Department of Computer Science
Series of Publications A
Report A-2006-2

Efficient Content-based Routing, Mobility-aware Topologies, and Temporal Subspace Matching

Sasu Tarkoma

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XIV, University Main Building, on April 29th, 2006, at 10 o'clock.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 51120

ISSN 1238-8645
ISBN 952-10-3054-2 (paperback)
ISBN 952-10-3055-0 (PDF)
Computing Reviews (1998) Classification: C.2.4, C.4, E.1
Helsinki 2006
Helsinki University Printing House

Efficient Content-based Routing, Mobility-aware Topologies, and Temporal Subspace Matching

Sasu Tarkoma

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
sasu.tarkoma@cs.helsinki.fi
http://www.cs.helsinki.fi/u/starkoma

PhD Thesis, Series of Publications A, Report A-2006-2
Helsinki, April 2006, 198 pages
ISSN 1238-8645
ISBN 952-10-3054-2 (paperback)
ISBN 952-10-3055-0 (PDF)

Abstract

Event-based systems are seen as good candidates for supporting distributed applications in dynamic and ubiquitous environments because they support decoupled and asynchronous many-to-many information dissemination. Event systems are widely used, because asynchronous messaging provides a flexible alternative to RPC (Remote Procedure Call). They are typically implemented using an overlay network of routers. A content-based router forwards event messages based on filters that are installed by subscribers and other routers. The filters are organized into a routing table in order to forward incoming events to proper subscribers and neighbouring routers.

This thesis addresses the optimization of content-based routing tables organized using the covering relation and presents novel data structures and configurations for improving local and distributed operation. Data structures are needed for organizing filters into a routing table that supports efficient matching and runtime operation. We present novel results on dynamic filter merging and the integration of filter merging with content-based routing tables. In addition, the thesis examines the cost of client mobility using different protocols and routing topologies.

We also present a new matching technique called temporal subspace matching. The technique combines two new features. The first feature, temporal operation, supports notifications, or content profiles, that persist in time. The second feature, subspace matching, allows more expressive semantics, because notifications may contain intervals and be defined as subspaces of the content space. We also present an application of temporal subspace matching pertaining to metadata-based continuous collection and object tracking.

Computing Reviews (1998) Categories and Subject Descriptors:
C.2.4 Distributed Systems.
C.4 Performance of Systems.
E.1 Data Structures.

General Terms:
Design, Algorithms, Experimentation

Additional Key Words and Phrases:
Publish/subscribe, event systems, content-based routing, mobility, performance

Acknowledgements

I would like to thank my supervisor Professor Kimmo Raatikainen for guidance and support. The creative environment of his research groups at HIIT and at the University of Helsinki inspired many of the ideas presented in this thesis. The innovative atmosphere at HIIT Advanced Research Unit, led by Professor Martti Mäntylä, was instrumental for the thesis work.

I thank the pre-examiners of this thesis, Professors Martti Mäntylä and Jukka Riekki. I also thank my colleagues at HIIT, especially Jaakko Kangasharju and Adjunct Professor Patrik Floréen, for helpful discussions. Marina Kurtén helped to improve the language in this dissertation. I also acknowledge the valuable comments and feedback provided by anonymous reviewers; they helped to shape and improve the scientific papers that form the main body of this work.

This work was funded and made possible by the series of Fuego Core projects starting from 2002, led by Professor Raatikainen. This work would not have been possible without funding from Tekes, Nokia, TeliaSonera, Elisa, Ericsson, Movial, and More Magic Inc. Especially I would like to thank TeliaSonera for the research grant in 2005.

Part of this work was presented at the Berkeley-Helsinki summer schools, organized by Professors Randy Katz and Kimmo Raatikainen, at the University of California at Berkeley. In addition, the MiNEMA programme has provided opportunities to present and discuss the work.

Lastly, this work would not have been possible without the support of my family and friends.

Contents

Part I

Introduction

Chapter 1

Introduction

Future mobile applications are anticipated to require mechanisms for information processing, gathering, and distribution in dynamic environments. The popularity of information services that use *content delivery* motivates the development of algorithms and protocols for efficient content dissemination and *publish/subscribe* (pub/sub) [51] in mobile environments. Example applications are news, stock market [10] and weather notification services, group discussions and collaboration, and monitoring and controlling sensors and actuators. Publish/subscribe has also been used for distributed metadata management [72], cyber battlefield awareness [106], Internet games [12], software agent communication [122], and automatic hyperlink creation [41].

Mobile computing creates new possibilities for applications and services; however, it also presents new requirements for software that need to be taken into account in applications and in the service infrastructure. In order to support the development and deployment of intelligent applications, a number of fundamental enabling middleware [5] services are needed. Two important services are *event monitoring* and *event notification*, which are vital for supporting adaptability in applications. Environment monitoring and notification are usually provided by the *event* or *notification service*, which allow software components to communicate asynchronously [51, 112, 150]. Event systems are examples of *middleware*, which is a generic and widely used term for services that aim to support the development of software applications.

Event-based systems [19, 31, 49, 93, 100, 126, 154] are seen as good candidates for supporting distributed applications in dynamic and ubiquitous environments because they support decoupled and asynchronous one-to-many and many-to-many information dissemination [44, 110]. Event systems are widely used, because asynchronous messaging provides a flexible

alternative to RPC (Remote Procedure Call) [39, 51]. In the general model of event notification, subscribers subscribe events by specifying their interests using filters. Event producers publish events (also known as notifications), which are matched against active subscriptions. Event *filtering* or *matching* is used to deliver information to the proper set of subscribers [4, 21, 28, 30, 33, 34, 52, 57, 90, 122, 131].

Filtering is a central core functionality for realizing event-based systems and accurate content-delivery. Filtering is performed before delivering a notification to a client to ensure that the notification matches an active subscription from the client. Filtering is therefore essential in maintaining accurate event notification delivery. Filtering may also be performed by a router before a notification is forwarded to another router. This increases the efficiency by avoiding to forward notifications to routers that have no active subscriptions for them. Filters and their properties are useful for many different operations, such as matching, optimizing routing, load balancing, and access control. For example: a firewall is an example of a filtering router and an auditing gateway is a router that records traffic that matches the given set of filters.

Most research on event systems has focused on event dissemination in the fixed network, where clients are stationary and have reliable, low-latency, and high bandwidth communication links. Recently, mobility support and wireless communication have become active research topics in many research projects [46, 66, 67, 110, 111] working with event systems, such as Siena [31] and Rebeca [53, 91]. A mobility-aware event system needs to be able to cope with a number of sporadic and unpredictable end systems, to provide fast access to information irrespective of access location, medium and time. Problems such as delayed events, events generated for offline systems and the delay posed by the transmission of events create synchronization and event delivery problems, need to be solved. *User mobility* occurs when a user becomes disconnected or changes the terminal device. *Terminal mobility* occurs when a terminal moves to a new location and connects to a new access point. Mobility transparency is a key requirement for the system and the middleware system should hide the complexity of subscription management caused by mobility. The reconfiguration of the publish/subscribe router or broker topology [45] and the routing of events through dynamic networks [142] are emerging research topics. In addition, ad hoc environments require novel solutions for event dissemination. Sensor networks [38] and proximity-based notification [87] are examples of ad hoc environments.

Event systems are an integral part of *context-aware* architectures. Context-awareness is considered as an important property of future mobile applications [47]. Context typically pertains to the physical and social situation in which computational entities are embedded. Context-awareness is an active research topic and many middleware systems address context-aware operation [16, 84, 114, 141]. The Context Toolkit defines a distributed infrastructure for hosting context widgets. The toolkit is used to provide applications with contextual information [117]. The GAIA system is used to manage heterogeneous sensors and support context reasoning [114]. Nearly all context-aware systems employ some kind of asynchronous communication abstraction, typically asynchronous events or tuple spaces [18, 26, 78, 94, 95]. Events support context-triggered actions [16], and allow run-time binding of components supporting modularity.

Communication between context providers and consumers may be facilitated using a publish/subscribe event-routing network [84]. Current research prototypes, such as Siena, Rebeca, and Elvin [121, 127], support mobile users and context-aware operation to various degrees. Some event systems are not very suitable for context-sensitive operation because of propagation delays and limitations of routing table algorithms, as discussed later in this thesis. Ideally this separation of concerns simplifies the development of higher level components, because mobility transparency and scalability is handled by the lower pub/sub layer.

This thesis builds on previous research on distributed event systems and presents mechanisms for efficient *content-based routing* and explores the impact of mobility on event systems. One of the first content-based routing data structures was presented in the Siena project [29]. The *filters poset* (partially ordered set) structure was used by event routers to manage subscriptions and advertisements from other routers. In event literature filters that represent subscriptions and advertisements are typically manipulated as sets and we are not aware of efficient data structures for processing frequent filter set additions and removals. The Siena filters poset was found to be limited in terms of scalability, which led to the development of the combined broadcast and content-based (CBCB) routing scheme [32].

The main research questions in this thesis for efficient content-based routing and matching are:

- Is it possible to develop more efficient data structures for routing?
- What routing table configurations are the most efficient?
- How to efficiently use filter merging with a routing data structure?
- How to do temporal matching instead of instantaneous matching?

- How to match subspaces instead of points?
- How to do context-based matching?

To answer these questions, we present the *poset-derived forest data structure* and variants that address the scalability problems of the filters poset and perform considerably better under frequent filter additions and removals than acyclic graph based structures. We present different routing table configurations that combine forests, posets, and dedicated matcher components for flexible and efficient routing. Furthermore, we present the new *DoubleForest* data structure for *temporal subspace matching*.

Most event systems have informal semantics and do not give guarantees on event delivery. Recently, formal semantics for content-based routing protocols and publish/subscribe systems have been proposed [9, 55, 90]. The formal semantics do not take mobility into account. Mobile components typically require that the pub/sub topology is updated and thus it is necessary to prove for a mobility protocol that the safety properties are not violated, which we call *mobility-safety*. Typically, a *stateful* mobility protocol is used that buffers messages for a disconnected client. The JEDI event system was one of the first pub/sub systems to support mobile components in a hierarchical topology of event brokers [43]. The Siena mobility support service was formally verified to maintain safety and liveness [23]. On the other hand, the protocol is based on basic pub/sub primitives and has a high cost in synchronizing the source and target servers. The Rebeca system, which is based on an acyclic graph topology with advertisement semantics, was also extended to support mobile clients, but the mobility-safety of the protocol was not established [54, 93]. Moreover, event literature typically focuses only on subscriber mobility. With advertisement semantics also a publisher mobility protocol is required, but it has not yet been analyzed. In this thesis we examine both subscriber and publisher mobility in different topologies and characterize mobility using mobility-safety and the notion of completeness of the topology.

The main research questions for mobility-aware routing are:

- Are stateful handover protocols mobility-safe?
- What optimizations can be performed and how do they affect mobility-safety?
- How do different router topologies affect the cost and mobility-safety of the handover protocol?
- What if the mobile client moves before an issued subscription or advertisement has been fully propagated?

- What are the upper and lower bounds for cost in terms of message exchanges for different router topologies and how does incompleteness of the routing topology affect these costs?

1.1 Structure of the Thesis

The thesis is structured into six parts as follows: in the first introductory part we present the publish/subscribe paradigm and examine content-based routing.

In the second part we give an overview of content-based routing tables and present a number of new data structures and configurations for efficient routing. We formally define the poset-derived forest and variants, which are useful and versatile structures for routing.

In the third part, we examine mobility in pub/sub topologies and compare the cost of mobility in different routing topologies. We also discuss the lower and upper-bound costs of mobility in terms of exchanged messages.

In the fourth part, we present the DoubleForest data structure, which is a more advanced structure for temporal subspace matching and context-aware matching. The structure supports several semantic operators, namely covering and overlapping. We also consider filter preloading and present a formal framework for filter merging.

In the fifth part, we present several applications for the results of this thesis. First, we consider change notification and context-aware collection and object synchronization using the DoubleForest for mobile clients. Second, we present the smart office scenario that highlights interactions in a modern office environment and motivates the use of content-based routing to realize context-aware computing.

The last part presents the conclusions.

1.2 Contributions

The original and new contributions of this thesis are the following:

- The poset-derived forest data structure and variants for efficient processing of partially ordered sets of filters defined by the covering relation. The forest is simpler and more efficient than the filters poset that was used in the Siena system. We present useful theorems for the data structures and an implementation of a visual tool for inspecting them called the PosetBrowser.

- Optimization of routing tables using posets, forests, and filter merging as the basic building blocks. We present useful designs and examine their performance.

- We present the first formal framework for filter merging and a dynamic algorithm for merging. The algorithm integrates with a content-based routing table.

- We present a new technique for routing called *temporal subspace matching*, which is an advanced technique that addresses the two main limitations of existing content-based routing systems, namely allowing content to be defined using subspaces instead of points, and allowing temporal routing instead of instantaneous routing.

- The DoubleForest data structure for temporal subspace routing and context-aware computing. We present formal definition and prove the correctness of the structure with optimizations. We also show how preloading may be used to achieve constant matching time.

- Characterization of pub/sub mobility using completeness of the topology and mobility-safety, and investigation of the cost of mobility in different topologies. We present the upper and lower bound costs for different topologies and simulation results. We also examine and analyze publisher mobility, which has not, at the time of writing, been addressed in event literature.

- As applications of the forests and temporal subspace routing, we present context-aware collection and object synchronization for mobile clients and a smart office scenario that highlights ubiquitous interactions.

1.3 Research History

The thesis research was carried out in the Fuego Core research project at the Helsinki Institute for Information Technology HIIT. The three-year (2002-2004) research project investigated middleware for mobile, wireless Internet. A public state-of-the-art review of middleware was prepared in the project. This review summarizes standardization and research efforts pertaining to distributed event systems [130]. The general challenges of the wireless and mobile environment, especially for software agents, are discussed in [136].

The main influences and starting points for the research were Antonio Carzaniga's and Gero Mühl's Ph.D. dissertations [27, 90]. The former defined the filters poset structure and investigated different event routing strategies. The latter formalized event routing and introduced filter merging. The presented research pertains to optimizing event routers, provides a formal framework for integrating filter merging into routers, and formalizes client mobility in pub/sub systems.

We initially proposed a mobility-aware event domain with event channel-based topology updates. This mechanism used linear hashing to map channels to servers based on their type [135]. This was motivated by the observation that the generic state-transfer protocol developed in the Siena project relied on flooding the pub/sub network and other solutions were required for efficient handovers. We also considered the benefits of the event system and mobility support for reactive software agents, which are essentially based on asynchronous events [128]. The filter covering, matching, and merging mechanisms were developed as basic building blocks for event systems. An outline of the mechanisms was presented in [129]. The poset-derived forest data structure is presented in [131]. An overview of chapter 5 of this thesis is presented in [134] and [133]. Chapter 8 of this thesis is outlined in [132]. Chapter 9 of this thesis is presented in [137]. The author also contributed to the Wireless World Research Forum's vision on adaptive computing [150].

