EDN itself. Authors propose to use event space partitioning, to partition the event space between the EDN servers so that a single event needs only to be distributed to the brokers storing filters matching the event. The prefix routing approach differs from the one presented in [WQA+02] in that it does not rely on the partitioning of the event space. Partitioning of the event space is a difficult problem for the content based publish/subscribe systems, especially if one considers that a filter is a k-dimensional subspace of the d-dimensional filter and event domain, where $1 \leqslant k \leqslant d$ and $1 \leqslant d \leqslant \infty$ is not fixed during the runtime of the system.

Chapter 5

Bloom Filter-Based Routing

The prefix forwarding architecture proposed in the previous chapter allows to forward events based on their content performing only one content-based match per the delivered event. The prefix forwarding approach preserves the decoupling properties of the underlying publish/subscribe system while simultaneously significantly increasing the speed of event forwarding.
However, the proposed prefix forwarding approach exposes several issues which are the main motivation for the further development of the prefix forwarding approach into the Bloom filter-based routing approach. Bloom filter-based routing approach contrasts the prefix forwarding approach in that it offers a unified solution to the problem of event forwarding and filter routing in the content-based systems. Moreover, the Bloom filter-based routing approach offers the room for further improvements, especially regarding the possible parallelization.
The remainder of this chapter is structured as follows: Section 5.1 discusses the motivation behind the development of the Bloom filter-based routing and presents the overview of the proposed approach. Section 5.2 introduces background information regarding the Bloom filters and their implementation in context of the Bloom filter-based routing XSiena system. Two subsequent sections detail the algorithms for the routing of filters (Section 5.3) and forwarding of events (Section 5.4). Subsequently, Section 5.6 discusses the potential for parallelization of the event forwarding process. This chapter is concluded with a brief overview of the related work in Section 5.7.

5.1 Overview and Motivation

The first issue which motivated the development of the Bloom filter-based alternative to the prefix forwarding (cf. Chapter 4) was the size of the forwarding tree. Based on the evaluation of the prefix-based routing strategy (see Chapter 9) it can be said that the forwarding tree size remains relatively stable, in the function of the number of elements in the routing tree. However,

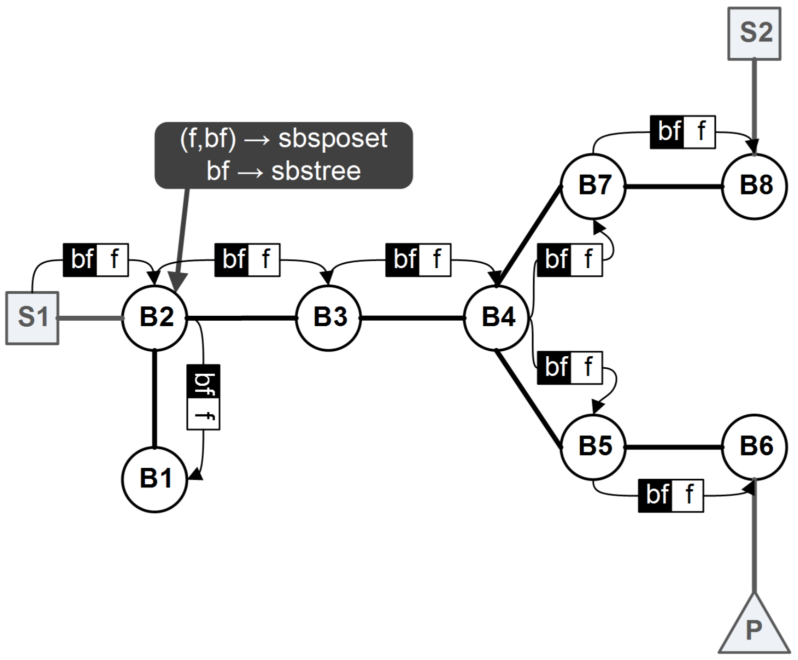


Figure 5.1: Routing of filters

increasing the number of attribute name and value pairs in the events results in the higher number of filters being matched and simultaneously, the higher number of entries in the forwarding tree. The extreme case being the forwarding tree containing all branches of the routing tree, i.e., a case when event matches all filters in the publish/subscribe system. Bloom filter-based routing copes with this issue by replacing the routing tree and the forwarding tree with data structures which allows to construct routing prefixes with an upper bound on their size. This implies that independently of the number of filters being matched by an event the size of the prefix attached to the event will not grow larger than some predefined value.

Second issue addressed with the development of the Bloom filter-based routing are the consistency guarantees provided by the tree optimizer. The tree optimizer remains a valid and decoupled approach for solving of the concurrency issues arising during the creation of the routing tree. However, an approach which does not rely on such constructs offers the potential for a better scalability, especially in the case of large scale distributed systems.

Therefore, the concept of the Bloom filter-based routing has been developed. Bloom filter-based routing upholds the principles presented in the Chapter 4. Specifically, Bloom filter-based routing follows the end-to-end argument in that it shifts the task of the content-based forwarding of events from the publish/subscribe network layer into the edge of the network. Moreover, thanks to the Bloom filter-based approach a unified framework for both event forwarding and subscription routing can be proposed.

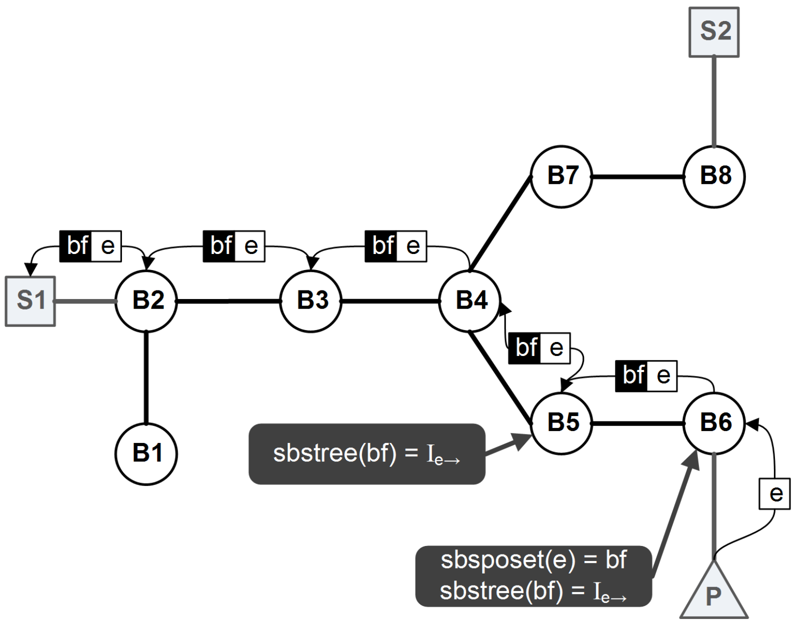

S2
B7
B8
bf e
bf e
bf e
S1
B2
B3
B4
bf e
bf e
B5
B6
B1
sbstree(bf) = $\mathbb{I}_{e\rightarrow}$
e
sbsposet(e) = bf
sbstree(bf) = $\mathbb{I}_{e\rightarrow}$
P

Figure 5.2: Forwarding of events

The basic forwarding and routing algorithm of the Bloom filter-based routing approach is similar to that of the prefix forwarding approach. Figure 5.1 presents the filter routing process using the Bloom filter-based routing approach. A subscriber (**S1**) issuing a filter f constructs a Bloom filter bf which summarizes the filter's content and attaches it to the filter prior to the filter dispatching. Every broker upon the reception of the subscription message containing the filter f with attached Bloom filter bf inserts the filter f into the `sbsposet` routing structure and the Bloom filter bf into the `sbstree` routing structure. The subscription message is subsequently routed further into the publish/subscribe network using the identity-based routing algorithm – see Section 2.5.4.

Figure 5.2 presents the forwarding of the events. Publisher **P** publishes event e which is subsequently delivered to the publisher connecting broker **B6**. Broker **B6** upon the reception of the event e matches the event with the `sbsposet` routing structure and extracts a Bloom filter bf containing a digest of the subscribers' interest matched by the event. The process of matching of the event e with the `sbsposet` is performed only once in the first broker encountered by the event e.

Subsequently, broker **B6** uses the Bloom filter bf in conjunction with the `sbstree` routing structure to calculate the set of interfaces $\mathbb{I}_{e\rightarrow}$ on which event e should be forwarded. The process of calculation of the set of interfaces $\mathbb{I}_{e\rightarrow}$ relies only on the contents of the Bloom filter b and does not inspect the content of the event e. All subsequent brokers forwarding event e towards subscriber **S1** use on the bloom filter bf attached to the event e.

5.2 Bloom Filters

This chapter relies heavily on the Bloom filters. Bloom filters [Blo70] are probabilistic data structures which represent a set of n elements using a vector of m bits. Bloom filters never exceed their predefined size of m bits and are able to represent the whole universe of elements within the m bits of their size. The trade-off for the compact data representation is a probability of false positives. A false positive occurs when a Bloom filter queried for the presence of the element e returns true, event though element e was never inserted into the Bloom filter. Bloom filters, however, never return false negatives, i.e., an element which was inserted into a Bloom filter is never reported as missing.
A Bloom filter of width m uses k independent hash functions which taking as argument an element to store in the Bloom filter produce a hash value of exactly m bits width. Storing an element in a Bloom filter equals to setting k bits in the Bloom filter, which correspond to the return values of the k hash functions. Specifically, if all m bits are set in the Bloom filter than it represents the whole universe of elements.

5.2.1 False Positives Probability

A false positive probability for a Bloom filter is calculated as a probability of two different elements having k identical hashes. Intuitively, the false positives probability is proportional to the number of elements n stored in the Bloom filter and inversely proportional to the size m of the Bloom filter. The probability p of a false positive is given by [FCAB00]:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{5.1}$$

The value of k which minimizes the false positives probability p is given by:

$$k = \frac{m}{n}\ln 2 \tag{5.2}$$

Using k from Equation 5.2 in Equation 5.1, one can obtain the probability as a function of only two parameters: m and n. This in turn allows us to easily calculate the desired size of the Bloom filter as a function of the false positives probability p and expected number of elements n. Figure 5.3 shows the relation between p, m and n for the value of k given in Equation 5.2.
One of the issues when using Bloom filters is the need to provide k different hash functions. In order to reduce the need for computation of possibly large number of different hash functions, additional hash functions are simulated with the help of the double hashing technique [DM04]:

$$g_i(x) = h_1(x) + ih_2(x) \tag{5.3}$$

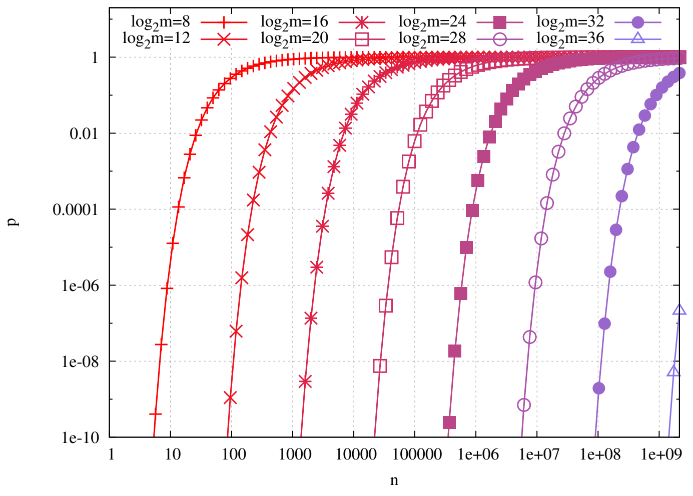


Figure 5.3: False positive probability p as a function of number of elements n and the size m of the Bloom filter

The authors of [KM06] have shown that only two hash functions $h_1(x)$ and $h_2(x)$ are necessary to effectively implement k hash functions for use with a Bloom filter without any loss in the asymptotic false positive probability.

5.2.2 Counting Bloom filters

Removal of elements from a Bloom filter in its above form is not possible. The intuition behind that reasoning is that removal of one of the elements which share one of the bits set by the k hash functions would result in a false negative for the second element. False negatives are not allowed in Bloom filters. In order to circumvent this problem counting Bloom filters can be used [FCAB00]. A counting Bloom filter stores not only the indices of the set bits, it also stores the number of times a given bit was set. The addition of the same element twice results in increasing the counter of the corresponding bits to 2.

The removal of the element does not result in the removal of the given bit from the Bloom filter unless the decremented counter associated with the given bit equals zero. In order to remove an element inserted twice it needs to be removed twice as well, as the first removal would keep all the corresponding bits with the counter decremented to one.

5.2.3 Bloom filter implementation

Typical Bloom filter implementations assume that m bits of the Bloom filter are represented as an array of $\frac{m}{\text{sizeof}(word)}$ words – a bit set. Using 64 bit integer values a Bloom filter could be represented as:

$$\underbrace{0\ldots63}_{0},\ \underbrace{0\ldots63}_{1},\ \cdots,\ \underbrace{0\ldots63}_{\frac{m}{64}} \tag{5.4}$$

Setting a bit number b in a bit set based Bloom filter corresponds to setting a bit number b AND 63 in the integer number $b \gg 64$, where AND is a bitwise AND operation and $\gg$ is a bitwise shift right operation. The above implementation outline is however relatively inefficient if sparse Bloom filters are considered. A sparse Bloom filter is a Bloom filter storing only a small number of elements, which corresponds to a small number of bits being set in the Bloom filter. From the above one can observe that setting of one bit number $m - 1$ requires initialization of $\frac{m}{\text{sizeof}(word)} - 1$ words with the value 0 and the last word with the value $1 \ll (\text{sizeof}(word) - 1)$.
Therefore, an alternative implementation, of Bloom filters using a sparse bit set is proposed. A sparse bit set represents a Bloom filter using a dynamic array of integers representing the indices of bits which are set. Setting of the bit number $m - 1$ requires only the addition of the $m - 1$ value into the sparse bit set. Intuitively, there is a trade-off in space efficiency when the traditional and the sparse bit set implementations of Bloom filters are concerned. A sparse representation is more efficient if a Bloom filter is holding less than $\frac{m}{log_2(m)}$ elements, where m is the size of the Bloom filter. For a sparse bit set using $log_2(m) = 32$ bit integer values this corresponds to a fill ratio of 3% – i.e., the number of bits set in the Bloom filter divided by the total numbers of bits m.
One of the major advantages of the sparse Bloom filter implementation over the traditional implementation is the cheap over provisioning of the Bloom filter size. Figure 5.3 can be used to calculate the size of the Bloom filter needed to stay below a certain false positives rate for a given number of elements which are to be inserted into the Bloom filter. Achieving low false positives rates translates to creating large Bloom filters, which in their traditional implementation require allocation of large arrays. The advantage of the sparse Bloom filter is that empty Bloom filter, *regardless* of its size m always contains the same size of bytes – zero. Setting the bit $m-1$ in the sparse Bloom filter increases its size by the $\text{sizeof}(word)$ bytes. Setting the bit $m - 1$ in the traditional Bloom filter increases its size by $\frac{m}{\text{sizeof}(word)}$ bytes. The size of the traditional Bloom filter remains subsequently fixed at this value. Such behavior is problematic, specifically for very large values of m, or when such Bloom filter is to be transmitted over the network. From the above it can be concluded that there is very little penalty which needs to be paid if very large Bloom filters are created using the sparse representation, as opposed to very large penalties when the

Filter	Bloom filter	Source address	Id
$\{x > 5\}$	$\{6066, 8581\}$	*f1@dot.com*	f_1
$\{x >= 0.7\}$	$\{2787, 12518\}$	*f2@dot.com*	f_2
$\{x > 15\}$	$\{6441, 10582\}$	*f3@dot.com*	f_3
$\{y < 5\}$	$\{5037, 8516\}$	*f4@dot.com*	f_4
$\{y = 0\}$	$\{5001, 8507\}$	*f5@dot.com*	f_5
$\{x > 15;\ y > 0\}$	$\{6441, 10582\}$ $\{5102, 8636\}$	*f6@dot.com*	f_6

Table 5.1: The set of subscriptions used in following examples

traditional representation is used. Therefore, it can be said that sparse Bloom filters allow for cheap over provisioning of the Bloom filter size.

5.3 Filter Routing

The routing of filters is performed using identity-based routing strategy. Filters are issued by the subscribers and are delivered to the routers in form of subscription messages. Every filter generated by the subscriber carries a set of Bloom filters representing the content of the filter predicates. The calculation of the Bloom filter bf_i for the predicate p_i of the filter f is performed using k hash functions:

$$\forall_{p_i \in f} \quad : \quad bf_i = \bigcup_{j=0}^{j=k-1} h_j(p_i) \tag{5.5}$$

where $h_j(p_i)$ is the j^{th} hash function of the predicate p_i belonging to the filter f. The calculation of the hash function over the predicate p is straightforward and can be, e.g., executed as calculation of the standard hash function [Riv92, DEEJ01] over the string literal representing the predicate p. It is important to note that the only requirement on the result of hash function calculation for the predicate p is that for two different predicates p_1 and p_2 the calculated hash functions are also different with probability proportional to the hash function width m.

Table 5.1 shows example filters with their source interfaces and corresponding Bloom filters. It can be observed that the number of hash functions k used in case of the filters in Table 5.1 is equal to 2. The size of the Bloom filter m is set to $2^{14} = 16384$. It can be observed that Bloom filters are calculated on a per predicate basis – the last filter $\{x > 15;\ y > 0\}$ has two Bloom filters: one for predicate $x > 15$ (equal to $\{6441, 10582\}$) and one for predicate $y > 0$ (equal to $\{5102, 8636\}$). Bloom filters are represented by the indices of the bits which

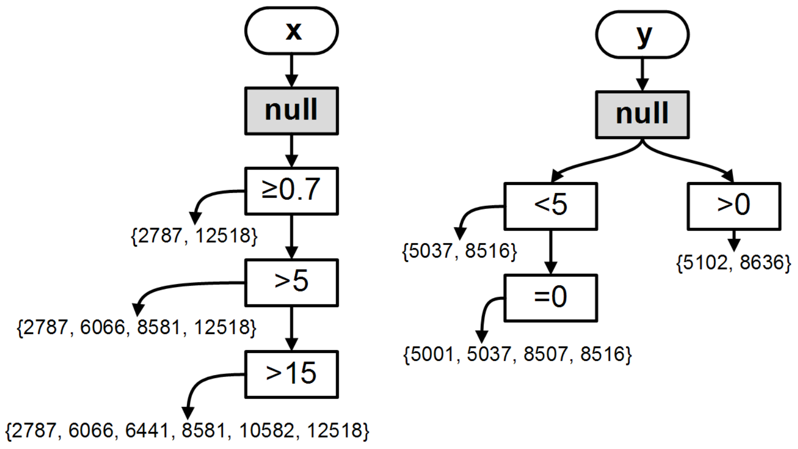


Figure 5.4: The `sbsposet` storing filters from Table 5.1

are set to one, enclosed in curly braces.
Subscription messages carrying filter f and the calculated Bloom filter bf arriving at every broker are stored in two data structures: the `sbsposet` and the `sbstree`. The `sbsposet` stores the predicates of filters along with the accompanying Bloom filters. The `sbsposet` is responsible for the content-based matching of incoming events. The `sbstree` stores a disjunction of conjunctions of Bloom filters representing filters' predicates and is responsible for fast forwarding of events without falling back to the events' content.

5.3.1 sbsposet

The `sbsposet` (see Figure 5.4) stores the predicates of filters by abstracting away the conjunctive form of subscriptions. The `sbsposet` routing structure stores predicates grouped by their attribute name. Figure 5.4 illustrates two attribute names: x and y. Every attribute name points to a partially ordered set of attribute constraints. Each partially ordered set of attribute constraints begins with a virtual root node (`null`) and stores attribute constraints sharing the same attribute name. The structure of the attribute constraints reflects the partial order resulting from the covering relation. Every attribute constraint has a Bloom filter associated with it. Bloom filter represents the whole predicate which contains the given attribute constraint and is calculated by the subscriber – hence attribute constraint > 5 in the x branch of the `sbsposet` has the Bloom filter of the $x > 5$ predicate.
Figure 5.4 shows the `sbsposet` storing predicates of filters shown in Table 5.1. It can be observed that filter $\{x > 15;\ y > 0\}$ is stored without the preservation of the conjunctive form between its predicates. Simultaneously, one can observe that the `sbsposet` maintains the coverage relation between the attribute constraints which is reflected in their Bloom filters.
The attribute constraint > 5 which is covered by the attribute constraint $\geqslant$

0.7 reflects this fact within the `sbsposet` by including (using a logical OR operation) the Bloom filter of the covering attribute constraint {2787, 12518} in its own. The Bloom filter of the > 5 attribute constraint encodes this way the fact that matching the > 5 attribute constraint implies matching of the covering $\geqslant 0.7$ attribute constraint. The `sbsposet`, similarly to the routing tree (cf. Section 4.3.1) by breaking the conjunctive form of the filters allows to represent filters in a more compact form – an example being filters: $\{x > 15;\ y > 0\}$ and $\{x > 15\}$ which in the `sbsposet` are represented using only two predicates: $x > 15$ and $y > 0$, instead of three as it would be the case if the poset [CRW01] or forest [TK06] data structures were used.
The removal of filters from the `sbsposet` is performed, similarly to insertion, on a per predicate basis. For the removal operation to work correctly (similarly as in the case of the counting Bloom filters) a counter associated with every predicate is implemented. The predicate counter, initially set to one, is incremented on every insertion of the identical predicate into the `sbsposet`. The predicate counter is decremented upon the predicate removal. A predicate is removed from the `sbsposet` if and only if upon its removal the counter is decremented to zero. Counting Bloom filters are also used in the `sbsposet` in order to cope with bit collisions during the removal of elements.

5.3.2 `sbstree`

The `sbsposet` stores the content of filters without regarding the conjunctions between the predicates of a single filter. The loss of this information could lead to a potentially large number of false positives, i.e., events delivered to subscribers which did not subscribe to them. Hence, there is a need for a data structure which would represent the conjunction between the predicates of filters. This section introduces such data structure – the `sbstree`. The main task of the `sbstree` is to represent the disjunction of conjunctions of predicate values. Therefore, in contrast to the `sbsposet`, the `sbstree` stores filters in their conjunctive form. Specifically, the `sbstree` does not store any predicates, instead, it works exclusively with the Bloom filters representing filters.
The disjunction of the predicates of a single filter can be expressed with a single Bloom filter using an OR operation between the Bloom filters of single predicates. However, this simple approach cannot be used to represent a conjunction of filter predicates. Therefore, the `sbstree` structure is used to cope with that issue – see Figure 5.5. The path from the root to the leaf of the `sbstree` forms a conjunction of the predicates of the given filter, or more precisely, the conjunction between the bits set in the Bloom filter of the given filter.
A Bloom filter of a filter is a logical OR between the Bloom filters of all filter predicates. For a given filter the bits set in its Bloom filter form a path rooted at the virtual `sbstree` root node – `null`. The leaf of the path represents an

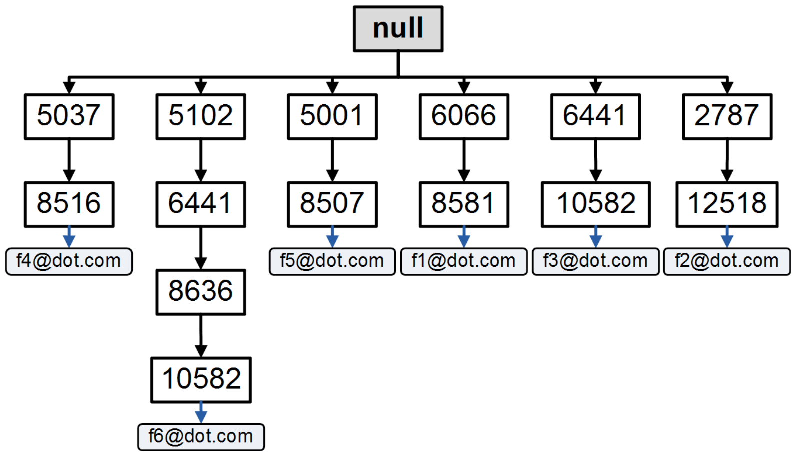


Figure 5.5: The `sbstree` storing subscriptions from Table 5.1

interface on which the given filter arrived – cf. Table 5.1. Hence, a path leading from the root node to the source interface forms a conjunction of all predicates of a given filter. As an example let us consider the longest path in the `sbstree`. It is formed by the bits forming the Bloom filters of the $f_6 \equiv \{x > 15;\ y > 0\}$ filter which arrived on the interface `f6@dot.com`. In order to arrive at the source interface of the filter f_6 one needs to match all bits of all predicates forming the filter f_6.

The filters are inserted into the `sbstree` according to their arrival order. Specifically, only the Bloom filters representing a logical OR between the Bloom filters of all predicates of the filter to be inserted plus the source interface are stored in the `sbstree`. In case of a bit collision, e.g., identical first bit in two different Bloom filters the `sbstree` branches.

The removal of filters from the `sbstree` is straightforward, and does not require the usage of additional counters. It is worth noting that identical filters arriving from same interfaces are never permitted into the `sbstree` as such filters are dropped (on the second occurrence) due to the application of the identity-based routing. Filters are therefore first inserted into the `sbstree` and upon success are stored in the `sbsposet` as well.

One can observe that unlike the routing tree presented in Section 4.2.2 both structures `sbstree` and `sbsposet` are deterministic with respect to the order of the filter arrival and insertion. This ensures that every broker in the system (assuming all filters issued by subscribers are delivered to all brokers – according to the identity-based routing scheme) has eventually the same layout of both structures.

```
 1 //calculate Bloom filter for event
 2 public BloomFilter match(Event event) {
 3    BloomFilter bf = new BloomFilter();
 4    for((attrName, attrValue) : event) {
 5       SBSPosetNode root = sbsposet.get(attrName);
 6       if(root != null) {
 7          root.match(attrValue, bf);
 8       }
 9    }
10    return bf;
11 }

13 //recursive attribute value matching with sbsposet
14 public match(AttrValue av, BloomFilter bf) {
15    boolean match = false;
16    for(SBSPosetNode child : this.children) {
17       if(Covering.apply(child.attrConstraint, av)) {
18          child.match(av, bf);
19       } else if(this.parent != null){
20          match = true;
21       }
22    }
23    if(match || this.children.isEmpty()) {
24       bf.OR(this.bloomFilter);
25    }
26 }
```

Listing 5.1: Bloom filter extraction during event matching

5.4 Event Forwarding

The process of event forwarding starts with the publisher publishing event *e*. Publication message carrying event *e* is subsequently received by the publisher connecting broker **B6** – see Figure 5.2. Broker **B6** performs two operations: (1) it matches the event *e*, based on the content of the event *e* with the `sbsposet` and extracts the Bloom filter bf, which it attached to the event *e* and (2) it calculates the set of interfaces on which to forward the event *e* based on the Bloom filter bf attached to the event. Subsequent brokers forwarding event *e* with attached Bloom filter bf perform only the second routing step.

The result of the process of matching of the event *e* with the `sbsposet` is a Bloom filter bf – see Listing 5.1. The Bloom filter bf encodes, in a compact form, a set of all predicates which match the event *e*. Event matching starts by selecting all branches in the `sbsposet` which have the attribute names equal to the attribute names of the attribute name and value pairs within the event – line 5. Subsequently, starting with the virtual root node (line 7) the corresponding attribute value of the event is matched with the attribute constraints forming the poset for the given filter attribute name.

The match operation finds the attribute constraint in the `sbsposet` which is the farthest one from the root attribute constraint and matches the attribute value of the event – lines 17 and 18. An attribute name and value pair $x = 7$ would therefore match the attribute constraint $x > 5$ in the Figure 5.4. The last step in the matching process is the assignment of the bloom filter of the attribute constraint to the Bloom filter of the event – line 24.
The Bloom filter bf returned as a result of the content-based matching between the event e and the `sbsposet` is subsequently used in the process of matching with the `sbstree`. The result of the matching process with the `sbstree` is the set of interfaces on which the event e should be forwarded.
The matching process of the Bloom filter bf starts at the virtual root node of the `sbstree` – see Figure 5.5. For every integer representing a bit set in the Bloom filter bf of the event e, a comparison with the children nodes of the root node is performed. In case of a match, a path starting at the matching node is followed until either: (1) the end of the path (indicated by the source interface) is reached or (2) a given node in the path does not have a corresponding matching bit in the event's e Bloom filter bf. In the first case the set of interfaces on which to forwarded event e is extended with the given source interface. In the latter case no action is taken.
As an example one can follow the process of matching of the event $e \equiv \{x = 7;\ y = -2\}$. The event is first matched based on its content with the `sbsposet` – see Figure 5.4. It selects the predicates $x > 5$ and $y < 5$ and the Bloom filter bf of the event e is assigned the Bloom filters of both matching predicates:

$$bf \equiv \{2787, 5037, 6066, 8516, 8581, 12518\} \tag{5.6}$$

The Bloom filter of the event e, due to the properties of the `sbsposet` contains also the Bloom filter of the predicate $x \geqslant 0.7$. The Bloom filter bf of the event e is subsequently matched with the `sbstree`. The matching starts with the first bit set in the Bloom filter: 2787. This bit selects the (rightmost) branch of the `sbstree` – see Figure 5.5. Subsequently, it is checked whether there is a bit in the bloom filter bf which satisfies the second node of the branch: 12518. Since such bit is found in the Bloom filter bf the source interface `f2@dot.com` is added to the list of interfaces on which the event e should be forwarded. The process continues with the remaining bits of the Bloom filter bf.

5.5 Improved Event Forwarding

The `sbstree` matching algorithm suffers however from the fact that for every bit set in the event's Bloom filter a path in the `sbstree` needs to be selected and possibly followed. From the `sbstree` traversal algorithm one can conclude that following a given path does not guarantee the reaching of the source interface at the end of it. Additionally, the complexity of the `sbstree` traversal

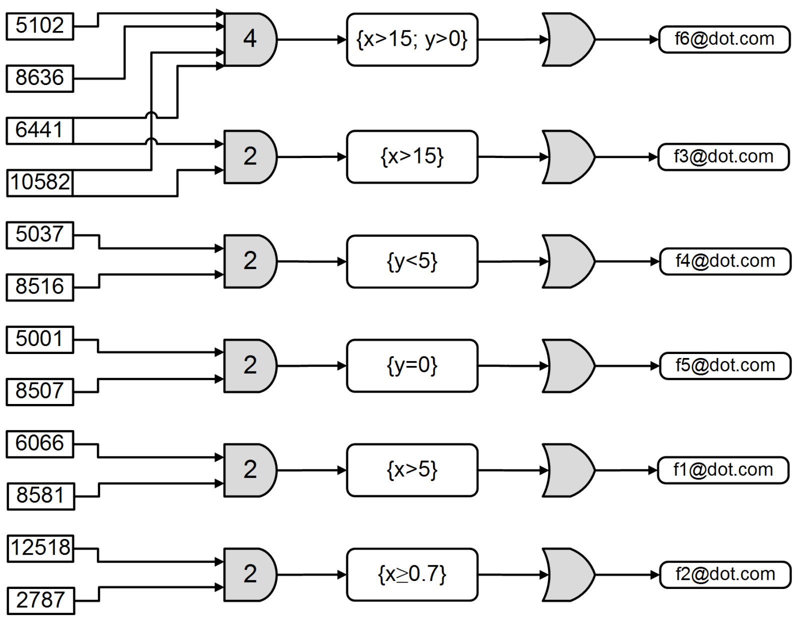


Figure 5.6: The counting algorithm variant of the `sbstree` from Figure 5.5 storing filters from Table 5.1

grows with the number of bits set in the event Bloom filters – as more and more paths need to be followed. The worst case being event Bloom filters with all bits present in the `sbstree` set, with the exception of the bits holding the filter source interfaces in the `sbstree`.

Therefore, in this section an improved version of the `sbstree`, based on a counting algorithm [YGM94, PFLS00, FJL+01] is proposed. The new version of the algorithm stores bits set in Bloom filters in a linear fashion – leftmost column on Figure 5.6. All bits from a Bloom filter forming a filter are assigned a counter. The value of the counter is equal to the number of bits set in the filter's Bloom filter. For the filter which arrived on interface `f6@dot.com` this value equals 4, as it has two predicates, each composed of a Bloom filter having 2 bits set. The counter itself represents a set of bits which need to be present in the event in order to trigger the counter. The triggered counter activates the filter it is connected to and thus the interface on which the filter arrived.

Upon arrival of an event e its Bloom filter bf is parsed for the values of the set bits. Every bit set in the event's Bloom filter bf triggers the corresponding bit in the counting `sbstree`. Triggering a bit corresponds to the decreasing of the counter which is connected to it. Whenever a counter reaches 0 the corresponding filter matches the given event. A filter matching an event e implies that the event should be forwarded on the filter's source interface – rightmost row in Figure 5.6.

The above algorithm, in contrast to the original `sbstree` variant, is linear with the number of bits set in the event. Section 9.2 presents the evaluation of this approach and contrasts it with the original `sbstree`. The major difference between the counting algorithms proposed in, e.g., [CW03] and the algorithm proposed in this section lies in that the counting variant of the `sbstree` does not require any content-based matching to be performed and therefore offers a very good potential for event forwarding speed-up.

5.6 Parallelization

In recent years we have been witnessing the exponential growth in the capabilities and processing speeds of the modern Graphics Processing Units (GPU). In 2004 a high-end 3GHz Pentium4 chip could peak 6 GFLOPS; a high-end NVIDIA GeForceFX 5900 graphic card could perform operations at 20 GFLOPS [Fer04]. In 2004 the ten year growth in the CPU speed reached 6.000%, while the ten year growth in the processing speed of the GPUs exceed 100.000%. The difference in the speed-up between the CPUs and the GPUs can be explained by the fact that GPUs are specialized designs focusing on the arithmetic operations. Therefore, the additional transistors in GPUs are used for computation instead of cache memory, like in the case of the CPUs [LHK+04]. These trends suggest the feasibility of off-loading the computationally intensive operations performed during the event forwarding in publish/subscribe systems to the Graphics Processing Units.
This section investigates the opportunities to excel the speed of event forwarding by combining the Bloom filter-based abstraction of the message content with the General-Purpose computation on Graphics Processing Units (GPGPU) [LHG+06]. The GPGPU takes advantage of hundreds of programmable vertex and fragment processors available on modern graphics cards in order to perform the arithmetically intensive non-rendering algorithms. The GPGPU introduces two main concepts which are the building blocks for all GPGPU applications: (1) streams and (2) kernels. Streams are the collections of records which require the same computation to be performed on them. Streams, due to their mapping on the underlying GPU architecture are represented as grids (matrices). Kernels are the functions which are applied to each element of the stream (grid).
The stream- and kernel-based architecture imposes a set of rules, which need to be fulfilled by the non-rendering algorithms which are to be mapped onto the GPU. The strict adherence to these rules is not necessary for the mapping itself. However, not following them will likely result in performance decrease, instead of increase in comparison to the sequential execution on the CPU. It is therefore desirable that the problem which is to be mapped onto the GPU exposes the following characteristics:

- high arithmetic intensity
- lack of dependencies between the stream elements

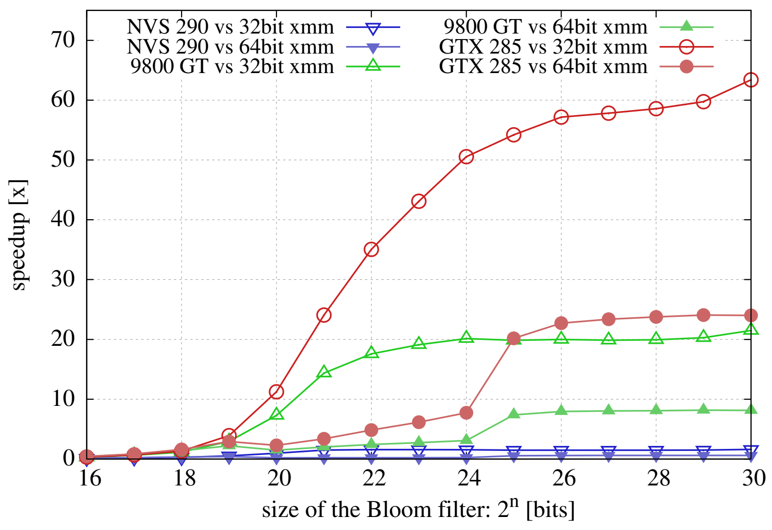


Figure 5.7: Speedup of Bloom filter operations using different CUDA devices

The high arithmetic intensity implies that algorithms which are mapped onto the GPU perform a large number of operations per transferred data word. In traditional rendering algorithms a single 8 bit word of data triggers at least 300 operations in the programmable fragment processors [FSH04, Han05]. The lack of dependencies between the stream elements implies that the kernels can exploit the parallelism of many cores working on the same data, without the need of blocking.
Figure 5.7 illustrates the speedup of the OR operation between two Bloom filters as a function of their size in bits. The speedup is calculated as the ratio of the Bloom filter OR operation speed using the 32 and 64 bit CPUs and different GPU devices. The `32bit xmm` and `64bit xmm` data series were obtained using the Intel(R) Celeron(R) 2.66GHz, 32 bit and the Intel(R) Core(TM)2 Quad Q9450 2.66GHz, 64bit CPUs, respectively. The OR operation in case of the both CPUs was performed in the XMM SIMD registers. The `NVS 290`, `9800 GT` and `GTX 285` data series represent different GPU devices with 16@0.9GHz, 112@1.5GHz and 240@1.47GHz cores, respectively.

5.6.1 `skiptree`

In order to use the GPGPU in content-based event forwarding the event forwarding algorithms must expose the characteristics presented in the previous section. The first step in that direction has already been performed: the content-based matching problem (with the help of the Bloom filters) has been transformed into a numerical algorithm – see Section 5.4. It is important to note that none of the contemporary GPGPU programming languages, like CUDA [NBGS08] or Brook [BFH+04], allows for an easy and straightforward mapping of prob-

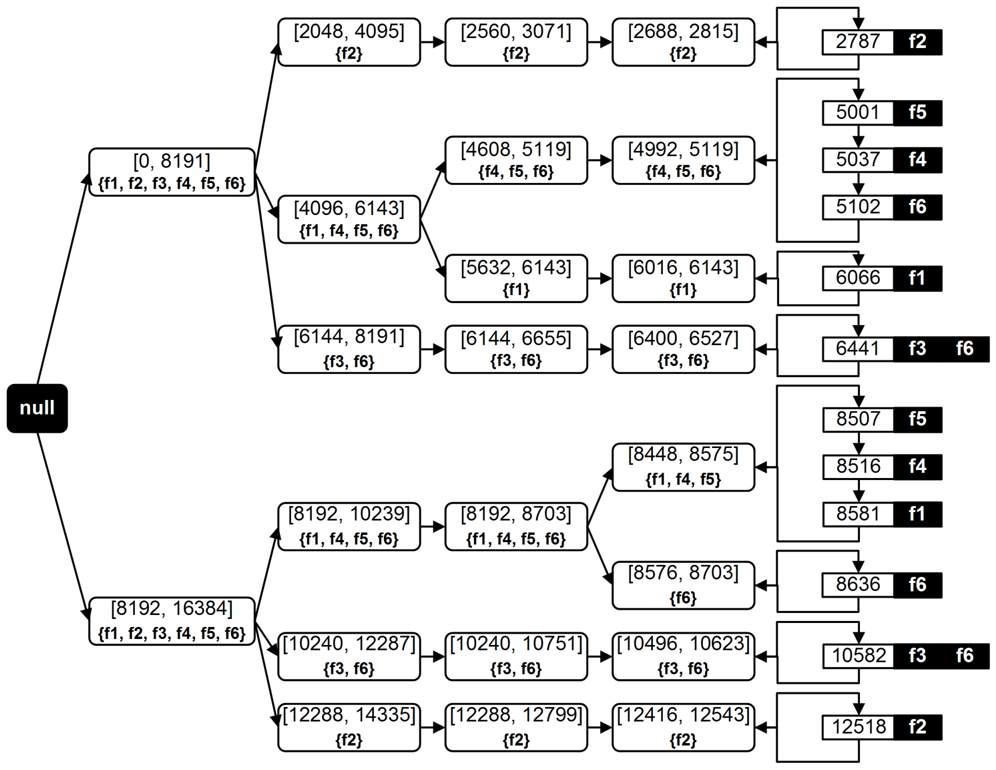


Figure 5.8: The `skiptree` containing filters from Table 5.1

lems like string comparison or recursion onto the GPGPU. Therefore, it is of crucial significance that the Bloom filter-based routing algorithms allow to transform the problem of content-based matching into a numerical, high arithmetic intensity algorithm, which can be executed in all but first broker on the path of the event.

However, even though the algorithms presented in Section 5.4 are not explicitly working with the content of events and subscriptions, they still expose a significant amount of data dependencies. Specifically, the presence of counters in the counting variant of the `sbstree` (see Figure 5.6) with the state dependent on the progress of the matching of the current event introduces additional overhead for the management of the routing structure. Moreover, it is not possible to start the matching of the new event until the previous one has been completely matched and the counters in the counting `sbstree` have been reset to their original state.

Therefore, this section proposes a new data structure – the `skiptree`. The `skiptree` is conceptually based on the design of a skip list [Pug90]. It follows two design considerations: it tries to achieve maximum event forwarding speed and it reduces to minimum the dependency and dynamism in the data structure. The `skiptree`, unlike skip list, can profit from the fact that the size of the data structure is bounded by the size of the underlying Bloom filter.

Figure 5.8 shows the `skiptree` after the insertion of the filters from the Table 5.1. It can be observed that the operations, similarly to the `sbstree` and counting `sbstree` are performed only on the bits of the Bloom filters. The `skiptree` maintains two data layers – the rightmost layer is an ordered linked list containing the bits of Bloom filters with the filter identifiers attached to every bit. Every bit stores the identifier of the filter (or filters, as in the case of bits 6441 and 10582) to which it belongs. For example bit 2787 belongs to the filter $f2$.
The left layer is a finger data structure which starting from the root node maintains fingers to the ranges of the bits. One can observe that starting from the root the fingers are getting more specific. The `skiptree` illustrated in Figure 5.8 was initialized with the minimum finger span of 128 and the finger span step of 4. This means that one finger at the lowest level spans exactly 128 bits and every higher level spans the previous level times four number of bits. Every finger contains the list of filters which are assigned to the Bloom filter bits within the range of the finger.
Since the size of the Bloom filter used in filters is known $m = 2^{14}$, the finger structure can be statically initialized during the `skiptree` construction. Subsequent additions of new Bloom filters into the `skiptree` result only in the updates to the rightmost linked list and the sets of filters stored by the fingers.

5.6.2 Filter routing and event forwarding

Filter routing using only the `skiptree` structure is possible only if the simple routing strategy is used. New filters are added to the `skiptree` on a per bit basis. New bit b to be inserted is compared with fingers starting from the level directly under the root node and upon match the comparison process is continued with the children of matching node, until the rightmost finger is found. The rightmost finger points to the first element of the linked list into which the new bit b is inserted. The filter identifier to which b belongs is attached to b and to the filter lists of all taken fingers. The above process is repeated for all remaining bits of the filter.
The forwarding of events is performed only based on the Bloom filter bf attached to the event. The goal of the matching process using the `skiptree` structure is the exclusion of filters which a given event does not match. This implies that at the start of the forwarding process it is assumed event e matches all filters and should be forwarded on all interfaces the broker **B** performing the matching:

$$\mathbb{I}_{e\rightarrow} = \mathbb{I}_{\mathbb{B}} \tag{5.7}$$

The matching process starts with the first bit b_1 of the event. The goal of the matching is to extract all bits (and their attached filters) which are between the bit number 0 and the first bit of the event b_1. In other words algorithm searches for all bits which are not satisfied by the event's Bloom filter. All unsatisfied bits imply filters which do not match an event. The search process is excelled by the use of fingers – algorithm tries to find a finger which is closest to the root and its range is enclosed by the exclusive range (0, b_1). Upon successful location of such finger its filter list is subtracted from the list of filters matching the event e.

Algorithm subsequently steps down the finger tree to fill up the whole exclusive range (0, b_1) using fingers. If the exclusive interval (0, b_1) is not completely covered by fingers and algorithm already searched the fingers at the lowest level than a binary search in the linked list is performed in order to find all bits (and filters) which are needed to fill the range. When the search is finished algorithm selects next bit b_2 from the event's Bloom filter. Starting at the last position b_1 it tries to fill the exclusive range (b_1, b_2) with the fingers and bits from the `skiptree`.

As an example the process of forwarding of the event $e \equiv \{x = 3\}$ can be considered. Event e after matching with the `sbsposet` presented in Figure 5.4 carries a Bloom filter $bf = \{2787, 12518\}$. The matching process of the event e with the `skiptree` starts with the exclusive range $(0, 2787)$. The algorithm starts at the root of the finger tree and tries to find the largest finger which is enclosed within the exclusive range $(0, 2787)$. The algorithm takes the top branch of the finger tree and descends it until it arrives at the lowest level finger – $[2688, 2815]$. However, since even this finger is not contained within the exclusive range $(0, 2787)$ algorithm needs to search the linked list pointed to by the finger in order to find all bits in the exclusive range $(0, 2787)$ present in the ordered linked list. Since none such bits are found (bit 2787 is outside of the range) algorithm proceeds by selecting the next bit 12518 from the event's Bloom filter and forming the new exclusive range: $(2787, 12518)$. The algorithm now walks up the tree to find the largest finger encompassed by the new exclusive range $(2787, 12518)$. The largest finger it finds is $[4096, 6143]$. It subtracts all filters from that finger from the set of filters (initially containing all filters – see Equation 5.7) matching event e. The set of filters matched by the event e contains now only two filters: $f2$ and $f3$. The algorithm continues in its search for other fingers encompassed by the exclusive range $(2787, 12518)$. It finds the fingers $[6143, 8191]$ (and removes filters $f3$ and $f6$ from the event matching set of filters) as well as $[6143, 8191]$, $[8192, 10239]$ and $[10240, 12287]$. The algorithm steps down along the last finger $[12288, 14335]$ until it reaches the linked list again. The result of the matching is the set of filters matching the event e containing only the filter $f2$ – as the only filter not removed. The event e is subsequently forwarded on the reverse address of the filter $f2$.

5.6.3 Discussion

The algorithms presented in this section are still under development. While on one hand side the combination of the Bloom filter-based routing and `skiptree` algorithm allows to reduce the problem of the content-based matching to the problem of numerical algorithm, there are still issues which prevent the straightforward implementation in the CUDA or Brook environments. One of the main issues is the need for dynamic data structures in the `skiptree` – the list of skip filters associated with the single bits and the list of skip filters associated with the fingers. Such dynamic data structures are expensive to maintain in the highly parallel programming environments and their optimization is the focus of the further research in this area.

5.7 Related Work

The related work for the fast event forwarding in content-based systems has been already partially covered in the Section 4.5. Therefore, this section focuses on the specific issue of fast event forwarding and filter routing using content summaries. In order to structure the comparison the related work has been divided into two groups: first group tries to tackle the problem of fast event forwarding and filter routing by altering the routing process and routing structures, while the second group aims at implementing a routing overlay which reorganizes the broker overlay, so as to exploit the filter similarities.

One related set of publications belonging to the first group is characterized by the following works: [TE02, TE04, AT05, AT07]. The authors propose to use Bloom filters to represent per broker, numeric filter summaries compacting filter information. The per broker filter summaries are subsequently exchanged by the brokers so that incoming events are matched against the summaries instead of the full content-based matching. The algorithm proposed by the authors assumes that it is possible to numerically encode events so that the numeric event representation can be matched with a numeric filter representation. This in turn requires that *the number of predicates in the system is predefined, as well as the specification of these predicates (attribute name and attribute constraint), and that the set of supported predicates is ordered and known for each broker*. The Bloom filter-based XSiena system does not make such assumptions, instead filter predicates are treated as black box entities, with external interface allowing for the computation of the coverage relation and the computation of the matching between the predicates and events.

A similar approach (based on [TE02]) to efficient content-based addressing in the environment of the mobile ad-hoc networks has been presented in [YB04]. The authors combine Bloom filter-based filter summaries and an on-demand, multicast routing protocol. However, due to the assumptions shared with the [TE02], resulting limitations concerning the expressiveness of the system are also similar.

The second group of publications concerned with the alteration of the broker network is best characterized by [CF05]. Authors propose a publish/subscribe system based on the peer-to-peer substrate with minimal overhead routing algorithm in which every peer subscribes to events it produces. Moreover, every peer forwards events to its neighbors only if the events match its own interest. Thus messages are effectively flooded within the community of interest. In order to minimize the false positives and false negatives authors organize peers based on their interest. Authors propose two methods: coverage-based organization of peers and similarity-based organization of peers. However, both approaches result in a relatively high overlay management overhead, with the second approach resulting in possible false negatives a condition not possible in the Bloom filter-based XSiena system. The first approach exposes also high number of false positives, especially for smaller networks ($N \leqslant 20000$ nodes) – a condition easily avoided in the Bloom filter-based XSiena system using sparse Bloom filter implementation – cf. Section 5.2.3. A similar approach has been proposed in [BBQV07] where authors propose algorithms which re-organize the broker spanning tree overlay according to the similarity of the brokers' interests.

Another class of publish/subscribe systems in the second group are those based on the structured overlays [TZWK07, Pie04]. Although DHT-based approaches usually expose a superior matching times they require typed filters and events (thus limiting the flexibility of the content-based publish/subscribe). Moreover, DHT-based systems require globally unique node ids ands additional mechanisms to account for the locality of the events. It is also important to recall (see Section 2.5.7) that peer-to-peer-based solutions always expose an additional overhead associated with the stretch of the underlying DHT.

Chapter 6

Fail-Awareness

For many critical applications, e.g., Critical Infrastructure Protection [Che05] (CIP) it is essential to be aware whether:

- one has received all information one was supposed to get
- the information that was received is timely, i.e., it is not older than some application-specific or context-specific threshold

The critical applications include, among others, transportation [EAP99], electric grid related systems [TBVB05] or embedded systems [KM99]. For example, one can recall the events of August 14, 2003 blackout in US and Canada. It has been stated [U.S04] that one of the major causes for the blackout was the fact that none of the system operators noticed failed components which in turn resulted in missed alarms and lack of adequate action. This clearly demonstrates that without being able to determine whether all information has been received, it is not possible to undertake appropriate actions. This in turn can result in a failure propagation and transformation [Wal05].
Moreover, it has been stated that second major cause for the August 2003 blackout was the lack of appropriate, timely actions due to the slow information exchange between operators. This becomes clear if one considers the fact that *when telemetry or electronic communications fail, some essential data values have to be manually entered into SCADA* [wide area electric grid control] *systems, state estimators, energy scheduling and accounting software, and contingency analysis systems* [U.S04]. This proves that management of complex wide area systems requires timely communication which in turn is only possible if a notion of timeliness is provided. Intuitively, timeliness implies the ability to make deterministic judgments about the information propagation time.
This chapter presents a fail-aware XSiena system which permits to compute upper bound on the age of received information and allows to determine the completeness of the received data. Specifically, the ability to calculate the upper bound on the age of publication, advertisement and subscription messages in

the XSiena system is introduced. The age (propagation delay) of the message is defined as the time between the message creation at the source host and the time message is delivered to the application at the destination host.

The fail-aware XSiena system introduced in this chapter does not require any prior information regarding the diameter of the content-based network nor any a priori known bounds on the message transmission delays. Fail-aware XSiena is fully decoupled with respect to space and synchronization. Moreover, the fail-aware XSiena system does not require any kind of clock synchronization (neither internal nor external) among the nodes of the content-based network.

There are several factors which speak for the application of the content-based systems in the context of the CIP. One of them is the fact that CIP applications and systems are typically geographically distributed. Therefore, they require a Wide Area Network (WAN) to collect and manage information from different administrative domains. This implies the need to support wide area, distributed control. Even in areas where its use has been so far very limited (e.g. electric grid control), the demand for the introduction of content-based solutions is clearly increasing [GDB+03]. Therefore, providing timeliness and fail-awareness for WANs is certainly within the scope of the content-based systems.

Simultaneously, Wide Area Networks pose a challenging environment. Due to their size (both in terms of geographical distribution and the number of nodes), WANs often suffer from failures, including, among others, delays and information loss. One reason for this are partitions and routing anomalies – instances where live nodes are not able to route packets to each other. For example, measurements have shown that an average wide area distributed system (PlanetLab [Ros05]) consisting of 280 nodes partitions at least once a day [MPHD06]. Moreover, within a ten day period, at least one node in a 192 node system suffers from a routing anomaly [MPHD06].

The choice to use the content-based XSiena system has been made because a traditional, point to point schemes with explicit information source and information sink suffer from a lack of flexibility in the presence of failures and restrict the reconfigurability of the system. Therefore, it will be demonstrated that a content-based network poses a promising alternative as it is well adapted to the loosely coupled nature of distributed interaction in wide area systems [EFGK03].

A simple approach to implement such a system would be to use external clock synchronization. Having all clocks synchronized with respect to some external time source allows to measure the message transmission time by calculating the difference between the reception time stamp at the receiver and sending time stamp at the sender. A *de facto* standard to provide such an external clock synchronization is NTP [Mil92, Mil06]. However, in this chapter it will be demonstrated that NTP does not provide any guarantees as to the results it delivers. The transmission time measured using NTP and the

real-time results always differ. Specifically, it is not possible to guarantee that the received message is not late because the transmission time computed using NTP might be lower than the actual transmission time. In this chapter it will be demonstrated that the proposed approach does not suffer from this problem. Moreover, the usage of the NTP is not always feasible due, to, e.g., security restrictions of the given system or the lack of the Internet access.
The remainder of this chapter is structured as follows: Section 6.1 defines the fail-awareness in context of the distributed systems. Section 6.2 presents the theory behind the calculation of the upper bound on the message transmission delays. Section 6.3 presents the application of the upper bound on the transmission delay in the context of the content-based publish/subscribe systems. The chapter is concluded with an overview of the related work in Section 6.4.

6.1 Definition

Fail-awareness provides an indicator allowing to implement services in distributed systems with uncertain communication and access to local hardware clocks only. The fail-aware indicator tells whether some safety property currently holds or if it might be violated [FC99b]. The fail-aware XSiena system is based on the by the Timed Asynchronous Distributed System Model (TADSM) [CF99], i.e., the communications channels expose the omission/performance failure semantics, while processes have crash performance failure semantics – see Chapter 3.
Intuitively, a fail-aware system is a system which is able to detect when it is not possible to depend upon properties provided by lower level services, e.g, to depend upon the provided information because its transmission time violates the predefined Δ_{max} threshold [FC03]. The TADSM model is suitably weak to represent the contemporary distributed systems – assumptions it makes on the type and frequency of failures, communication infrastructure and time flow allow it to be implemented in a wide range of existing systems without the need to introduce any extra hardware and without the need to alter the existing system's interconnections. This chapter demonstrates that it is also sufficiently strong to solve the problem of failure detection in a content-based system.

6.2 The Upper Bound on Transmission Delay

The goal of the fail awareness in the content-based XSiena system is to provide an indicator, whether is it not possible to rely on the safety property of the system. The safety property of the fail-aware XSiena system is defined as the ability to estimate whether messages received by the components of the system are *late* or *timely*. A message *m* is late if its transmission delay is greater than a predefined value Δ_{max}. It is timely otherwise. It is only possible

to detect a message that is late if it is possible to calculate the transmission delay of that message. However, it is not possible to calculate the transmission delay by subtracting send and receive time stamps in a distributed system with unsynchronized clocks.
Therefore, in order to provide fail-awareness in the content-based XSiena system, it is proposed to calculate an upper bound ub(m) on the transmission delay td(m) of the message m between the source of the message m and the destination of the message m. The upper bound on the message m transmission delay states that the calculated upper bound value is always higher than the actual transmission delay experienced by the message m. Specifically, it is possible that the actual transmission delay is lower than the calculated upper bound:

$$\mathrm{td}(m) \leqslant \mathrm{ub}(m) \tag{6.1}$$

Therefore, the upper bound can be seen as a conservative estimation of the transmission delay of a given message. Following the above reasoning it can be said that message m is late if the upper bound on its transmission delay is greater than a predefined value $\Delta_{\max}$ and that it is timely otherwise:

$$\mathrm{ub}(m) \leqslant \Delta_{\max} \Rightarrow m \text{ is timely} \tag{6.2}$$

$$\mathrm{ub}(m) > \Delta_{\max} \Rightarrow m \text{ is late} \tag{6.3}$$

In content-based publish/subscribe system there might exist multiple receivers for a message m and they might experience different transmission delays for m. To simplify the description, it is assumed that the transmission delay and the upper bound is always specific to some implicitly or explicitly given receiver. Computing an upper bound on the message transmission delay for a fail-aware content-based XSiena system must not require external clock synchronization. A method to provide such an upper bound has been presented in [FC99b] and subsequently improved in [CMRV01]. In what follows a brief outline of the proposed approach will be given. It will be also further extended so that it can be used in the context of the XSiena content-based system.
To compute an upper bound $ub(m)$ on a transmission delay of the message m between two connected processes p and q (see Figure 6.1) each of them requires a monotonic local clock with a bounded drift rate ρ with respect to real time – see Chapter 3.1. A drift rate of process q is bounded by ρ_q and the drift rate of process p is bounded by ρ_p.
The intuition behind the upper bound method is that process q sends a helper message n to process p at time A – measured with its local hardware clock. Subsequently, p sends a message m to q which receives it at time D indicated by q's local clock. The transmission delay td(m) of message m is bounded by the real-time period between D and A. The error due to q's clock drift equals $(D - A)\rho_q$. Hence:

$$\mathrm{td}(m) \leqslant (D - A)(1 + \rho_q) \tag{6.4}$$

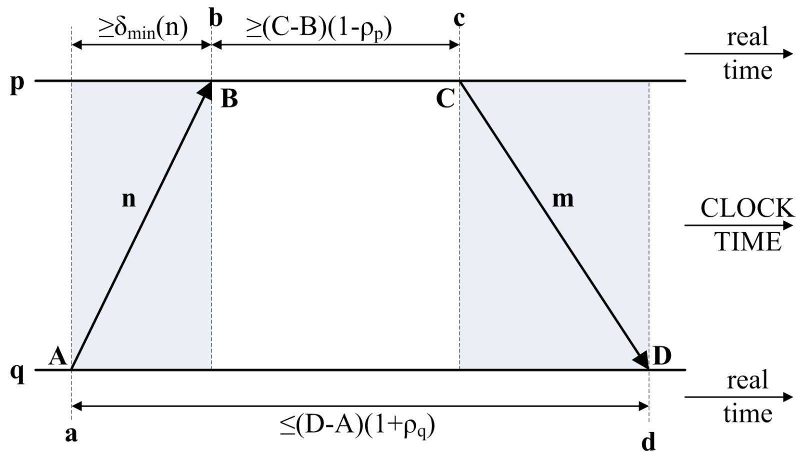


Figure 6.1: Calculating an upper bound on the transmission delay $td(m) = d-c$.

The upper bound calculation can be improved by subtracting the processing time at process p. This time can be calculated by p as $(C - B)$. The error due to p's clock drift equals $(C - B)\rho_p$. Hence:

$$\mathrm{td}(m) \leqslant (D - A)(1 + \rho_q) - (C - B)(1 - \rho_p) \tag{6.5}$$

In order to guarantee that the message transmission delay is indeed not higher than $\mathrm{ub}(m)$, a lower bound on the message processing time at node p is calculated using the factor $(1 - \rho_p)$.
A further improvement can be achieved if one is able to calculate a lower bound $\mathrm{lb}(n)$ on the transmission delay of the helper message n, i.e., $\mathrm{lb}(m) \leq \mathrm{td}(m)$. Knowing the bandwidth of the network (*bndwdth*) connecting processes p and q and the size ($\mathrm{size}(n)$) of the helper message n, a lower bound on the transmission delay of the helper message n can be set to $\delta_{\min}(n) = \dfrac{\mathrm{size}(n)}{bndwdth}$.
Hence, the upper bound $\mathrm{ub}(m)$ of a message m can be defined as:

$$\mathrm{ub}(m) = (D - A)(1 + \rho_q) - (C - B)(1 - \rho_p) - \delta_{\min}(n) \tag{6.6}$$

An error $e_{\mathrm{ub}}(m)$ on the upper bound $\mathrm{ub}(m)$ of the transmission delay $\mathrm{td}(m)$ of the message m is defined as the difference between the calculated upper bound and the real time transmission delay, i.e.: $e_{\mathrm{ub}} = \mathrm{ub}(m) - \mathrm{td}(m)$ – see Figure 6.1. It can be seen from Equation 6.6 that with the increase of the transmission delay $\mathrm{td}(n)$ (more precisely, with the increase of the uncertainty $\mathrm{td}(n) - \delta_{\min}(n)$) the upper bound $\mathrm{ub}(m)$ and the error value $e_{\mathrm{ub}}(m)$ will increase, as well. To cope with this problem, a helper message n with a smaller uncertainty $\mathrm{td}(n) - \delta_{\min}(n)$ is needed. This principle is illustrated on Figure 6.2. Instead of using more recent helper n, one can optimize $\mathrm{ub}(m)$ by using an older helper message n' with smaller uncertainty. Specifically, process p upon reception of each helper message determines whether the current one is characterized by a smaller

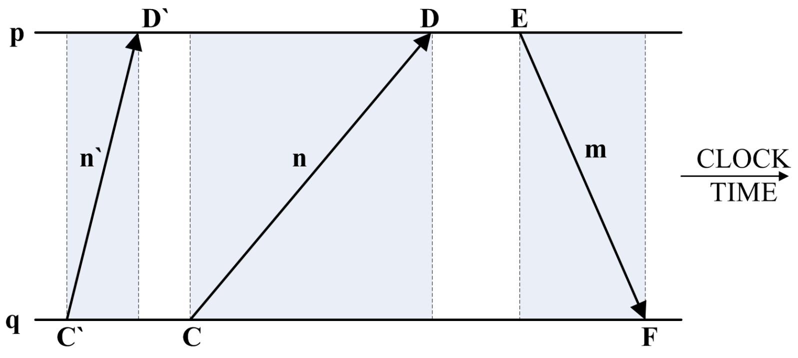


Figure 6.2: Using faster helper message to improve upper bound calculation

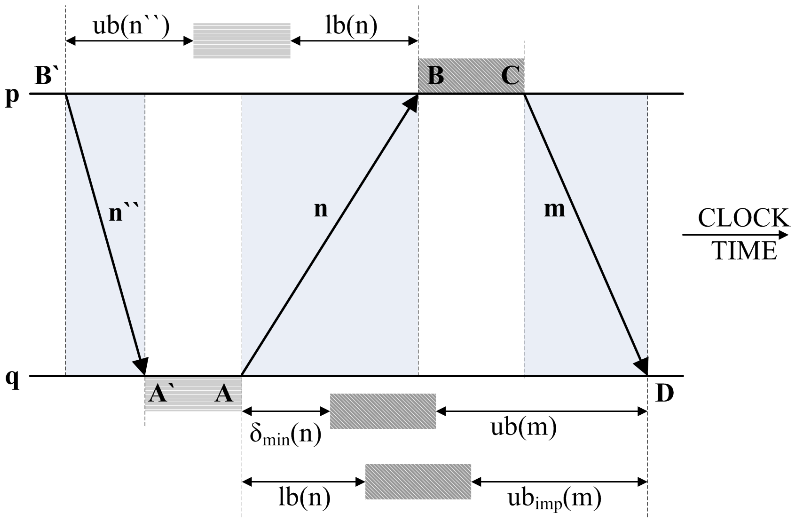


Figure 6.3: An improved upper bound (it is assumed that $\rho = 0$)

uncertainty than the old helper messages it already has received. The criterion for choosing n' in favor of n can be expressed as choosing a helper message which provides a better (lower) upper bound:

$$\begin{aligned}&[(F - C')(1 + \rho_q) - (E - D')(1 - \rho_p) - \delta_{\min}(n')]\\ &\qquad < [(F - C)(1 + \rho_q) - (E - D)(1 - \rho_p) - \delta_{\min}(n)]\end{aligned} \tag{6.7}$$

Which results in:

$$(C - C')(1 + \rho_q) - \delta_{\min}(n') < (D - D')(1 - \rho_p) - \delta_{\min}(n) \tag{6.8}$$

So far a simple definition of $\delta_{\min}(n)$ has been used, assuming it to be the lower bound for the helper message n. In [CMRV01] it is proposed to calculate a lower bound which can be greater than $\delta_{\min}(n)$. In following it will be shown how to use this improved lower bound to improve the upper bound.