The filter mechanisms presented in this thesis were used in the Fuego event system [129]. The Fuego event system was demonstrated at the sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004) using a smart office scenario, which showed various interactions in the office environment and illustrated the use of context-sensitive messaging realized using pub/sub primitives. Chapter 10 presents a summary of the Smart Office demonstration. Wireless communication was demonstrated using a GPRS connection with a J2ME (Java 2 Micro Edition) MIDP (Mobile Information Device Profile) phone.

The work presented in this thesis was done by the author with the following qualifications. The definitions and theorems of the poset-derived forest were joint work with Jaakko Kangasharju from HIIT and Theorem 3.7 is by him. In addition, Proposition 5.5, Lemmas 5.6 and 5.7, and Theorem 5.8 were proposed by the author, but the final proofs are by Kangasharju.

Chapter 2

Content-based Event Routing

In this chapter we present and discuss the routing of event messages in a distributed system. We give an overview of well-known distributed router configurations and discuss how the routing decision is made and how the routing information is propagated in the environment. We also briefly consider design patterns and IP multicast.

2.1 Overview

The main entities in a publish/subscribe (pub/sub) system are the publishers and subscribers of information. A publisher publishes an event and a subscriber receives notifications of events that have occurred. There are many names for the entities in pub/sub or event systems, so in this thesis the terms subscriber, consumer, and sink are synonymous. Similarly, publisher, producer, source, and supplier are synonymous. The semantic meaning of an event and its notification is application- and domain-specific, and the mutual agreement on the interpretation of a given notification between the recipients is outside the scope of this work. Each event may be published only once.

An event system may be centralized or distributed in nature and the notification responsibility may be provided by different entities in the environment: producers, a centralized router, or a sequence or a set of routers. In distributed environments a published event is communicated in an event message, also called a notification, using a message transport protocol. This is one of the defining characteristics of event and publish/subscribe systems — the use of asynchronous message passing. The entities may employ point-to-point messaging in communication, but communication may also be based on various multicast and broadcast technologies.

The *event router* is a component that connects the publishers and subscribers and mediates event messages between them. Typically, an event router consists of two parts: a set of connections to neighbouring routers and a set of local clients. Both sets are associated with a *routing table* that contains information about which event messages should be forwarded to which neighbouring router or local client. Neighbouring routers may also be called interfaces or destinations and these are taken to be synonymous in subsequent examination. In filter-based routing the routing table contains a set of filters for each interface and local client. A router with only a single neighbouring router is called an *edge router* or *border router*.

The distribution of event routers is necessary to achieve scalability, reliability, and high-availability. For example, if a producer is responsible for directly notifying a set of subscribers, it is clear that the centralized nature of this kind of *direct notification* is limited in terms of scalability and performance. The scalability of direct notification may be improved by using intermediary components, but this is just a step towards a routing infrastructure. Indeed, many research projects have focused on infrastructure-based notification and investigated different distribution mechanisms for connecting publishers and subscribers efficiently.

In many cases, the subscriber is interested in a very specific event and if the event system does not provide any mechanism for defining interests, the subscriber will receive all event messages published by the producer or producers in question. This is called *flooding* and it is the trivial way to ensure that every subscriber will receive the correct notifications. A message that is sent to a client that does not match the client's interests is called a *false positive*. Similarly, a message that was not delivered, but should have been received by the client is called a *false negative*.

Flooding every event message everywhere is not a scalable solution, which has led to the development of various filtering languages and filter matching algorithms. The scalability limitation is obvious, because the forwarding of event messages requires processing time on various entities of the environment and message transmission uses network resources. Excess and uncontrolled messaging may lead to congestion. Congestion in turn may cause event messages to be dropped.

Filtering allows the subscribers to specify their interest beforehand and thus reduce the number of uninteresting event messages that they will receive. A filter or a set of filters that describes the desired content is included with the subscription message. Filters may also be used to *advertise* the future publication of events. This advertisement information given by publishers may be used to further optimize messaging and the processing over-

Table 2.1: Infrastructure interface operations.

Operation	Description	Semantics
Sub(X,F)	X subscribes filter F	Sub/Adv
Pub(X,n)	X publishes notification n	Sub/Adv
Notify(X,n)	X is notified about notification n	Sub/Adv
Unsub(X,F)	X unsubscribes filter F	Sub/Adv
Adv(X,F)	X advertises filter F	Adv
Unadv(X,F)	X unadvertises filter F	Adv

head of routers. Many filtering languages have been developed, specified, and proposed. We give a brief overview of filtering languages in Section 2.6.

In order to support event filtering and event delivery, an event router needs to provide an interest-registration service and also have an interface for publishing events. Subscribers define their interests using this interest-registration service. Table 2.1 presents the pub/sub operations used by most event systems. The table presents the operations for two different semantics: *subscription semantics* and *advertisement semantics*. The advertisement semantics adds the operations for advertising and unadvertising a filter.

Depending on the expressiveness of the filtering language, a specific field, header, or the whole content of the event message may be filterable. In *content-based routing* the whole content of the event message is filterable. With the introduction of filtering we face the problem of how to propagate this filtering information in the distributed environment. It is not feasible to expect that a producer or a single router is capable of filtering event messages for a large number of subscribers.

The two important parts of a distributed pub/sub system are the router topology, by which we mean the exact nature of how the routers are connected with each other, and how routing information is propagated by the routers. By propagating routing information we mean how the interests, filters, of the subscribers are conveyed towards the publishers of that information. In essence, the routing information stored by a router must enable it to forward event messages either to other routers or to local clients that have previously subscribed to the event messages. The *routing problem* may be described as follows from the viewpoint of a single router: given an input event message, find the correct set of neighbouring routers and local clients that should receive the event message.

Expressiveness and scalability are important characteristics of an event

system [29]. Expressiveness deals with how well the interests of the subscribers are captured by the notification service, and scalability deals with federation, resources and issues such as how many users can be supported and how many routers are required. In addition to expressiveness and scalability, an event system needs to be relatively simple to be manageable, implementable, and to be able to support rapid deployment. Moreover, the system needs to be extensible and interoperable. Other non-functional requirements are: timely delivery of notifications (bounded delivery time), support for Quality of Service (QoS), high availability and fault-tolerance. Event order is an important non-functional requirement and many applications require support for either causal order or total order.

2.2 Router Topologies

A number of different router topologies have been proposed in event literature. Well-known router topologies include: *centralized*, *hierarchical*, *acyclic*, *cyclic*, and *rendezvous point-based* topologies. Centralized routers represent the trivial case for distributed operation, in which subscribers and producers use a client-server protocol for sending and receiving event messages and invoke the interest-registration service provided by the router.

In hierarchical systems each router has a master and a number of slave routers. Notifications are always sent to the master. Notifications are also sent to slaves that have previously expressed interest in the notifications. The basic hierarchical design is limited in terms of scalability, because one master router is the root of the distribution tree and will receive all the notifications produced in the system.

For acyclic and cyclic topologies routers employ a different, peer-to-peer, protocol to exchange interest propagation information and control messages. In this context, the peer-to-peer protocol denotes that the topology is not hierarchical. Acyclic topologies allow more scalable configurations than hierarchical topologies, but they lack the redundancy of cyclic topologies. On the other hand, topologies based on cyclic graphs require techniques, such as the computation of minimum spanning trees, to prevent loops and unnecessary messaging.

The rendezvous point model differs from acyclic and cyclic topologies, because the routing of a specific type of event is constrained by a special router, the *Rendezvous Point* (RP). The RP serves as a meeting point for advertisements and subscriptions and avoids the flooding of advertisements throughout the system. The rendezvous-based model is presented in more detail in Section 5.6. Rendezvous-based systems limit the propagation of

messages using the RP and thus attempt to address scalability limitations presented by the flooding of subscriptions or advertisements. Typically, an RP is responsible for a pre-determined event type. RPs may be used to create a type hierarchy. In this case, a message needs to be sent to the proper RP and any super-type RPs, which may increase messaging cost and limit scalability

The hierarchical topology was used in the JEDI system [15, 43], and an acyclic topology with advertisements in Rebeca [54, 90, 93]. The Siena project investigated and evaluated the topologies with different interest propagation mechanisms [27, 31]. In general, the acyclic and cyclic topologies have been found to be superior to hierarchical topologies [15, 27, 92]. The router topology in Gryphon [68, 125] is based on clusters called *cells* and redundant *link bundles* that connect cells. Most research has focused on static connections between routers. Dynamic connections between routers have been investigated in [45] and [142].

A number of *overlay*-based routing algorithms and router configurations have been proposed. An application layer overlay network is implemented on top of the network layer and typically overlays provide useful features such as fast deployment time, resilience and fault-tolerance. An overlay-routing algorithm leverages underlying packet-routing facilities and provides additional services on the higher level, such as searching, storage, and synchronization services.

Good overlay routing configuration follows the network level placement of routers. Many overlays are based on *Distributed Hash Tables* (DHTs), which are typically used to implement distributed lookup structures. Many DHTs work by hashing data to routers/brokers and using a variant of *prefix-routing* to find the proper data broker for a given data item. Hermes [109] and Scribe [116] are examples of publish/subscribe systems implemented on top of an overlay network and are based on the rendezvous point routing model. The Hermes routing model is based on advertisement semantics and an overlay topology with rendezvous points. This model was found to compare favourably to the Siena advertisement semantics using an acyclic topology [109].

2.3 Interest Propagation

The main functions of a router are to match notifications for local clients and to route notifications to neighbouring routers that have previously expressed interest in the notifications. The interest propagation mechanism is an important part of the distributed system and heart of the routing

algorithm. The desirable properties for an interest propagation mechanism are small routing table sizes and forwarding overhead [92], support for frequent updates, and high performance.

With subscription semantics the routers propagate subscriptions to other routers, and notifications are sent on the *reverse path* of subscriptions. In *simple routing* each router knows all active subscriptions in the distributed system, which is realized by flooding subscriptions. In *identity-based routing* a subscription message is not forwarded if an identical message was previously forwarded. This requires an identity test for subscriptions. Identity-based routing removes duplicate entries from routing tables and reduces unnecessary forwarding of subscriptions. In *covering-based routing* a covering test is used instead of an identity test. This results in the propagation of the most general filters that cover more specific filters. On the other hand, unsubscription becomes more complicated because previously covered subscriptions may become uncovered due to an unsubscription. *Merging-based routing* allows routers to merge exiting routing entries. Merging-based routing may be implemented in many ways and combined with covering-based routing [92]. Also, merging-based routing has more complex unsubscription processing when a part of a previously merged routing entry is removed.

With advertisement semantics the routers first propagate advertisements and then, on the reverse path of advertisements, the subscriptions. Notifications are forwarded on the reverse path of subscriptions in both semantics. Advertisements may be used with various routing mechanisms. Advertisements typically have their own routing table and they are managed using the same algorithms as subscriptions. The removal of an advertisement causes a router to drop all overlapping subscriptions for the neighbour that sent the unadvertisement message. Similarly, an incoming advertisement requires that overlapping subscriptions are forwarded to the neighbour that sent the advertisement message. The use of advertisements considerably improves the scalability of the event system [15, 27, 92].

One of the first formulations of a wide-area pub/sub system based on these two semantics with optimizations was presented in the Siena system, which used covering relations between filters to prevent unnecessary signalling. The Siena system used the notion of covering for three different comparisons: matching a notification against a filter, covering relation between two subscription filters, and overlapping between an advertisement filter and a subscription filter. Covering and overlapping relations have been used in many later event systems, such as Rebeca [93] and Hermes [108, 109]. The combined broadcast and content-based (CBCB) routing

scheme extends the Siena routing protocols by combining higher-level routing using covering relations and lower-level broadcast delivery [32]. The protocol prunes the broadcast distribution paths using higher-level information exchanged by routers.

2.4 Definitions

We follow the basic concepts defined in the Siena system [29] and later refined and extended in Rebeca [90]. A filter F is a stateless Boolean function that takes a notification as an argument. Later in the thesis, we also use lower-case letters to denote filters. Many event systems use the operators of Boolean logic, *AND*, *OR*, and *NOT*, to construct filters. A filtering language specifies how filters are constructed and defines the various predicates that may be used. A predicate is a language-specific constraint on the input notification. We present the notification data model and the filtering language used in the experimentation part of this thesis in Appendix A.

A filter is said to match a notification n if and only if $F(n) = true$. The set of all notifications matched by a filter F is denoted by $N(F)$. A filter F_1 is said to cover a filter F_2, denoted by $F_1 \sqsupseteq F_2$, if and only if all notifications that are matched by F_2 are also matched by F_1, i.e., $N(F_1) \supseteq N(F_2)$. We also say that F_1 has *equal or greater selectivity* than F_2. Similarly, F_2 has *equal or lesser selectivity* than F_1. The filter F_1 is equivalent to F_2, written $F_1 \equiv F_2$, if $F_1 \sqsupseteq F_2$ and $F_2 \sqsupseteq F_1$. The filter F_1 is *incomparable* with F_2, if $F_1 \not\sqsupseteq F_2$ and $F_2 \not\sqsupseteq F_1$. The $\sqsupseteq$ relation is reflexive and transitive and defines a partial order.

A set of n filters $S_F = \{F_1, \ldots, F_n\}$ covers a filter F_k if and only if $N(S_F) \supseteq N(F_k) \Leftrightarrow \bigcup_i^n N(F_i) \supseteq N(F_k)$. Covering of two sets follows from this.

An advertisement A is said to overlap with the subscription S, denoted by $A \simeq S$, when their filters overlap. Two filters, F_1 and F_2, are overlapping if and only if $N(F_1) \cap N(F_2) \neq \emptyset$. The data structures presented in this thesis are filter-language agnostic. The covering, overlapping, and merging mechanisms used in the experimentation part of the thesis are discussed in Appendix A.

Example 2.1 *We define three filters using the notation (filter, constraint):* $(F_1, x < 10)$*,* $(F_2, x \in [5, 9])$*, and* $(F_3, x \in [8, 15])$*. The constraints are defined for the variable* x *over integers. We have* $F_1 \sqsupseteq F_2$*, since the range* $[5, 9]$ *is contained in* $x < 10$*. We have* $F_1 \not\sqsupseteq F_3$*, because the range* $[8, 15]$

is not totally contained in $x < 10$. *It is also clear that the ranges do not contain each other, hence* $F_2 \not\supseteq F_3$ *and* $F_3 \not\supseteq F_2$. *On the other hand, it is clear that* $F_1 \simeq F_2$. *Also* $F_1 \simeq F_3$ *since* $x < 10$ *and* $[8, 15]$ *overlap.*

2.5 Routing Decision

Message routing systems may be classified into four categories: channel-based, subject-based, header-based, and content-based [130]. Channel-based systems make the routing decision based on channel names that have been agreed by the communicating participants. Subject-based systems make the routing decision based on a single field. Header-based systems use a special header part of the message in order to make the routing decision. For example, SOAP [149] supports header-based routing of XML-messages. Finally, content-based systems use the whole content of the message in making the decision [30]. Next, we describe the four well-known categories of message routing systems.