Figure 6.3 illustrates a pair of messages n'' and n. The intuition behind the improved lower bound method is that when the lower bound $\mathrm{lb}(n)$ of helper message n is greater than $\delta_{\min}(n)$, the improved upper bound $\mathrm{ub}_{\mathrm{imp}}(m)$ is smaller than the original $\mathrm{ub}(m)$.
The lower bound on the transmission delay of message n represented as $\mathrm{lb}(n)$ can be calculated as:

$$\mathrm{lb}(n) = (B - B')(1 - \rho_p) - (A - A')(1 + \rho_q) - ub(n'') \tag{6.9}$$

It is possible that the calculated lower bound $\mathrm{lb}(n)$ is smaller than the $\delta_{\min}(n)$. Specifically, it can be less than zero. Therefore, an improved lower bound $\mathrm{lb}_{\mathrm{imp}}(n)$ can be defined as:

$$\mathrm{lb}_{\mathrm{imp}}(n) = \begin{cases} \delta_{\min}(n) & \text{if} \quad \delta_{\min}(n) \geqslant \mathrm{lb}(n) \\ \mathrm{lb}(n) & \text{if} \quad \delta_{\min}(n) < \mathrm{lb}(n) \end{cases} \tag{6.10}$$

Process p can subsequently attach the calculated value $\mathrm{lb}_{\mathrm{imp}}(n)$ on to the message m it sends to the process q – see Figure 6.3. Process q knowing the value of $\mathrm{lb}_{\mathrm{imp}}(n)$ which it has received with the message m, can calculate an improved upper bound $\mathrm{ub}_{\mathrm{imp}}(m)$:

$$\begin{aligned} procT &= (D - A)(1 + \rho_q) - (C - B)(1 - \rho_p) \\ \mathrm{ub}_{\mathrm{imp}}(m) &= procT - \mathrm{lb}_{\mathrm{imp}}(n) \end{aligned} \tag{6.11}$$

The ability to calculate the lower bound on a helper message influences the choice of the faster helper message presented in Equation 6.8. Using the improved lower bound $\mathrm{lb}_{\mathrm{imp}}(n)$ the criterion in Equation 6.8 can be rewritten as:

$$(C - C')(1 + \rho_q) - \mathrm{lb}_{\mathrm{imp}}(n') < (D - D')(1 - \rho_p) - \mathrm{lb}_{\mathrm{imp}}(n) \tag{6.12}$$

where $\mathrm{lb}_{\mathrm{imp}}(n')$ and $\mathrm{lb}_{\mathrm{imp}}(n)$ are the improved lower bounds on helper messages n' and n – see Figure 6.2.
It is important to note that the upper bound computation requires only local clocks with a bounded drift rate. Specifically, this means that it is possible to use a common time stamp counter (available on most modern CPUs) which typically has a bounded drift rate. For example, a High Precision Event Timer is supported on many systems and this timer has a specified maximum drift rate of 500ppm for intervals over 1 millisecond [Cor04].

6.3 Fail-Aware Publish/Subscribe

In order to achieve fail-awareness in a content-based publish/subscribe system, it is necessary to be able to detect and signal when it is not possible to depend

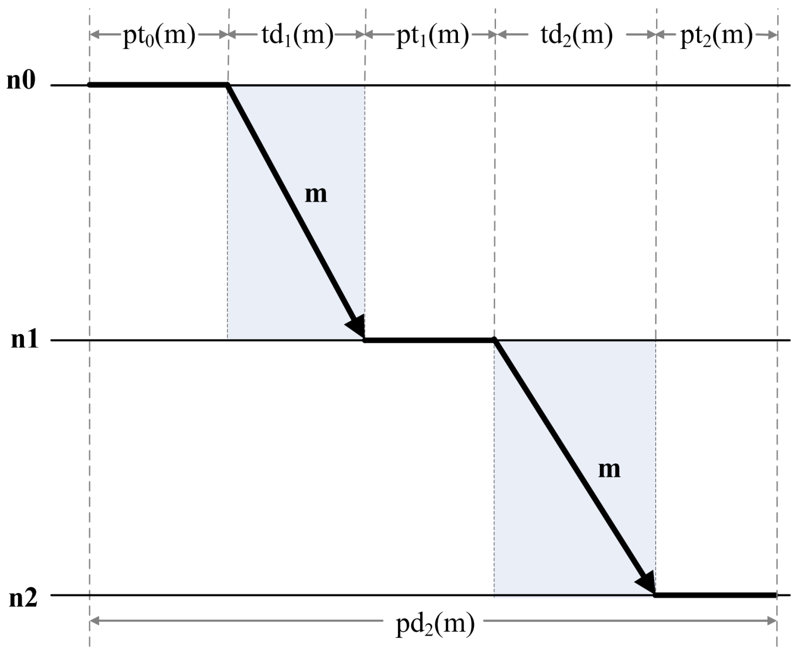


Figure 6.4: The components of the propagation delay for a three host network

upon real-time properties of lower level services due to unmasked failures. A fail-aware content-based system uses the upper bound on the transmission delay ub(m) to detect failures with respect to the timeliness of message transmission delay. It has been shown in Chapter 6.2 that such detection is possible when two processes communicate with each other. However, when recalling the architecture of a content-based XSiena system presented in Section 3.3, it becomes clear that the upper bound on the transmission delay alone is not sufficient for the timeliness detection.

Timeliness properties need to be preserved between the sender of the message and its receiver. This in turn implies that an upper bound on the transmission delay needs to be computed for messages which pass multiple hosts on their way from sender to receiver in the XSiena system. Moreover, neither sender nor the receiver know the path taken by the message for which the timeliness property should be evaluated.

The calculation of the timeliness of the message transmission delay between multiple hosts requires the definition of the propagation delay pd(m) of the message m. Propagation delay of the message m can be defined as a sum of the transmission delay td(m) and the processing time pt(m) of the message m. The transmission delay is the actual time spent by the message in transit between two nodes. Processing time is the time spent by the message at the given node from its reception until its dispatching. Specifically, if the current node is the nth node encountered by the message m, then the propagation delay $\mathrm{pd}_n(m)$ of the message m calculated by that node equals:

$$\mathrm{pd}_n(m) = \sum_{i=1}^{i=n} \mathrm{td}_i(m) + \sum_{i=0}^{i=n} \mathrm{pt}_i(m) \tag{6.13}$$

where $\mathrm{pt}_0(m)$ is the time between the scheduled creation of the message m and its dispatching at the producing node, $\mathrm{pt}_n(m)$ is the time between the reception of the message m at the destination node and the placement of the message m in the routing table of the destination node and $\mathrm{pt}_i(m)$ for $i \in \{1, \dots, n-1\}$ is the time between the reception of the message m and its dispatch at the node i. The transmission delay $\mathrm{td}_i(m)$ is the transmission delay between nodes $i-1$ and i – see Figure 6.4.
Applying the techniques presented in Chapter 6.2 to the Equation 6.13 one can calculate the upper bound on the propagation delay of the message m. The upper bound on the propagation delay $\mathrm{ub}(\mathrm{pd}(m))$ determines the upper bound on the total age of the message m since its creation. In order to calculate the upper bound on the propagation delay of the message m the $\mathrm{td}_i(m)$ is replaced with the corresponding upper bound $\mathrm{ub}_i(m)$ on the transmission delay – see Equation 6.6. The upper bound on the local processing time $\mathrm{ub}(\mathrm{pt}_i(m))$ of the message m, can be calculated by every node using its own local hardware clock. The calculation is performed by including the compensation for the hardware clock drift rate $(1 + \rho_i)$:

$$\mathrm{ub}(\mathrm{pt}_i(m)) = (1 + \rho_i)\mathrm{pt}_i(m) \tag{6.14}$$

Combining Equations 6.6, 6.13 and 6.14 one can calculate the upper bound on the propagation delay of the message m as:

$$\mathrm{ub}(\mathrm{pd}_n(m)) = \sum_{i=1}^{i=n} \mathrm{ub}_i(m) + \sum_{i=0}^{i=n} \mathrm{ub}(\mathrm{pt}_i(m)) \tag{6.15}$$

6.4 Related Work

Despite the broad range of publish/subscribe systems only a few provide guarantees related to the delivery of messages or their timeliness in a Wide Area Network environment. The approach presented in [ZSB04] allows for an in order, gapless delivery of information. The presented approach relies on the algorithm which explicits the set of filters a message should be matched against and a set of filters which is used by the broker to perform the matching. In case a broker contains only a subset of filters which are required to match the message, the message is flooded without the filtering taking place. The algorithm trades-off the flooding approach for the consistency maintenance with respect to the routing tables of the brokers – cf. Section 4.4. The fail-aware approach differs significantly from the above work in that the fail-aware

XSiena system is agnostic to the underlying routing mechanism and that it aims at providing the timeliness information regarding the events in the system.
Author in [Fet98] presents a topic-based publish/subscribe system with the ability to detect if all messages that are at least some $\Delta_{\max}$ seconds old where received. Fail-aware XSiena system extends the approach proposed in [Fet98] to work with a content-based publish/subscribe system and alleviates the requirements for each process having to calculate an upper bound $\epsilon(T)$ on the current deviation of its clock from real-time. Specifically, the fail-aware XSiena system relies solely on the local clocks of the processes without any reference to the real-time.
There exist other publish/subscribe systems providing real-time properties, e.g., [PC05a], however, they require dedicated hardware and use closed source software, so it is neither possible to evaluate their functioning nor feasible to use them with the existing infrastructure, e.g., the Internet). Similarly, [GDB+03] requires the use of dedicated infrastructure and externally synchronized clocks.

Chapter 7

Soft State

The specific environment of distributed systems, with the high probability of communication and node failures [MPHD06, WSH08], implies the need for an approach which could be used to build survivable content-based systems. A survivable content-based system continues to provide service despite transient components and links failures [KSS03, ALRL04].
This chapter presents a soft state [Cla88, RM99], content-based XSIENA system based on the fail-aware approach presented in Chapter 6. Following [Cla88] soft state can be defined as a system state which can be lost in a crash without *permanent* disruption of system features. Specifically, soft state implies the survivability of the system in the face of failures. The proposed soft state-based design allows the XSIENA content-based system to recover from node, link and timing failures so that a correct system state is never permanently lost. The soft state approach is used for the creation and maintenance of the routing state stored in the routing tables of the XSIENA system. The soft state XSIENA system is deployed in the Wide Area Network environment of the PlanetLab.
In the soft state approach both publishers and subscribers determine the validity (lease time) of periodically published advertisement and subscription messages. Advertisements and filters need to be refreshed by the publishers and subscribers before their lease time expires, otherwise they are removed by the brokers from their routing tables. The lease-based approach does not restrict the expressiveness of advertisements and subscriptions, including the coverage-based routing optimizations.
In order to adaptively calculate the validity of subscription and advertisement messages the fail-aware XSIENA system is used. Specifically, no prior information regarding the diameter of the publish/subscribe network, nor any a priori known bounds on the message transmission delays are required. The proposed soft state approach is fully decoupled with respect to space and synchronization and lightweight in that it does not require any kind of clock synchronization (global clock) among the nodes of the publish/subscribe network. In the soft state XSIENA system both periodic re-advertisement and re-subscription messages are idempotent with respect to the correct system state, thus ensuring the

soft-state properties of the content-based system.
An interesting property of the soft state XSiena system is that it automatically handles the effects of subscribers' and publishers' failures and unannounced departures [MÖ2]. In traditional, hard state content-based systems such failures would result in the pollution of the brokers' routing tables. In hard state systems filters and advertisements are stored by the brokers in their routing tables until a matching unsubscription or unadvertisement is received. If a subscriber or a publisher which issued a subscription or an advertisement crashes, neither a matching unsubscription nor a matching unadvertisement is issued. This results in a broker routing table indefinitely holding entries for inactive subscribing and publishing nodes. Soft state approach allows for the automatic expiration of such entries, thus alleviating the need for explicit unsubscriptions and unadvertisements. This in turn results in a more flexible system, as the departures of subscribers and publishers are lightweight in that no messages need to be exchanged upon such event.
The remainder of this chapter is structured as follows: Section 7.1 outlines the issues related to message propagation delays and link failures in the hard-state publish/subscribe systems. Section 7.2 presents the soft state approach towards solving of the problems presented in previous section along with the detailed analysis of implications of the soft state publish/subscribe system. Section 7.3 discusses the implications of the Timed Asynchronous Distributed System Model onto the safety and liveness properties of the soft state XSiena system. Section 7.4 presents the practical aspects related to the implementation of the soft state XSiena system including the choice of the API, implementation of the clocks and the routing of unsubscriptions and unadvertisements. This chapter is concluded with the overview of the related work in Section 7.5.

7.1 Motivation

The goal of the proposed soft state XSiena system is the creation and the maintenance of the soft logical routing state. The logical routing state is created using advertisements and filters and it is stored in the routing tables of publish/subscribe brokers. Routing tables, in turn, form advertisement and subscription trees which are the foundation upon which the actual information exchange takes place – see Sections 2.5.3 and 2.5.6. The lack of or incomplete advertisement and filter trees imply that nodes are not able to forward events and thus it is not possible for the publishers and subscribers to communicate. It is therefore of vital importance for the survivability of the publish/subscribe system to ensure the eventually proper establishment of filter and advertisement routing tables.
Figure 7.1 illustrates the possible issues when a simple publish/subscribe network is considered. Publishers **P1** and **P2** issue advertisements $a1$ and $a2$ at 12:00, real-time. Since the propagation delays between the publishers **P1** and

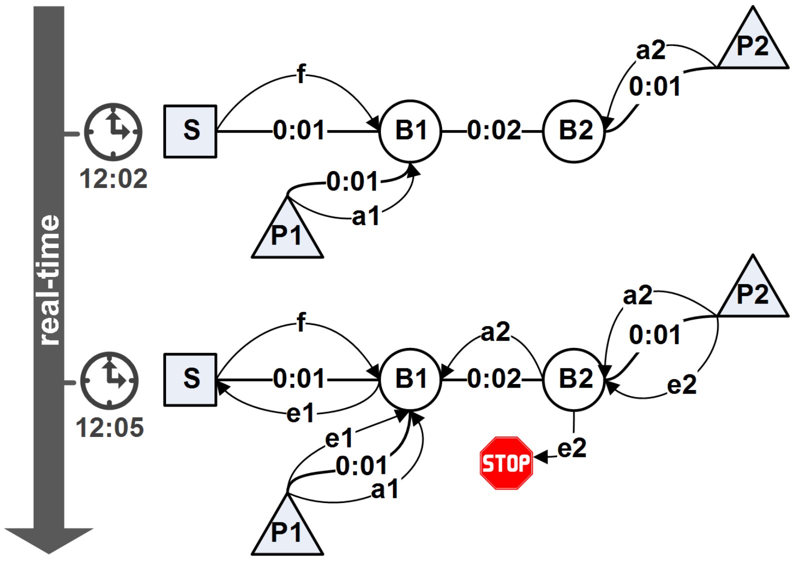


Figure 7.1: Subscription propagation issues due to timing relations with advertisements

P2 and their connecting brokers **B1** and **B2** equal 0:01 both advertisements arrive at respective brokers at 12:01, real-time. Simultaneously, at 12:01, subscriber **S** issues a subscription message containing the filter $f1$ matching both advertisements $a1$ and $a2$. Filter f arrives at the broker **B1** at 12:02, real-time – see the upper part of the Figure 7.1. Simultaneously, the advertisement $a1$ from publisher **P1** has been already delivered to the broker **B1**. However, due to a larger propagation delay between brokers **B1** and **B2** (equal to 0:02) the advertisement $a2$ from the publisher **P2** did not reach the broker **B1** yet. As a result filter f will only be visible for the publisher **P1**. Specifically, the subscription message containing the filter f will not be forwarded to broker **B2** as the matching advertisement $a2$ has not been yet delivered to the broker **B1**. Moreover, due to the decoupling properties of publish/subscribe systems, the subscriber **S** will never be able to tell that it is missing publications matching its filter f and issued by the publisher **P2** – see the lower part of Figure 7.1.
Even if the broker **B1** forwards the filter f on the reverse path of the advertisement $a2$ a subsequent transient failure on the link between the brokers **B1** and **B2** can lead to the loss of the subscription message containing the filter f. This in turn, again, results in the publication messages from the publisher **P2** never being delivered to the subscriber **S**.

7.2 Routing State Validity

The main issue in designing a soft state content-based system is the determination of the validity T an the re-issue period τ of subscription and advertisement

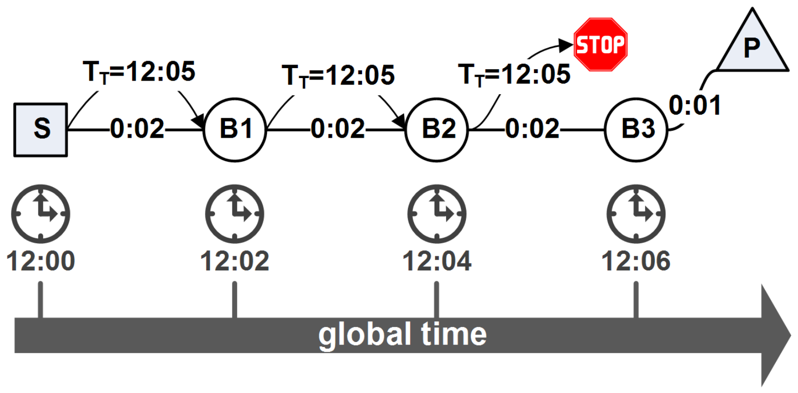


Figure 7.2: Propagation of a subscription using the validity time approach

messages. The validity T determines how long should a filter or advertisement be leased to the node of the soft state XSiena system. The re-issue period τ specifies how often the owing application tries to refresh the leased filter or advertisement. The validity T of a filter or advertisement can be expressed either as a validity time T_T or validity interval T_I.

7.2.1 Validity Time

A validity time T_T describes a point in time when a given filter should expire. In order to be meaningful to all nodes in the content-based system the validity time must be expressed with respect to a synchronized clock. A synchronized clock is shared by all nodes in the system and ensures that every node has the same view of the time – called global time.
An exemplary T_T value could read 12:05, meaning that the message associated with it should expire on every node at 12:05, global time. Assuming all processes in the system have their clocks synchronized within a given imprecision Φ_{max}[1], such message would indeed expire on all nodes within [12:05 – Φ_{max}, 12:05 + Φ_{max}]. However, the use of the validity time is difficult due to two issues: (1) it is difficult to provide a low imprecision, internal clock synchronization in large distributed systems and (2) message propagation delays significantly influence the validity time.
To better illustrate the latter case one can consider the scenario illustrated in Figure 7.2. For the clarity of presentation it is assumed that the imprecision Φ_{max} of the clock synchronization equals zero, i.e., all clocks are perfectly synchronized with respect to the real-time. The subscriber **S** subscribes at 12:00 (global time) with a filter which has the validity time T_T set to 12:05 (global time). The subscription message propagation delay between subscriber **S** and the next broker **B1** equals 0:02, which implies that the subscription message

[1]the imprecision Φ_{max} is defined as the maximum difference between the synchronized clocks of all processes

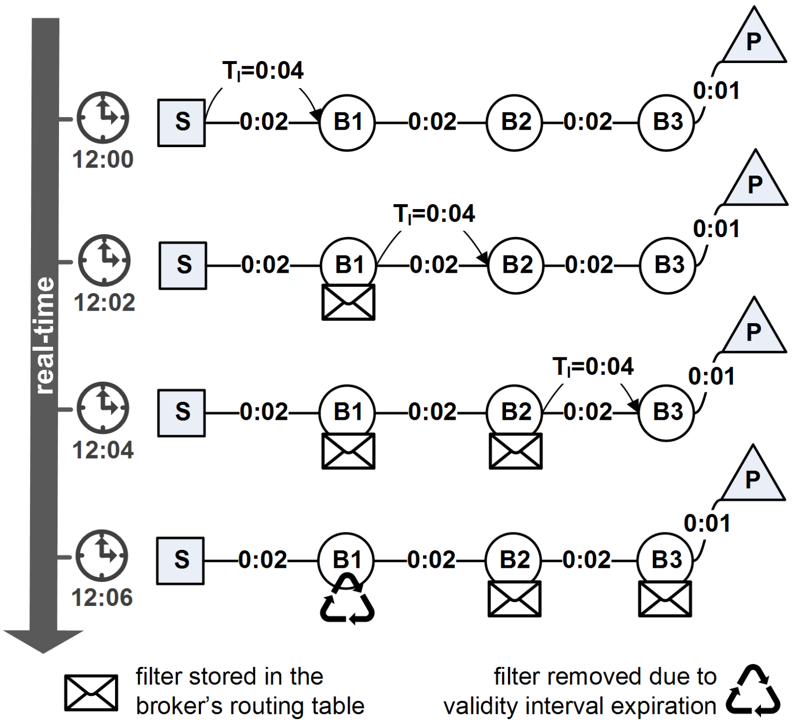

real-time
12:00
12:02
12:04
12:06
S
B1
B2
B3
P
0:02
0:01
T_l=0:04
filter stored in the broker's routing table
filter removed due to validity interval expiration

Figure 7.3: Propagation of a filter using the validity interval approach

sent by **S** will arrive on broker **B1** at 12:02, global time. Analogously, the next broker **B2** will receive the subscription message at the 12:04, global time. The last broker **B3** cannot accept the subscription message as upon its reception at the 12:06, global time, the filter carried by the subscription message will be already expired – as 12:05 < 12:06. As a result events published by the publisher **P** will never reach subscriber **S**.

From the above one can observe that the straightforward validity time approach is not applicable to large networks (with many brokers connecting publishers and subscribers) or to networks with varying transmission delays – see Figure 3.2. The validity time approach implicitly requires the knowledge of the network diameter in terms of latency, which is clearly not practical for decoupled publish/subscribe systems. Moreover, due to the N-to-M communication type of the content-based publish/subscribe systems a single advertisement message is delivered to multiple brokers with varying latency distances from the issuing publisher. This in turn can easily result in part of the brokers being overwhelmed with refresh messages arriving well before the expiration of the validity time and another part of the brokers not receiving the refresh messages on time, due to their large latency distance from the publisher.

7.2.2 Validity Interval

Based on the observations in previous section, the validity interval approach will be taken under further consideration. The validity interval T_I does not rely on the synchronized clocks. Instead, every process in the publish/subscribe network calculates the validity interval attached to a filter or an advertisement independently. The validity interval specifies the amount of time each broker should keep a filter or an advertisement until it expires and is removed from its routing tables. However, this approach is also susceptible to issues caused by the propagation delays. Figure 7.3 shows the influence of the filter propagation delay on the routing tables. One can observe that subscription message issued by the subscriber **S** at real-time 12:00 has the validity interval T_I set to 0:04. At real time instance 12:02 it reaches the broker **B1** which installs it in its routing table and starts decrementing the validity interval of the filter on its local clock. Subsequently, it forwards the filter to the broker **B2**, which repeats the procedure. At real-time instance 12:06 filter issued by the subscriber **S** reaches the broker **B3**. However, at that time broker **B1** removes the filter from its routing table since the the filter validity interval (equal to 0:04) has been decremented (since 12:02) to 0. This in turn results in publisher's **P** events never being delivered to the subscriber **S**.
It is therefore required to set the re-issue period τ to a value which is lower than the specified validity interval T_I. Otherwise, the subscription message propagation will result in messages not being refreshed before the expiry of the validity interval. However, even the setting of the re-issue period to low values does not guarantee that the refresh messages arrive before the expiry of the validity interval. Varying link latencies and link overload may lead to significant delays of the refresh messages which in turn leads to the expiry of the advertisements and filters.

7.2.3 Extending Validity Interval

In order to cope with the above issues the message propagation delays (see Equation 6.15) are correlated with the validity interval approach for both filters and advertisements. Specifically, the advertisement and subscription propagation delays are calculated for every link traversed by those messages. The estimated delays are added to the validity interval T_I of each filter and advertisement and are stored along the filters and advertisements in the routing tables of the processing brokers. As a result the validity interval of filters and advertisements is *extended* by the amount equal to the expected link and processing delays. This in turn ensures that filters and advertisements do not expire before the arrival of the re-subscription and re-advertisement messages. Moreover, such approach is transparent to both subscribers and publishers.
As an example we shall consider the scenario presented in Figure 7.4. For the simplicity of the presentation it is assumed that all processing times (see

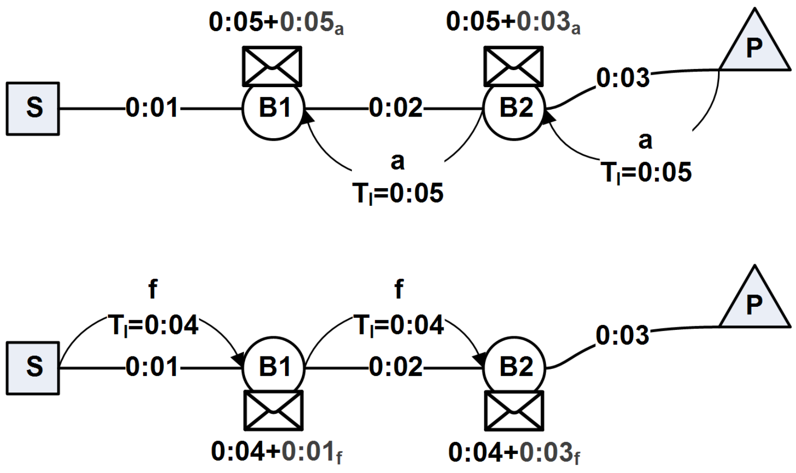


Figure 7.4: Extending validity interval with the upper bound on the propagation delay

Equation 6.14) are equal to zero. The upper part of the Figure 7.4 shows the process of the propagation of the advertisement a issued by the publisher **P**. The advertisement a has been assigned a validity interval T_I equal to 0:05. Upon reception of the advertisement a by the broker **B2** it calculates the upper bound on the propagation delay of the advertisement a.

The upper bound on the propagation delay $ub(pd_{\mathbf{B2}}(a))$ is the sum of the upper bound on the processing time of the advertisement a (equal to zero) and the upper bound on the transmission delay between the publisher **P** and the broker **B2** equal to 0:03. The calculated upper bound on the propagation delay $0:03_a$ of the advertisement a is subsequently added to the validity interval $T_I = 0:05$ of the advertisement a. The extended validity interval is stored alongside advertisement a in the routing table of the broker **B2**. Broker **B2** will remove the advertisement a from its routing table only after the expiry of the extended validity interval 0:05 + $0{:}03_a$.

The advertisement a is subsequently propagated by the broker **B2** towards the broker **B1**. Broker **B1** performs the analogous operations to those of the broker **B2**, the only difference being the value of the upper bound on the transmission delay of the advertisement a including now both upper bounds on transmission delay between the publisher **P** and the broker **B2** (0:03) and between the broker **B2** and the broker **B1** (0:02) plus the upper bound on the processing delays at the three nodes **P**, **B2** and **B1**. The calculation of the validity interval T_I for filters is done in the analogous way – see lower part of the Figure 7.4.

The above algorithm allows filters and advertisements to remain valid at every node despite the varying latencies on the communication links and despite the varying processing latencies. Presented algorithm intuitively extends the validity interval by the upper bound on the possible delay experienced by messages on its way from the publisher (or subscriber) until the current broker.

The algorithm does not guarantee that a filter or advertisement will never expire before the arrival of the refresh message. However, such guarantee is impossible to satisfy with unbounded link and processing delays inherent to the TADSM model. Nonetheless, subsequent refresh messages will adapt to the changed link and processing characteristics so that the correct operation of the content-based pub/sub network will be eventually restored, providing the processing times and transmission delays do not grow infinitely.

7.2.4 Utilization and Uncertainty

The algorithm presented in the previous section makes a pessimistic assumption regarding the propagation of refresh messages. The assumption being made is that a refresh message propagation time might span the range from zero to the calculated upper bound on the propagation delay. Such assumption while providing large margin of safety is certainly an overestimation. Therefore, in this section an alternative algorithm for the calculation of the validity interval extension and a new metric for evaluation of the application of the validity interval extension technique are proposed.
The new metric proposed in this section is called the validity interval utilization. The validity interval utilization is defined as the amount of the extended validity interval $T_I(m)$ of the message m which has elapsed before the arrival of the refresh message:

$$U_{T_I} = \frac{T_I(m) - \mathrm{wait}(m)}{T_I(m)} \tag{7.1}$$

where the wait(m) is the amount of time message m has spent in the routing table before the arrival of the refresh message. The validity interval utilization U_{T_I} is expressed in percent. Intuitively, the higher the validity interval utilization, the less bandwidth is wasted for the unnecessary (too early) refresh messages. On the other hand, high validity utilization implies that the given filter or advertisement was more likely to expire before the arrival of the refresh message – as the delay of the refresh message might not have been compensated by the small remaining validity utilization.
The validity interval extension algorithm presented in the Section 7.2.3 lowers the values of the validity interval utilization for filters and advertisements by extending the validity interval with the upper bound on the respective message propagation delay. Moreover, the further the given message travels from the source (in terms of latency) the lower will the validity interval utilization be.
The above observation is the motivation for the introduction of the new algorithm for the extension of the validity interval. The new algorithm is based on the upper bound on the propagation delay, however its main goal is the estimation of the *uncertainty* of the upper bond for the given message. The uncertainty ub_Δ of the upper bound on the propagation delay for a given message m is defined as the difference between the minimum and maximum upper

bound on the propagation delay values over time:

$$\mathrm{ub}_\Delta(m) = \frac{\mathrm{ub}_{\max}(\mathrm{pd}(m)) - \mathrm{ub}_{\min}(\mathrm{pd}(m))}{\Delta t} \tag{7.2}$$

where Δt is the period of time (window) over which the uncertainty is calculated. Intuitively, the uncertainty serves as an estimation of the dispersion of the latency for the given path limited to a certain time window. Unlike the extension calculation algorithm presented in Section 7.2.3 for stable paths the value of the uncertainty-based extension will be low as the difference between the minimum and maximum upper bound on the propagation delay will remain small. This in turn will increase the values of the utilization for the given path. The window Δt for which the uncertainty ub_Δ is calculated can be expressed either in terms of time units or in terms of messages. The choice of the right window size and calculation method is left to the application programmer, with the time-based and message-based windows already included in the system and selectable via the configuration parameters.

7.3 Liveness and Safety

The correctness of distributed systems is often specified in terms of liveness and safety [Lam79]. In what follows the liveness and safety definitions for the correct publish/subscribe systems (following [Jae07, MÖ2]) are given. The definitions are formalized using temporal logic [Pnu81].
The safety property is composed of the conjunction of the following definitions: (1) an event e is delivered to the subscriber $\mathbf{S}$ at most once:

$$\Box\left[\mathrm{deliver}(e, \mathbf{S}) \Rightarrow \bigcirc\Box\neg\mathrm{deliver}(e, \mathbf{S})\right] \tag{7.3}$$

(2) the subscriber $\mathbf{S}$ only receives events which have been previously published:

$$\Box\left[\mathrm{deliver}(e, \mathbf{S}) \Rightarrow \exists_{\mathbf{P}} : \mathrm{publish}(e, \mathbf{P})\right] \tag{7.4}$$

(3) the subscriber $\mathbf{S}$ only receives events for which it has subscribed (cf. Equation 2.2):

$$\Box\left[\mathrm{deliver}(e, \mathbf{S}) \Rightarrow \exists_{f \in \mathbb{F}_\mathbf{S}} : f(e) = 1\right] \tag{7.5}$$

The liveness property states that: subscriber $\mathbf{S}$ which subscribed with filter f and did not issue an unsubscription message for that filter, will eventually receive every event e which is published and matches f:

$$\Box\left[\Box\left(f \in \mathbb{F}_\mathbf{S}\right) \Rightarrow \Diamond\Box\left(\mathrm{publish}(e, \mathbf{P}) \wedge f(e) = 1 \Rightarrow \Diamond\mathrm{deliver}(e, \mathbf{S})\right)\right] \tag{7.6}$$

The soft state XSiena system satisfies the liveness property. Moreover, following [Jae07] it can be said that the soft state XSiena system satisfies a modified version of the safety property – the eventual safety property, defined as: starting

from an arbitrary state the system eventually satisfies the safety properties as defined in Equations 7.3, 7.4 and 7.5.
However, the soft state XSiena system, unlike systems proposed in [MJH+05, Jae07], does not (and cannot) guarantee an upper bound on the time for which system remains in the incorrect state. This is a direct implication of the unbounded network delays in the Timed Asynchronous Distributed Systems Model – the underlying model for the soft state XSiena system – see Chapter 3. Instead, it can be said that the soft state XSiena system achieves the eventual safety property assuming that: (1) network delays do not expose an infinite growth and (2) the size of the network does not expose an infinite growth either.
The first assumption on the finite growth of the network latencies implies that the validity extension technique eventually calculates an extension large enough so that subsequent refresh messages can reach the filter to be refreshed within its specified validity interval T_I plus the calculated extension. The second assumption is parallel to the first one and implies that the growth in the network size allows the filters to finally propagate to all brokers in the network.

7.4 Practical Aspects and Implementation

The implementation of the soft state XSiena system is based on the fail-aware approach presented in Chapter 6. Figure 7.5 outlines the most important parts of the soft state XSiena architecture. The soft state-related code is well isolated and has been placed in the `soft state I/O filter` within the Apache MINA[2] [Lec09] network stack. The `soft state I/O filter` is placed directly before the serialization filter and is responsible for: (1) calculation of the upper bound on the message transmission delay, (2) calculation of the upper bound on the message processing time and (3) scheduling of the unsubscritptions and unadvertisements of the expired messages.
The use of the soft state I/O filter allows to calculate the upper bound on the message transmission delays on a per link basis. Brokers in the soft state XSiena system manage the upper bound on transmission delay for every incoming link. The upper bound on transmission delay for the given link is updated upon reception of every message on this link. Specifically, the upper bound calculations are not limited to the advertisement or subscription messages, instead, every message exchanged on the given link is used to calculate the upper bound. A link-oriented approach allows to scale the implementation of the transmission delay calculation independently of the number of different advertisements and filters in the system. This is especially valid if one considers potentially large number of helper messages which need to be stored – see Chapter 6.2.