Channel/topic-based. Routing decision is made based on the channel on which the event is published. A channel is a discrete communication line with a name. Named channels are also called topics, and they represent an abstraction of numeric network addressing mechanisms. Usually with channel-based messaging, new channels need to be added to the address space, because the producers and consumers must agree on a channel. Channel-based messaging allows the use of IP multicast groups [103]. The channels can be allocated to multicast addresses.

Subject-based. Routing decision is made based on the subject of the event. Subject-based routing is more expressive than channel-based routing. On the other hand, a single field may not be enough to properly describe the content of a message.

Header-based. Routing decision is made based on a number of fields in the message header. In header-based routing the message has two distinct parts: the header and the body. Only fields in the header are used for making routing decisions. Header-based routing is more expressive than subject-based and has performance advantage to content-based routing, because only the header of a message is inspected.

Content-based. Routing decision is made based on the whole content of a message, for example strongly typed fields in the event message. Content-based routing is the most expressive of the four types.

Content-based event routing has been proposed as one of the requirements for advanced applications, in particular for mobile users [33, 44] and context-sensitive messaging [84]. The latter mechanism formulates the current and future context of entities as event filters and subscribes to them. The Elvin event broker [121] is used to deliver messages to the recipients based on the subscribed context filters. Context-sensitive messaging may be used, for example, to control and monitor a set of mobile robots in a particular location [84].

2.6 Filtering and Merging

Event filtering is used in most current event architectures. The CORBA Notification Service uses the extended Trader Constraint Language (TCL) [100]. The Java Messaging Service (JMS) supports a subset of SQL-92 for event filtering [126]. These two specifications do not define any particular way of doing distributed event delivery although distributed filtering may be implemented based on them.

Research efforts such as JEDI [43], Elvin [127], Rebeca [91], Gryphon [68], and Siena [31] have investigated distributed event filtering. Wide-area scalability of event filtering was investigated in the Siena architecture and they define filter relationships formally using covering relations. Filter covering is used in many systems to find the non-covered set of filters, or minimal cover set, that is propagated by event routers. Attribute counter-based algorithms for finding the set of covering filters for a given input filter and the set of mergeable filters were presented in [90]. On the other hand, these algorithms work only in the context of the specific attribute filter model and performance results for frequent additions and deletions were not discussed.

The first matching algorithms supported only equality tests and relational operators for integers. Recently an extended attribute counter algorithm was proposed that supports substring matching and uses a selectivity table for removing unmatchable predicates [34]. In general, filter matching is done by counting attributes using the counting algorithm [28, 34, 88, 90, 104], counting and clustering [69], using a tree-based data structure [4], or a *binary decision diagram* (BDD) [20]. Fast matching algorithms combine client-side processing, caching, and filter clustering [52]. An algorithm has been proposed for RSS-feeds and RDF-based metadata [107]. Recently, a unified approach to routing, covering, and merging was presented in [79].

Many matching mechanisms do not take the distribution and selectivity of filters into account. Efficient selectivity-based filtering has been exam-

ined in [65]. Selectivity-based filtering evaluates the most general filters first that have the highest selectivity. A high selectivity can be estimated based on different information: the distribution of events, the distribution of subscriptions, or both. In addition to exact event matching also approximate matching has been proposed based on fuzzy logic [83]. The Elvin [127] filtering language is based on Lukasiewicz's tri-state logic with values *true*, *false*, and *undecidable*.

W3C is specifying and working on two XML query languages: XPath [143] and XQuery [144], which may also be used in the routing of events that are represented using XML. Efficient XPath filtering is an active research topic. Most XPath query evaluation implementations run in exponential time to the size of input queries [59]. XPath query covering and merging are computationally demanding, which motivates simpler schemes. Tree pattern aggregation is a recent research area and covering algorithms and a minimization algorithm have been presented for conjunctive tree queries [35].

Filter merging is a technique to find the minimum number of filters and constraints that have maximal selectivity in representing a set of subscriptions by modifying constraints in the filters. Merging and covering are needed to reduce processing power and memory requirements both on client devices and on event routers. These techniques are typically general and may be applied to subscriptions, advertisements, and other information represented using filters.

A filter-merging-based routing mechanism was presented in the Rebeca distributed event system [90]. The mechanism merges conjunctive filters using perfect merging rules that are predicate-specific. Routing with merging was evaluated mainly using the routing table size and forwarding overhead as the key metrics in a distributed environment. Merging was used only for simple predicates in the context of a stock application [90, 92]. The integration of the merging mechanism with a routing data structure was not elaborated and we are not aware of any results on this topic.

The optimal merging of filters and queries with constraints has been shown to be NP-complete [42]. Subscription partitioning and routing in content-based systems have been investigated in [145, 146] using Bloom filters [13] and R-trees for efficiently summarizing subscriptions.

Bloom filters are an efficient mechanism for probabilistic representation of sets, and support membership queries, but lack the precision of more complex methods of representing subscriptions. To take an example, Bloom filters and additional predicate indices were used in a mechanism to summarize subscriptions [138, 139]. An Arithmetic Attribute Constraint Summary (AACS) and a String Attribute Constraint Summary (SACS)

structures were used to summarize constraints, because Bloom filters cannot capture the meaning of other operators than equality. The subscription summarization is similar to filter merging, but it is not transparent, because routers need to be aware of the summarization mechanism. Filter merging, on the other hand, does not necessarily require changes to other routers. In addition, the set of attributes needs to be known a priori by all brokers and new operators require new summarization indices. The benefit of the summarization mechanism is improved efficiency, since a custom-matching algorithm is used that is based on Bloom filters and the additional indices.

A BDD-based merging algorithm has been proposed in [79]. The exact rules for filter merging were not elaborated in this work. The algorithm removes all subscriptions, which are covered by a new merger. This requires that all routers are aware of the merging technique in order to support safe unsubscriptions.

2.7 Design Patterns

Design patterns are software engineering designs that have been observed to work well. Patterns are found in different contexts, they provide a solution for a well-defined problem area, and digress the various dimensions of the problem [119]. Patterns are classified into different groups based on their level of abstraction. Architectural patterns summarize good architectural designs; for instance the broker pattern that is used in the CORBA architecture [99]. Design patterns capture the essence of medium level, language independent, design strategies in object-oriented design. Moreover, idioms represent programming-language-level aspects of good solutions [119].

The three well-known patterns for event notification are: the *observer pattern* [58], the *event-channel pattern*, and the *notifier pattern* [60]. The observer pattern allows subscribers to directly register with a producer. This pattern couples the entities together and does not define how the producers are located. The pattern does not scale to large numbers of subscribers per object; however, it allows the use of a mediator that improves flexibility of the system. The observer-pattern is used, for example, in the Java and Jini event models [113]. The *publish-register-notify*, a pattern similar to the observer pattern, is used in the Cambridge Event Architecture [8, 63].

The event-channel and notifier patterns, on the other hand, decouple subscribers and producers by introducing a broker that mediates events on their behalf. The event channel and notifier also support various non-functional requirements, such as QoS and disconnected operation. The

event-channel and notifier patterns are similar, but the notifier also abstracts the location and distribution of event brokers, whereas with channels the client must first obtain the reference of the channel. The notifier pattern may be realized by using the observer pattern and mediators or proxies [151]. The event channel pattern is used in the CORBA Event Service [99] and Notification Service [100]. A separate specification defines how CORBA event channels are connected to form communication topologies [101].

2.8 Multicast

IP multicast is a simple, scalable and efficient mechanism to realize simple group-based communication. IP multicast routes IP packets from one sender to multiple receivers. Participants join and leave the group by sending a packet using the IGMP (RFC 1112) protocol to a well-known group multicast address. IP multicast groups are not very expressive. They partition the IP datagram address-space and each datagram belongs at most to one group. Moreover, IP multicast is a best-effort unreliable service, and for many applications a reliable transport service is needed.

Event systems may use multicast to deliver notifications to appropriate event routers or servers. Not many event systems take advantage of network level IP multicast. An evaluation of different algorithms for mapping subscribers to multicast groups is presented in [103]. Multicast works well in closed networks, however, in large public networks multicast or broadcast may not be practical. In these environments universally adopted standards such as TCP/IP and HTTP may be better choices for all communication [68].

Part II

Posets and Forests: Towards Efficient Routing

Chapter 3

Posets and Forests

This chapter presents the central building blocks of a content-based routing table: the filters poset and forest data structures. We start with an overview of routing tables and present a number of interesting forest and poset configurations for efficient routing. After the overview, we present the filters poset in more detail and then formally define new data structures for content-based routing: the poset-derived forest and variants of the forest.

3.1 Routing Tables

Most research on content-based routing has focused on distributed routing with various semantics or the efficient matching of filters. The routing tables of content-based routers are typically represented as sets and the mechanisms for inserting and removing filters are left unspecified. For example, JEDI [43] and Hermes [109] keep filters in a simple table, and Rebeca uses sets and a counting algorithm for finding covering filters and mergeable filters [90]. Two counting-based algorithms are needed for routing. One to determine the covered filters, and one to determine the covering filters. A unified approach based on Binary Decision Diagrams (BDDs) has been proposed in [79], which we present in more detail in Section 4.8.

The desirable characteristics for a content-based routing table are efficiency, small size, support for frequent updates, and extensibility and interoperability. The routing table data structure should be generic enough to support a wide range of filtering languages.

The filters poset (FP) data structure was used in the Siena system to store filters by their covering relations and manage information related to forwarded messages. The filters poset can be thought of as the routing table for a Siena router. The poset stores filters by their generality and

may also be used to match notifications against filters by traversing only matching filters in the poset starting from the most general filters. We call the set of most general filters that covers other filters the *root set* of the data structure in question. The root set is also called the *non-covered set* or the *minimal cover set*.

The filters poset is a generic data structure and may be used with various filter semantics, which makes it attractive for dynamic environments. The poset may also be used for various interest propagation mechanisms, such as subscription and advertisement semantics. On the other hand, this generality has a performance drawback. One of the findings in Siena was that the filters poset algorithm limits the performance of routers and more efficient solutions are needed [32].

We have specified and developed data structures and mechanisms for improving the scalability of content-based routers in hierarchical and peer-to-peer routing:

Poset-derived Forest (PF): This is the basic forest data structure for finding the non-covered set of filters. Emphasis is on very fast additions, deletions, and computation of the non-covered set. The main usage scenario for the PF is the management of filters from local clients and border routers.

Balanced Poset-derived Forest (BF): Similar to the PF, but for each node maintains an index of interfaces that are reachable through the node towards the descendants of the node. The index is useful when performing interface-specific operation, such as pruning, because it allows to quickly locate the required elements in the forest. The main usage scenario for the BF is the management of filters from local clients and border routers.

Non-redundant Poset-derived Forest (NRF): Similar to the former structure, but guaranteed not to contain any redundant filters. This makes the NRF equivalent to the FP in hierarchical routing. It may also be used for peer-to-peer routing.

3.2 Siena Filters Poset

The filters poset data structure was used in the Siena distributed event system for maintaining covering relations between filters [29]. In Siena's peer-to-peer configurations the poset stores additional information for each subscription that is inserted into the poset. The *subscribers(f)* set gives the set

of subscribers for the given subscription filter f, and similarly, *forwards(f)* contains the subset of peers to which f needs to be sent. Algorithm 1 presents the steps needed to process a subscription *subscribe(X,f)* where X is the subscriber and f is the filter representing the subscription [29, 31].

Algorithm 1 Filter processing in the subscription *subscribe(X,f)*.

1. If a filter f' is found for which $f' \sqsupseteq f$ and $X \in$ *subscribers(f')* then the procedure terminates, because f for X has already been subscribed by a covering filter.

2. If a filter f' is found for which $f' \equiv f$ and $X \notin$ *subscribers(f')* then X is added to *subscribers(f')*. The server removes X from all subscriptions covered by f. Also, subscriptions with no subscribers are removed.

3. Otherwise, the filter f is placed in the poset between two possibly empty sets: immediate predecessors and immediate successors of f. The filter f is inserted and X is added to *subscribers(f)*. The server removes X from all subscriptions covered by f, and subscriptions with no subscribers are also removed.

In distributed operation based on an acyclic graph router topology, the Siena server defines the set *forwards(f)* as presented in the equation

$$forwards(f) = neighbours - NST(f) - \bigcup_{f' \in P_s \wedge f' \sqsupseteq f} forwards(f'). \qquad (3.1)$$

The *neighbours* set contains the event brokers connected to the current broker (one application-level hop distance). The functor *NST* (Not on any Spanning Tree) means that the propagation of f must follow the computed spanning trees rooted at the original subscribers of f. With acyclic topologies *NST* contains the neighbour that sent f. P_s denotes the subscription poset. Using the equation, f is never forwarded to the neighbour that sent it. Due to the last term of the equation the subscription is not forwarded to any routers that have already been sent a covering subscription.

Because X is removed from all subscriptions covered by f, an intermediary server does not know which subscriptions should be forwarded due to unsubscription. This information is essentially lost by this optimization; however, the origin of the subscriptions has this information and propagates any subscriptions due to the unsubscription in the same message, which is applied atomically by other servers. The *unsubscribe(X,f)*

removes X from the *subscribers* set of all subscriptions that are covered by f. Filters with empty subscriber sets are removed. Algorithm 2 gives an outline of subscription processing. The model may be extended with advertisements [31].

Algorithm 2 Message handlers for subscription semantics.

IncomingSub(f,*source*)

1. Add (f,*source*) to P_s.
2. Forward subscription message using *forwards(f)* to any new neighbours in the set.

IncomingUnsub(f,*source*)

1. Remove (f,*source*) from P_s.
2. Let F_O denote the old forwards set and F_N a newly computed forwards set for f after the subscriber *source* has been removed from the *subscribers* set. If the *subscribers* set is empty then $F_N = \emptyset$. The unsubscription is forwarded to $F_O \setminus F_N$. The set may be empty if there are subscriptions from other neighbours that cover f. The *forwards* sets of subscriptions covered by f may change, which may require the forwarding of new subscriptions. Any uncovered subscriptions in P_s are forwarded with the unsubscription message. An uncovered subscription is such that its forwards set gains an additional element due to the removal of a covering filter.

3.2.1 Forwards Sets

The message-forwarding behaviour of hierarchical routing is simple. This behaviour becomes more complex when a router has multiple neighbouring routers. Siena uses the *forwards* set to compute destinations for messages in peer-to-peer routing.

The *forwards(f)* set is determined using Equation 3.1. The last term of the equation means that the removal of an entry in a *forwards* set may affect the *forwards* sets of other subscriptions. This happens during unsubscriptions and may require some of the uncovered subscriptions to be forwarded.

Figure 3.1 presents a routing scenario with an event server or router S with three neighbours *I1*,*I2*, and *I3*. Figure 3.2 illustrates the use of the *forwards* set in subscription in this scenario. Five subscription operations are sent to the server and the trace is shown in the figure. The first two subscriptions are root filters and they are forwarded to other output servers except the one that sent them. *I1* sent filter a and therefore a is forwarded to *I2* and *I3* but not to *I1*. The third and fourth subscriptions need to be forwarded to *I1* in order to avoid false negatives. Finally, the *forwards* set for the last subscription is empty, so it is not forwarded.

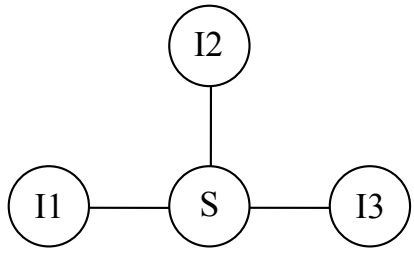

Figure 3.1: Example routing scenario with three neighbours.

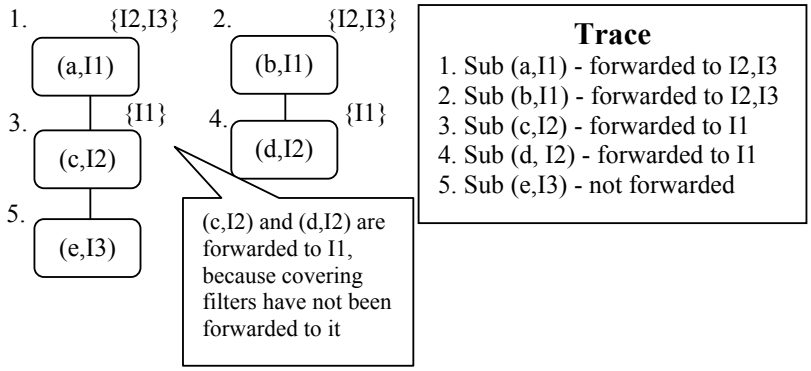

Figure 3.2: Example of the forwards set in subscription.

Figure 3.3 gives an example of an unsubscription operation. The first subscription of the previous example is removed and the unsubscription is sent to *I2* and *I3*. The subscription $(c, I2)$ is uncovered and since it has only been forwarded to *I1* it has to be sent also to *I3* but not to *I2*

that originally sent it. The *forwards* set of the direct descendant of the uncovered subscription also changes and the subscription needs to be sent to *I2*.

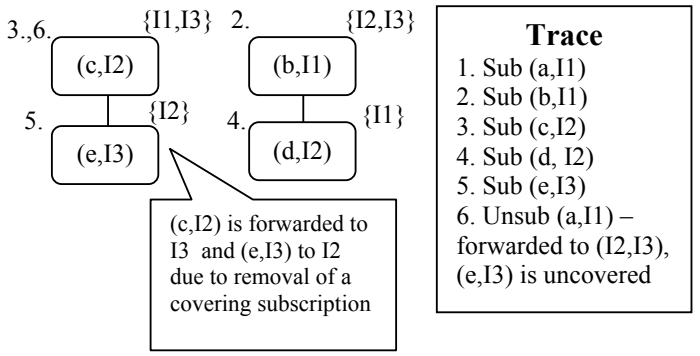

Figure 3.3: Example of the forwards set in unsubscription.

The following example illustrates how *forwards* sets are computed using Algorithm 2. Let the *subscribers* set be $\{I1,I2\}$ for the filter f. The old *forwards* set is $F_O = \{I1,I2,I3\}$. When f is unsubscribed by *I1*, the new *subscribers* set contains only *I2*, and the new *forwards* set is $F_N = \{I1,I3\}$. Therefore, the unsubscription is sent to $F_O \setminus F_N = \{I1,I2,I3\} \setminus \{I1,I3\} = \{I2\}$. Similarly, should a issue a new subscription for f, the new *forwards* set will have *I2* as a new neighbour.

It is not necessary to store the *forwards* sets for filters and they may be computed at run-time. A recursive function is not needed to compute the *forwards* sets, because only the first two levels may have elements in their *forwards* sets as discussed in Section 3.2.3.

Correctness of Forwards Sets The forwarding behaviour of FP is correct for a single neighbour. This is the case for hierarchical routing. Correctness follows from the observation that FP computes the correct non-covered set and maintains it during additions and deletions. Any redundant filters are removed by sub-poset pruning.

Peer-to-peer routing may be modelled by constructing the root set for each interface. We call this the *naive forwarding mechanism*. The set must be maintained at the router behind the designated interface. It is evident that if communication delay is not taken into account, by propagating root sets, the forwarding information of the distributed system is correct. The

correctness of set-based content-based pub/sub is discussed more formally in Section 5.2.

The filters poset aggregates the interface-specific root sets by computing the covering relations (direct predecessors and successors) for filters from all interfaces. FP also uses two additional sets for subscription semantics:

- the *subscribers* set is used to associate filters to the interfaces that sent them and sub-poset pruning is used to remove redundancy,
- the *forwards* set is used to store forwarding information.

Thus FP aggregates routing information and is more compact than the naive approach.

Theorem 3.1 *FP has the same forwarding behaviour as the naive forwarding mechanism.*

Proof. It is clear that given the sets of the naive mechanism, we can build an FP that has the same forwarding behaviour. We show that given an FP, we can construct the sets required by the naive mechanism.

The construction is performed by computing the root sets for each interface separately. The iteration is done over the FP root set for each interface and ignoring any filters installed only by the current interface. In this case, the second level of the poset may need to be inspected. The resulting interface-specific sets are the sets used by the naive mechanism. □

3.2.2 Poset Algorithm

We could not locate a detailed analysis of the add and remove algorithms or benchmark results. The following description is based on the Siena Java implementation [140]. The implementation supports hierarchical operation. The insert operation follows the rules presented at the beginning of this section. The relations between filters are maintained using two lookup structures: *predecessor(f)* and *successor(f)*, where the former maintains a list of immediate predecessors of f and similarly the latter maintains the immediate successors of f. Elements in the *predecessor(f)* set cover f, and similarly elements in the *successor(f)* set are covered by f.

The delete (*del*) operation can be made efficient using these two lookup structures by simply removing the filter and connecting the predecessors and successors accordingly. First f is disconnected from every successor of f. If f is a root filter this adds those successors of f that have empty predecessor sets to the root set. Otherwise, f is removed from the successor

sets of its predecessors and a predecessor x of f is connected with a successor y of f only if x does not have an immediate successor x' that covers y.

The predecessor and successor sets are determined by the *add* operation. The set *predecessor(f)* is located by walking covering filters in the poset from the root set in breadth-first order and adding the last covering filter in the poset to the predecessor set for every visited branch. The *successor* set starts from the *predecessor(f)* set or, if it is empty, the root set of the poset and walks the poset in breadth-first order looking for the direct successors of f.

The add operation inserts f between the two sets *predecessor(f)* and *successor(f)*. This operation simply updates relevant lookup structures to reflect the new node. After insertion the sub-poset defined by successors of f is pruned from empty nodes according to the rules. Duplicate filter processing may be optimized by detecting duplicates using a hash table and only updating the *subscribers* set.

3.2.3 Useful Properties

In this section we present and give proofs for useful properties of the FP. We assume subscription semantics, but similar proofs may be constructed also for advertisement semantics. The results in this section may be used to simplify and optimize the data structures.

We use the (F, I) notation to denote that the filter F was received from the interface I. In subsequent examination, by saying that a *node* of a data structure covers another node, we mean that the filter contained in the node covers a filter contained in the other node.

Property 3.2 *Minimum interface property: For every node x in the poset, if interface c is in subscribers(x), then c cannot be in the subscriber set subscriber(y) of any descendant y of x.*

Proof. It is sufficient to show that the *subscribe* operation presented in Algorithm 1 cannot break this property. The three conditional cases of the algorithm need to be considered. In the first case no modification of the poset is made, so the property cannot be broken. In the second and third cases the removal of c from covered filters and the removal of nodes with no subscribers will re-establish this property if it was broken temporarily by the addition. □

Property 3.2 states that no interface appears twice in any transitive closure of the covering relation in the poset. The algorithms for FP ensure that this property is maintained. Theorem 3.3 states that when local clients are treated as one external interface, the poset has the maximum depth of

k, where k is the number of external interfaces. We call this the *external client* assumption. With this assumption all active filters from a client are incomparable.

From Property 3.2 it follows that covering relations exist only for filters from different interfaces. In other words, filters from the same interface are incomparable in the data structure. This fact may be used to optimize the data structure.

Theorem 3.3 *The poset has a maximum depth of k nodes when local clients are represented using a single interface (external client assumption), where k is the number of interfaces.*

Proof. According to Property 3.2, any transitive closure of the covering relation in the poset has at most one appearance of each interface, and every node has at least one interface. The maximum depth is achieved for linear order, and the depth is exactly k when each node has a different interface. As an example, consider the sequence $(x > 0, i_1)$,$(x > 5, i_2)$,$(x > 10, i_3)$, $(x > 12, i_4)$. □

Theorem 3.4 *Only elements in the root set or the direct successors of elements in the root set may have a non-empty forwards set.*

Proof. The *forwards* set is computed according to Equation 3.1. It is clear that the elements of the root set always have non-empty *forwards* sets. In our case the size must be either $|neighbours|$ for a filter subscribed by a local client or more than one neighbour, or $|neighbours|-1$ if the filter has only one neighbouring broker as a subscriber. This means that the direct descendants of root elements can have at most one entry in their *forwards* sets. The direct successor must be of different interface than the root filter because of Property 3.2. □

Theorem 3.4 shows that it is necessary to compute the forwards set only for the first two levels: the root level and the level directly under the root level.

3.3 Poset-derived Forests

3.3.1 Poset-derived Forest Data Structure

The poset-derived forest data structure is used to store filters by their covering property with other filters and is similar to the Siena filters poset presented in the previous section with the exception that each node has only one parent (Figure 3.4). The forest is a generic data structure and

may be extended with the sets *subscribers(f)*, *forwards(f)*, *advertisers(a)*, and *forwards(a)*.

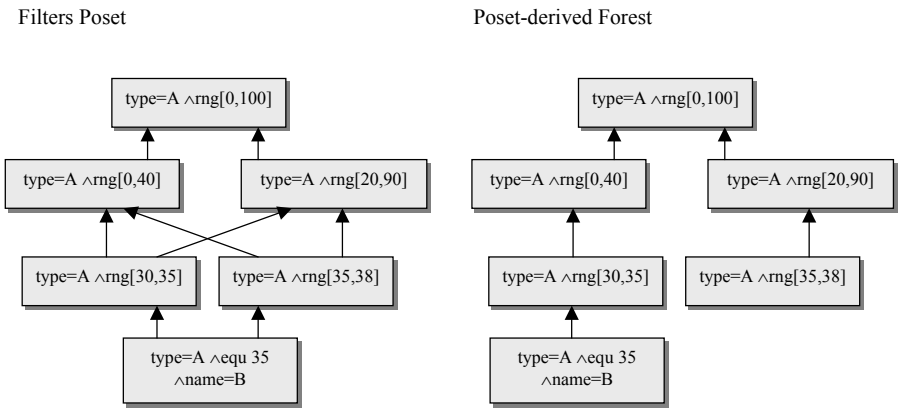

Figure 3.4: Filters poset and poset-derived forest for the same set of filters.

A pair $(\mathcal{F}, \succ)$ represents the poset-derived forest, where $\mathcal{F}$ is a finite set of filters and $\succ$ is a subset of the covering relation. More formally:

Definition 3.5 *A pair $(\mathcal{F}, \succ)$ is a poset-derived forest with base set $\mathcal{F}$, if*

1. *$\mathcal{F}$ is a finite set of filters and $\succ$ is a relation between filters in $\mathcal{F}$.*
2. *For each $a \in \mathcal{F}$ there is at most one $b \in \mathcal{F}$ for which $b \succ a$, i.e., $(\mathcal{F}, \succ)$ is a forest with the relation $\succ$ going from parent to child.*
3. *If $a, b \in \mathcal{F}$ and $b \succ a$, then $b \sqsupseteq a$.*

It is convenient for uniformity of treatment to imagine the roots of the trees belonging to $(\mathcal{F}, \succ)$ to be children of a node not in $\mathcal{F}$, which we will call the *imaginary root* of $(\mathcal{F}, \succ)$.

$(\mathcal{F}, \succ)$ is called *maximal in* $\mathcal{F}$ if there do not exist $a, b \in \mathcal{F}$ for which $(\mathcal{F}, \succ \cup \{(a, b)\})$ is a poset-derived forest. It is clear that any poset-derived forest can be extended to a maximal one by adding pairs to the relation $\succ$. While there may exist several maximal forests for the same base set, Theorem 3.7 shows that it is still meaningful to speak of a "root set". The forest can be used to easily compute the minimal cover for F (Corollary 3.8). Theorem 3.9 is useful when using the poset-derived forest to detect and compare the overlap of filters. The theorems of Section 3.2.3 are also applicable to forests.

In applications we typically require the maximality criterion to hold. The maximality criterion may be generalized to apply at any level of the forest with the following definition.

Definition 3.6 *A poset-derived forest $(\mathcal{F}, \succ)$ is sibling-pure at node a (a may be the imaginary root) if there do not exist $b, c \in \mathcal{F}$ for which $a \succ b$, $a \succ c$, and either $c \sqsupseteq b$ or $b \sqsupseteq c$. The forest is sibling-pure if it is sibling-pure at every node, including the imaginary root.*

Sibling-purity at a node means that the node's children in the forest are incomparable with each other. When this holds for every node, the forest is sibling-pure. In other words, a sibling-pure forest ensures that nodes are locally placed as far away from the root nodes as possible. In subsequent examination we assume that the forest is sibling-pure.

Theorem 3.7 *The set of roots of the trees in a poset-derived forest maximal in $\mathcal{F}$ depends only on $\mathcal{F}$ and not on the forest relation $\succ$. Furthermore, this set is a subset of the set of roots of any poset-derived forest with base set $\mathcal{F}$.*

Proof. Let $(\mathcal{F}, \succ)$ be a maximal poset-derived forest, and let $(\mathcal{F}, \succ')$ be another poset-derived forest having a root a that is not a root in $(\mathcal{F}, \succ)$. Then there exists a $b \in \mathcal{F}$ for which $b \sqsupseteq a$. These two observations mean that adding (b, a) to $\succ'$ does not break any of the properties of Definition 3.5, so $(\mathcal{F}, \succ')$ is not maximal in $\mathcal{F}$.