[2]See: `http://mina.apache.org`

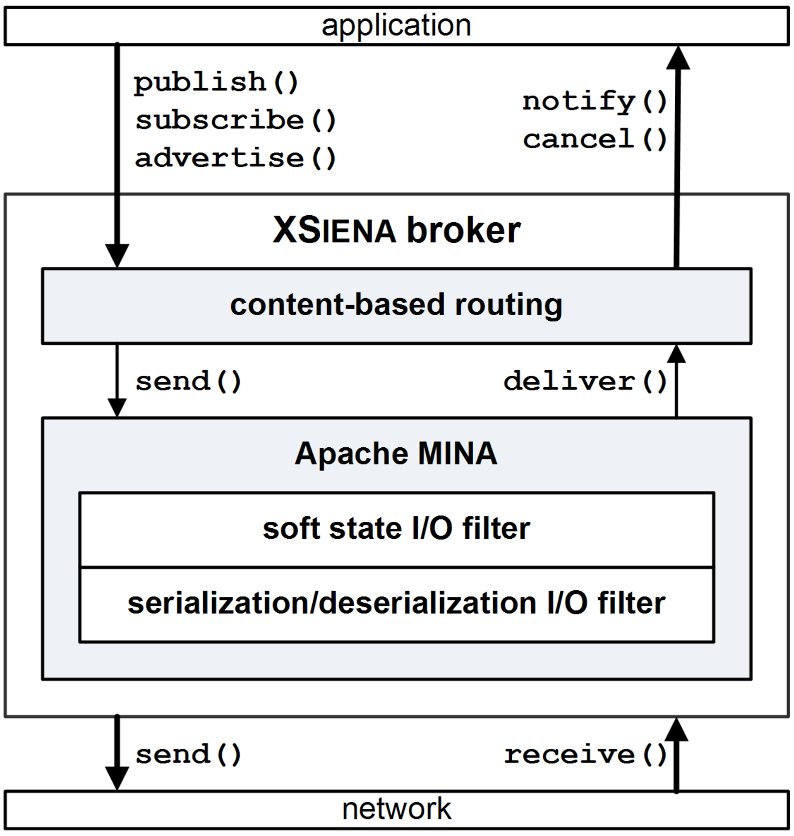


Figure 7.5: The overview of the soft state XSIENA architecture

The soft state XSIENA system allows the user to select between two hardware clocks. First hardware clock is based on the Java `System.nanoTime()` method. The second hardware clock uses a C-based wrapper for the `rdtsc` assembler instruction. The C-based wrapper is accessible from within the Java application via the JNI interface.
It is up to the developer of the soft state XSIENA application to implement the periodic re-subscriptions and re-advertisements. Such implementation can be easily performed using, e.g, Java's `ScheduledThreadPoolExecutor`. Such approach follows the lines of the soft state system philosophy, in that the crash of the application should result in the lack of the refresh messages.

7.4.1 API

The nature of the soft state publish/subscribe system determines the exposed API. The API presented in the Listing 7.1 is a direct result of the properties of the soft state XSIENA system. Since the system uses both advertisement and subscription messages the API follows that presented in Listing 2.3, including calls to advertise new content and to subscribe to events. The subscribe operation includes as parameters: (1) a new filter `CFilter` and (2) a callback

```
public interface XSiena {
  void publish(Event e);
  void subscribe(Filter f, Deliverable d);
  void advertise(Filter f);
}
```

Listing 7.1: The soft state XSiena broker interface.

```
public interface ExtendedXSiena {
  void unsubscribe(Filter f, Deliverable d);
  void unadvertise(Filter f);
}
```

Listing 7.2: The extended soft state pub/sub broker interface.

interface `Deliverable`. The `Deliverable` interface is implemented by the application programmer in order to receive events matching the issued filters. Unlike the API proposed in [PEKS07] the validity of the filters and advertisements is not explicitly included in the corresponding `advertise` and `subscribe` API calls. In soft-state XSiena system the validity interval T_I is part of the subscription and advertisement messages and can be manipulated via the `Filter` API. Specifically, the validity interval T_I is not a subject to the content-based matching. It is the opinion of the author that such design is more natural for the soft state publish/subscribe systems, as the validity of the filter or advertisement is an integral part of the given message and must travel with it across multiple nodes in the publish/subscribe network.

7.4.2 Unsubscriptions and Unadvertisements

An important aspect of the soft state publish/subscribe system is the lack of unadvertisement and unsubscription methods in the API. Such methods are strictly speaking not necessary, as every advertisement and filter will expire within its validity interval T_I plus the broker dependent extension. However, it is possible to provide an extended API to the brokers (see Listing 7.2) which allows to unadvertise or unsubscribe messages prior to the expiration of their validity intervals. It is important to stress that soft state publish/subscribe brokers do not rely on those calls, which are only a means to speed up removal of filters and advertisements with large validity intervals.

Another, related, design decision is made with respect to the removal of filters and advertisements for which the validity interval expires. Such cases are handled by the brokers via the broker internal, private unsubscribe and unadvertise methods. An important aspect of those methods is that their effects are never propagated outside of the brokers. If, for example, a filter f_1 covering filter f_2 expires than fact of the deletion of the filter f_1 and uncovering of the filter f_2 is never propagated to other downstream brokers. Even though the propagation of this information would be a correct action from the perspective of the soft state publish/subscribe system (see Section 2.5.5) and could increase the responsiveness of the publish/subscribe system it is chosen not to implement this behavior in the soft state XSiena system. It is important to note that the correct routing information will be eventually set up with the propagation of the refresh messages for the filter f_2. The motivation for the removal of the visibility of the unsubscription and unadvertisement external actions it is the fact that it is desirable to try to avoid the storm of unsubscription and unadvertisement messages propagated throughout the publish/subscribe network whenever a filter or advertisement expires due to the high variance in the propagation delays of the refresh messages. In other words soft state XSiena system deliberately trades-off the responsiveness of the system for its scalability.

7.5 Related Work

Timed filters and events have been first addressed in [FJL+01]. Authors assume that filters and events are associated with time intervals, which limit their validity. However authors do not consider the renewal of filters which leads to the issues presented on Figure 7.1 in Section 7.1. Another approach for the creation of the soft state publish/subscribe system has been presented in [BSB+02] where authors assumed a fixed filter structure for each of the publish/subscribe system nodes and focused on the exactly once delivery of events. Specifically, authors did not consider dynamic content-based routing using the coverage relation and coupled the publishers with the subscribers by the requirement for the latter to confirm reception or lack of events.
In [ST04] authors design a self-stabilizing publish/subscribe system. The proposed system assumes however, that all subscription messages issued by subscribers are always broadcast into the network. Such algorithm is not efficient, as similar or identical filters need not to be resent on links where a filter covering the given one has already been sent – cf. Section 2.5.5. Moreover, authors propose to exchange whole routing tables in order to detect potential inconsistencies, which can be expensive in publish/subscribe systems with large number of subscribers and large number of distant brokers.
Author in [MÖ2] proposes a subscription leasing scheme to achieve eventual stability. Author assumes the existence of a global clock (internal or external clock synchronization providing a notion of global time accessible to all participants of the publish/subscribe network) and assumes that message transmission time is always bounded and stays within $[\delta_{\min}, \delta_{\max}]$. It has been shown that such system model is impossible to satisfy [FC03], which implies the need for a weaker set of assumptions regarding the system model.
In [ZSB04] authors show how to provide a reliable (in-order, gapless) delivery

in a content-based publish/subscribe system. In contrast to the solutions presented in [ZSB04] the soft state XSiena system does not assume a redundant overlay network and proposes an active countermeasure for the problem of filter propagation failures.
In [XFZ04] authors propose a soft state approach in context of a publish/subscribe system based on the Pastry DHT overlay. Authors assume that both subscriptions and advertisements are assigned timeouts which are evaluated with respect to the node's local clock. However, the presented approach does not account for the influence of the propagation delays on the timeout values which makes it impractical in a typical distributed system.
In [Pie04] authors propose a soft state design in that publish/subscribe brokers periodically exchange heartbeat messages in order to keep the state of the routing tables intact. The presented approach does not allow to cope with the issues presented in Figure 7.1 as the heartbeat messages and not data carrying messages (filters and advertisements) are used to refresh the state of the routing tables.
In a more recent approach presented in [MJH^{+}05, Jae07] authors simulate a publish/subscribe system achieving eventual stability using advertisements and filter leases. Authors, similarly to [MÖ2], assume that every entry has globally uniform filter timeout value π. Every clock in the system can take values between 0 and $\pi - 1$. Every entry in the routing table has a one bit counter, initially set to one. When the clock overruns the counters of all entries are decremented. Entries which counters overflow are removed from the routing tables of the respective brokers. Similarly to [MÖ2] authors assume that there exist global upper and lower bounds on the propagation delay of messages. Whenever those bounds are violated authors assume that the system has suffered a failure. The soft state XSiena system does not make such assumptions. Instead, a per filter and per advertisement timeout values are proposed which allow to adaptively account for varying communication delays. Authors in [MJH^{+}05, Jae07] have also assumed a known network diameter, which allows them to calculate an upper bound on the stabilization time of the soft-state publish/subscribe system. The soft state XSiena does not allow for estimation of such value as due to the decoupled nature of publish/subscribe systems a fixed network diameter value is impossible to obtain.

Chapter 8

Implementation

This chapter presents the most important aspects of the implementation of the family of the XSIENA systems. The XSIENA system has been implemented in Java, however this chapter focuses on the language independent aspects of the implementation. The implementation of events and filters along with the coverage relation stems from the Java version of the SIENA publish/subscribe system. All other parts of the XSIENA system, including the communication framework, routing framework and message handling code have been contributed by the author of this thesis.
This chapter is divided into two parts. First part (Section 8.1) discusses the implementation of the XSIENA system from the perspective of the application developer wishing to use the XSIENA system as a communication backbone in his project. The second part (Section 8.2) discusses the internals of the XSIENA project which are of relevance for developers willing to modify the behavior of the XSIENA publish/subscribe system.

8.1 Programming Applications

Every application created in the XSIENA system uses either the publisher or the subscriber API. In practice, every XSIENA application is connected directly to the XSIENA broker – the `CDispatcher`. The XSIENA broker implements the complete publish/subscribe API (cf. Listing 2.3, Section 2.2.3) with applications implementing the necessary methods.
Figure 8.1 presents the basic interfaces used by the `CDispatcher` class. The complete publish/subscribe API is contained within the `IXSiena` interface. Applications can use the `publish()` method to disseminate events `CEvent` into the publish/subscribe network. When advertisement-based routing is used (cf. Section 2.5.6) applications summarize the content they are going to produce using the `advertise()` and `unadvertise()` methods taking the `CFilter` describing the content as parameter. In order to receive data an application subscribes using the `subscribe()` method, passing the `CFilter`

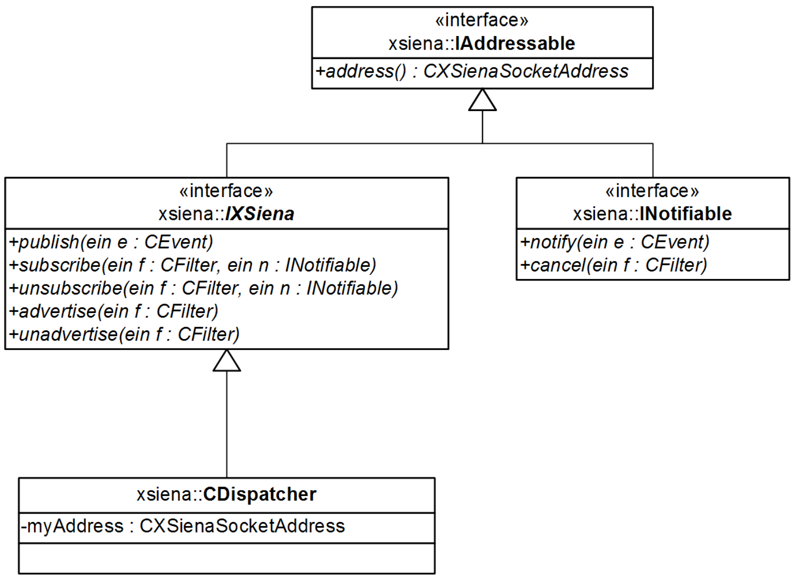

«interface»
xsiena::IAddressable
+address() : CXSienaSocketAddress
«interface»
xsiena::IXSiena
+publish(ein e : CEvent)
+subscribe(ein f : CFilter, ein n : INotifiable)
+unsubscribe(ein f : CFilter, ein n : INotifiable)
+advertise(ein f : CFilter)
+unadvertise(ein f : CFilter)
«interface»
xsiena::INotifiable
+notify(ein e : CEvent)
+cancel(ein f : CFilter)
xsiena::CDispatcher
-myAddress : CXSienaSocketAddress

Figure 8.1: XSIENA broker-related interfaces

describing its interest and a return address (`INotifiable`) for the delivery of matching `CEvents` as parameters. Applications willing to stop receiving the data, can revoke their interest using the `unsubscribe()` method. Unsubscribe method uses the `INotifiable` parameter to identify the application revoking its interest.
Every application in the XSIENA system must implement the `INotifiable` interface. The `INotifiable` interface requires applications to provide the code for handling of two cases: (1) the delivery of an event matching the interest of the application (`notify()`) and (2) the cancellation of the issued filter due to the lack of a matching advertisement (`cancel()`). Additionally, every application, as well as every `CDispatcher` must return its own address – the `CXSienaSocketAddress`.
The address of the XSIENA application or broker has the following form:

$$\underbrace{\texttt{xsiena}}_{\text{protocol specification}} :// \underbrace{\texttt{application}}_{\text{application name}} @ \underbrace{\texttt{host.ip}-\texttt{or}-\texttt{name.com}}_{\text{host address}} : \underbrace{\texttt{port}}_{\text{host port}} \tag{8.1}$$

where the currently supported protocol specifications include `xsiena` for the TCP-based communication and `xsienau` for the UDP-based communication. The application name is used to differentiate between multiple applications attached to the same `CDispatcher`. It is ignored in case of the `CDispatcher` address. The host address can take either the form of the DNS name or the IP address. Optionally, a port can be specified, although both standard protocols supported by the XSIENA have their default ports: 3420 for the `xsienau` and 3421 for the `xsiena`.
Listing 8.1 shows an example publish/subscribe application which measures

```
class CLatencyApp implements INotifiable {
   CXSienaSocketAddress myAddress = null; int c;

   CLatencyApp(CXSienaSocketAddress sa) {
      myAddress = sa; c = 0;
   }

  public CXSienaSocketAddress address() { return myAddress; }

  public void cancel(CFilter f) { /*ignore*/ }

  public void notify(CEvent e) {
    long RT = System.nanoTime();
    long ST = e.getAttribute("ST").longValue();
    System.out.println( (++c) + "\t" + (RT-ST) );
  }
}
```

Listing 8.1: Latency measuring application

latency of the received events. The presented application (CLatencyApp) implements the address() method in order to return the address under which it can be notified for events matching its interest. The presented applications ignores the case when there is no advertisement matching its interest. Whenever it receives a new event (notify) it saves the event receive time and based on the send time contained in the event it calculates the total time between the sending and the reception of event.
Listing 8.2 presents the corresponding main part of the publish/subscribe application. It can be observed that in the first part of the program a network composed of three brokers **B1**, **B2** and **B3** communicating over the TCP/IP is created. All brokers are placed on the local host, so as to make the comparison of the time stamps (cf. Listing 8.1) feasible. Brokers are connected so as to form a chain starting with broker **B1** and ending with broker **B3**. After the creation of the network the latency application presented in Listing 8.1 is instantiated. Subsequently, the publisher of events advertises the content it is going to produce (B1.advertise(filter)) and the latency application located on the broker **B3** subscribes to the published events which have the send time stamps greater than 0 – B3.subscribe(filter, LA). In the last part of the program an endless publishing loop is started with every event being attached the send time stamp.
The above example shows the ease of the creation of publish/subscribe network using the XSiena publish/subscribe system. Changing the, e.g., transport protocol in the above example from TCP to UDP requires only the change of the protocol specification (xsiena to xsienau) for the three broker addresses in the Listing 8.2. Specifically, the publish/subscribe application is completely decoupled from the underlying implementation of the publish/subscribe net-

```
public static void main(String[] args){
   CXSienaSocketAddress B1A, B2A, B3A;
   B1A = CXSienaURI.parse("xsiena://localhost:4050");
   B2A = CXSienaURI.parse("xsiena://localhost:4060");
   B3A = CXSienaURI.parse("xsiena://localhost:4070");

   CDisptacher B1 = new CDisptacher(B1A, null);
   CDisptacher B2 = new CDisptacher(B2A, B1A);
   CDisptacher B3 = new CDisptacher(B3A, B2A);
   CLatencyApp LA = new CLatencyApp(B3A.setID("latencyApp"));

   CFilter filter = new CFilter("ST", COp.GT, 0);
   B1.advertise(filter);
   B3.subscribe(filter, LA);
   while(1) {
      Thread.sleep(100);
      CEvent event = new CEvent("ST", System.nanoTime());
      B1.publish(event);
   }
}
```

Listing 8.2: Latency measuring system

work.

8.2 Developer View

Figure 8.2 presents the overview of the main packages which constitute the XSiena system. The main part, including the `CDispatcher` class is contained in the `xsiena` package. The communication layer classes are placed in the `xsiena::comm` package. The upper bound calculation and related methods have been placed in the `xsiena::comm::ub` package reflecting the tight cou-

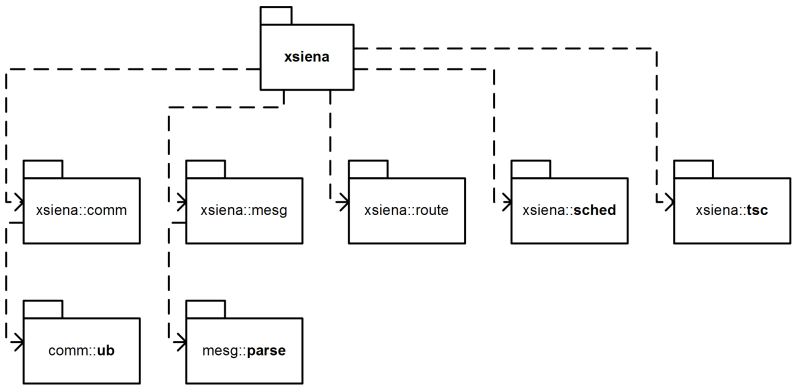


Figure 8.2: Packages in the XSiena system

pling between the calculation of the upper bound and the communication layer – cf. Section 7.4.
Events, filters and their relation reflecting classes are contained within the `mesg` sub-package of the `xsiena` package. This package contains also the network packet class `CSENPPacket` which reflects the unsterilized wire format of the XSiena messages. Package `parse` (contained with the `xsiena::mesg` package) contains the grammar specification of the event and filter textual representation. The grammar specification is used by the JavaCC (`https://javacc.dev.java.net/`) parser generator in order to generate parsers for reading of events and filters from, e.g., configuration files.
The `xsiena::route` package contains the routing code of the XSiena system. Specifically, it contains implementations of the different routing structures used by the versions of the XSiena system, including: the poset-derived forest data structure (cf. Section 2.5.5) the `sbsposet` and `sbstree` (cf. Section 5.3) and the counting variant of the `sbstree` – cf. Section 5.5. However, unlike in the case of the system presented in [MÖ2], the different routing structures constitute different versions of the XSiena system and are not unified within one routing framework.
The `xsiena::sched` and `xsiena::tsc` packages are specific to the soft state version of the XSiena system. The `xsiena::sched` package contains the scheduler which is responsible for the removal of the expired filters and advertisements and for extending the validity of the existing ones upon the arrival of the refresh messages. The `xsiena::tsc` package contains the implementation of the different local hardware clocks. The current version provides support for the CPU time stamp counter (`rdtsc`) and the built-in Java nanosecond timer – `System.nanoTime()`.
Figure 8.3 illustrates the main components of the `CDispatcher` class. Every XSiena dispatcher maintains a reference to the packet sender and the packet receiver. Which decouple the specific protocol being used from the XSiena broker. The packet receiver uses the `IDeliverable` interface of the `CDispatcher` in order to deliver all `CSENPPackets` arriving via the network to the broker. The broker itself holds a reference to all of its neighbor brokers, using their addresses. Moreover, it holds the reference to the `CRouting` class which decouples the different routing algorithms from the broker. The `CRouting` class also implements the `IXSiena` interface, however it uses the `CDispatcher` methods to send the `CSENPPackets` to the neighboring brokers.

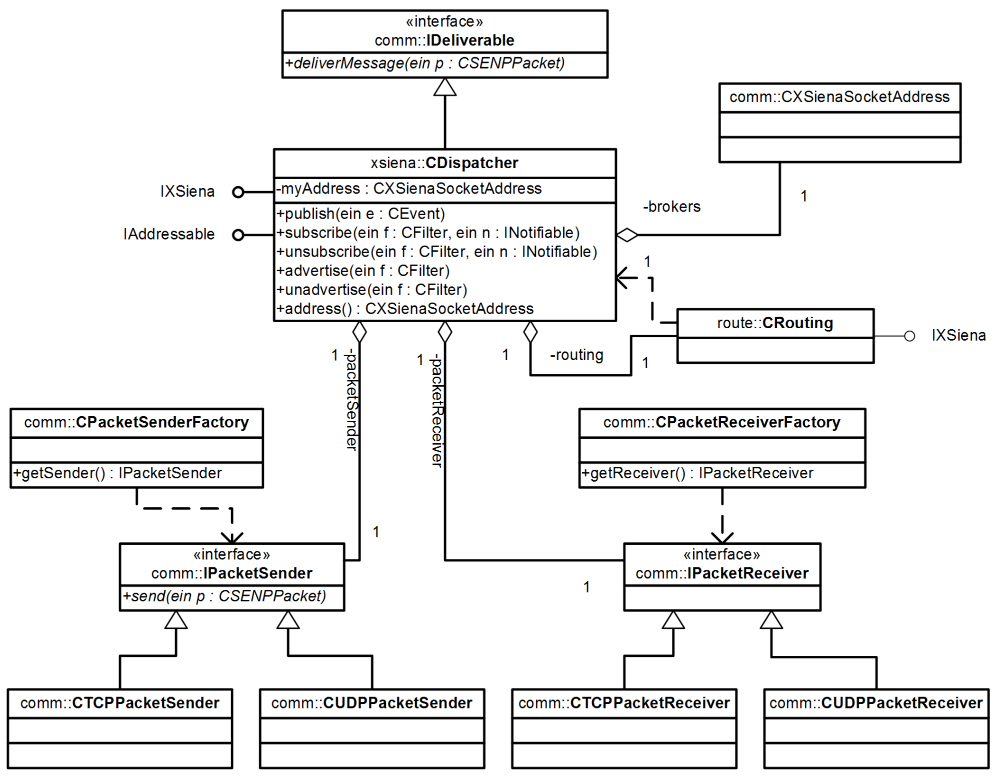

«interface»
comm::IDeliverable
+deliverMessage(ein p : CSENPPacket)
comm::CXSienaSocketAddress
xsiena::CDispatcher
-myAddress : CXSienaSocketAddress
+publish(ein e : CEvent)
+subscribe(ein f : CFilter, ein n : INotifiable)
+unsubscribe(ein f : CFilter, ein n : INotifiable)
+advertise(ein f : CFilter)
+unadvertise(ein f : CFilter)
+address() : CXSienaSocketAddress
IXSiena
IAddressable
-brokers
1
1
route::CRouting
IXSiena
-routing
1
1
1
-packetSender
1
-packetReceiver
comm::CPacketSenderFactory
+getSender() : IPacketSender
comm::CPacketReceiverFactory
+getReceiver() : IPacketReceiver
1
«interface»
comm::IPacketSender
+send(ein p : CSENPPacket)
1
«interface»
comm::IPacketReceiver
comm::CTCPPacketSender
comm::CUDPPacketSender
comm::CTCPPacketReceiver
comm::CUDPPacketReceiver

Figure 8.3: The developer view of the `CDispatcher`

Chapter 9

Evaluation

This chapter presents the evaluation of the algorithms and overall system performance for the XSiena family of the content-based systems. The evaluation section follows the layout of the thesis: Section 9.1 presents the evaluation of the prefix forwarding (based on Chapter 4) followed by Section 9.2 presenting the evaluation of the Bloom filter-based routing (based on Chapter 5). In the second part of the evaluation (Section 9.3) the performance of the fail-aware XSiena system is investigated (based on Chapter 6) followed by the evaluation (Section 9.4) of the application of the fail-aware principles to the construction of the soft-state XSiena system (based on Chapter 7).
The evaluation focuses on the properties of the proposed algorithms and whenever possible it tries to compare algorithms presented in this thesis with previous, existing solutions. The evaluation of the XSiena family of publish/subscribe systems is performed using both simulation and existing system prototypes. The existing prototypes are available for download at the `http://wwwse.inf.tu-dresden.de/xsiena/` address.
For the evaluation purposes a generic library for the generation of random predicate-based filters and events has been created. The library generates filters and events according to the parameters presented in the Table 9.1. Since, the family of the XSiena publish/subscribe systems is derived from the Siena publish/subscribe system, the set of generated filters and events follows the predicate-based semantics of the Siena system. The only difference being that the generated filters are in the form presented in Section 4.2.1, i.e., no logically inconsistent filters are created and no range operators, nor inequality operators are used. The minimum and maximum values for the given parameter, e.g., attribute value, can be set to the same value resulting in the selected value being always returned. The set of available distributions includes uniform, normal and Pareto distributions. The normal distribution function is given by the probability density function:

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \qquad \text{for} \quad -\infty < x < \infty \tag{9.1}$$

Filter Generator	**Event Generator**
minimum number of filters	minimum number of events
maximum number of filters	maximum number of events
distribution of the number of filters	distribution of the number of events
minimum predicate count (per filter)	minimum attribute name and value pair count (per event)
maximum predicate count (per filter)	maximum attribute name and value pair count (per event)
distribution of the number of predicates	distribution of the number of attribute name and value pairs
unique attribute names count	unique attribute names count
distribution of the attribute names	distribution of the attribute names
minimum attribute value	minimum attribute value
maximum attribute value	maximum attribute value
distribution of the attribute values	distribution of the attribute values
range of operators	$\emptyset$ (not used)
distribution of the operators	$\emptyset$ (not used)

Table 9.1: Parameters of the filter and event generation library

where, unless explicitly stated, the standard deviation σ is set to 0.3 and mean μ is set to 0. Pareto distribution is given by the probability density function:

$$p(x) = \frac{\alpha\beta^{\alpha}}{x^{\alpha+1}} \qquad \text{for} \quad x > \beta \tag{9.2}$$

where, unless explicitly stated, the parameter α is set to 3 and the parameter β is set to 1. The distribution functions are calculated using the Stochastic Simulation in Java library [LMV02, LB05] by Pierre L'Ecuyer.
The predicate count and the attribute name and value count distributions are, unless explicitly stated chosen as uniform. Attribute names for both events and filters are selected from the Automatically Generated Inflection Database (AGID)[1], based on the `aspell` word list, containing 112505 English words

[1]`http://wordlist.sourceforge.net`

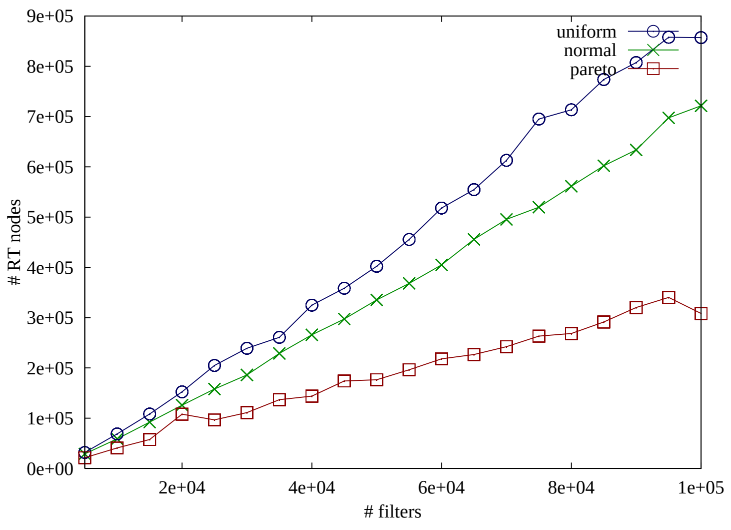


Figure 9.1: Routing tree size (number of nodes)

and acronyms. The distribution of the attribute names can also be chosen from the uniform, normal and Pareto set. The attribute values are generated from the [minimum, maximum] range using the integer values. The distribution of the attribute values is also subject to the three possibilities: uniform, normal and Pareto. The range of the operators, unless explicitly stated, contains: $\{>, \geqslant, <, \leqslant, =\}$, and the distribution of the operators is set to uniform.

9.1 Prefix Forwarding

The evaluation of the prefix forwarding XSiena system has been performed in a dedicated, discreet event simulator, conceptually and architecturally based on the OMNeT++ [Var99] simulator by András Varga. The simulation included the networking layer and message based communication between the components of the publish/subscribe system. Specifically, a fully functional routing tree and forwarding tree structures have been implemented. Both structures operate on the standard predicate-based messages.
The first experiments which were performed included testing for the feasibility of the forwarding trees for the event forwarding. Specifically, it has been evaluated whether the size of the forwarding tree will allow it to be piggybacked on the events. For that purpose, routing trees with increasingly large number of filters stored in them have been created. Subsequently, randomly generated events have been matched against the routing trees in order to obtain the forwarding trees. The size and the height of the routing trees are illustrated in Figures 9.2 and 9.1. The results are plotted for all three distributions available in the filter generation library for attribute names and attribute values, i.e., both

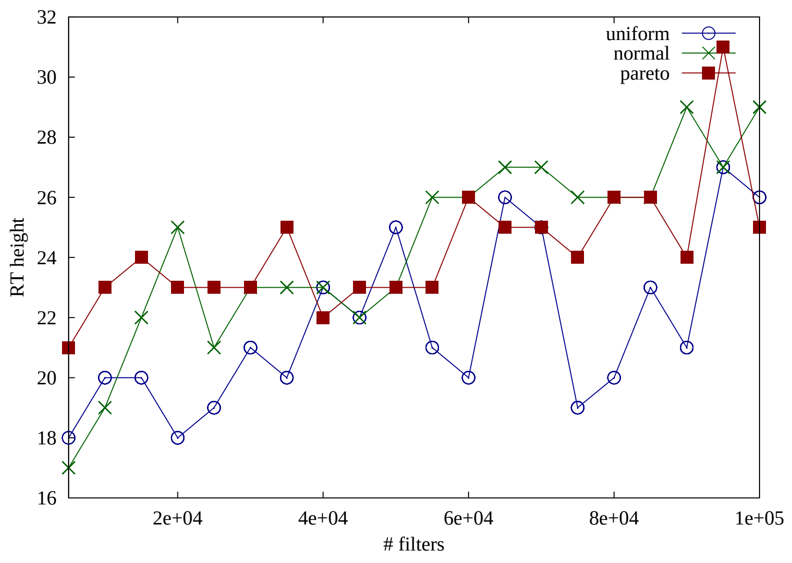


Figure 9.2: Routing tree height

parameters shared the same distribution. The amount of predicates in the filters has varied from 2 to 8. The attribute values were selected from a continuous interval [−50, +50]. All tests assume a pessimistic scenario, in which every filter originates in a different subscriber.

One can observe that the size of the routing tree is proportional to the distribution used, with the Pareto distribution having the least number of nodes. The explanation for this fact is the higher probability of creating identical predicates, and thus limiting the size of the routing tree. On the other hand, the count of the predicates in the routing tree is higher than it would seem from the number of filters (2–8 predicates per filter) due to the fact that the split operation duplicates parts of filters and thus increases the total number of nodes in the routing tree. Similarly, the more dense distributions (Pareto, normal) tend to produce higher routing trees, than the uniform distribution.

The following experiment generated a set of events and matched them against the routing trees presented in Figures 9.2 and 9.1. The amount of attribute name and value pairs in events varied from 2 to 12. Attribute values were selected from a continuous interval [−50, +50]. The events were also generated in three sets (uniform, normal and Pareto) which were matched with the corresponding routing trees. The parameters of event generation, apart from those already mentioned, were identical with those of the filter generation.