For the second part, let $a \in \mathcal{F}$ be a root in a maximal poset-derived forest. As above, since the forest is maximal, there cannot exist a $b \in \mathcal{F}$ for which $b \sqsupseteq a$. Hence a must be a root of a tree in any poset-derived forest with base set $\mathcal{F}$. □

A *cover* for a set of filters $\mathcal{F}$ is defined to be a set $\mathcal{G} \subseteq \mathcal{F}$ such that for each $f \in \mathcal{F}$ there exists a $g \in \mathcal{G}$ for which $g \sqsupseteq f$. This cover is *minimal* if it does not contain a proper subset that is also a cover of $\mathcal{F}$. It is clear that $\sqsupseteq$ cannot hold between two members of a minimal cover.

Corollary 3.8 *For any set of filters $\mathcal{F}$ there exists a unique minimal cover. This minimal cover is the set of root nodes of any maximal poset-derived forest with base set $\mathcal{F}$.*

Proof. Any root set of a poset-derived forest with base set $\mathcal{F}$ is a cover of $\mathcal{F}$ and conversely, for each cover it is possible to construct a poset-derived forest with that cover as its root set. The claim now follows directly from Theorem 3.7. □

For two sets of filters $\mathcal{F}, \mathcal{G}$ the *overlap of* $\mathcal{F}$ *with* $\mathcal{G}$ is defined to be the subset of those elements in $\mathcal{G}$ which overlap with some element of $\mathcal{F}$.

Theorem 3.9 *For any sets of filters $\mathcal{F}, \mathcal{G}$ the overlap of $\mathcal{F}$ with $\mathcal{G}$ is the same as the overlap of the minimal cover of $\mathcal{F}$ with $\mathcal{G}$.*

Proof. Obviously the overlap of the minimal cover is contained in the overlap of the full set. Now let $g \in \mathcal{G}$ belong to the overlap of $\mathcal{F}$ with $\mathcal{G}$. Then there exists an $f \in \mathcal{F}$ for which $f \simeq g$, or $N(f) \cap N(g) \neq \emptyset$. Now there exists an f' in the minimal cover of $\mathcal{F}$ for which $f' \sqsupseteq f$, or $N(f) \subseteq N(f')$. From this it follows that $N(f) \cap N(g) \subseteq N(f') \cap N(g)$, and therefore $f' \simeq g$ and g belongs to the overlap of the minimal cover of $\mathcal{F}$ with $\mathcal{G}$. □

The two central operations for the poset-derived forest are the addition of new elements to the forest, and deleting existing items from it. The operations are presented in Algorithm 3.

Sibling-purity is very easy to maintain for the *add* operation, but more complicated for the *del* operation. It is expected that for some application areas, such as hierarchical routing or the management of filters from local clients, it is not necessary to maintain sibling-purity for the *del* operation. This simplifies the *del* operation in Algorithm 3.

Algorithm 3 assumes that there is an efficient way to find if a filter has already been placed into the structure. This is possible using *syntactic equivalence* using hashtables. In syntactic equivalence, canonical representations of filters are compared. Syntactic equivalence is not necessarily implied by *semantic equivalence*, e.g., $F_1 \sqsupseteq F_2 \wedge F_2 \sqsupseteq F_1$. Semantic equivalence is computationally more complex to determine, whereas syntactic equivalence may be achieved in constant or near-constant time and it detects all semantically identical filters with simple filtering languages. We note that this restriction to syntactic equivalence does not break the data structure or the routing algorithms. Filters that fail the equivalence testing will be simply placed into the structure.

3.3.2 Poset-derived Forest with Multiple Interfaces

The basic poset-derived forest does not take into account the interface-processing present in the Siena filters poset. Each input node has one interface, but within the data structure a node may have several interfaces associated with it. The number of interfaces is bounded by the number n of nodes in the forest. Let $[n]$ denote the set of the n smallest natural numbers, i.e. $[n] = \{0, ..., n-1\}$.

Algorithm 3 *Add* and *del* procedures for the forest.

Let $(\mathcal{F}, \succ)$ be a poset-derived forest. It is assumed that there is an efficient way to find a node in $\mathcal{F}$ based on its identifier. In subsequent examination, references to "larger" and "smaller" are to be taken with respect to the relation $\sqsupseteq$. We define the following algorithms with inputs $\mathcal{F}$ and a filter x and output a poset-derived forest:

add($\mathcal{F}$, x): This algorithm maintains a *current node* during its execution. First, set the current node to be the imaginary root of $\mathcal{F}$.

1. If x is already in the forest, return without changes.
2. Else if x is incomparable with all children of the current node, add x as a new child of the current node.
3. Else if x is larger than some child of the current node, move all children of the current node that are smaller than x to be children of x and make x a new child of the current node.
4. Else pick a child of the current node that is larger than x, set the current node to this picked child and repeat this procedure from step 2.

del($\mathcal{F}$, x): Let C be the set of children of x and r be the parent of x. Then run add for each of the elements of C starting from step 2 and setting r as the current node. In this an element of C carries the whole subtree rooted at it with the addition. To preserve sibling-purity, any siblings of a relocated node that are smaller than the node must be relocated deeper into the tree using *add*.

Definition 3.10 *A triple* $(\mathcal{F}, \succ, G)$ *is a poset-derived forest with multiple interfaces, if*

1. $(\mathcal{F}, \succ)$ *is a poset-derived forest.*
2. G *is a function that associates a subset of* $[n]$ *with every filter, and* $G(x) \neq \emptyset$ *if and only if* $x \in \mathcal{F}$.
3. *If* $x \in \mathcal{F}$ *and an interface* $k \in G(x)$, *then* $k \notin G(y)$ *holds for all descendants of* x *in the relation* $\succ$.

To satisfy Property 3.2 we extend the *add* and *del* operations accordingly. A node is not inserted if a covering node with the same interface is already present. If a node is inserted, nodes that are covered by the new node and have the same interface are removed. A forest is either *redundant* or *non-redundant*. A redundant forest may contain a redundant filter, whereas a non-redundant forest may not contain such a filter. A redundant filter is such a (F_1, i) that there exists a (F_2, i) for which $F_2 \sqsupseteq F_1$.

The process of removing redundant filters is called *interface pruning* or *interface elimination* and it involves scanning the data structure for filters that are covered by the input filter and have been received from the same interface.

Operations

The *add* operation inserts a new filter x into the forest and if the interface c of the input filter is also new it is inserted into the set of interfaces. More formally, the *add* operation creates a new forest $(\mathcal{F}', \succ', G')$, where $\mathcal{F}' = \mathcal{F} \cup \{x\}$, $G'(x) = G(x) \cup \{c\}$, and $G'(y) = G(y), y \neq x$. Similarly, the *del* operation results in a new structure without the deleted filter.

The elimination of redundant filters may be implemented in *add*, *del*, or both. We distinguish two interesting cases. First, for filters from local clients no elimination is necessary. Second, for hierarchical and peer-to-peer routing the structure should be non-redundant.

Interface-based Balancing

Interface-based balancing is a technique for optimizing interface-specific operations in the forest. Each node maintains a set of interfaces used by its descendants. The index may be used to cluster filters from the same interface near each other. The *add* operation sorts the set representing the current level of the forest using the interface of the node to be inserted. Nodes that have descendants of the same interface are processed first. This

allows to quickly find filters from a specific interface in the forest. A forest that implements the interface index is called a balanced forest (BF).

We use an index presented in Definition 3.11 to optimize the performance of the data structure. For each filter in the structure the index keeps a record of the interfaces of its descendants. The index is updated for every addition and deletion. The add operation uses the index in deciding which subtree to traverse. An interface index entry is not necessarily needed for leaf nodes. The index requires at most $n - 1$ entries where n is the number of nodes in the forest and n bits per entry if each node has a unique interface. The total number of bits required by the index in the worst case is $n(n - 1)$. The index may be implemented using a bit vector of at most k bits for representing the interfaces, where k is the maximum number of interfaces in the forest.

Definition 3.11 *Interface-index(x): the input is node x and the output is the set of interfaces used by x's descendants.*

The maintenance of the index has both memory and processing overhead. During *add* and *del* operations the index is updated from the inserted node to the root so the depth of insertion is important. The index update is simple to implement for *add* and *del*. The index is updated for the input node and for all predecessors.

3.3.3 Non-redundant Forest

The non-redundant poset-derived forest (NRF) extends the BF with an *eliminate-interfaces-all* procedure during *add*. This procedure ensures that there are no redundant filters in the data structure. The naive implementation of this procedure simply tests each filter from the interface in question whether or not they are covered and should be eliminated.

This extension results in performance loss, but may be optimized by using the interface index in balanced forests. In this case, the *eliminate-interfaces-all* procedure checks only those parts of the forest that advertise the interface. In a descent into the subforest selected using the given interface, all the descendants of a node (and the node itself) covered by the input filter may be interface eliminated.

We note that it is not necessary to invoke the operation if it is known that the input filter is incomparable with existing filters from the interface associated with the filter. This is the case when, e.g., inserting the uncovered set after a *del* operation.

3.4 Discussion

The directed acyclic graph (dag) approach of the Siena filters poset gives a more complete model of the partial order of the base set $\mathcal{F}$ based on the covering relation than the forest. On the other hand, this completeness also complicates the addition and deletion algorithms and the internal representation of the data structure. The algorithms for *add* and *del* are simple and efficient for the forest.

We cannot say much about the running times of the *add* and *del* algorithms for the poset-derived forest without knowing about the structure of $\mathcal{F}$. The data structure does not perform well for totally unordered or ordered sets. In this case the *add* algorithm runs in worst-case time linear in the size of the input data set while *del* is constant-time. The retrieval of the non-covered set may be implemented in constant time by simply returning the data structure that holds the root nodes.

The covering set of filters for a given filter may be determined using the poset-derived forest data structure; however, since the structure uses only one-to-one relationships between filters, not all covering relations are captured. For a given filter the covering set may be found by traversing those trees that have covering nodes. The procedure is similar for covered nodes. If filters have only one-to-one relationships with other filters the covering set may be determined by simply traversing a single tree.

We also propose a representation for the forest data structure that seems to give good expected running times for the *add* and *del* algorithms, namely, to keep the children of any node in a linked list ordered by the size of the subtree rooted at each child from smallest to largest, and in the *add* algorithm always add the new node to the first, i.e. smallest tree encountered. This requires that any change in the size of a subtree is propagated towards the root node.

Since the addition and removal of filters is a frequent activity, concurrency control is an important factor in the performance of the data structure. Concurrent modifications may be controlled separately for different trees and also for subtrees. Deletion only affects the current level in the tree. Any balancing operations that are required may be performed to one tree (or one branch) at a time. In addition, since there may be several covering root nodes, if a heavy operation is performed on the first covering tree the filter may be inserted into the next possible tree. This kind of control is more difficult to implement with dags.

3.5 Equivalence of Forests and Posets

We consider the equivalence of the poset-derived forest and the filters poset. The analysis concentrates on the *forwards* sets of filters, because they determine the forwarding behaviour of the router.

The Siena filters poset supports the *unsubscribe* operation with an arbitrary filter. Our implementations differ and they only allow removing a filter that has been previously inserted. This modification does not change the routing semantics, because typically end systems subscribe and unsubscribe the same filters.

Hierarchical Routing The operation of the forest is similar to the poset for hierarchical routing environments. Here, the redundant subscriptions stored by PF and BF, in some cases, result in false positives: redundant subscriptions being forwarded to the master router, and false positives sent by the master to a slave. This depends on how the forest is used. We present two possibilities that prevent false positives:

PF and BF: The root set of a client is always installed at the master replacing the old set. The old set may be removed efficiently by simply walking through the forest and removing the given interface. This is the case for periodic updates.

NRF: Redundant filters are removed during *add*.

Peer-to-peer Routing We employ a simplified model here, in which we assume that the second *NST* term of Equation 3.1 contains only the interface that sent the filter in question. The former assumption corresponds to subscription semantics and the latter assumption corresponds to acyclic-graph-based routing.

The forest has two limitations when compared with the filters poset. First, it may have a number of redundant filters. This is solved by using NRF. Second, the *forwards* set management is more complicated for peer-to-peer environments, because the forest does not record the full relations between filters. This incompleteness may result in additional entries in the *forwards* sets of some filters and thus cause false negatives and positives.

Figure 3.5 gives an example of an unnecessary update. In this case, there are two possible places for inserting a subscription and one of them is more favourable. This may result in an additional element in the *forwards* set. Example 3.2 deals with the problem of finding the set of subscriptions that are uncovered by an unsubscription.

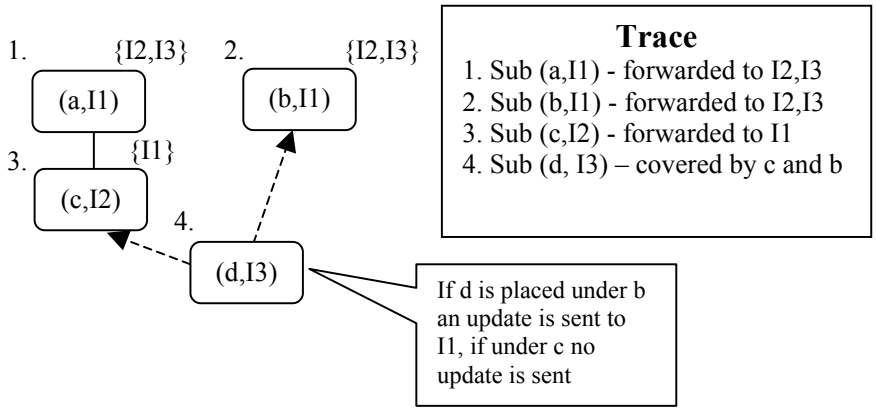

Figure 3.5: Unnecessary update using the forest.

The situation is the same when advertisements are used as discussed in Section 3.6, because advertisement forwarding is essentially the same and subscription forwarding uses the constrained $neighbours_s$ set. Since a root filter has the most overlap with any advertisements (Theorem 3.9) it will have the largest $neighbours_s$ set and hence the largest *forwards* set.

Example 3.2 *Let us consider the following scenario for filters F_1,F_2,F_3,and F_4 and the set of neighbours $\{a,b,c\}$. Superscript denotes the neighbour that sent the filter. F_1^c and F_2^c are root filters. F_1^c and F_2^c cover F_3^a and F_3^a covers F_4^b. When F_3^a is unsubscribed, the forwards set of F_4^b has a missing entry and it will be forwarded as a subscription to $\{c\}$. A forest, on the other hand, may be constructed with the following relations: F_1^c covers F_3^a and F_2^c covers F_4^b. When F_3^a is unsubscribed there are no covered elements and thus F_4^b is not forwarded. This means that the routing table of $\{c\}$ is missing an element.*

The following three rules simplify the determination of the *forwards* set:

- The set is $neighbours - NST$ for any root node if it has only one element in the *subscribers* set, which is contained in the *neighbours* set. The *forwards* set is equal to the *neighbours* set if the *subscribers* set has more elements or the only element is a local client.

- A candidate set is $neighbours - NST - forwards_{root}$ for any direct successor of a root node. The set contains at most one element. The

forwards set is empty if there exists a covering root node or a direct successor to a root node that has been sent to the neighbour already.

- Otherwise the set is empty.

To address these limitations, we extend NRF by using the following mechanisms:

- For each filter added using *add* as a direct successor of a root filter the *forwards* set is checked for a covering node. The covering node is a root node or a direct successor to a root node, if it exists.

- During *del*, the *forwards* set of the deleted node is computed using the same mechanism as for *add* (or retrieved from memory). If the *forwards* set is empty, no further action is needed. If it is not empty, the removed node was either a root node or a direct successor to a root node. In this case, the set of direct successors covered by the deleted filter needs to be found. This set is needed to send any uncovered subscriptions to the neighbour or neighbours. There are at most $|neighbours|$ entries in the *forwards* set of a root node and at most one in the *forwards* set of a successor to a root node. The set is found by iterating over the set of root nodes after *del* has been performed. A root node cannot be covered, unless it was a child of the deleted node. A node that is covered by the filter that was deleted, and is a direct successor of a root node, is added to the uncovered set, if the intersection of the forwards sets (the unsubscription and the covered node) is non-empty. The intersection determines where the uncovered filter is sent.

Computation of the *forwards* set is more difficult for the forest than for the poset, because the covering test has to be performed for any non-empty *forwards* set computed for a direct successor of a root node. On the other hand, the *forwards* set may be maintained in the data structure and thus the cover check needs to be performed only once. If the set is maintained and stored it needs to be updated when a covering node is deleted or the node is moved.

3.6 Advertisements

The basic subscription semantics may be optimized by using advertisements. In this model, advertisements are propagated to every node, and subscriptions are propagated only towards advertisers that have previously

advertised an overlapping filter. The idea is to use the additional advertisement information to prevent subscription flooding. The model uses two poset data structures, one for each type of message. Since the poset-derived forest can be made equivalent to the filters poset, it is also a useful data structure for advertisement semantics. Advertisements from local clients can be stored in a redundant forest. In addition, the filter merging framework presented in Chapter 8 may be used for both subscriptions and advertisements.

In advertisement semantics a second poset P_a is used for advertisements [29]. The sets *advertisers(a)* and *forwards(a)* are needed for each advertisement $a \in T_A$, where T_A is the set of all advertisements in the poset. Instead of forwarding subscriptions to a global set *neighbours*, a set constrained by advertisements is used as presented by the equation

$$neighbours_s = \bigcup_{a \in T_A : a \simeq s} advertisers(a) \cap neighbours. \tag{3.3}$$

In this case, Equation 3.1 uses the $neighbours_s$ set instead of the *neighbours* set. An advertisement may thus result in a number of subscriptions being forwarded to the sender of the advertisement. The process of unadvertisement is similar to unsubscription. Algorithms 4 and 5 give an outline of message processing with advertisement semantics. The algorithms are derived from [29] and [90].

3.7 Poset-based Matching

The two important use scenarios for event matching are:

- Event forwarding to neighbouring routers.
- Event forwarding to local clients.

These two scenarios differ, because generalized filter sets are used for neighbouring routers, but local filtering requires specific filters. Remote filter sets are envisaged to be considerably smaller than local filter sets. Moreover, for neighbouring brokers the filters are typically stored in a poset whereas local filtering may require optimized filtering structures. Optimized matching structures may also be used for filter sets from neighbouring brokers.

The counter algorithm is the basic mechanism for efficiently matching events [28, 34, 88, 90, 104]. The counter algorithm keeps track of the number of attribute filter matches for each filter. Counting is based on the

Algorithm 4 Subscription message handlers for advertisement semantics.

IncomingSub(f,*source*)

1. Add (f,*source*) to P_s.
2. Calculate $neighbours_s$ using P_a and Equation 3.3.
3. Send subscription message to *forwards(f)*.

IncomingUnsub(f,*source*)

1. Remove (f,*source*) from P_s.
2. Forward unsubscription following the procedure in Algorithm 2. The set may be empty if there are subscriptions from other neighbours that cover f. The *forwards* sets of subscriptions covered by f may change, which may require the forwarding of new subscriptions. An uncovered subscription is such that its forwards set gains an additional element due to the removal of a covering filter.

fact that filters are conjunctions of attribute filters. Typically, the counting algorithm is divided into a preliminary elimination phase in which unmatchable filters and interfaces are removed, and a counting phase. If the counter of a filter becomes equal to the number of attribute filters in the filter, the filter matches the input notification and the corresponding interface is added to a set of output interfaces. The counter algorithm returns either the identifiers of matching filters or a set of output interfaces. Optimized matchers use efficient data structures for different predicates, for example hashtable lookup for equality tests and interval trees [40] for range queries.

The data structure and posets in general have two interesting properties for matching that follow from the definition of the covering relation (Property 3.12 and 3.13).

Property 3.12 *If a node n_1 matches a notification then all the predecessors of n_1 must also match the notification.*

Property 3.13 *If a node n_1 does not match a notification then none of the descendants of n_1 matches the notification.*

The node in this case may be any object that is comparable using the covering relation, for example: filters, attribute filters, and disjuncts.

Algorithm 5 Advertisement message handlers.

IncomingAdv(a,*source*)

1. Add (a,*source*) to P_a.
2. Forward advertisement message to *forwards(a)*.
3. Determine the set of overlapping subscriptions using P_s for which a is the only advertisement from the *source* that overlaps and send them to the *source*. In other words, any subscriptions that have not yet been sent are forwarded to the advertising node (*source*). Those subscriptions that overlap with an existing advertisement from the *source* have already been forwarded so they are not processed. The overlapping set is found by iterating over the first two levels of P_s and testing the overlap of subscriptions with the advertisement.

IncomingUnadv(a, *source*)

1. Remove (a,*source*) from P_a.
2. Forward unadvertisement in a similar fashion than the unsubscription is forwarded. The *forwards(a)* set may be empty if there are advertisements that cover a from other neighbours. Forward any uncovered advertisements in P_a.
3. Remove any subscriptions for *source* that are no longer needed. All subscriptions are removed from neighbours other than the *source* that do not have an associated overlapping advertisement from some other neighbour.

The Siena poset-based matcher uses Property 3.13 in order to optimize matching. The pseudocode for the forest is given by Algorithm 6. The poset-based matcher is similar, but requires the testing of nodes that have already been visited.

The forest or poset also supports approximate matching. For example, we may walk the forest with the notification breadth-first and define a time bound for matching. When this time expires the algorithm simply walks the remaining nodes and records the interfaces as matched. This is approximate, because it may result in false positives.

Algorithm 6 Pseudocode for forest-based matching.

```
MATCH-FOREST(n)
 1  let S be an empty sequence
 2  let FW be an initially empty set of forward interfaces
 3  let I_max be the # of interfaces for the event type
 4  let q = FALSE
 5
 6  R = GET-ROOTS(n.type)
 7  let Im be an imaginary root of a tree
 8  Im.children = R
 9  ADDLAST(S, Im)
10  while S is non-empty and not q
11        do
12           o = REMOVEFIRST(S)
13           while o has unprocessed children and not q
14                 do
15                    c = NEXTCHILD(o)
16                    if subscribers(c) ⊆ FW
17                       then ADDLAST(S, c)
18                    elseif MATCH(c, n)
19                       then
20                             ADDLAST(S, c)
21                             ADDTOSET(FW,subscribers(c))
22                             if |FW| ≥ I_max
23                                then q=TRUE
24  return FW
```

The interesting feature of the algorithm is that the matching mechanism does not know the details of the filtering language — it only assumes that there are covering relations between nodes. This makes the algorithm suitable for environments where the filtering language and the operators (predicates) are dynamic and change. In addition, adding new operators does not require complicated changes to the matching algorithm, such as creating new indexing structures.

We propose two improvements to the basic algorithm. First, the matching test does not need to be performed for a filter whose interfaces have already been added to the result set. We found that this modification resulted in better performance. The second improvement is to use the interface index to prevent the processing of those subtrees in a balanced forest, which have been already matched. This is easily accomplished by simply checking whether the interfaces of a particular node are already contained in the result set; if they are the node is not processed further.

3.8 Rate-control Using Posets

The matching technique described in the previous section may be combined with rate-control policies to improve the scalability of content-based systems. Rate-control rules define how many notifications per second or time unit should be forwarded to the subscribing interface. Some rate-control rules are set by system designers and policies, and some rate-control rules are set by applications.

Rate control rules are represented using attribute filters and thus parts of filters [93]. The rate control rules support the covering relation and are also mergeable. The covering relation is a simple inequality, where a bigger rate-value covers smaller rate-values. For example: (20 notifs/s) $\sqsupseteq$ (10 notifs/s).

Therefore, the rate-control extension may be used with the forest or poset-based matchers with some modifications. For each filter in the data structure we keep track of the notification rate per time interval. If the rate limit value has been exceeded, it is not necessary to check the filters rooted at that node, because they have also exceeded their limits. This provides a convenient way to prevent unnecessary messaging between brokers. To balance the forest properly during insertion, a covering subtree is selected with the closest rate-control filter.

Rate measurement is important for load balancing decisions. The time-window to monitor is important, and this information is typically required at least for the root set of the poset or forest and in some cases for each

subscription. Monitoring of the rate for the root-set requires that the covering subscriptions of the root set are updated (their counters are increased) during matching. When the monitored time-period restarts, the counters are reset and the old values may be stored into a history. The current rate value may not be very interesting for a load balancer, but rather knowledge of the recent behaviour of the rate is important and may be used to extrapolate future behaviour, for example, using moving averages or other statistical methods.

Chapter 4

Experimentation

This chapter presents the experimental results with the filter mechanisms presented in this thesis. First, we present the workload generator that was used for the experimentation. After this we present the benchmarks for forests and posets and then examine data-structure performance. We present performance results for the following data structures: filters poset (FP), poset-derived forest (PF), and non-redundant forest (NRF).

4.1 Workload Generator and the Environment

A custom workload generator was used for the experimental results. The generator creates sets of filters and notifications using the given parameters. The key parameters of the workload generator are the number of filters, the number of attribute filters in the filters, the number of schemas, the number of unique names that are used in generating attribute filter constraints, the range of values for number tests, the number of notifications to match, and the number of interfaces. Interfaces are assigned to filters in a round-robin fashion.

The filters were generated using the structure enforced by a schema. Each attribute filter has a type and a name. Each attribute name is associated with a single type. In the experiments, each attribute filter corresponds to a range query in the interval $[0, 100]$. We have also experimented with other atomic predicates, such as the set $\{<, >, \leq, \geq, =, \neq, [a, b]\}$. We present results for the range predicate, because range queries are used by many applications, for example, spatial applications.

We experimented with the following filter schemas: a 2-dimensional range, a 3-dimensional range, and a variable number (1-3) of range attribute filters.

Table 4.1: Data structures used in the experimentation.

Name	Description
Poset-derived Forest (PF)	The basic forest.
Balanced Forest (BF)	PF with 1-bit balancing.
Non-redundant Forest (NRF)	The non-redundant BF.
Poset	The filters poset.

We used the following equipment: an HP laptop with a 2 GHz Pentium III processor with 512 MB of main memory, Windows XP, and Java JDK 1.4.2. Table 4.1 presents the different data structures used in experimentation. The filters poset algorithm is based on the Siena Java implementation [140] and extended with hashtable-based duplicate filter detection.

4.2 Benchmarks

The two important benchmarks were the *add scenario* and *add/remove scenario*. The former consists of the creation of the forest or poset data structure with the given set of input filters. The latter consists of repeated insertions and deletions to the data structure. In the add/remove scenario a filter is removed and a new random filter with the same interface identifier is created and added to the data structure. The state after the add scenario is the initial state for the add/remove scenario.

We distinguish between *local* and *distributed* operation. Interface elimination and computation of the *forwards* sets are not needed for the local case. Distributed operation consists of the two routing models, hierarchical and peer-to-peer. The latter requires the computation of the *forwards* set whereas the former does not.

Figure 4.1 presents an overview of the add and add/remove scenarios. The workload generator is used to generate filters and notifications. First, filters are inserted or inserted and removed from the data structure. We measure the time spent in the operations and the total number of covering operations. After the first phase we match notifications using the data structure. Correctness is checked after each phase and between add and add/remove scenarios.

The two important cases in experimentation were a variable number of filters with unique interfaces and a variable number of interfaces with a static number of filters. In each experiment, we used a single schema

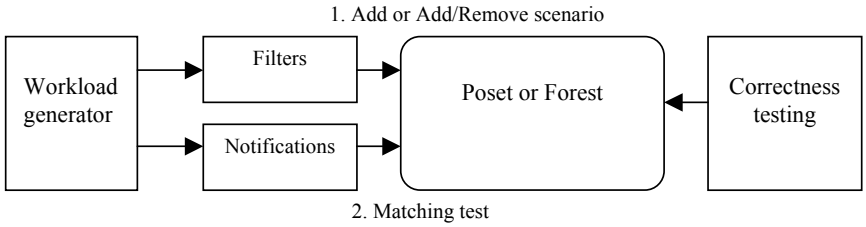

Figure 4.1: Add and add/remove scenario tests.

to generate the filters. The single schema situation is the most difficult scenario for matching and covering, because two schemas are by definition independent and a notification may match only one schema (event type) at a time, but filters of the same schema have to be analyzed.

Correctness of operation was tested using assertions on root set size invariance using Theorem 3.7, data structure size invariance, and existence of false positives and negatives in matching. We recorded the set of matching interfaces for each notification using a naive algorithm and used this set to determine any false positives or negatives for the other algorithms. Existence of nodes not connected with the root set, existence of cycles, and the minimum interface property (Property 3.2) were also tested. For the balanced forest also the proper functioning of the index was tested. We also tested the correctness of the filters poset's direct successor and predecessor sets. The time spent in testing the data structures is not included in the benchmark results.

Each experimentation run was replicated 10 times. The original input set and the set of filters to be added after removals, and the set of notifications to test the structures were created for each replication. The figures present results using the arithmetic mean over the replications.