Figures 9.3 and 9.4 illustrate the size and height of the resulting forwarding trees. One can observe on both figures that although the amount of nodes in the routing tree approaches one million, the number of nodes (size) of the largest forwarding tree barely exceeds 150 nodes. One can contribute this behavior to

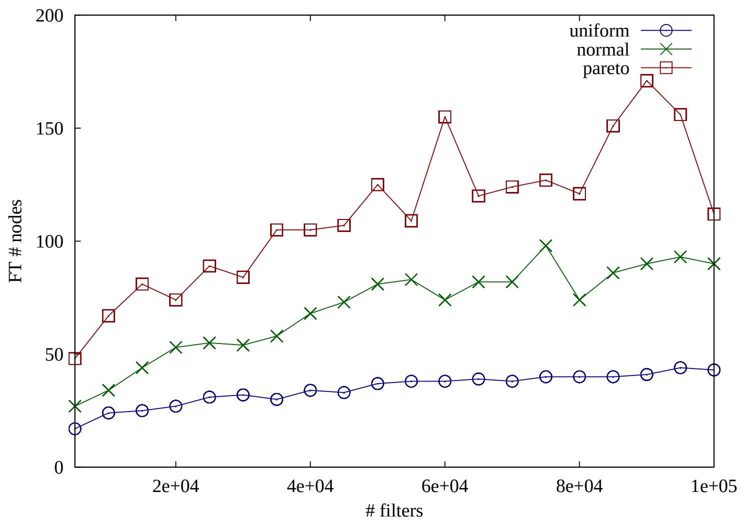


Figure 9.3: Forwarding tree size (number of nodes)

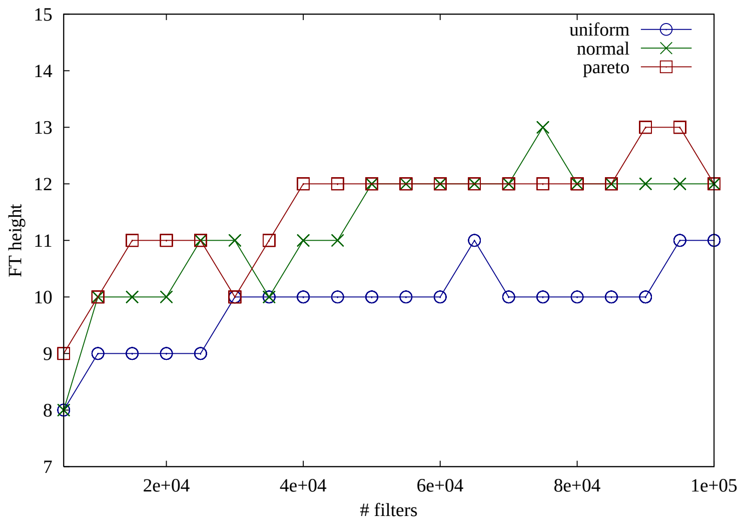


Figure 9.4: Forwarding tree height

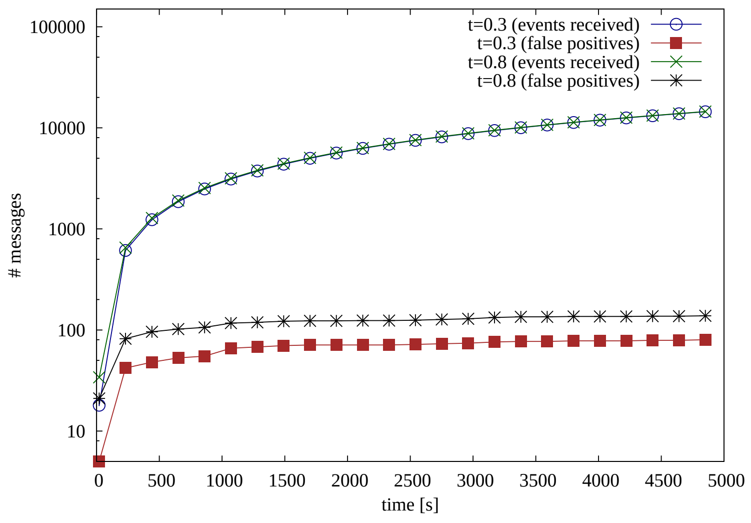


Figure 9.5: False positives count for different tree optimizer thresholds

the fact that the size of the forwarding tree is mostly determined by the number of the attribute name and value pairs in the events. This is confirmed by the relatively stable forwarding tree size despite increasing size of the routing tree. This, in turn, allows us to conclude that the size of the forwarding tree exhibits a small payload and allows it to be piggy-backed on the events.

9.1.1 Simulation Results

The next experiment simulated the operations of a content-based publish/subscribe system. For the purpose of the experiment a network with 9 subscribers, 9 publishers, 27 routers and one tree optimizer has been created. Each of the subscribers subscribed every 3 seconds with a filter generated using the Pareto distribution. Every 3 seconds each of the publishers issued an event generated using the Pareto distribution. Figure 9.5 shows the averaged rate of the false positives as perceived by the subscribers. The experiment has been performed with two different tree optimizer update thresholds (see Equation 4.2): 30% and 80%. One can observe that the rate of false positives is higher with the higher update threshold of the tree optimizer – this is expected behavior of the system, as the higher update thresholds imply longer time spans when brokers use the `inertR` method on the old routing tree.
It can be also observed that with the growth of the routing trees the rate of false positives falls achieving less than 1% after 15000 events publications. The obtained results are similar, despite the varying threshold values of the tree optimizer. This behavior can be contributed to the fact that the routing trees are getting more stable with the time. Specifically, the changes in the routing tree are more likely to be performed at the deeper three levels, thus reducing

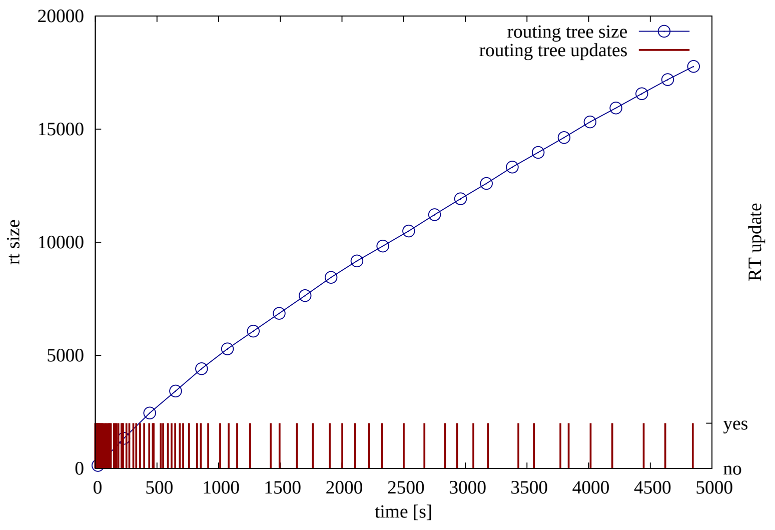


Figure 9.6: Tree optimizer updates of the routing tree – 30% threshold

the number of false positives.

During the simulation the number and frequency of routing tree updates performed by the tree optimizer has been counted as well. Figures 9.6 and 9.7 show the number of updates and the size of the routing tree when two different update thresholds are used. Using the lower update threshold (Figure 9.6) of 30% the tree optimizer initiated 84 updates. When using the higher update threshold (Figure 9.7) of 80% the tree optimizer initiated only 19 updates. In both cases it can be clearly observed that with the growth of the routing tree, the number of necessary updates decreases. This behavior can be explained by the fact that the larger the routing tree grows the more new filters it takes to reach the update threshold. More importantly, as already mentioned large routing trees are more unlikely to have new filters added close to the root, which in combination with the weighted threshold calculation algorithm (see Equation 4.2) decreases the update rate. One way to avoid too infrequent updates for large routing trees is to add a second, time-based threshold.

9.1.2 Using Real-Life Data

Testing publish/subscribe systems has always been a challenging task, mainly due to the lack of real-life data. There exist many examples of publish/subscribe systems being implemented in a real-life environment (e.g., Gryphon [ZSB04] has been used in US Tennis Open, Ryder Cup, and Australian Open [2], see also Section 2.1.1), however, the data gathered by those systems has not been made publicly available. Therefore, the testing of the publish/subscribe systems has

[2]`http://www.research.ibm.com/distributedmessaging/`

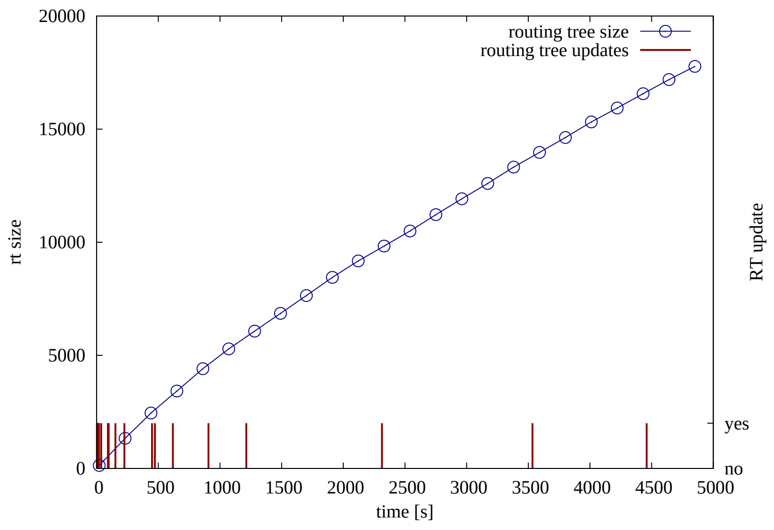


Figure 9.7: Tree optimizer updates of the routing tree – 80% threshold

```
24967317   anderson real estate   2006-05-31 13:03:42
24967317   www.xtremeteam.cus     2006-05-31 13:06:47
24967317   www.etremeteam.us      2006-05-31 13:07:18
24967317   celia murphy           2006-05-31 22:05:12
24967641   donut pillow           2006-05-31 14:08:53
24967641   dicontinued dishes     2006-05-31 14:29:38
24969374   orioles tickets        2006-05-31 12:31:57
24969374   baltimore marinas      2006-05-31 12:43:40
```

Listing 9.1: An extract from the AOL log data

in many cases relied on the statistical generators of event and filter patterns. However, such generators usually cannot represent the relations between filters and events as found in the systems running in the real-life environments.

In order to cope with this issue, a real-life data for the creation of filters and events is used. The source of the real-life data is the log data from an Internet search engine. The idea to use the search engine data stems from the fact, that queries issued by users of a search engine bear a certain similarity to filters. A query contains data which summarizes the interest of a given user, which allows it to be regarded as a filter. Such user issued filters are matched by the search engine against its database of all known publications (web pages). Subsequently, results (links to the content of interest) are delivered to a user on the results web page.

Therefore, being given a log containing user queries issued to a search engine one can easily build a set of subscriptions. In the following experiments the data released on the 4th of August 2006 by the AOL search engine is used. The released data contains 20,000,000 search keywords for over 650,000 users

Attribute Name	**Operator**	**Attribute Value**
query	*substring*	anderson real estate
query	*substring*	www.xtremeteam.cus
query	*substring*	celia murphy
query	*substring*	donut pillow
query	*substring*	dicontinued dishes

Table 9.2: Example filters using the AOL data

spanning a 3-month period between March 1, 2006 and May 31, 2006 [PCT06]. From the above data set two sets of experiment data have been extracted: (1) set A: 43238 queries from 28 users with more than 4000 queries issued per user and (2) SET B: 77301 queries from 11276 users with exactly 10 queries issued per user. Listing 9.1 shows an exempt from the data log made available by the AOL.

For the filter creation only one predicate with attribute name set to *query* and operator set to *substring* has been used. The attribute value was set to the content of the query a given user issued to the search engine. Table 9.2 illustrates example filters created from the AOL data set. The format of queries presented in table 9.2 allows for an easy integration of queries into the predicate-based semantics of the XSiena system.

The main open issue remaining after the generation of filters is the generation of events. Being given a set of filters, there is no direct way to derive neither a publication pattern, nor the content of publications. The problem of the event generation originates in the lack of the feasible event domain. Being given a set of the user queries to a search engine, one is simultaneously given the filter domain. However, the straightforward use of the database of the search engine (the natural domain for the set of events) is not a practical choice for the testing of publish/subscribe system.

Therefore, a different approach towards the creation of the events has been assumed. Given the set of user queries issued to the AOL search engine every query in this set is reissued to the Google[3] search engine. Subsequently, for each query issued the number of hits returned by the Google search engine is recorded. Assuming that the size of the Google index equals approximately 2^{10} entries, one can divide the number of hits returned for the given query by the total number of available entries. The resulting number is a probability that a randomly issued publication from the set of all content indexed by the search engine will match the given query.

This in turn allows to devise a simple algorithm for the creation of events:

[3]`http://www.google.com`

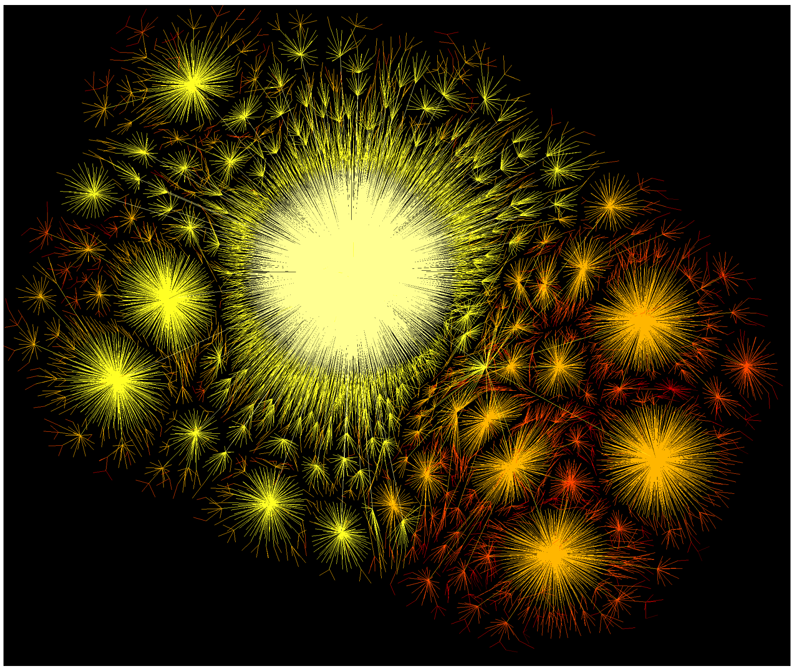


Figure 9.8: Routing tree layout – set A

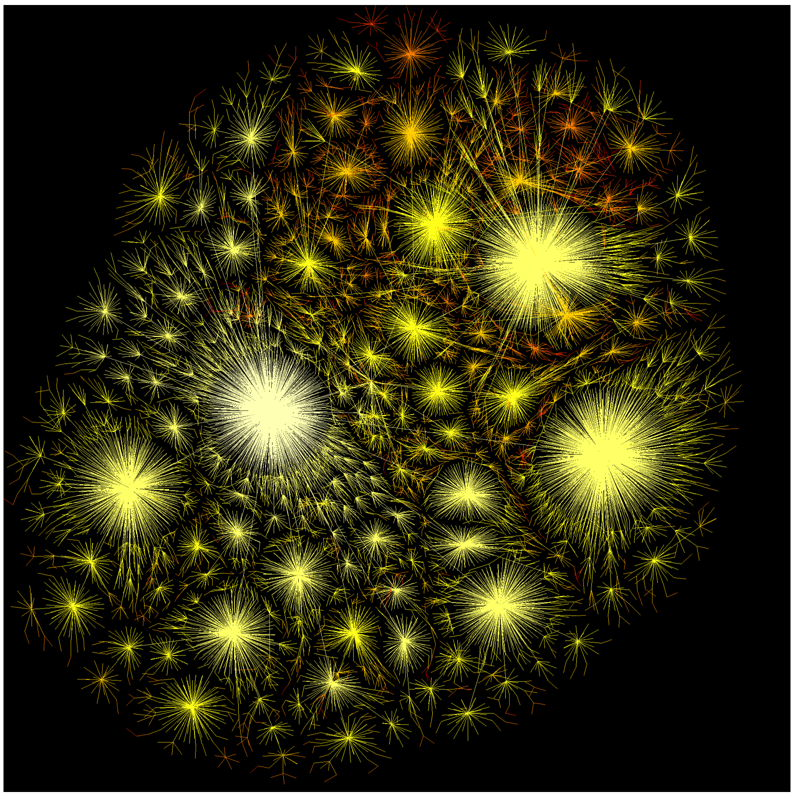


Figure 9.9: Routing tree layout – set B

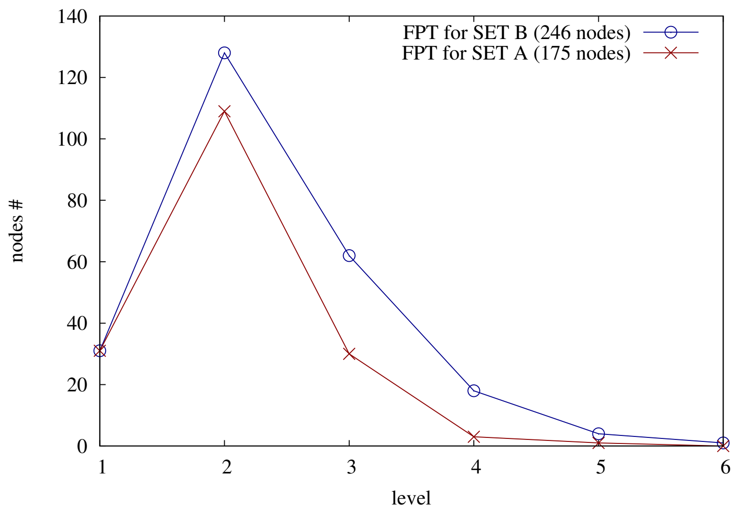


Figure 9.10: Forwarding tree sizes – set A and set B

1. for every filter contained in the set of filters
    (a) get the probability (0.0–1.0) of an event matching a given filter (calculated using the Google search engine query)
    (b) generate a random number (0.0–1.0) using a uniform distribution
    (c) if the random number is smaller than the calculated probability add the content of the filter to the event

such generated events consists of strings containing filter queries, which would match filters (using the *substring* operator) with a probability proportional to the number of matches a given filter returns when issued into a search engine. The so generated sets of events and filters have been used in the subsequent tests.
For thee first test the routing trees using the filters from set A and B have been created. Figure 9.8 shows the routing tree after insertion of unique filters from the set A, while figure 9.9 shows the routing tree after the insertion of the unique filters from the set B. The lighter the color of the edge, the closer it is placed to root of the routing tree. It can be observed that large number of different users (set B) tends to have more contention in their queries, compared to more differentiated queries (set A) coming from the smaller user base.
Figure 9.10 shows the average sizes of the forwarding trees obtained by matching events with routing trees obtained for the set A and set B. The sizes of the forwarding trees constitute approximately 1‰ of the number of filters stored in the routing tree. It can be observed that the smaller user base with larger amount of queries per user (set A) results in a smaller average forwarding tree. This can be contributed to the fact that the smaller user population has a more

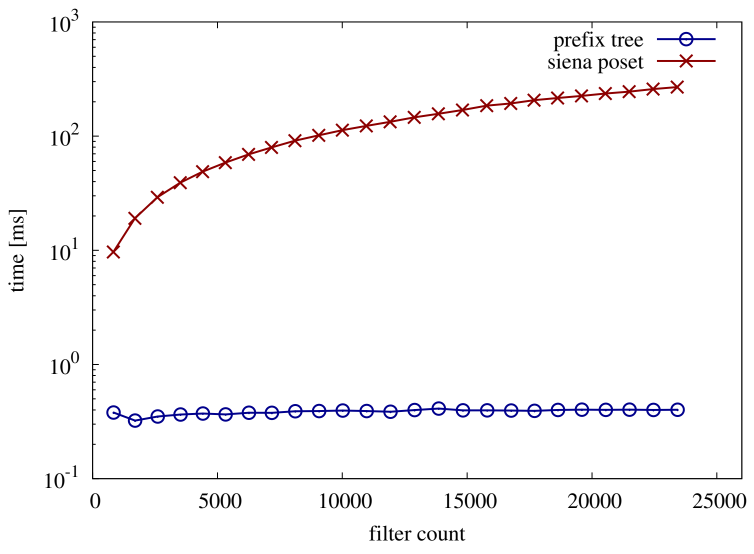


Figure 9.11: Event forwarding speed

widespread interest than the large user group which uses the search engine only occasionally.

The following three tests compare the event forwarding speed, the filter insertion speed and the forwarding tree extraction speed. Figure 9.11 shows the time needed to forward single event using the forwarding tree piggybacked by the event compared to the standard content-based forwarding using the SIENA poset – see Section 2.5.5. It can be observed that with increasing number of filters stored in the routing structure the SIENA-based poset is outperformed by the order of two magnitudes by the forwarding tree. The filter insertion time illustrated in Figure 9.12 shows that the need to maintain links between multiple filters in poset gives the routing tree a noticeable speed advantage. Moreover, per predicate insertion is more efficient in comparison to poset structure where large filters have to be compared multiple times with each other. The last Figure 9.13 compares the speed of the content based matching when routing trees and SIENA-based poset is used. One can observe, that the content-based matching using the routing tree is only marginally faster from the poset-based matching, probably due to the fact that identical predicates are not stored in the routing tree. On the other one has to recall that the content-based matching using the routing tree implies the construction of the forwarding tree for the event being matched, which in turn is not necessary in the case of the poset-based approach.

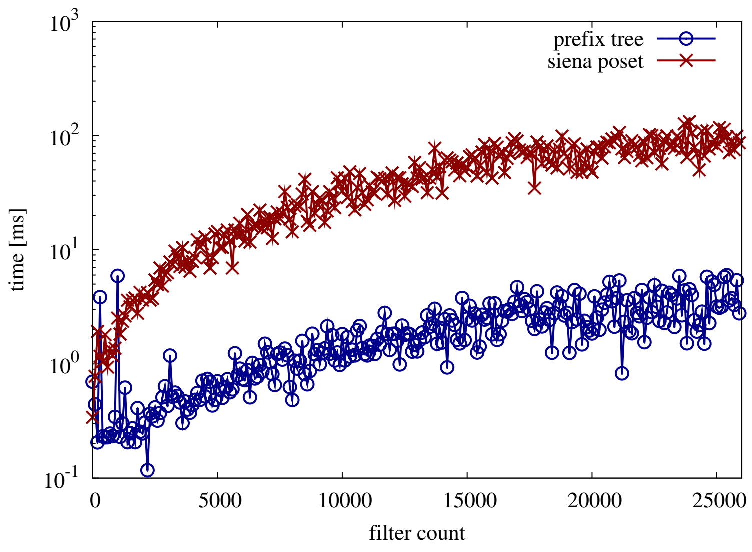


Figure 9.12: Poset versus routing tree creation time

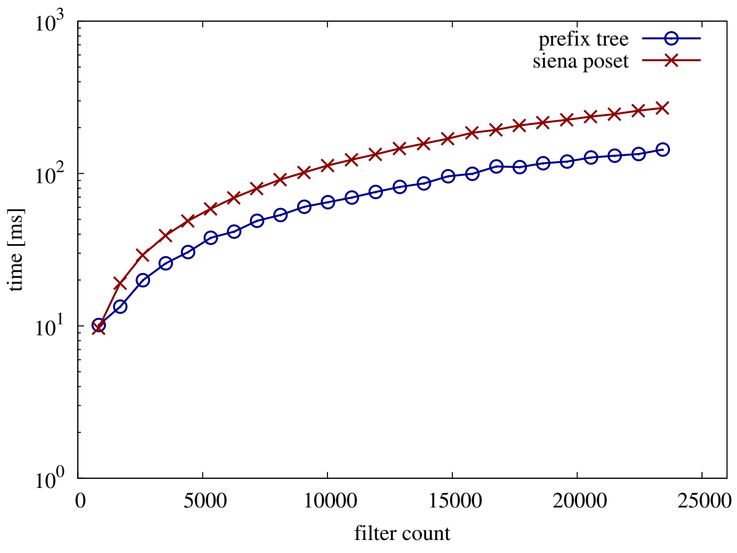


Figure 9.13: Content-based matching using routing tree

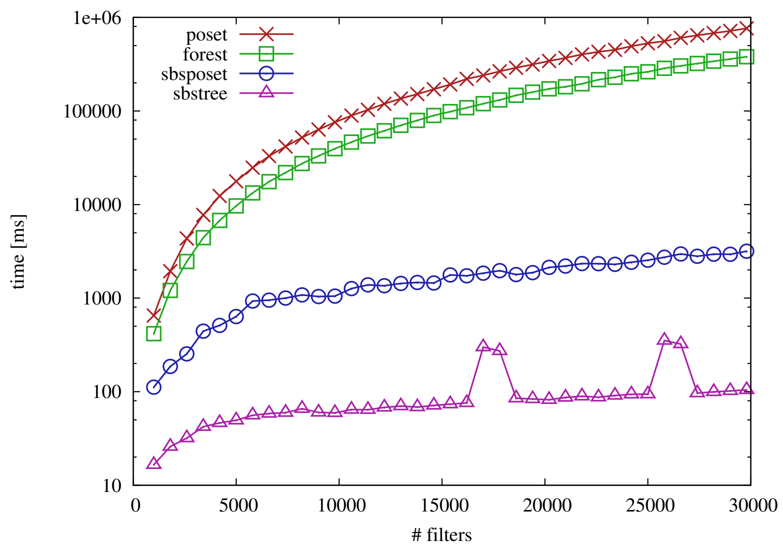


Figure 9.14: Filter insertion (routing) time

9.2 Bloom Filter-Based Routing

The evaluation of the Bloom filter-based routing is based on the XSiena system implementation which takes advantage of the `sbsposet` and `sbstree` data structures. The XSiena-based implementation of the Bloom filter-based routing is a part of the StreamMine [JF08a, FBFJ] project. StreamMine project[4] is developed within the Scalable Automatic Streaming Middleware for Real-Time Processing of Massive Data Flows[5] (STREAM) framework developed within the Seventh Framework Programme for Research and Technological Development (FP7) of the European Commission.

The evaluation of the `sbsposet` and `sbstree` data structures had been contrasted with two standard implementations of content-based routing publish/subscribe systems. The first one being the traditional Siena routing algorithm based on the poset structure [CRW01]. The second one being a recently published improvement implementation based on the forest data structure [TK06]. All three routing strategies (Bloom filter-based, poset-based and forest-based) were implemented in the XSiena system so as to create a uniform test environment operating within the same communications framework and with identical message semantics. All experiments have been executed on the same hardware and software: Java HotSpot(TM) Server VM (build 1.6.0_03-b05, mixed mode) running on a DELL Optiplex 745 with Core 2 Duo E6400 processor and 2GB of RAM. All workloads used in this section assume the approach outlined in Section 9, with a distinct difference that every subscription message carrying a filter originates from a different subscriber/interface.

[4]`http://streammine.inf.tu-dresden.de/`

[5]`http://www.streamproject.eu/`

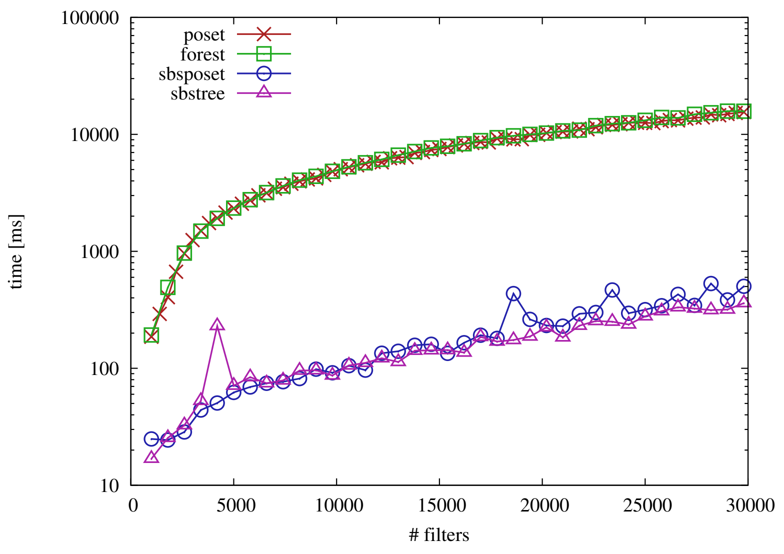


Figure 9.15: Event matching (forwarding) time

The first test was aimed at verifying the filter routing speed using the `sbsposet` and `sbstree` data structures. Figure 9.14 shows the time needed to insert a given number of filters into different routing structures. The `poset` data series represents the original SIENA poset [CRW01]. The `forest` data series represents the improved implementation presented in [TK06]. The `sbsposet` and `sbstree` data series represent the Bloom filter-based routing structures. In Section 5.3, it has been shown that only after insertion into both `sbsposet` and `sbstree` structures a filter can be used for matching against incoming events. It can be observed that although the original SIENA data structure has the worst performance, the difference between the more optimal `forest` and `sbstree` combined with `sbsposet` is still of an oder of magnitude. The poor performance of the SIENA poset is further stressed by the fact that its routing structure allows only for hierarchical networks of brokers, while both `forest` and Bloom filter-based structures are designed to work with arbitrary acyclic graph topologies.

The test was performed using the Pareto distribution for the attribute names and attribute values. The attribute names were selected from the set of 20.000 attribute names using the `aspell` word list. The number of filter predicates varies between 2 and 5 – using the uniform distribution. Attribute values are selected using normal distribution from the range $[0, 200000]$. All tests assume a pessimistic scenario, i.e., that every subscription message arriving at the broker originates in a different subscriber/interface.

The second test, allows us to verify the applicability of the Bloom filter-based routing approach towards the fast forwarding of events. Figure 9.15 shows the event matching speed, e.g., time needed to match 1000 events with data structure storing indicated number of filters. In case of traditional content-

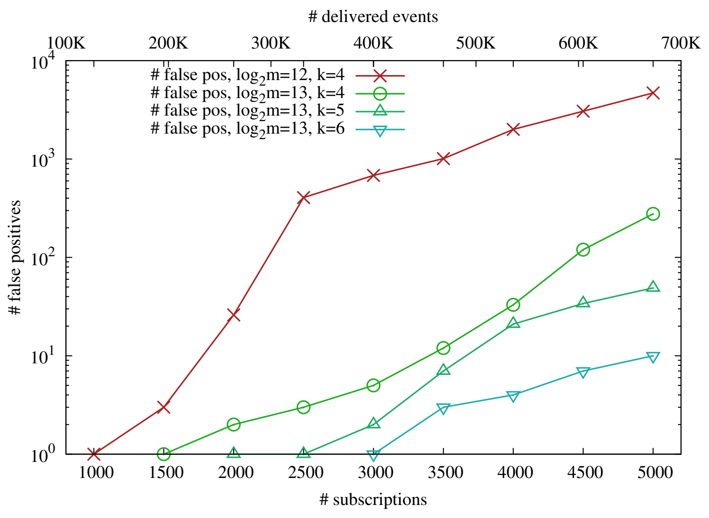


Figure 9.16: False positives probability

based routing, the matching time is defined as the time needed to calculate the forwards set for an event. In case of the `sbsposet` it is a time needed by the broker to assign a Bloom filter to an event and in case of the `sbstree` it is a time needed to calculate the forwards set for an event given its Bloom filter. Figure 9.15 shows results for the same set of filters as Figure 9.14. The number of event attribute name and value pairs varies between 2 and 5 – using the uniform distribution. Event attribute names are selected from a set of 20.000 attribute names using Pareto distribution. Event attribute values are selected using normal distribution from the range $[0, 200000]$.

In all cases, the matching time using Bloom filter based structure (`sbsposet`) is significantly faster than the matching time for traditional SIENA `poset` and `forest`. The difference in the performance improvement can be attributed to the way the `sbsposet` is built: attribute names are stored in a hash set and serve as pointers to the lists of respective attribute constraints posets. Clearly, with increasing number of different attribute names the advantage of the `sbsposet` is more substantial as with a single lookup in a hash set it can filter out more irrelevant predicates. One can also observe that the matching speed using the `sbstree` follows with the same performance.

The main advantage of using Bloom filters to represent the sets of predicates matching an event is the fact that they are size-bounded and can represent the whole universe of predicates by having all bits set. However, this feature does not come without a price. In case of the Bloom filters, this is the possibility of false positives. As illustrated in Figure 5.3 the false positives probabilities for small filters are relatively high, even for a low number of elements.

The false positives in a Bloom filter result in false positives in the publish/subscribe system using Bloom filters for routing. In this case a false positive is defined as delivery of an event which was not subscribed to. In order to verify that issue, 10,000 events were routed through a broker which stored an increasing number of filters. After the delivery of events, it has been checked whether they have been really subscribed for. This experiment, assumed a pessimistic case where every subscription message is issued by a different subscriber. In order to maximize the number of matches, subscriptions had only one predicate selected using a Pareto distribution from 2,000 English words. Events were constructed in a similar way: every event contained from 1 to 10 attribute name and value pairs (uniform distribution), and attribute names were chosen from 2,000 English words using the Pareto distribution. In all experiments, according to the expectations, the number of false negatives was equal to zero.