4.3 Local Clients

To experiment with the impact of local clients, we created 500 filters using the schema and performed the add and add/remove scenarios. Since we are dealing with local clients, each filter has its own interface and interface pruning does not take place.

Figure 4.2 presents the results for 2D range filters and Figure 4.3 the results for 3D range filters. We measured both the number of *covering*

operations required by the add and add/remove scenarios and *processing time*.

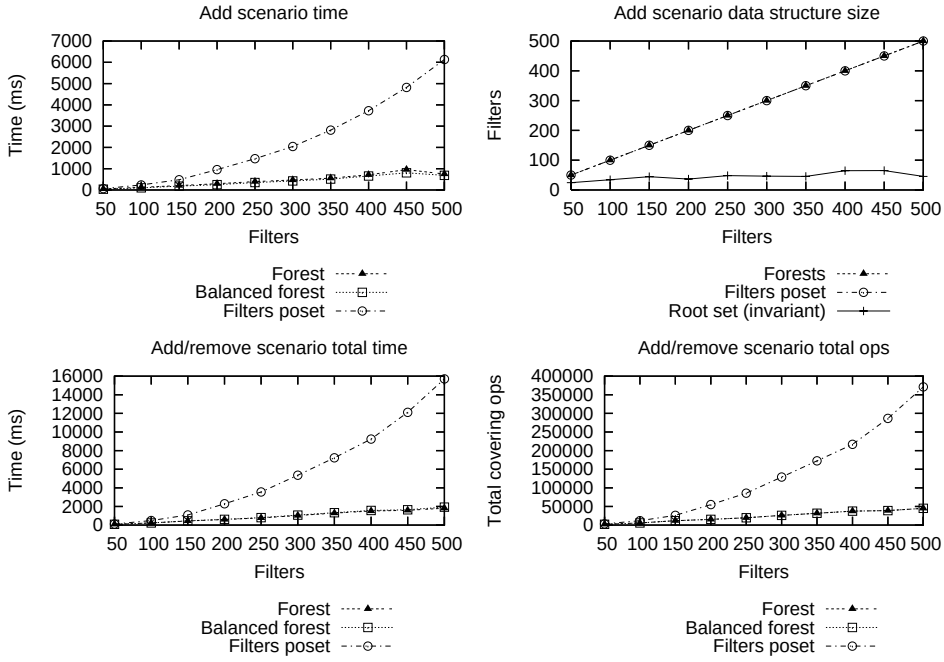

Figure 4.2: Results for a variable number of 2D range filters.

The filters poset performs considerably slower than the forest structures in both add and add/remove scenarios. The data structures have the same size, which is shown in the figures for the add scenario. With 3D filters the structures require considerably more processing time. Similar results to the 2D case were also observed with the variable number of attribute filters case, but with reduced processing overhead.

We observe that the balanced forest and non-redundant forest do not have overhead compared to the plain forest data structure in these benchmarks. This is explained by how the interface-based balancing is realized. For local clients, the balancing is very useful and allows to locate any other filters from the same interface rapidly, or to detect if there are none.

Figure 4.4 gives the results for the variable interfaces scenario with a static number of filters (500). The x-axis shows the number of interfaces. Here the number of operations and times required by the forests are approximately constant. This is due to the fact that the number of filters is static. The filters poset is considerably slower in this scenario as well, be-

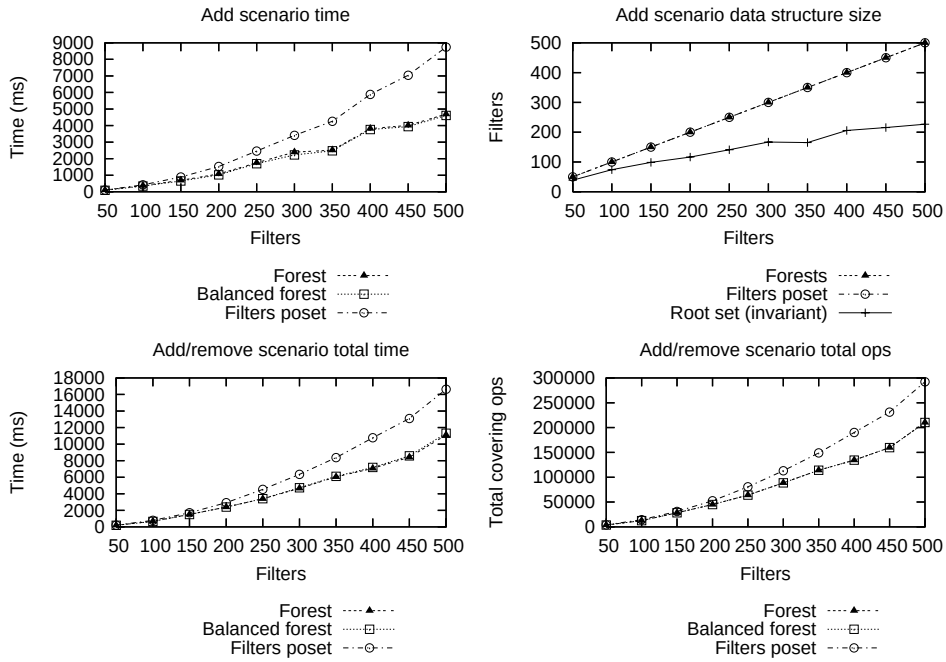

Figure 4.3: Results for a variable number of 3D range filters.

cause the size of the structure grows when the number of interfaces grows. The filters poset is able to remove all filters with covering filters that have the same interface.

4.4 Distributed Operation

In distributed operation, a router receives new filters or requests to remove existing filters from neighbouring routers. Typically, there are a few neighbours and many local clients. These may be analyzed separately. We used 1000 filters and 3–15 neighbouring routers. Since there are many filters per interface, interface-based filter pruning is needed to remove unnecessary filters.

In order to achieve more realistic distributed filter updates, we prevent the addition of a filter that is covered by an existing filter from the same interface. This is motivated by the update semantics for distributed operation, in which a neighbour only sends a covering or incomparable filter, or deletes an existing filter and sends the uncovered filter set. The uncovered filter set sent after *del* is incomparable with existing filters sent by

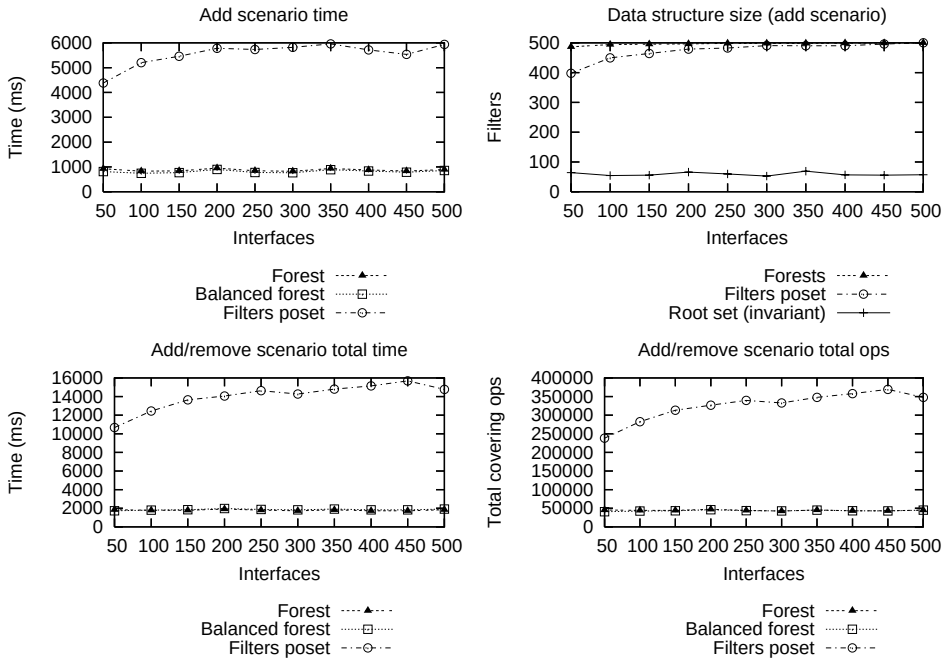

Figure 4.4: 2D filters with a variable number of interfaces.

the neighbour. The workload generator functions as before and generates a number of filters (1000), but we use an intermediate layer that drops any additions that violate these semantics. As before, the data structures process exactly the same set of filters.

Interface-based filter elimination or pruning is needed for distributed operation to remove unnecessary filters. Pruning can be implemented in at least two places in a non-redundant forest. If implemented in the *add* operation, the forest has the same size as the filters poset, and redundant filters are removed prior the computation of the forwards set. If the pruning is implemented in the *del* operation, the *add* operation is very simple, but the size of the data structure may grow larger than the filters poset. The *add* case is more useful for peer-to-peer routing, whereas the *del* case is expected to be more efficient for hierarchical routing. We use a simple interface-based elimination in conjunction with the *add* operation. The procedure scans those parts of the forest that have the input filter's interface and eliminates them if they are covered by the filter.

Figure 4.5 presents the results for a variable number of interfaces (3–15) for a static number of 2D range filters. Figure 4.6 presents the same

benchmark for 3D range filters. As a summary for the results using different schemas, the non-redundant forest has better performance than the filters poset for both the add and add/remove scenarios. On the other hand, the interface-based filter elimination is more difficult for the forest, because it cannot use the information provided by the direct successor sets of the filters poset. The overall performance, however, is better due to the simple and efficient nature of the *add* and *del* operations. We note that filter elimination may also be implemented in conjunction with the *del* operation, which may be used to further simplify and improve the performance of the *add* operation. Results for the variable number of range attribute filters workload was similar to the 2D case, and updates were faster, because some of the filters were simpler.

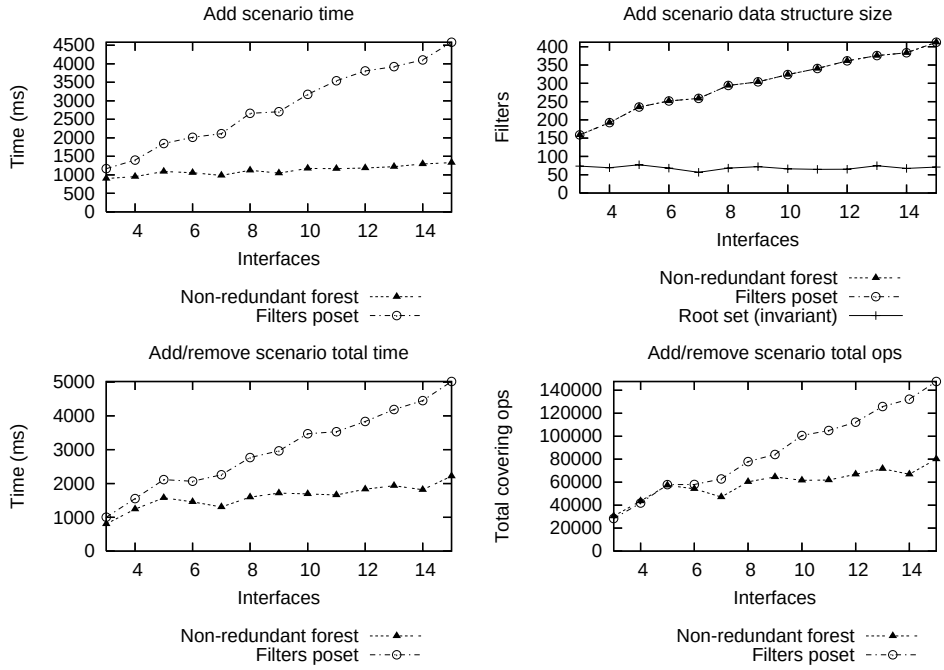

Figure 4.5: Hierarchical operation with 2D range filters for a variable number of interfaces.

4.5 Forwards Sets

The computation of the *forwards* set for the non-redundant forest is more complex than for the poset, because the complete successor sets are not

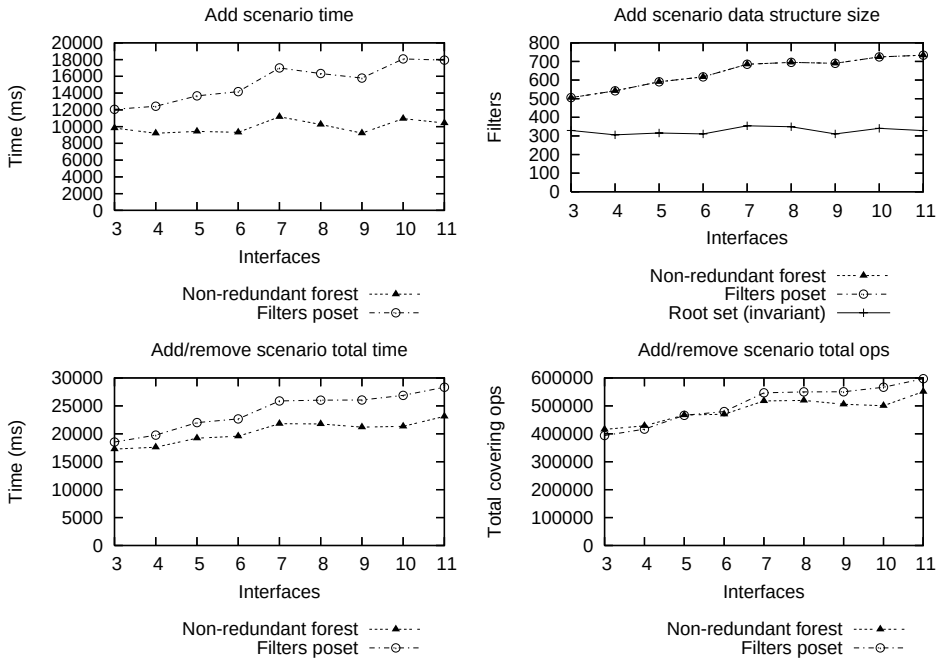

Figure 4.6: Hierarchical operation with 3D range filters for a variable number of interfaces.

maintained. Especially, the determination of the uncovered set in unsubscription is heavy. The performance of the *forwards* set determination may be improved by simply computing the *forwards* set based on the relations stored in the forest. This means that in some cases there may be false positives, extra messages, that the poset would not send. False negatives do not occur due to this behaviour, because updates are idempotent and the minimum-interface property is maintained.

We have implemented the *forwards* set computation for the forest and experimented with two cases. First, we have the complete *forwards* set computation. Second, we have the incomplete *forwards* set computation for the *add* operation and full uncovered set computation for *del*. In the complete forest-based *forwards* set computation, the first two levels of the forest are scanned for any covering nodes when a second level node is inserted. The uncovered set in *del* is located similarly when a root node or a second-level node is removed by scanning the first two levels for any nodes that are covered by the removed node. The scanning is optimized by using the interface-index. In the approximative case, there may be a number

false positives for added filters or filters in the uncovered set. The computation of the set for both FP and the non-redundant forest was tested using a naive algorithm for computing the set. The naive algorithm builds the successor and predecessor sets at run time.