Figure 9.16 shows the number of false positives in the function of the number of subscriptions in the broker. At first one can observe that the number of false positives is lower than it would be assumed when looking at the Figure 5.3. For example the number of false positives events for 5,000 subscriptions, $\log_2 m = 12$ and $k = 4$ is equal to 4695, which is less than 50% of the sent events. Looking at the Figure 5.3 one might expect to receive almost exclusively false positives. This contradiction might be explained by three facts: (1) probability of false positives is lower if one considers multiple predicates forming a subscription, as their conjunction results in a multiplication of false positives probabilities; (2) the high number of matches in the experiment results in false positives being covered by non-false positives, thus reducing their number; (3) filters generated with Pareto distribution will tend to have more identical predicates, reducing the number of different elements.

Figure 9.17 allows us to estimate the average overhead for the transmission of an event when Bloom filter-based routing. It can be observed that for the test scenario depicted on the Figures 9.14 and 9.15 an average event will have up to 200 bits set in its Bloom filter. Using sparse Bloom filter representation (see Section 5.2.3) with 16 bit integer values used as bit indices would result in the average overhead of 400 bytes per event. It is the believed by the author that the increase in the event size is justified and offset by the improvement in the forwarding speed.

9.2.1 Counting `sbstree`

Figure 9.18 illustrates the test which highlights the importance of the counting `sbstree` optimization. In the illustrated test the average number of bits set per event has been varied, whilst maintaining a constant number of filters in the original `sbstree` and counting `sbstree`. The average number of bits set per event e corresponds to the average number of predicates matching the event e

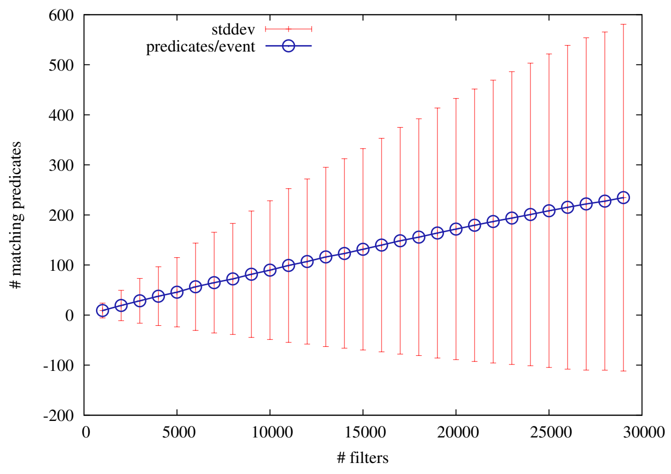


Figure 9.17: Number of predicates matching a single event

times the number of bits per predicate *k*.

It can be observed that the matching time in case of the original `sbstree` (indicated as `sbstree` in the Figure 9.18) increases exponentially with the number of bits set in the event. The optimized variant (indicated as `counting` in the Figure 9.18), using a counting algorithm is linear with the number of bits set implying the correctness of the proposed matching algorithm.

Figure 9.19 illustrates the matching time with stable number of bits set per event. It can be observed that the matching time of the counting variant of the `sbstree` is practically linear with respect to the number of filters stored in the counting `sbstree`. This confirms the observation that the matching time is proportional to the number of bits set in the event. The original variant of the `sbstree` is, on the other hand, linear with the number of filters stored in the `sbstree`, despite the stable number of bits set in the event.

9.2.2 System benchmarks

In order to evaluate the performance of the XSiena system as a fully functional, content-base publish/subscribe communication middleware a series of tests involving a distributed network of physical brokers running the Bloom filter-based XSiena system has been performed.

Figure 9.20 illustrates the experiment setup in three different scenarios. First, a varying number of brokers (**B1**, **B2**, **B3**) has been connected to form the acyclic network presented in Figure 9.20. Subsequently, the subscriber **S1** issued a single subscription which was propagated through the whole network. Finally publisher **P1** was issuing events (the flow of events is depicted by arrows in the Figure 9.20) which were always received by the subscriber $S1$,

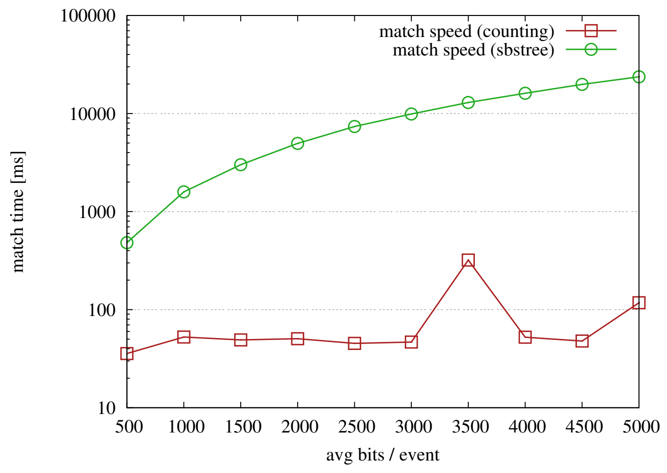


Figure 9.18: Event forwarding with increasing number of bits set per event

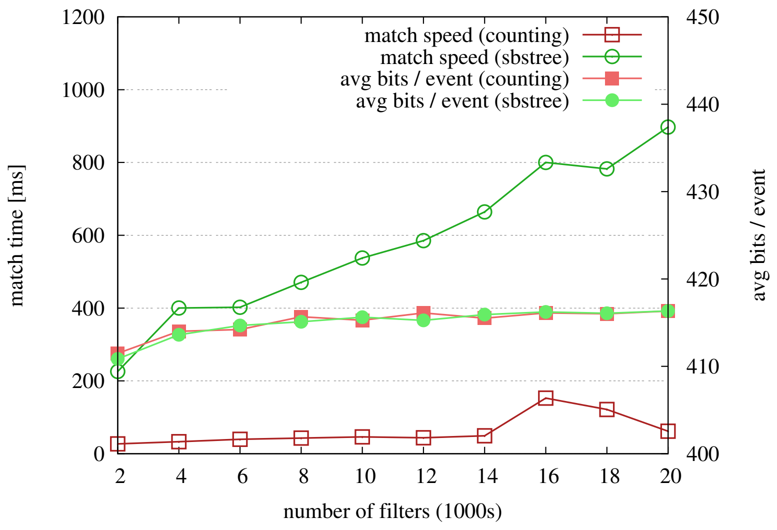


Figure 9.19: Event forwarding with constant number of bits set per event

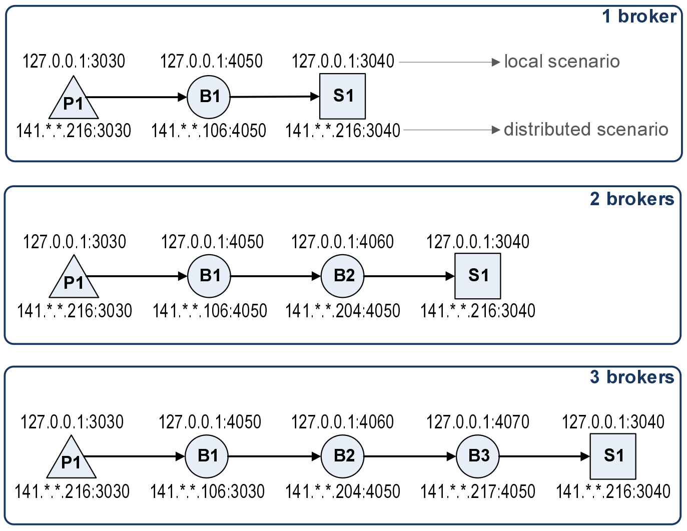


Figure 9.20: Benchmarks setup

i.e., events issued by **P1** always matched the filter issued by the $S1$. An event consisted of ten different attribute name and value pairs, with the total wire size of approximately 740 bytes (including the Bloom filter).

Figure 9.21 compares the latencies of the TCP-based communication for different number of brokers. For the latency measurement, publisher attached a send time stamp (using the Java `System.nanoTime()` method) to every event it has published. The subscriber, upon reception of the event calculated the reception time stamp and logged the difference between the two. The source code for the latency experiment is presented in Section 8.1. Obviously, for the above scenario to work, both publisher and subscriber had to be placed on the same physical machine (`141.*.*.216`). It can be observed that for the local scenario, even with three brokers placed between the publisher and the subscriber, the majority of the latencies stays in the sub-millisecond range. The exposed latencies are a good starting point towards meeting of the requirements of the modern real-time event processing systems [ScZ05].

The second important parameter which had to be considered was the ability to provide high throughput for the events. Figure 9.22 presents the throughput tests for two scenarios: (1) communication via sockets with all brokers running on the same physical host (`local`) and communication via sockets with all brokers running on different physical hosts (`distributed`). It can be observed that the throughput of the distributed runs approaches the network saturation values (100BASE-T full duplex Ethernet). For the local setup, the overhead of running the multiple brokers on the same host results in the observable throughput decrease.

A series of throughput tests for the communication via direct upcall has been also performed – see Figure 9.23. The direct upcall test simulated two hundred

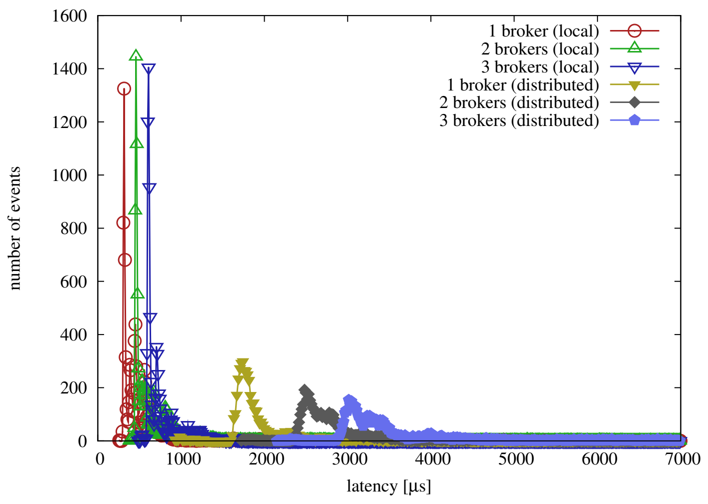


Figure 9.21: Latency for varying number of brokers

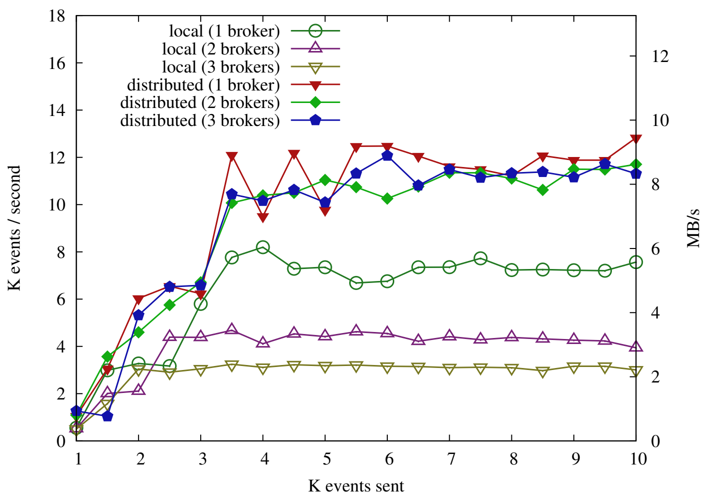


Figure 9.22: Throughput for varying number of brokers

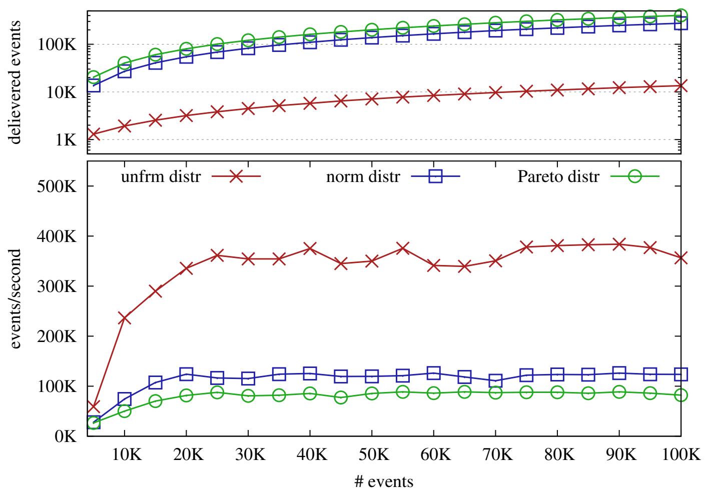


Figure 9.23: Local throughput for varying distributions

different subscribers registering their filters with the XSIENA broker. Three tests with varying number of events matching the registered filters have been performed. Each of the tests was performed with one of the following distributions: (1) uniform (least overlap), (2) normal (medium overlap) and (3) Pareto (highest overlap). The type of the distribution chosen strongly influences the total number of the delivered events – see the upper part of the Figure 9.23. The 300000 delivered events in case of the Pareto distribution indicates that on average a single event was delivered to three components. It can be observed that the achieved throughput in all cases should satisfy even very demanding applications.

9.3 Fail-Awareness

The evaluation of the fail-aware XSIENA system is divided into three parts – first the issues related to the clock drift rate are investigated. Specifically, the behavior of the time stamp counter, accessible via assembler command `rdtsc` was evaluated. The time stamp counter is used throughout the remainder of the experiments as the local hardware clock. The second part of the experiments focuses on the use of the upper bound and the influence of the optimizations discussed in Section 6.2 on the precision of the upper bound. In the last part of the evaluation the focus is put on the comparison of the upper bound method and the external clock synchronization using the NTP [Mil92] protocol. Specifically, the maximum error values of both methods are compared.
The upper bound measurements and their comparison to the results obtained

via the NTP protocol has been performed in the PlanetLab environment [Ros05, PBFM06]. Distributed applications deployed in the PlanetLab environment i are usually subject to a much harsher conditions (with respect to experienced transmission delays and omission failures) than the same application deployed on legacy hosts. The reason for this fact is that in PlanetLab environment physical hosts are shared by many users which often leads to unexpected spikes in the transmission or precessing delays. This is, however, ideal for our purpose because fail-aware XSiena system is specifically targeted towards such "harsh" environments.
All communication between the nodes of the fail-aware XSiena system deployed over the PlanetLab is carried out using the UDP protocol. For the following experiments, two sets of hosts have been used. First set is named *local*, the second is named *global*. Hosts belonging to the *local* set were selected from within German PlanetLab nodes placed in: Bremen, Dresden, Berlin and Passau. Hosts belonging to the *global* set were placed in: Delhi (India), Helsinki (Finland), Krakow (Poland) and Saarbrücken (Germany).
On each host a single fail-aware XSiena broker process has been started. Additionally, on each underlined host a single subscriber and a single publisher process was started as well. Each underlined host acted as a starting and ending point for any event which was also routed via remaining hosts forming a closed ring. In this way, when measuring an upper bound on the transmission delay, one could compare it against the real-time transmission delay. The real-time transmission delay was calculated using the time stamp counter of the underlined host.

9.3.1 Clock Drift

Since the evaluation of the fail-aware XSiena system is carried out in the PlanetLab environment, one must first investigate whether PlanetLab nodes used for experiments have time stamp counters which expose drift rates stable enough to allow for their use as the local hardware clocks.
In order to measure the drift rate of the time stamp counters the code of the SNTP [Mil06] client has been modified. The modifications included using the time stamp counter as the time source for the client application so that one could compare it against real time stratum 1 servers[6]. The measurements were performed every hour – the minimum frequency NTP stratum 1 servers allowed for.
Figure 9.24 shows the drift rate of the three PlanetLab hosts over the period of ten days at the end of March 2009. It can be observed that two hosts (`ssvl.kth.se` and `iit-tech.net`) expose relatively stable drift rates, while

[6]The NTP stratum levels define the distance from the reference clock and the associated accuracy. Stratum 1 servers are computers attached directly (e.g. via RS-232) to stratum 0 devices, such as atomic clocks or GPS clocks.

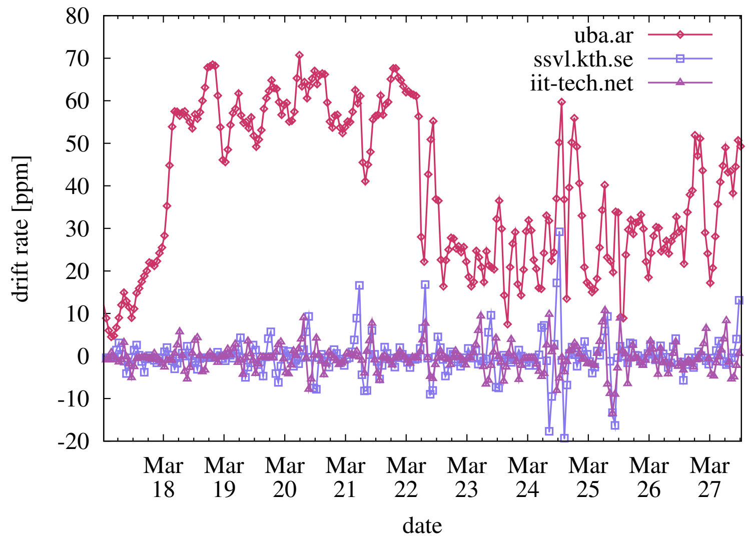


Figure 9.24: Drift rates of three PlanetLab hosts

the host `uba.ar` exposes has large fluctuations of the drift rate.
Figure 9.25 shows the drift rate of a single (`cs.princeton.edu`) PlanetLab host. The drift rate of the host's time stamp counter has been measured simultaneously against three NTP stratum one servers: `t1.timegps.net`, `nist1.symmetricom.com` and `swisstime.ethz.ch`. It can be observed that the drift rates stay within a few parts per million of each other which allows to assume that the drift estimation method can be relied upon. A reliable drift estimation technique confirms that time stamp counters are a valid source of local time for the nodes of the fail-aware XSiena system.
Some architectures using ACPI compliant power management, e.g., AMD K8 multi-core, might expose drift issues related to performance state (P-state) and power state (C-state) changes. Suggested solution to this issue might be switching from time stamp counter to other counters, unaffected by the power management, such as HPET [Cor04] or PMTimer [Nag06]. Another problem affecting multi-core systems is context switching when subsequent `rdtsc` calls are executed on two different cores with unsynchronized time stamp counters. A solution to that problem might be the new `rdtscp` instruction [Bru04] or the time stamp counter synchronization present in current operating systems.

9.3.2 The Upper Bound on Transmission Delay

Our tests of the upper bound on the transmission delay focus on the comparison of the calculated upper bound and the real time transmission delay in the PlanetLab system. Specifically, it is important to investigate whether the upper bound properties hold – see Equation 6.1. The experiment has been performed the experiments in two different scenarios, varying especially in the terms of

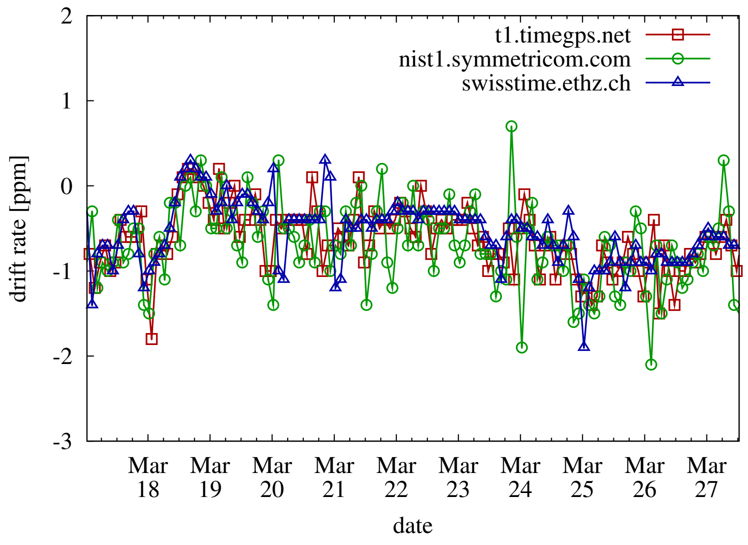


Figure 9.25: Drift rate of a single PlanetLab host

the experienced network delays.

Figure 9.26 shows the comparison of the calculated upper bound and the real-time transmission delays for the *local* hosts. It can be observed that the calculated upper bound on transmission delay holds, i.e., it is never lower than the real transmission delay. Moreover, it can be observed that fail-aware XSiena system maintains low error values (defined as difference between transmission delay td(m) and the upper bound ub(m)) regardless of the transmission delay variations. Moreover, the helper optimization mechanism (see Section 6.2) helps to quickly lower the initial, high error values. For comparison, the same measurement performed with an unoptimized (no helper message optimization, no lower bound on helper message computation) fail-aware XSiena system version has been included – see Figure 9.27. It is clearly visible that in case of an unoptimized protocol error values depend directly on the message transmission time and are much higher than in the optimized version.

Figure 9.28 illustrates the relation between the upper bound on the transmission delay ub(m) and the upper bound on the processing time ub(pt(m)) of the events forwarded in the *local* setup. It can be observed that although the upper bound on transmission delay is generally higher than the upper bound on the processing time, it is not uncommon for the nodes to experience sudden spikes (three orders of magnitude) in the processing time of events. Such spikes can be contributed to the scheduling algorithms and interference from other slices on the PlanetLab nodes on which the experiment was executed.

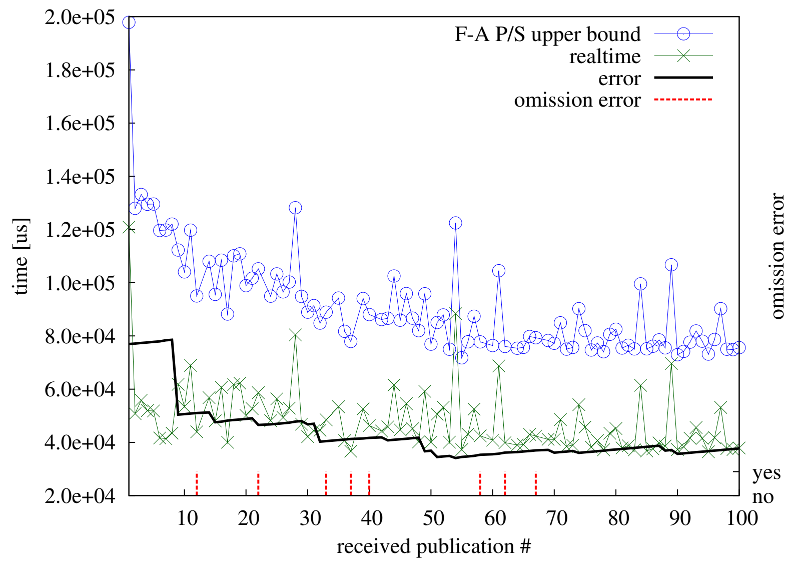


Figure 9.26: Optimized boot up of the upper bond – *local* setup

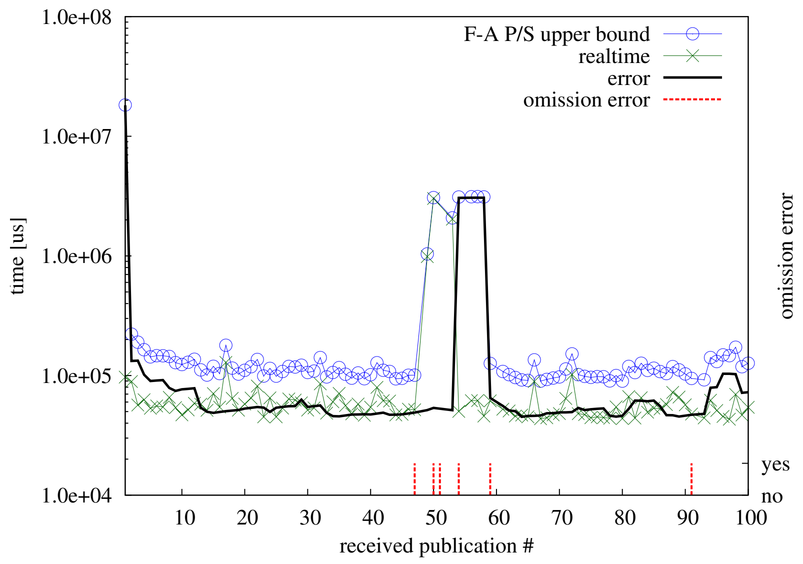


Figure 9.27: Unoptimized boot up of the upper bond – *local* setup

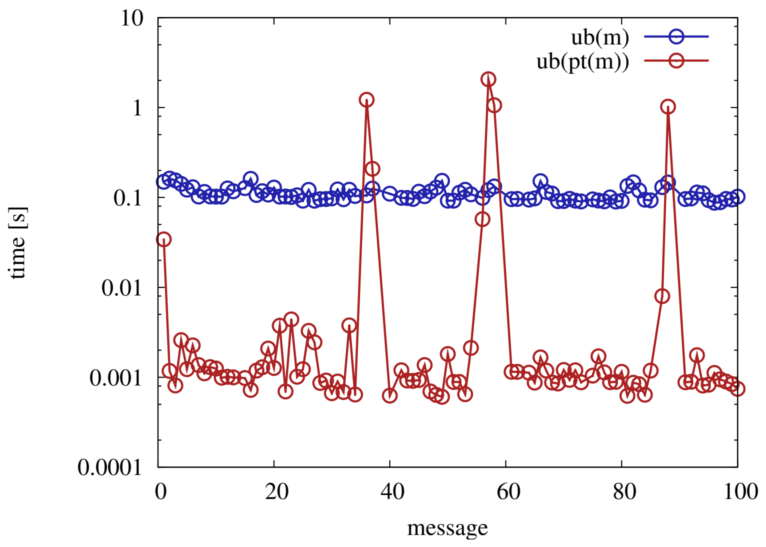


Figure 9.28: Upper bound on transmission delay and processing time

9.3.3 Upper Bound versus NTP

The following set of experiments evaluates the real-time error of the upper bound on the transmission delay and compares it to the error provided by the NTP service. The intuition behind the error value is that it represents how precisely it is possible to read a remote clock. In other words, measuring message transmission time can be either performed using the upper bound technique with the given upper bound error, or it can by performed by subtracting receive and send time stamps of the processes synchronizing their clocks with NTP with the given NTP error.

The error of the upper bound is calculated as the difference between the upper bound on the message transmission delay and the real-time transmission delay. The error provided by the NTP is defined as: *the root dispersion plus one-half the root delay* [. . .] *increased by a small amount (time_tolerance) each second to reflect the clock frequency tolerance* [Mil94]. The root dispersion is defined as: *maximum error relative to the primary reference source at the root of the synchronization subnet* [Mil94]. The NTP error values are obtained via the `ntp_gettime()` system call. It is important to note that the error provided by the NTP, is not the upper bound error value, i.e., it is possible that the actual error is higher than the one provided by the NTP.

Figures 9.29 and 9.30 show the error of the fail-aware XSiena publish/subscribe system and its comparison to the NTP error, respectively. Both experiments have been performed in the *local* setup. It can be observed that the majority of the error values in case of the fail-aware XSiena system is placed around 50 milliseconds mark. The comparison with the NTP error clearly illustrates that the upper bound provides a much tighter (over 1 second) estimation.

Figures 9.31 and 9.32 demonstrate the same experiment for the *global* scenario.

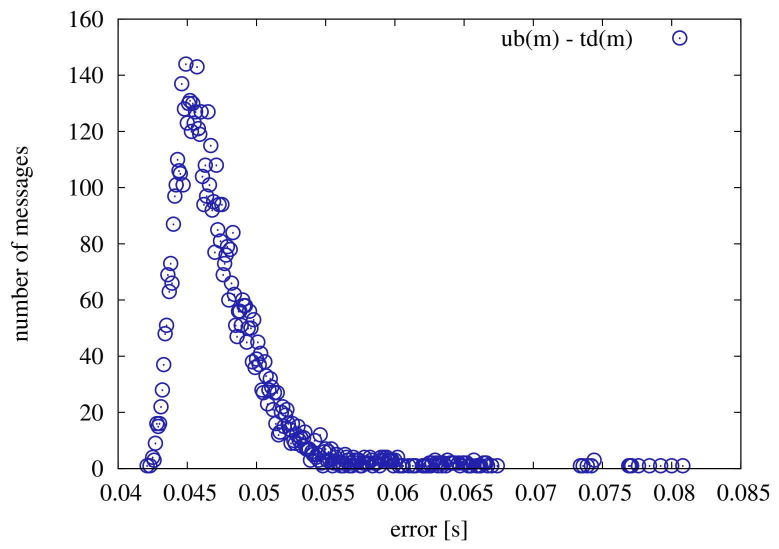


Figure 9.29: Upper bound on transmission delay error – *local* setup

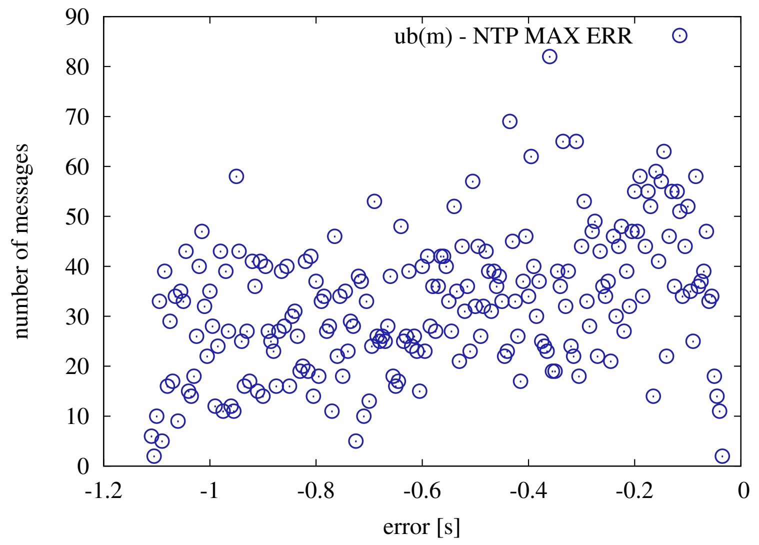


Figure 9.30: Upper bound versus NTP – *local* setup

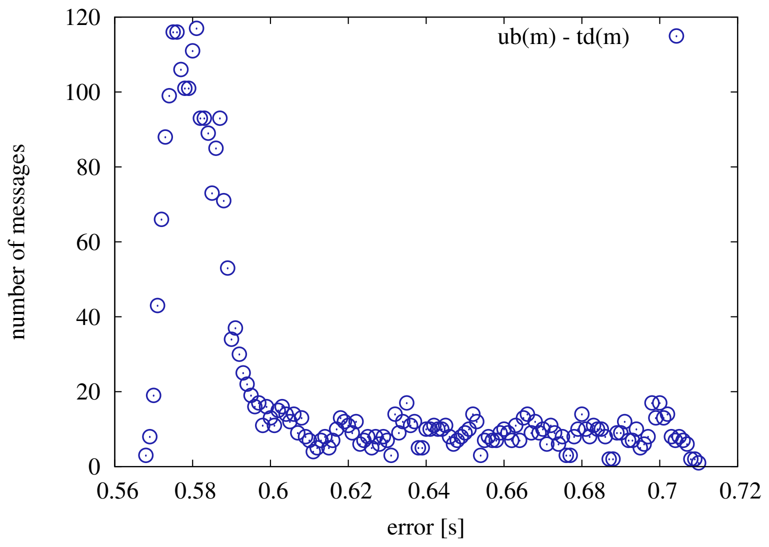


Figure 9.31: Upper bound on transmission delay error – *global* setup

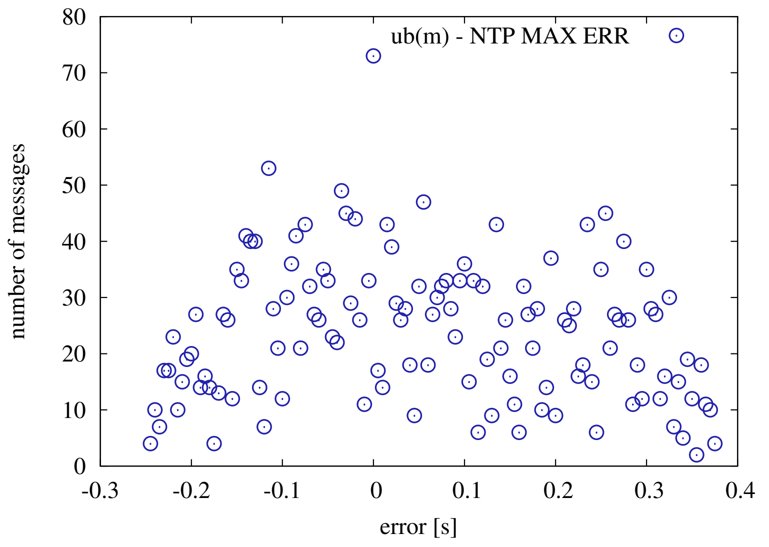
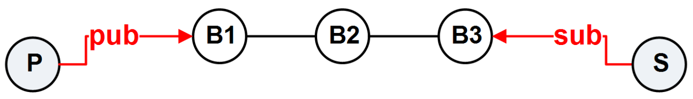
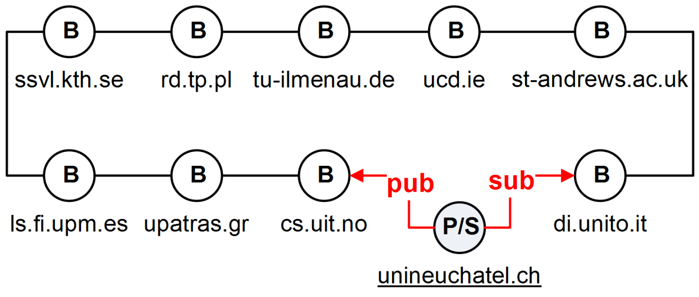


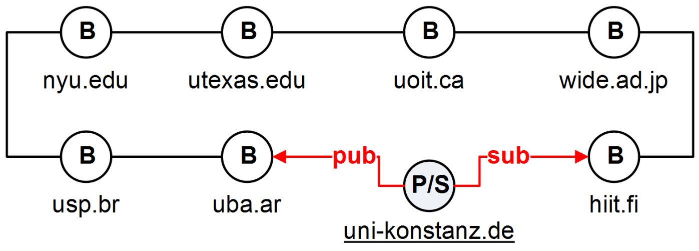
Figure 9.32: Upper bound versus NTP – *global* setup



(a) *local* setup – all nodes on the same physical host



(b) *ipeurope* setup – European PlanetLab hosts



(c) *ipworld* setup – planetary PlanetLab hosts

Figure 9.33: Different network setups for the soft state XSiena experiments

It can be observed that due to the increased transmission delays (an order of magnitude) the tightness of the upper bound has also worsened. Nonetheless, even though both approaches provide similar precision it must be remembered that the fail-aware XSiena system provides guarantees as to the upper bound values which are not given by the NTP.

9.4 Soft State

The evaluation of the soft state XSiena publish/subscribe system has been performed in three basic setups – see Figure 9.33. In the *local* setup (Figure 9.33(a)) all XSiena nodes have been started on the same physical machine and connected via TCP/IP network. This scenario was motivated by the fact that sharing the same physical machine implies sharing the same hardware clock, which allows to measure the message transmission times by subtracting the send and receive times. Such measurement based on the local clock allows for the verification of the upper bound values for every node. In the *local* setup one subscriber **S** and one publisher **P** were connected by three brokers: **B1**, **B2** and **B3**.
The *ipeurope* setup (Figure 9.33(b)) involves 10 PlanetLab hosts placed in

Europe, connected via TCP/IP. One host (`unineuchatel.ch`) is used to start both publisher and subscriber processes, so that one can compare the send and receive times of the messages using the local hardware clock. This scenario can be seen as an example distribution path in a large scale, content-based publish/subscribe system. Similarly, the *ipworld* setup (Figure 9.33(c)) includes hosts distributed across multiple continents.

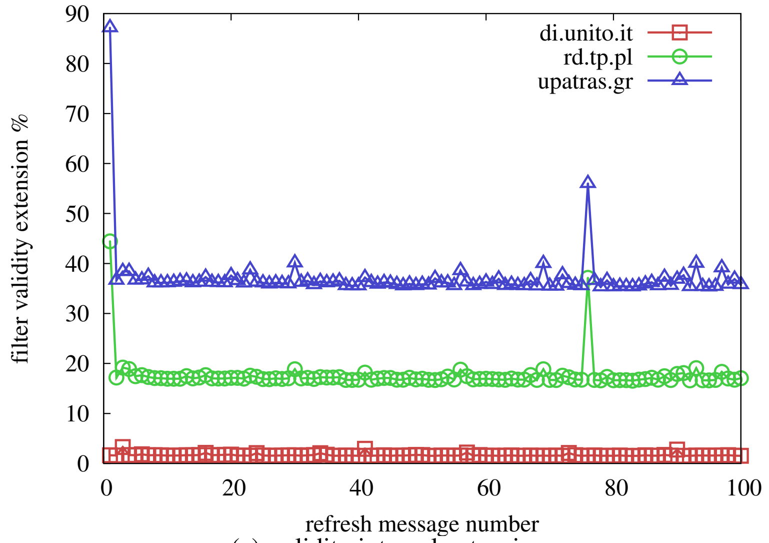
The first experiment used the conservative calculation of the validity interval extension (see Section 7.2.3) and aimed at visualization of the validity extension and resulting validity utilization in different brokers of the soft-state XSiena system – see Figure 9.34(b). For this experiment the filter validity interval T_I was set to 1 second and the refresh period τ to 0.8 seconds. In a system with zero transmission and processing delays such settings would result in all brokers sharing the same validity interval utilization of 80%.

However, due to the non-zero delays, one can observe that the utilization of the subscription at the first broker encountered by the filter (`di.unito.it`) is close to the 80% as the upper bound on the filter propagation delay is much lower than the actual validity interval T_I. The utilization of the same filter at one of the last brokers on its path (`upatras.gr`) is around 60%. This can be explained by the fact that upper bound on the processing delay which is added to the filter's validity interval is much larger (due to the larger number of brokers traversed by the filter) and therefore the upper bound overestimation also grows. The growing overestimation for the brokers further from the message source compensates for a higher probability of a sudden increase in the message propagation delay, which is related to the number of hops a message has to traverse.

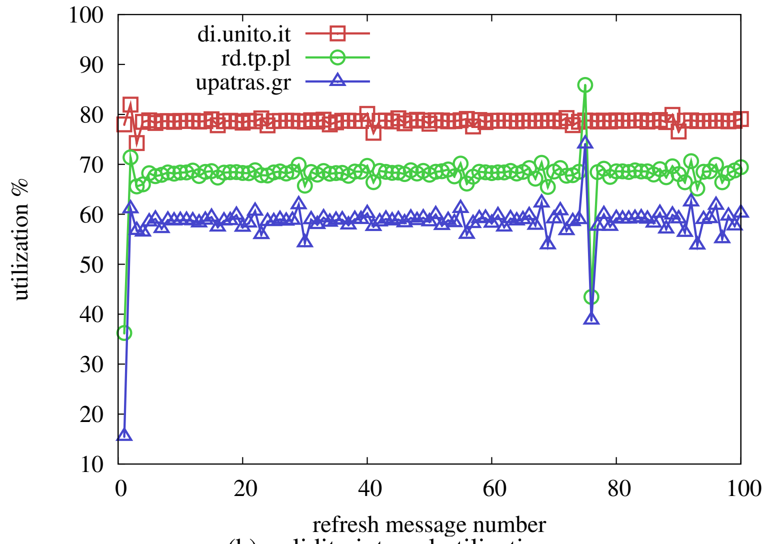
Figure 9.34(a) shows how for the same experiment (as in Figure 9.34(b)) different brokers extend the validity interval of a single filter. It can be observed that the first broker (`di.unito.it`) encountered by the filter barely extends its validity, while one of the last ones (`upatras.gr`) extends it by almost 40%. Remembering that the validity interval of the filter was set to 1 second, one can conclude that the average upper bound on the propagation delay of that filter when it arrived at `upatras.gr` node was equal to 0.4 seconds.

On Figure 9.34(a) one can also observe that the refresh message number 75 was significantly slowed down (upward spike) between the `upatras.gr` and `rd.tp.pl` nodes. This translates to the utilization of the filter refreshed by that message growing proportionally to the refresh message delay – see Figure 9.34(b). Subsequent refresh message arrived timely and therefore the utilization calculated for it significantly dropped – see downward spike in Figure 9.34(b). Since the utilization for message 75 remained under 100% no unsubscription due to the timeout of the validity interval has taken place.

The second experiment conducted using the soft state XSiena system aimed at verifying the uncertainty-based validity interval extension algorithm – cf Section 7.2.4. The experiment has been performed in the *ipeurope* setup, using

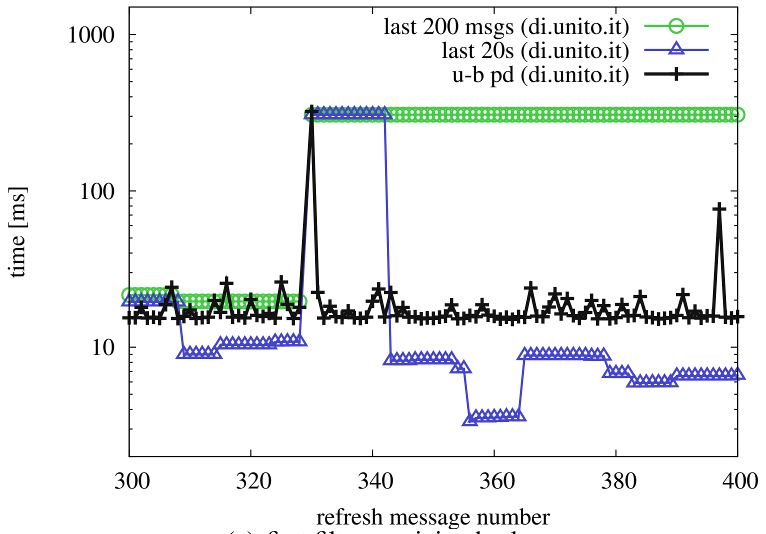


(a) validity interval extension

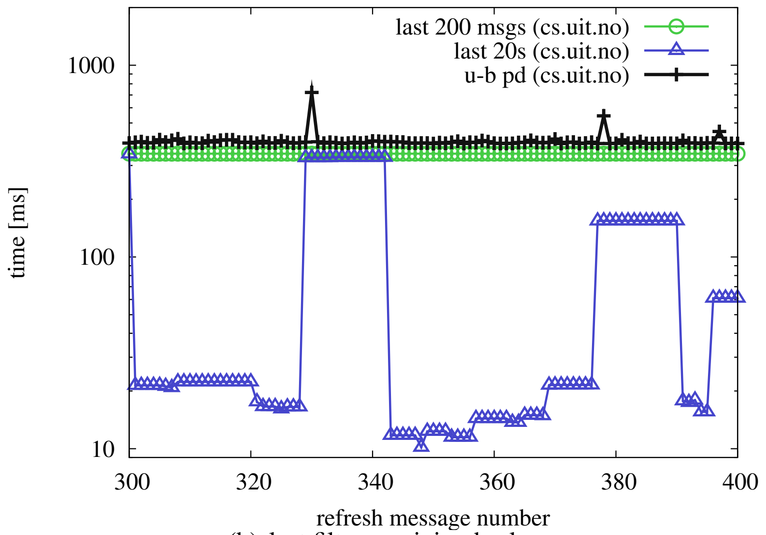


(b) validity interval utilization

Figure 9.34: Filter statistics for the *ipeurope* setup



(a) first filter receiving broker



(b) last filter receiving broker

Figure 9.35: Uncertainty-based validity interval extension

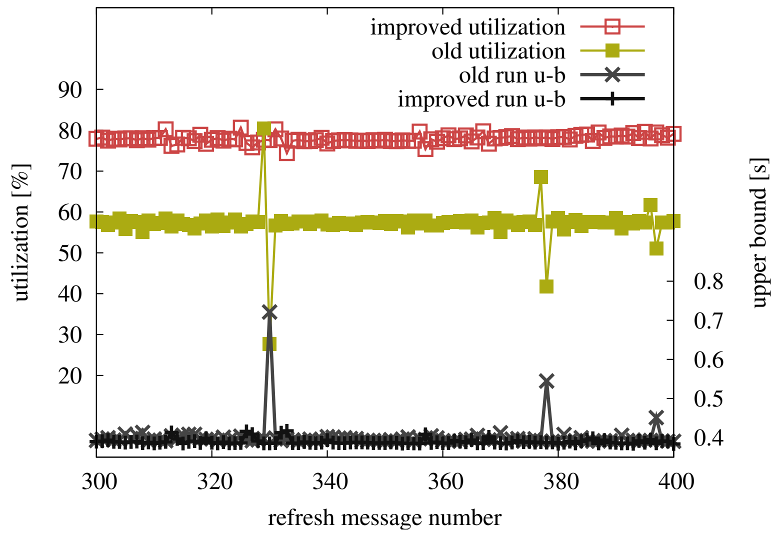


Figure 9.36: Comparison of uncertainty-based and upper bound-based utilization

the same filter validity and reissue setting as in the case of Figure 9.34 – filters were re-issued every 0.8 seconds and their validity interval was set to one second.

Figure 9.35 compares the validity interval extension calculated by the first broker encountered by the subscription message (Figure 9.35(a)) and by the last broker encountered by the subscription message (fig:eval:ss:uncert02). Comparing both cases one can observe that although the calculated upper bound on the propagation delay values are significantly different, the validity interval extension based on the uncertainty algorithm remains similar in case of the both brokers. Specifically, in the case of the Figure 9.35(b) it does not exceed the calculated upper bound on the subscription message propagation delay.

The validity interval extension has been plotted for two time windows: (1) using the last 20 seconds time window (`last 20s`) and (2) using the last 200 messages (`last 200 msgs`) time window. Since filter reissue period equaled 0.8 seconds the 20 seconds time window was more responsive than the 200 messages time window. It can be clearly see in case of Figure 9.35(a) where the increase in the interval extension quickly returns to normal in case of the time-based window, as opposed to the message count-based one.

Figure 9.36 compares the utilization of the two approaches: the validity interval extension based on the upper bound (`old`) and the validity interval extension based on the upper bound uncertainty (`new`). The data for the experiment were gather for the filters and their refresh messages on the `cd.uit.no` broker in the *ipeurope* setup. In order to gather the data two consecutive runs have been performed: one using the uncertainty-based extension and one using the upper

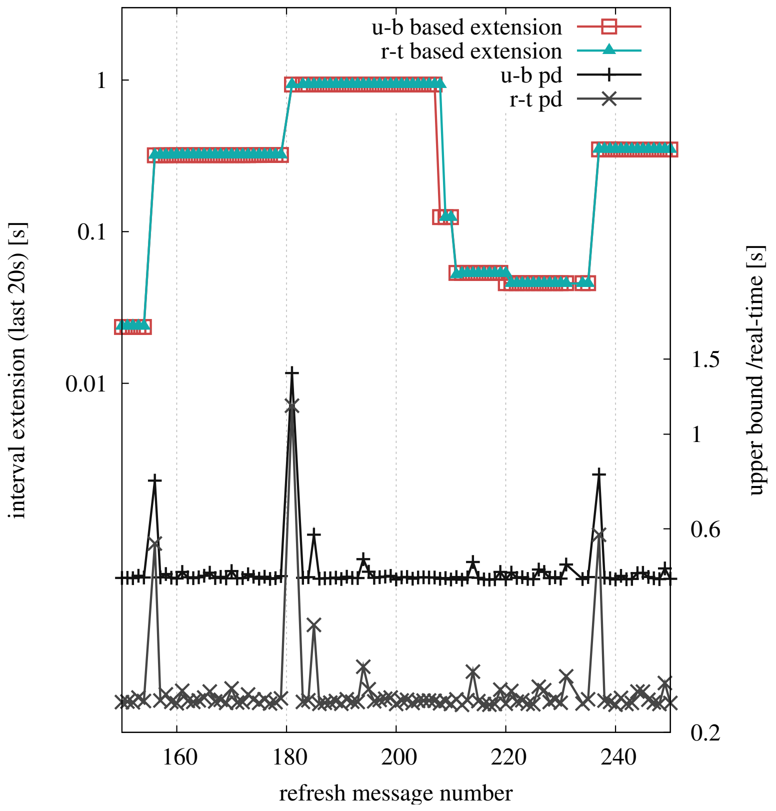


Figure 9.37: Upper bound-based and real-time-based uncertainty calculation

bound-based extension. In both cases the validity interval T_I of the filter was set to 1 second and the re-issue period τ was set to 0.8 seconds.

The relatively high upper bound values (0.4 seconds) in combination with the upper bound-based approach have contributed to the drop of the utilization from the application programmer specified 80% to about 60%. On the other hand, the uncertainty-based extension method retains the application designer specified utilization, despite upper bound values reaching 40% of the originally specified validity.

Figure 9.37 compares the uncertainty-based interval extension calculated using the upper bound on the propagation delay (`u-b based extension`) and the uncertainty-based interval extension calculated using the real-time propagation delay (`r-t based extension`). It is only possible to plot the above data for the host on which both publisher and subscriber are running. In the case of the *ipeurope* setup this is the `unineuchatel.ch` node. It contains both the source and the sink of the subscription messages, thus allowing to use its local hardware clock to calculate the real-time propagation delay of filters by subtracting the send time from the receive time.

It can observed that both calculated values are virtually identical, which confirms the upper bound as the right technique for use in the soft state XSiena

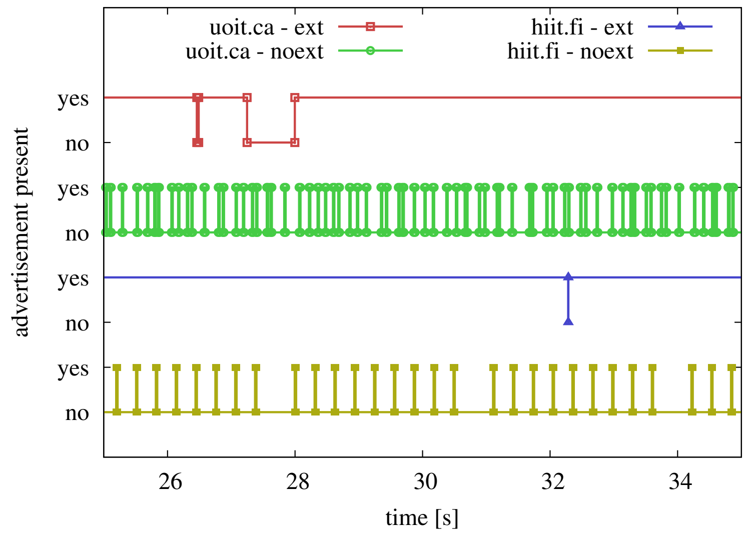


Figure 9.38: Advertisements expiration over time

system. Figure 9.37 shows also the actual values of the upper bound on (`u-b pd`) and the real-time (`r-t pd`) filter propagation delays. Both left and right y axes have logarithmic scale. The very well performance of the upper bound technique can be explained by the use of optimization techniques (see Section 6.2) and the resulting fact that the error of the upper bound is very stable especially is sudden increase in the message propagation delay is experienced – see Figure 9.26.

The last experiment compares the efficiency of the validity extension technique to the simple approach where no extension is used. For that experiment the *ipworld* setup has been used. During the experiment all brokers were connected using the TCP protocol. The measurement has been performed for the advertisement messages. Advertisement messages have been assigned the validity interval T_I of 20 milliseconds and the refresh period τ has been set to 16 milliseconds, which results in the 80% utilization.

Figure 9.38 shows the presence of advertisements in the two brokers of the *ipworld* setup. It can be observed that when no validity interval extension (`noext`) is used the advertisement presence in the brokers is very brief, with the validity interval expiration quickly causing an unadvertisement. In contrast, the validity interval extension technique (`ext`) assures that the expiration is postponed until the refresh message is able to arrive. Figure 9.38 illustrates also the fact that the further away from the source of the advertisement a given broker is, the lower the amount of time an advertisement is present at that broker when no validity interval extension is used.

Figure 9.39 plots the data from Figure 9.38 across a longer period of time as a percentage of time a given broker has spent without an advertisement. Time without advertisement implies the amount of time when broker is not

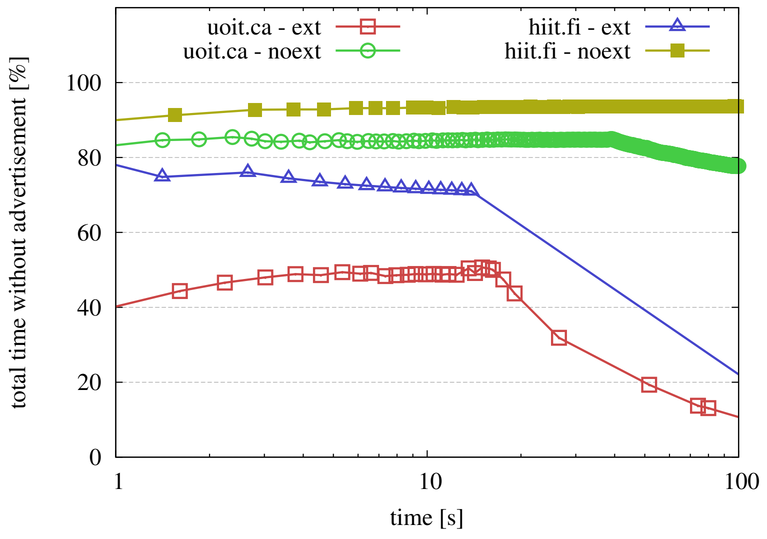


Figure 9.39: Total time without advertisement

able to forward filters towards matching publishers and thus it is not able to deliver events to the interested subscribers. It can be observed that without the extension technique the total broker time without valid advertisement remains very high, while the validity interval extension technique quickly helps to extend the advertisement intervals so as to accommodate for the large latency experienced by the refresh messages.

Chapter 10

Conclusions

This chapter summarizes the main contributions of this thesis and provides a brief outline of the possible future work with respect to the content-based matching and routing of information as well as the privacy and security issues.

10.1 Summary

During the last fifteen years, since the introduction of the publish/subscribe systems, the recognition for their importance for the distributed systems research agenda has been constantly increasing. The publish/subscribe concept was first introduced in [OPSS93] and was driven by the four design principles: (1) minimal semantics of the core communication protocols, (2) self-describing objects, (3) dynamically defined types and (4) anonymous communication. Shortly thereafter a content-based naming model for an Internet-scale event observation and notification facility [RW97] was introduced extending the expressiveness of the publish subscribe systems.
Subsequently, a number of advanced routing concepts including the filter coverage relation and the concept of advertisements summarizing the content to be produced by the event publishers [CRW01] has been introduced. Simultaneously, a range of new underlying architectures for publish/subscribe systems including structured [RHKS01, CDKR02] and unstructured overlays [BEG04] has been proposed.
The practical evolution of the publish/subscribe systems has been closely followed by the theoretical publications including the formal specification of the publish/subscribe systems with the concepts of safety and liveness [MÖ2], allowing to precisely define the specification of the correct publish/subscribe system. A formalization of the decoupling properties of the publish/subscribe systems, including the specification of the time, space and synchronization decoupling properties has been proposed in [EFGK03].
The above developments have led to the widespread adoption of the publish/subscribe systems as a communication middleware for the loosely coupled

distributed systems. Publish/subscribe paradigm is used by multiple applications [Bar07, SL07, MSSB07, DXGE07] wishing to decouple their components and take the advantage of the event-based communication style exposed by the publish/subscribe paradigm.

This thesis made two basic contributions to the field of the content-based publish/subscribe systems. Firstly it proposed a set of event forwarding and filter routing algorithms which are based on the end-to-end principle. The proposed algorithms (cf. Chapters 4 and 5) migrate the expensive content-based event matching from the network layer into the application layer, reducing the overall cost for the filter routing and for the event forwarding. Secondly, a fail-aware, content-based publish/subscribe system has been proposed – cf. Chapter 6. Fail-aware publish/subscribe system is based on the Timed Asynchronous Distributed System Model [CF99] and allows to detect the end-to-end timeliness of advertisements, filters and events. It has been also shown how to use the fail-aware publish/subscribe system to remove the hard state from the publish/subscribe network – cf. Chapter 7.

The result of this thesis is the XSiena family of the content-based publish/subscribe systems. The XSiena family of content-based publish/subscribe systems has been used as a proof-of-concept implementation of the two main contributions of this thesis: the Bloom filter-based routing framework and the soft state publish/subscribe system. This thesis presented an extensive evaluation of the implemented prototypes including evaluation in the Local Area Network and the Wide Area Network environment of the PlanetLab. The results obtained via the evaluation have confirmed that the algorithms presented in this thesis allow to excel the content-based forwarding of events and provide an applicable model for the soft state in the content-based publish/subscribe systems.

10.2 Outlook

Content-based publish/subscribe systems are becoming an increasingly mature technology which is used in an increasing number of applications. However, the widespread use of the publish/subscribe systems is still hindered by the lack of the production grade implementations with well established target group.

Publish/subscribe systems are not an end-user interface or a middleware which would be exposed and thus appreciated as such by a broad spectrum of users. The role of the publish/subscribe systems is to remain a tool for the developers of the distributed systems. A tool, which provides fast, decoupled, and content-based communication between the components of a distributed system. Therefore, publish/subscribe systems should and will find their way as a communication abstraction into more and more loosely coupled systems. One of the examples being the role XSiena plays in the StreamMine project – cf. Section 9.2.

There exist two basic paths which are certainly still posing a number of challenges in the context of the content-based publish/subscribe systems. The first one focuses on further improvements in the speed of event and filter routing along with the optimization of bandwidth usage. New technologies, like: many-core processors [WES08], GPUs [FH08], transactional memory [RFF07] and resulting parallelization possibilities [Bri08] are becoming widely available in the commercial off-the-shelf segment and offer a variety of new tools and approaches in search for more efficient algorithms.
The second path is implied by the possibilities offered through the use of the end-to-end approach towards the content-based routing. The ability to abstract the content of the messages can be seen as a foundation allowing for the new approach towards the development of secure content-based publish/subscribe services. Current attempts at providing security and privacy in content-based systems [CW01, Hom02, BEP05] are still in preliminary stadium. Moreover, aforementioned approaches are still based on the point-to-point paradigms taken directly from the TCP-based communication. The publish/subscribe paradigm, based on the N-to-M communication, as opposed to the 1-to-1 model of the TCP, requires a new approach towards the provision of privacy and security in large scale, content-based networks.

Symbols

Term	Definition (Section)	Description
e	2.2.2	event
f	2.2.1	subscription filter
a	2.2.3	advertisement filter
v	2.3.2	value: a part of event in predicate-based semantics
p	2.3.2	predicate: a part of a filter in predicate-based semantics
$\mathbf{P}$	2.1.3	publisher
$\mathbf{S}$	2.1.2	subscriber
$\mathbb{P}$	2.5.6	set of all publishers in the system
$\mathbb{S}$	2.5.2	set of all subscribers in the system
$\mathbf{B}$	2.1.4	broker
$\mathbb{E}_f$	2.5.3	set of events selected by filter f
$\mathbb{E}_a$	2.5.6	set of events selected by advertisement a
$\mathbb{F}_{\mathbf{S}}$	2.2.1	set of active filters issued by the subscriber $\mathbf{S}$
$\mathbb{A}_{\mathbf{P}}$	2.2.3	set of active advertisements issued by the

continued on next page

Term *contd.*	**Definition** *contd.*	**Description** *contd.*
		publisher **P**
$\mathbb{F}_{\mathbf{B}}$	2.5.4	set of active filters stored in the routing table of the broker **B**
$\mathbb{A}_{\mathbf{B}}$	2.5.4	set of active advertisements stored in the routing table of the broker **B**
$\mathbb{I}_{\mathbf{B}}$	2.5.1	set of all interfaces of the broker **B**
$I_{\leftarrow e}$	2.5.1	source interface of the event *e*
$I_{\leftarrow f}$	2.5.4	source interface of the filter *f*
$\mathbb{I}_{f\rightarrow}$	2.5.4	set of interfaces on which filter *f* was forwarded
$\mathbb{I}^{\text{unsub}}_{f\rightarrow}$	2.5.3	set of interfaces on which unsubscription message containing filter *f* was forwarded
$I_{\leftarrow a}$	2.5.6	source interface of the advertisement *a*
$\mathbb{I}_{a\rightarrow}$	2.5.6	set of interfaces on which advertisement *a* was forwarded
$\mathbb{E}_f$	2.5.4	set of events selected by filter *f*
$\sigma_{\max}$	3.1	maximum (upper bound) processing time of a correct process
$\delta_{\max}$	3.1	maximum (upper bound) message transmission delay
$\delta_{\min}$	3.1	minimum (lower bound) message transmission delay
$\Delta_{\max}$	6.1	message classification threshold
$H(t)$	3.2	value of the hardware clock at the real-time *t*
$\rho_{\max}$	3.2	upper bound on the drift rate

continued on next page

Term *contd.*	**Definition** *contd.*	**Description** *contd.*
		of all hardware clocks
td(m)	6.2	transmission delay of the message m
ub(m)	6.2	upper bound on the transmission delay of the message m
$\mathrm{ub}_\Delta(m)$	7.2.4	uncertainty of the upper bound on the transmission delay of the message m
lb(n)	6.2	lower bound on the transmission delay of the message m
pd(m)	6.3	propagation delay of the message m
pt(m)	6.3	processing time of the message m
T_T	7.2	validity time
T_I	7.2	validity interval
$\Phi_{\max}$	7.2	maximum deviation between correct, synchronized clocks, precision
U_{T_I}	7.2.4	validity interval utilization

Index

Bibliography

[AEM99] 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, 1999.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB '00: Proceedings of 26th International Conference on Very Large Data Bases*, pages 53–64, Cairo, Egypt, September 2000. Morgan Kaufmann.

[AGK$^+$01] Micah Adler, Zihui Ge, James F. Kurose, Donald F. Towsley, and Steve Zabele. Channelization problem in large scale data dissemination. In *ICNP '01: Proceedings of the 9th International Conference on Network Protocols*, pages 100–109, Riverside, CA, USA, November 2001. IEEE Computer Society.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan.–Mar. 2004.

[AT05] Ioannis Aekaterinidis and Peter Triantafillou. Internet scale string attribute publish/subscribe data networks. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 44–51, New York, NY, USA, 2005. ACM.

[AT07] Ioannis Aekaterinidis and Peter Triantafillou. Publish-subscribe information delivery with substring predicates. *IEEE Internet Computing*, 11(4):16–23, August 2007.

[Bar64] Paul Baran. On distributed communications networks. *IEEE Transactions on Communications Systems*, 12(1):1–9, March 1964.

[Bar07] David Barnett. Publish-subscribe model connects tokyo highways. *Industrial Embedded Systems*, March 2007.

[BBPV05] Roberto Baldoni, Roberto Beraldi, Sara Tucci Piergiovanni, and Antonino Virgillito. On the modelling of publish/subscribe communication systems: Research articles. *Concurrency and Computation: Practice and Experience*, 17(12):1471–1495, April 2005.

[BBQ+07] Roberto Baldoni, Roberto Beraldi, Vivien Quéma, Leonardo Querzoni, and Sara Tucci Piergiovanni. Tera: topic-based event routing for peer-to-peer architectures. In Hans-Arno Jacobsen, Gero Mühl, and Michael A. Jaeger, editors, *DEBS '08: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*, volume 233 of *ACM International Conference Proceeding Series*, pages 2–13. ACM, 2007.

[BBQV07] Roberto Baldoni, Roberto Beraldi, Leonardo Querzoni, and Antonino Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to siena. *The Computer Journal*, 50(4):444–459, 2007.

[BBV+05] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop advanced architecture. In *DSN '05: International Conference on Dependable Systems and Networks*, pages 12–21, Yokohama, Japan, June 2005. IEEE Computer Society.

[BCM+99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 262, Washington, DC, USA, 1999. IEEE Computer Society.

[BCSS99] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A case for message oriented middleware. In Prasad Jayanti, editor, *DISC '99: 13th International Symposium on Distributed Computing*, volume 1693 of *Lecture Notes in Computer Science*, pages 1–18, Bratislava, Slovak Republic, September 1999. Springer.

[BCV03] Roberto Baldoni, Mariangela Contenti, and Antonino Virgillito. The evolution of publish/subscribe communication systems. In André Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 137–141. Springer, 2003.

[BEG04] Sébastien Baehni, Patrick Th. Eugster, and Rachid Guerraoui. Data-aware multicast. In *DSN 2004: International Conference on Dependable Systems and Networks*, pages 233–242. IEEE Computer Society, 2004.

[BEP05] Jean Bacon, David Eyers, and Ken Moodyand Lauri Pesonen. Securing publish/subscribe for multi-domain systems. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 1–20, Grenoble, France, November–December 2005. Springer.

[BFC93] Tony Ballardie, Paul Francis, and Jon Crowcroft. Core based trees (cbt). In *SIGCOMM*, pages 85–95, 1993.

[BFG07] Silvia Bianchi, Pascal Felber, and Maria Gradinariu. Content-based publish/subscribe using distributed r-trees. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par '07: Proceeding of the 13th International Euro-Par Conference*, volume 4641 of *Lecture Notes in Computer Science*, pages 537–548. Springer, August 2007.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[BFS07] Scott Boag, Don Chamberlinand Mary F. Fernandez, and Daniela Florescuand Jonathan Robieand Jerome Simeon. Xquery 1.0: An xml query languag. Online, January 2007. http://www.w3.org/TR/xquery/.

[BH06] Sven Bittner and Annika Hinze. Pruning subscriptions in distributed publish/subscribe systems. In Vladimir Estivill-Castro and Gillian Dobbie, editors, *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, volume 48 of *CRPIT*, pages 197–206, Hobart, Tasmania, Australia, January 2006. Australian Computer Society.

[BHL95] Burnie Blakeley, Harry Harris, and Rhys Lewis. *Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development.* Mcgraw-Hill Series on Computer Communications. The McGraw-Hill Companies, June 1995.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BN84] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BPSMY06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Malerand Francois Yergeau. Extensible markup language (xml) 1.0. Online, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816/.

[Bri08] Andrey Brito. Optimistic parallelization support for event stream processing systems. In Sam Michiels, editor, *The Proceedings of the Doctoral Symposium of the ACM/IFIP/USENIX 9th International Middleware Conference*, pages 7–12, Leuven, Belgium, December 2008. ACM.

[Bru04] Rich Brunner. AMD64 status report for kernel summit. USENIX - 2004 Linux Kernel Developers Summit, July 2004.

[BSB$^+$02] Sumeer Bhola, Robert E. Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *DSN 2002: International Conference on Dependable Systems and Networks*, pages 7–16, Bethesda, MD, USA, June 2002. IEEE Computer Society.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Seventh Symposium on Operating System Design and Implementation*, pages 335–350, Seattle, WA, USA, November 2006. USENIX Association.

[CABB04] M. Cilia, M. Antollini, C. Bornhövd, and A. Buchmann. Dealing with heterogeneous data in pub/sub systems: The concept-based approach. In A. Carzaniga and P. Fenkam, editors, *DEBS '04: Proceedings of the 3rd International Workshop on Distributed Event-Based Systems*. IEE, 2004.

[Car93] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[CBGM03] Arturo Crespo, Orkut Buyukkokten, and Hector Garcia-Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):174–191, Jan.–Feb. 2003.

[CCR04] Miguel Castro, Manuel Costa, and Antony Rowstron. Peer-to-peer overlays: structured, unstructured, or both? Technical Report MSR-TR-2004-73, Microsoft Research, Cambridge, UK, July 2004.

[CDHR03] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony I. T. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. In André Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 103–107. Springer, 2003.

[CDKR02] M. Castro, P. Druschel, A.M. Kermarrec, and AIT Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, October 2002.

[CDKR03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. Scalable application-level anycast for highly dynamic groups. In Burkhard Stiller, Georg Carle, Martin Karsten, and Peter Reichl, editors, *Networked Group Communication*, volume 2816 of *Lecture Notes in Computer Science*, pages 47–57. Springer, 2003.

[CDNF01] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[CDS74] Vinton Cerf, Yogen Dalal, and Carl Sunshine. Specification of internet transmission control program. RFC 675, dec 1974.

[CF99] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[CF04] Raphaël Chand and Pascal Felber. XNET: a reliable content-based publish/subscribe system. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 264–273, Florianpolis, Brazil, October 2004. IEEE Computer Society.

[CF05] Raphaël Chand and Pascal Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In Jose C. Cunha and Pedro D. Medeiros, editors, *Euro-Par '05: Proceedings of the 11th International Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 1194–1204, Lisbon, Portugal, August–September 2005. Springer.

[CFGR02] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient filtering of xml documents with xpath expressions. In *Proceedings of the 18th International Conference on Data Engineering*, pages 235–244, San Jose, CA, USA, 2002. IEEE Computer Society.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CH06] Antonio Carzaniga and Cyrus P. Hall. Content-based communication: a research agenda. In Eric Wohlstadter, editor, *SEM '06: Proceedings of the 6th International Workshop on Software Engineering and Middleware*, pages 2–8. ACM, November 2006.

[Che05] Michael Chertoff. The national plan for research and development in support of critical infrastructure protection. Department of Homeland Security, Online, April 2005.

[CHY$^+$98] P. Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang, and Yi-Min Wang. Dcom and corba side by side, step by step, and layer by layer. *C++ Report*, 10(1):18–29, January 1998.

[CK74] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, May 1974.

[CK05] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *Computer Communication Review*, 35(2):71–82, 2005.

[Cla88] D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 106–114, New York, NY, USA, 1988. ACM.

[CLS03] Mao Chen, Andrea S. LaPaugh, and Jaswinder Pal Singh. Content distribution for publish/subscribe services. In Markus Endler and Douglas C. Schmidt, editors, *Middleware '03: Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*, pages 83–102, Rio de Janeiro, Brazil, June 2003. Springer.

[CMPC04] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 552–561, Hachioji, Tokyo, Japan, March 2004. IEEE Computer Society.

[CMRV01] Antonio Casimiro, Pedro Martins, Luís Rodrigues, and Paulo Veríssimo. Measuring distributed durations with stable errors. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 310–319, London, UK, 2001. IEEE Computer Society.

[Cor04] Intel Corporation. IA-PC HPET (High Precision Event Timers) Specification. Online, October 2004. http://www.intel.com/hardwaredesign/hpetspec_1.pdf.

[CP06] Gianpaolo Cugola and Gian Pietro Picco. Reds: a reconfigurable dispatching system. In Eric Wohlstadter, editor, *SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware*, pages 9–16, Portland, Oregon, USA, November 2006. ACM.

[CPB$^+$05] David D. Clark, Craig Partridge, Robert T. Braden, Bruce Davie, Sally Floyd, Van Jacobson, Dina Katabi, Greg Minshall, K. K. Ramakrishnan, Timothy Roscoe, Ion Stoica, John Wroclawski, and Lixia Zhang. Making the world (of communications) a different place. *SIGCOMM Comput. Commun. Rev.*, 35(3):91–96, 2005.

[CQL08] Daniel Cutting, Aaron Quigley, and Björn Landfeldt. Spice: Scalable p2p implicit group messaging. *Computer Communications*, 31(3):437–451, 2008.

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors, *SIGCOMM '03: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication,*, pages 407–418, Karlsruhe, Germany, August 2003. ACM.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[CRW04] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, pages 7–11, Hong Kong, China, March 2004.

[CS05] Fengyun Cao and Jaswinder Pal Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. In *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 292–313, Grenoble, France, November 2005. Springer.

[CW01] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In Birgitta König-Ries, Kia Makki, S. A. M. Makki, Niki Pissinou, and Peter Scheuermann, editors, *IMWS 2001: NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, volume 2538 of *Lecture Notes in Computer Science*, pages 59–68, Scottsdale, AZ, USA, October 2001. Springer.

[CW03] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors, *Proceedings of ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 163–174, Karlsruhe, Germany, August 2003. ACM.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[DEEJ01] 3rd Donald E. Eastlake and Paul E. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001. Updated by RFC 4634.

[DeR99] James Clark Steve DeRose. Xml path language (xpath). Online, November 1999. http://www.w3.org/TR/xpath.

[DF03] Yanlei Diao and Michael J. Franklin. Query processing for high-volume xml message brokering. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 261–272, 2003.

[DGH+06] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *EDBT '06: 10th International Conference on Extending Database Technology*, volume 3896 of *Lecture Notes in Computer Science*, pages 627–644, Munich, Germany, March 2006. Springer.

[DGP$^+$07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR '07: Third Biennial Conference on Innovative Data Systems Research*, pages 412–422, Asilomar, CA, USA, 2007.

[DM78] Yogen K. Dalal and Robert Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, 1978.

[DM04] Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In Alan J. Hu and Andrew K. Martin, editors, *FMCAD '04: Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 367–381, Austin, Texas, USA, November 2004. Springer.

[DPW04] A. Davis, J. Parikh, and W. E. Weihl. Edgecomputing: extending enterprise applications to the edge of the internet. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *WWW Alt. '04: Proceedings of the 13th international conference on World Wide Web. Alternate track papers & posters*, pages 180–187, New York, NY, USA, May 2004. ACM.

[DRF04] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internet-scale xml dissemination service. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 612–623, Toronto, Canada, August 2004. Morgan Kaufmann.

[DXGE07] Gan Deng, Ming Xiong, Aniruddha S. Gokhale, and George Edwards. Evaluating real-time publish/subscribe service integration approaches in qos-enabled component middleware. In *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 222–227, Santorini Island, Greece, May 2007. IEEE Computer Society.

[EAP99] D. Essame, J. Arlat, and D. Powell. Padre: a protocol for asymmetric duplex redundancy. *Dependable Computing for Critical Applications*, 7:229–248, 1999.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[EG02] Patrick Th. Eugster and Rachid Guerraoui. Probabilistic multicast. In *DSN '02: Proceddings of the International Conference on Dependable Systems and Networks*, pages 313–324, Bethesda, MD, USA, June 2002. IEEE Computer Society.

[Eug07] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Transactions on Programming Languages and Systems*, 29(1):1–50, January 2007.

[FBFJ] Christof Fetzer, Andrey Brito, Robert Fach, and Zbigniew Jerzak. *To appear in: Handbook of Research on Advanced Distributed Event-Based Systems, Publish/Subscribe and Message Filtering Technologies*, chapter StreamMine. IGI Global.

[FC99a] Christof Fetzer and Flaviu Cristian. Building fault-tolerant hardware clocks from cots components. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 67–86, San Jose, CA, USA, Nov 1999.

[FC99b] Christof Fetzer and Flaviu Cristian. A fail-aware datagram service. In Iain Bate and Alan Burns, editors, *IEE Proceedings - Software Engineering*, volume 146, pages 58–74. IEE, April 1999.

[FC03] Christof Fetzer and Flaviu Cristian. Fail-awareness: An approach to construct fail-safe systems. *Journal of Real-Time Systems*, 24(2):203–238, March 2003.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, June 2000.

[FCMB06] Ludger Fiege, Mariano Cilia, Gero Mühl, and Alejandro P. Buchmann. Publish-subscribe grows up: Support for management, visibility control, and heterogeneity. *IEEE Internet Computing*, 10(1):48–55, 2006.

[Fer04] Randima Fernando, editor. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004.

[Fet98] Christof Fetzer. Fail-aware publish/subscribe in erlang. In *Proceedings of the Fourth International Erlang User Conference*, Stockholm, Sweden, September 1998.

[FH08] Kayvon Fatahalian and Mike Houston. A closer look at gpus. *Communications of the ACM*, 51(10):50–57, 2008.

[FJL$^+$01] Francoise Fabret, H. Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126, New York, NY, USA, 2001. ACM Press.

[FJLM05] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The padres distributed publish/subscribe system. In *Feature Interactions in Telecommunications and Software Systems*, pages 12–30, Leicester, UK, July 2005.

[FSH04] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In Dieter W. Fellner and Stephen N. Spencer, editors, *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, Grenoble, France, August 2004. Eurographics Association.

[GDB$^+$03] K. Harald Gjermundrød, Ioanna Dionysiou, David Bakken, Carl Hauser, and Anjan Bose. Flexible and robust status dissemination middleware for the electric power grid. Technical Report EECS-GS-003, School of Electrical Engineering and Computer Science Washington State University, Pullman, Washington 99164-2752 USA, September 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Orineted Software.* Addison-Wesley Professional, 1995.

[Gra03] Jim Gray. Distributed computing economics. Technical Report MSR-TR-2003-24, Microsoft Research, Redmond, WA, USA, March 2003. http://research.microsoft.com/pubs/70001/tr-2003-24.pdf.

[Gro04] Object Management Group. Corba event service specification. Online, October 2004. http://www.omg.org/docs/formal/04-10-02.pdf.

[GSGM03] Prasanna Ganesan, Qixiang Sun, and Hector Garcia-Molina. Yappers: A peer-to-peer lookup service over arbitrary topology. In *INFOCOM '03: Proceedings of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1250–1260, March-April 2003.

[Han05] Pat Hanrahan. Why is graphics hardware so fast? In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–1, Chicago, IL, USA, 2005. ACM.

[Hom02] Alexis B. Hombrecher. *Reconciling Event Taxonomies Across Administrative Domains*. PhD thesis, University of Cambridge Computer Laboratory, Cambridge, United Kingdom, June 2002.

[Jac03] Hans-Arno Jacobsen. Tutorial: Omg data distribution service. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, pages 198–199, May 2003.

[Jac06] Van Jacobson. A new way to look at networking. Google Tech Talk, August 2006.

[Jac07] Van Jacobson. Content-centric networking at ntt. Online, 2007. http://www.parc.com/research/projects/networking/contentcentric/.

[Jae07] Michael A. Jaeger. *Self-Managing Publish/Subscribe Systems*. PhD thesis, Technische Universität Berlin, 2007.

[Jer05] Zbigniew Jerzak. Highly available publish/subscribe. In *HASE '05: Supplement Proceedings of the Ninth IEEE International Symposium on High Assurance Systems Engineering*, pages 11–12, Heidelberg, Germany, October 2005. IEEE Computer Society Press.

[JF06] Zbigniew Jerzak and Christof Fetzer. Handling overload in publish/subscribe systems. In *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, pages 32–37, Lisbon, Portugal, June 2006. IEEE Computer Society.

[JF07] Zbigniew Jerzak and Christof Fetzer. Prefix forwarding for publish/subscribe. In Hans-Arno Jacobsen, Gero Mühl, and Michael A. Jaeger, editors, *DEBS '07: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*, volume 233 of *ACM International Conference Proceeding Series*, pages 238–249, Toronto, Ontario, Canada, June 2007. ACM.

[JF08a] Zbigniew Jerzak and Christof Fetzer. BFSiena: a communication substrate for StreamMine. In Roberto Baldoni, editor, *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, volume 332 of *ACM International Conference Proceeding Series*, pages 321–324, Rome, Italy, July 2008. ACM.

[JF08b] Zbigniew Jerzak and Christof Fetzer. Bloom filter based routing for content-based publish/subscribe. In Roberto Baldoni, editor, *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, volume 332 of *ACM International Conference Proceeding Series*, pages 71–81, Rome, Italy, July 2008. ACM.

[JFF07] Zbigniew Jerzak, Robert Fach, and Christof Fetzer. Fail-aware publish/subscribe. In *NCA '07: Sixth IEEE International Symposium on Network Computing and Applications*, pages 113–125, Cambridge, MA, USA, July 2007. IEEE Computer Society.

[JFF08] Zbigniew Jerzak, Robert Fach, and Christof Fetzer. Adaptive internal clock synchronization. In *SRDS 2008: 27th International Symposium on Reliable Distributed Systems*, pages 217–226, Naples, Italy, October 2008. IEEE Computer Society.

[JKKR04] T. S. Jayram, Subhash Khot, Ravi Kumar, and Yuval Rabani. Cell-probe lower bounds for the partial match problem. *Journal of Computer and System Sciences*, 69(3):435–447, November 2004.

[KHCS05] Satyen Kale, Elad Hazan, Fengyun Cao, and Jaswinder Pal Singh. Analysis and algorithms for content-based event matching. In *ICDCS '05 Workshops: 25th International Conference on Distributed Computing Systems Workshops*, pages 363–369, Columbus, OH, USA, June 2005. IEEE Computer Society.

[KM99] J. Kaiser and M. Mock. Implementing the real-time publisher/subscriber model on the controller area network (can). In *ISORC '99: Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 172–181, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[KM06] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In Yossi Azar and Thomas Erlebach, editors, *ESA '06: Proceedings of the 14th Annual European Symposium on Algorithms*, volume 4168 of *Lecture Notes in Computer Science*, pages 456–467, Zurich, Switzerland, September 2006. Springer.

[KSS03] John C. Knight, Elisabeth A. Strunk, and Kevin J. Sullivan. Towards a rigorous definition of information system survivability. In *DISCEX 2003: 3rd DARPA Information Survivability Conference and Exposition*, pages 78–89, Washington, DC, USA, April 2003. IEEE Computer Society.

[Lam79] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, 1979.

[LB05] Pierre L'Ecuyer and Eric Buist. Simulation in java with ssj. In *Proceedings of the 37th Winter Simulation Conference*, pages 611–620, Orlando, FL, USA, December 2005. ACM.

[Lec09] Emmanuel Lecharny. Using mina 2.0 in real life. ApacheCon Europe, March 2009. http://mina.apache.org/documentation.html.

[Lei07] Tom Leighton. The akamai approach to achieving performance and reliability on the internet. In Indranil Gupta and Roger Wattenhofer, editors, *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 2–2, Portland, Oregon, USA, August 2007. ACM.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LHG$^+$06] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. Gpgpu: general-purpose computation on graphics hardware. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208, Tampa, FL, USA, 2006. ACM Press.

[LHJ05] Guoli Li, Shuang Hou, and Hans-Arno Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 447–457, Washington, DC, USA, 2005. IEEE Computer Society.

[LHK$^+$04] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM.

[LMJ08] Guoli Li, Vinod Muthusamy, , and Hans-Arno Jacobsen. Adaptive content-based routing in general overlay topologies. In Valérie Issarny and Richard E. Schantz, editors, *Middleware '08: Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference*, volume 5346 of *Lecture Notes in Computer Science*, pages 1–21, Leuven, Belgium, December 2008. Springer.

[LMV02] Pierre L'Ecuyer, Lakhdar Meliani, and Jean G. Vaucher. Ssj: a framework for stochastic simulation in java. In Jane L. Snowdon and John M. Charnes, editors, *Proceedings of the 34th Winter Simulation Conference: Exploring New Frontiers*, pages 234–242, San Diego, California, USA, December 2002. ACM.

[LRS02] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make gnutella scalable? In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *IPTPS '02: First International Workshop on Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 94–103, Cambridge, MA, USA, March 2002. Springer.

[MÖ2] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Technisches Universität Darmstadt, 2002.

[MFB02] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Filter similarities in content-based publish/subscribe systems. In Hartmut Schmeck, Theo Ungerer, and Lars C. Wolf, editors, *ARCS '02: Proceedings of the International Conference on Architecture of Computing Systems, Trends in Network and Pervasive Computing*, volume 2299 of *Lecture Notes in Computer Science*, pages 224–240, Karlsruhe, Germany, April 2002. Springer.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[Mil92] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. RFC 1305, March 1992.

[Mil94] David L. Mills. A kernel model for precision timekeeping. RFC 1589, March 1994. http://www.ietf.org/rfc/rfc1589.txt.

[Mil06] David L. Mills. Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi. RFC 4330, January 2006.

[MJH$^+$05] Gero Mühl, Michael A. Jaeger, Klaus Herrmann, Torben Weis, Andreas Ulbrich, and Ludger Fiege. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. In *Euro-Par 2005 Parallel Processing*, volume 3648/2005, pages 664–674. Springer Berlin / Heidelberg, 2005.

[MLC08] James Moscola, John W. Lockwood, and Young H. Cho. Reconfigurable content-based router using hardware-accelerated language parser. *ACM Transactions on Design Automation of Electronic Systems*, 13(2):1–25, April 2008. Article 28.

[MPHD06] Alan Mislove, Ansley Post, Andreas Haeberlen, and Peter Druschely. Experiences in building and operating a reliable peer-to-peer application. In Yolande Berbers and Willy Zwaenepoel, editors, *EuroSys*, pages 147–159, Leuven, Belgium, April 2006. ACM.

[MSSB07] J. Darby Mitchell, Marc L. Siegel, M. Curran N. Schiefelbein, and Armen P. Babikyan. Applying publish-subscribe to communications-on-the-move node control. *Lincoln Laboratory Journal*, 16(2):413–430, 2007.

[MZV07] Tova Milo, Tal Zur, and Elad Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 749–760, Beijing, China, June 2007. ACM.

[Nag06] Bhavana Nagendra. AMD TSC Drift Solutions in Red Hat Enterprise Linux. Online, December 2006.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[OAA$^+$00] Lukasz Opyrchal, Mark Astley, Joshua S. Auerbach, Guruduth Banavar, Robert E. Strom, and Daniel C. Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In Joseph S. Sventek and Geoff Coulson, editors, *Middleware '00: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, volume 1795 of *Lecture Notes in Computer Science*, pages 185–207, New York, NY, USA, April 2000. Springer.

[OPSS93] Brian M. Oki, Manfred Pflügl, Alex Siegel, and Dale Skeen. The information bus – an architecture for extensible distributed systems. In B. Liskov, editor, *Proceedings of the 14th Symposium on the Operating Systems Principles*, pages 58–68. ACM Press, December 1993.

[PB03a] Peter Pietzuch and Sumeer Bhola. Congestion control in a reliable scalable message-oriented middleware. In Markus Endler and Douglas C. Schmidt, editors, *Middleware '03: Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*, pages 202–221, Rio de Janeiro, Brazil, June 2003. Springer.

[PB03b] Peter R. Pietzuch and Jean Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In Hans-Arno Jacobsen, editor, *DEBS '03: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems*, pages 1–8. ACM, 2003.

[PBFM06] Larry L. Peterson, Andy C. Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planetlab. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 351–366, Seattle, WA, USA, November 2006. USENIX Association.

[PC05a] G. Pardo-Castellote. DDS Spec Outfits Publish/Subscribe Technology for the GIG. *COTS Journal*, 4, April 2005.

[PC05b] Gerardo Pardo-Castellote. Omg data distribution service: Real-time publish/subscribe becomes a standard. *RTC Magazine*, 14, January 2005.

[PCM03] Gian Pietro Picco, Gianpaolo Cugola, and Amy L. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 234–243, Providence, RI, USA, May 2003. IEEE Computer Society.

[PCT06] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In Xiaohua Jia, editor, *Infoscale '06: Proceedings of the 1st International Conference on Scalable Information Systems*, volume 152 of *ACM International Conference Proceeding Series*, pages 1–7, Hong Kong, May–June 2006. ACM. Article No. 1.

[PEKS07] Peter Pietzuch, David Eyers, Samuel Kounev, and Brian Shand. Towards a common api for publish/subscribe. In Hans-Arno Jacobsen, Gero Mühl, and Michael A. Jaeger, editors, *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, volume 233 of *ACM International Conference Proceeding Series*, pages 152–157, Ontario, Canada, June 2007. ACM.

[PFH07] Gert Pfeifer, Christof Fetzer, and Thomas Hohnstein. Exploiting host name locality for reduced stretch p2p routing. In *NCA '07: Sixth IEEE International Symposium on Network Computing and Applications*, pages 134–144, Cambridge, MA, USA, July 2007. IEEE Computer Society.

[PFLS00] João Pereira, Françoise Fabret, François Llirbat, and Dennis Shasha. Efficient matching for web-based publish/subscribe systems. In Opher Etzion and Peter Scheuermann, editors, *CooplS '02: Proceedings of the 7th International Conference on Cooperative Information Systems*, volume 1901 of *Lecture Notes in Computer Science*, pages 162–173, Eilat, Israel, September 2000. Springer-Verlag.

[Pie04] Peter R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pow96] David Powell. Group communication. *Communications of the ACM*, 39(4):50–53, 1996.

[PSB03] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In Markus Endler and Douglas C. Schmidt, editors, *Middleware '03: Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*, pages 62–82, Rio de Janeiro, Brazil, June 2003. Springer.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[QB06] Yi Qiao and Fabián E. Bustamante. Structured and unstructured overlays under the microscope: a measurement-based view of two p2p systems that people use. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 31–31, Berkeley, CA, USA, 2006. USENIX Association.

[Que07] Leonardo Querzoni. *Techniques for Efficient Event Routing*. PhD thesis, Universita di Roma "La Sapienza", 2007.

[Que08] Leonardo Querzoni. Interest clustering techniques for efficient event routing in large-scale settings. In Roberto Baldoni, editor, *DEBS 2008: Proceedings of the Second International Conference on Distributed Event-Based Systems*, volume 332 of *ACM International Conference Proceeding Series*, pages 13–22, Rome, Italy, July 2008. ACM.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[RFF07] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA '07: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 221–228, San Diego, California, USA, June 2007. ACM.

[RHKS01] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop*, volume 2233 of *Lecture Notes in Computer Science*, pages 14–29, London, UK, 2001. Springer Berlin / Heidelberg.

[Riv92] Ronald L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[RLW$^+$02] Anton Riabov, Zhen Liu, Joel L. Wolf, Philip S. Yu, and Li Zhang. Clustering algorithms for content-based publication-subscription systems. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 133–142, Washington, DC, USA, July 2002. IEEE Computer Society.

[RM99] Suchitra Raman and Steven McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 15–25, New York, NY, USA, 1999. ACM.

[Ros05] Timothy Roscoe. The planetlab platform. In Ralf Steinmetz and Klaus Wehrle, editors, *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*, pages 567–581. Springer, 2005.

[RW97] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In Mehdi Jazayeri and Helmut Schauer, editors, *6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 344–360, Zurich, Switzerland, September 1997. ACM.

[SA97] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *AUUG '97: Proceedings of the 1997 Australian UNIX User Group*, pages 243–255, September 1997.

[SB89] Marion D. Skeen and Mark Bowles. Apparatus and method for providing decoupling of data exchange details for providing high performance communication between software processes. U.S. Patent No. 5,557,798, July 1989.

[SCG01] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh based content routing using xml. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 160–173, Banff, Alberta, Canada, October 2001. ACM.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stanley B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[SK08] Ao-Jan Su and Aleksandar Kuzmanovic. Thinning akamai. In Konstantina Papagiannaki and Zhi-Li Zhang, editors, *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 29–42, Vouliagmeni, Greece, October 2008. ACM.

[SL07] Heejin Son and Xiaolin Li. Parmi: A publish/subscribe based asynchronous rmi framework for cluster computing. In Ronald H. Perrott, Barbara M. Chapman, Jaspal Subhlok, Rodrigo Fernandes de Mello, and Laurence Tianruo Yang, editors, *HPCC '07: Third International Conference on High Performance Computing and Communications*, volume 4782 of *Lecture Notes in Computer Science*, pages 19–29, Houston, USA, September 2007. Springer.

[SM02] Inc. Sun Microsystems. Download java message service specification - version 1.1. Online, April 2002. http://java.sun.com/products/jms/.

[SMK+01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, San Diego, California, United States, 2001. ACM.

[SMRP08] Arnd Schröter, Gero Mühl, Jan Richling, and Helge Parzyjegla. Adaptive routing in publish/subscribe systems using hybrid routing algorithms. In François Taïani and Renato Cerqueira, editors, *ARM '07: Proceedings of the 7th Workshop on Adaptive and Reflective Middleware*, pages 51–52, Leuven, Belgium, December 2008. ACM.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[ST04] Zhenhui Shen and S. Tirthapura. Self-stabilizing routing in publish-subscribe systems. In *International Workshop on Distributed Event-Based Systems*, 2004.

[ST06] Zhenhui Shen and Srikanta Tirthapura. Faster event forwarding in a content-based publish-subscribe system through lookup reuseevent. In *NCA '06: Fifth IEEE International Symposium on Network Computing and Applications*, pages 77–84, Cambridge, Massachusetts, USA, July 2006. IEEE Computer Society.

[STA05] Zhenhui Shen, Srikanta Tirthapura, and Srinivas Aluru. Indexing for subscription covering in publish-subscribe systems. In Michael J. Oudshoorn and Sanguthevar Rajasekaran, editors, *ISCA PDCS '05: Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems*, pages 328–333, Las Vegas, NV, USA, September 2005. ISCA.

[SVvS05] Daniela Gavidia Spyros Voulgaris and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[TA90] B. H. Tay and Akkihebbal L. Ananda. A survey of remote procedure calls. *ACM SIGOPS Operating Systems Review*, 24(3):68–79, July 1990.

[Tar07] Sasu Tarkoma. Chained forests for fast subsumption matching. In Hans-Arno Jacobsen, Gero Mühl, and Michael A. Jaeger, editors, *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, volume 233 of *ACM International Conference Proceeding Series*, pages 97–102, Toronto, Ontario, Canada, June 2007. ACM.

[Tar08a] Sasu Tarkoma. Dynamic filter merging and mergeability detection for publish/subscribe. *Pervasive and Mobile Computing*, 4(5):681–696, 2008.

[Tar08b] Sasu Tarkoma. Fuego toolkit: a modular framework for content-based routing. In Roberto Baldoni, editor, *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, volume 332 of *ACM International Conference Proceeding Series*, pages 325–328, Rome, Italy, July 2008. ACM.

[TB00] David A. Thompson and John S. Best. The future of magnetic data storage technology. *IBM Journal of Research and Development*, 44(3):311–322, 2000.

[TB07] Salman Taherian and Jean Bacon. State-filters for enhanced filtering in sensor-based publish/subscribe systems. In Christian Becker, Christian S. Jensen, Jianwen Su, and Daniela Nicklas, editors, *MDM '07: Proceedings of the 8th International Conference on Mobile Data Management*, pages 346–350, Mannheim, Germany, May 2007.

[TBVB05] K. Tomsovic, D. Bakken, V. Venkatasubramanian, and A. Bose. Designing the next generation of real-time control, communication and computations for large power systems. *Proceedings of the IEEE – Special Issue on Energy Infrastructure Systems*, 93(5):965–979, May 2005.

[TE02] P. Triantafillou and A. Economides. Subscription summaries for scalability and efficiency in publish/subscribe systems. In *Proceedings of Workshops of 22nd International Conference on Distributed Computing Systems*, pages 619–624, Vienna, Austria, July 2002. IEEE Computer Society.

[TE04] Peter Triantafillou and Andreas A. Economides. Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *Proceedings of 24th International Conference on Distributed Computing Systems*, pages 562–571, Hachioji, Tokyo, Japan, March 2004. IEEE Computer Society.

[TK05] Sasu Tarkoma and Jaakko Kangasharju. Filter merging for efficient information dissemination. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, Özalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *On the Move to Meaningful Internet Systems '05: Proceedings of the OTM Confederated International Conferences CoopIS, DOA, and ODBASE*, volume 3760 of *Lecture Notes in Computer Science*, pages 274–291, Agia Napa, Cyprus, October 2005. Springer.

[TK06] Sasu Tarkoma and Jaakko Kangasharju. Optimizing content-based routers: posets and forests. *Distributed Computing*, 19(1):62–77, September 2006.

[TZWK07] Christos Tryfonopoulos, Christian Zimmer, Gerhard Weikum, and Manolis Koubarakis. Architectural alternatives for information filtering in structured overlays. *IEEE Internet Computing*, 11(4):24–34, 2007.

[U.S04] U.S.-Canada Power System Outage Task Force. Final report on the august 14th blackout in the united states and canada. Online, April 2004.

[Var99] Andras Varga. Using the omnet++ discrete event simulation system in education. *IEEE Transactions on Education*, 42(4):372, 1999.

[VRKvS06] Spyros Voulgaris, Etienne Riviére, Anne-Marie Kermarrec, and Maarten van Steen. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, 2006.

[Wal05] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, December 2005.

[WES08] Ian Watson and Hisham El-Shishiny, editors. *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies, IFMT 2008, Cairo, Egypt, November 24-25, 2008*, ACM International Conference Proceeding Series. ACM, 2008.

[WF07] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In Francesca Saglietti and Norbert Oster, editors, *SAFECOMP '07: 26th International Conference on Computer Safety, Reliability, and Security*, volume 4680 of *Lecture Notes in Computer Science*, pages 356–369, Nuremberg, Germany, September 2007. Springer.

[WJL04] Jinling Wang, Beihong Jin, and Jing Li. An ontology-based publish/subscribe system. In Hans-Arno Jacobsen, editor, *Middleware '04: Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science*, pages 232–253, Toronto, Canada, October 2004. Springer.

[WJLS04] Jinling Wang, Beihong Jin, Jing Li, and Danhua Shao. A semantic-aware publish/subscribe system with rdf patterns. In *COMPSAC '04: Proceedings of the 28th International Computer Software and Applications Conference, Design and Assessment of Trustworthy Software-Based Systems*, pages 141–146, Hong Kong, China, September 2004. IEEE Computer Society.

[WQA$^+$02] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen J. Wang. Subscription partitioning and routing in content-based publish/subscribe systems. In *16th International Symposium on DIStributed Computing (DISC'02)*, 2002.

[WQV$^+$04] Yi-Min Wang, Lili Qiu, Chad Verbowski, Dimitris Achlioptas, Gautam Das, and Paul Larson. Summary-based routing for content-based event distribution networks. *SIGCOMM Comput. Commun. Rev.*, 34(5):59–74, 2004.

[WSH08] Timo Warns, Christian Storm, and Wilhelm Hasselbring. Availability of globally distributed nodes: An empirical evaluation. In *SRDS '08: IEEE Symposium on Reliable Distributed Systems*, pages 279–284, Naples, Italy, October 2008.

[XFZ04] Tao Xue, Boqin Feng, and Zhigang Zhang. P2pens: Content-based publish-subscribe over peer-to-peer network. In Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun, editors, *GCC 2004: Third International Conference on Grid and Cooperative Computing*, volume 3251 of *Lecture Notes in Computer Science*, pages 583–590, Wuhan, China, October 2004. Springer.

[YB04] Eiko Yoneki and Jean Bacon. An adaptive approach to content-based subscription in mobile ad hoc networks. In *PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, pages 92–97, Washington, DC, USA, 2004. IEEE Computer Society.

[YGM94] Tak W. Yan and Hector Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, June 1994.

[YSTH87] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, and Yasuaki Honda. *Object-oriented concurrent programming*, chapter Modelling and programming in an object-oriented concurrent language ABCL/1, pages 55–89. MIT Press, Cambridge, MA, USA, 1987.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[ZSB04] Yuanyuan Zhao, Daniel C. Sturman, and Sumeer Bhola. Subscription propagation in highly-available publish/subscribe middleware. In Hans-Arno Jacobsen, editor, *Middleware '04: Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science*, pages 274–293, Toronto, Canada, October 2004. Springer.