Figure 4.7 presents the results for peer-to-peer operation with a static number of 2D range filters and no local clients. The x-axis shows the number of interfaces. All the interfaces are neighbouring routers in this case. The results are shown for the filters poset and the non-redundant forest. The forest was experimented with two different modes: exact *forwards* sets and approximative *forwards* sets. The former mode corresponds to the filters poset and computes the same *forwards* set as the poset. The latter computes an approximative set that may result in false positives. A false positive in this case means a new filter or an uncovered filter that was already sent to a neighbour.

Figure 4.8 presents the results for the *forwards* set computation. We count the total number of forwarded filters in order to understand the performance of the two modes. The figure also shows the total number of added filters.

The forest has better performance for the *add* operation, whereas the *del* is faster for the poset. The approximative *forwards* set is faster than the exact set in both add and add/remove scenarios; however, the filters poset is faster for *del* when the number of interfaces is small.

The results with one interface representing local clients are similar to the previous figure, but fewer filters are forwarded to neighbouring routers. This happens because filters from local clients are forwarded to all neighbours, which simplifies the computation of the *forwards* set. We also experimented with 3D range filters and a variable number of range attribute filters. The results were similar to the 2D scenario with 3D filters requiring more processing time and the variable scenario less.

Figure 4.9 presents the results with 2D range filters with 4 neighbouring routers and a variable number of local clients. This scenario represents a combined benchmark of the previous local client and distributed benchmarks. The semantics for the local clients follows the Siena model, and clients behave in a similar fashion to routers. The forests are faster for both *add* and *del* when the number of local clients grows. This was expected from the results with local clients.

The poset performance degrades when the number of local clients grows. Local clients have two implications. First, the number of filters in the data structure grows because interface-based filter pruning cannot be performed for filters from different clients. Second, it may become more likely that

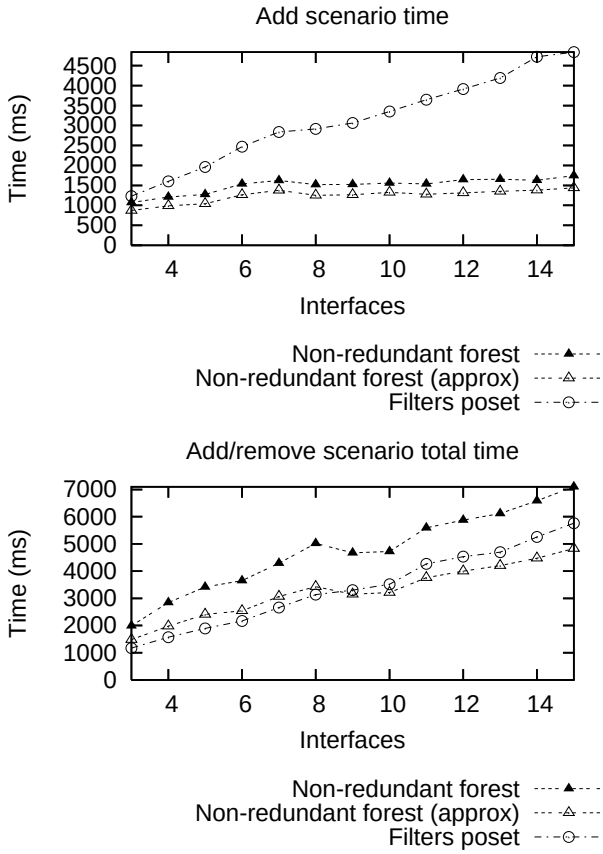

Figure 4.7: Peer-to-peer operation with 2D range filters for a variable number of interfaces.

the *forwards* sets of root filters are equal to the *neighbours* set. In our experimentation, the forest gives considerably better performance when the number of local clients grows. Figure 4.10 presents the results for the *forwards* set computation. The approximative *forwards* set does not give much additional performance when there are many local clients.

4.6 PosetBrowser

We have developed a graphical tool for experimenting with various content-based routing data structures. The Java-based tool is called *PosetBrowser*[1]

[1]Available at `http://www.hiit.fi/fuego/fc/demos`

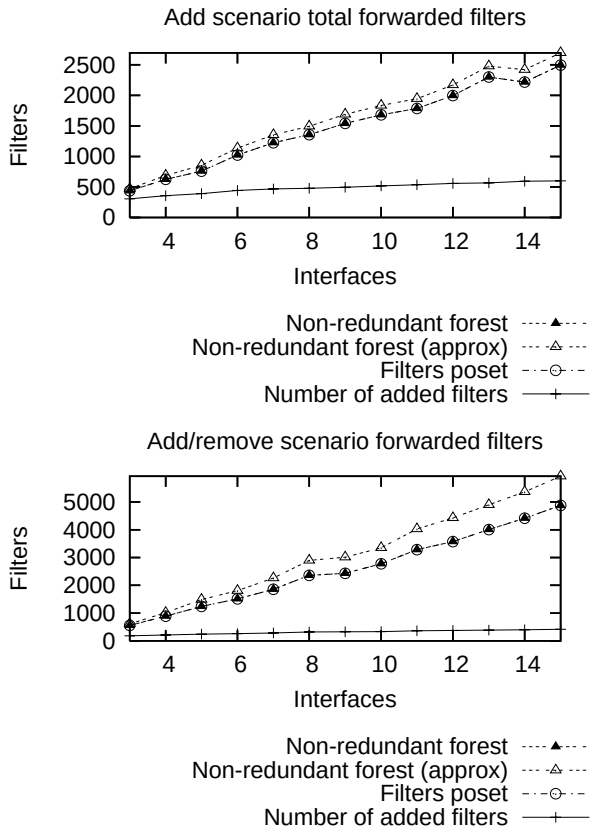

Figure 4.8: Forwards set results for peer-to-peer with 2D range filters with a variable number of interfaces.

and uses the JGraph toolkit [6] for graph visualization. The PosetBrowser tool supports the following data structures: FP, PF, BF, and NRF.

Figure 4.11 shows the different data structures supported by the software. PosetBrowser uses a simple custom layout algorithm for drawing the nodes. The supported operations are viewing and graphically comparing data structures, changing the filter set size at run-time, data-structure root-set-merging using perfect and imperfect merging techniques, and *forwards* set computation. The *forwards* set computation is done using three different mechanisms: naive, poset, and forest.

The tool contains tests for ensuring the correctness of the algorithms. The tests include addition and deletion of filters. Figure 4.12 shows the filter generation features of the software.

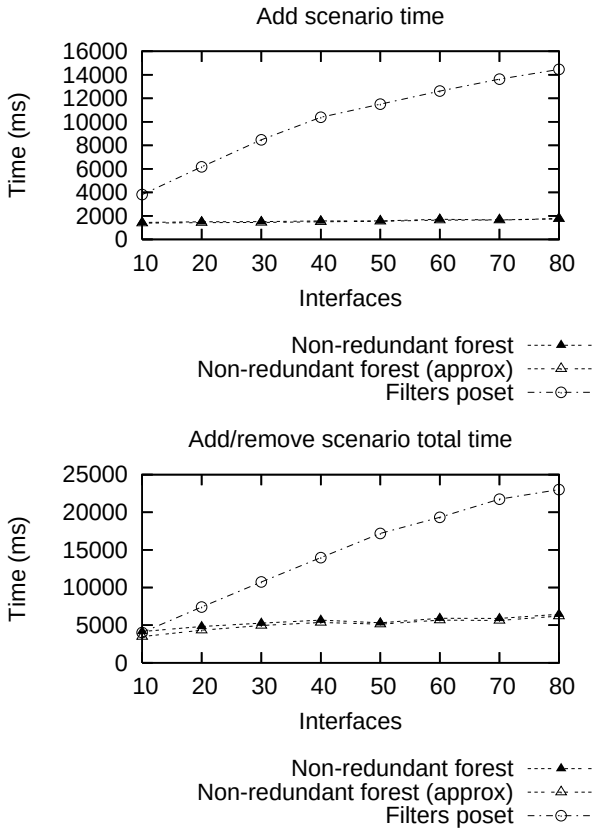

Figure 4.9: 4 neighbours and a variable number of local clients with 2D range filters.

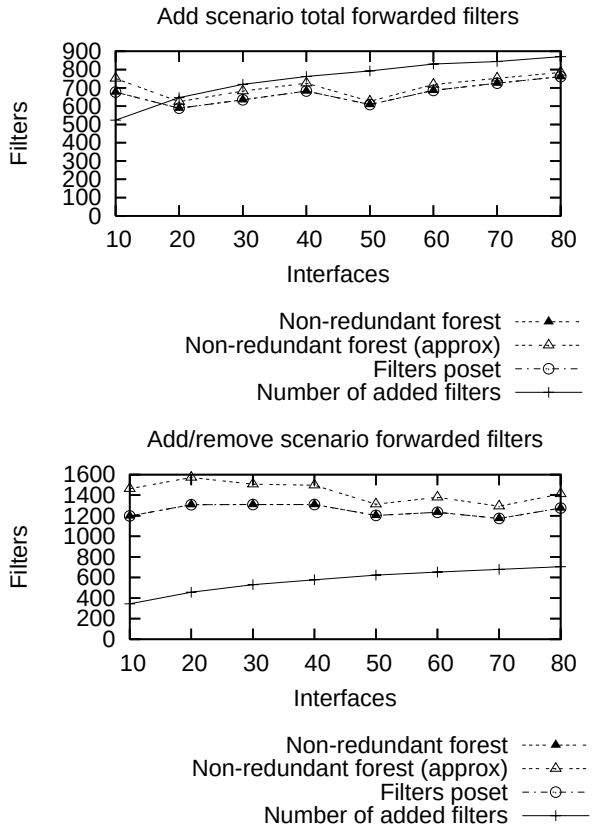

Figure 4.10: Forwards set results for 4 neighbours and a variable number of local clients with 2D range filters.

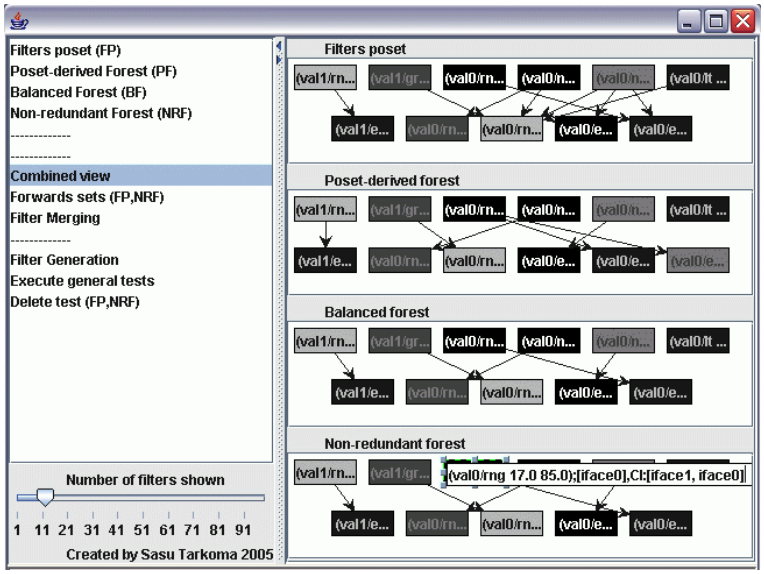

Figure 4.11: PosetBrowser with four different data structures.

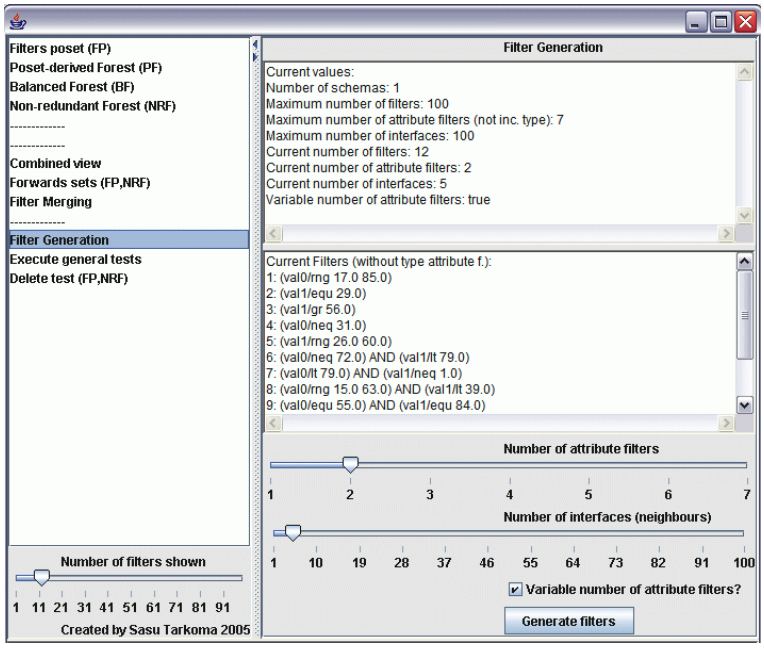

Figure 4.12: Filter generation in the PosetBrowser.

4.7 Discussion

The poset-derived forests are more efficient than the filters poset for maintaining local clients. The non-redundant variants are also useful in hierarchical routing. The situation is more complicated for peer-to-peer environments, in which the *del* operation has overhead for the forests.

The experimental results suggest three engineering guidelines for content-based routing systems based on filter covering:

Hierarchical Routing: The non-redundant balanced forest data structure should be used for hierarchical routing and for finding the non-covered set.

Peer-to-peer Routing: For peer-to-peer routing, the forest structures should be used when there are many local clients. The forest requires more complicated *forwards* set management. Alternatively, the forest should be used to find the non-covered set of the clients and then the filters poset should be used to manage external routing information only.

Matching: The data structures have similar matching performance, but they are outperformed by more optimized matchers. Matching performance is proportional to the number of filters and also to the number of interfaces. The best matching performance was achieved when the number of interfaces was small. The forest should be used to find the non-covered set that is propagated and an additional matcher data structure should be used to quickly match notifications to the local clients. This is a two-phase process: first notifications are matched for the covering set by the poset or forest, and then they are matched by the matching data structure.

4.8 Routing Configurations

The hierarchical routing proposed in the Siena system uses the FP for the computation of the minimal cover set. In an acyclic topology, the poset also stores the *forwards* set and thus computes a per-neighbour minimal cover set.

In Siena, the subscriptions of a local client are not handled independently of each other. Any filter from a local client that is covered by a new filter from the same client will be removed. Similarly, when a client removes a filter (unsubscribes), any filters that are covered by the removed filter will be removed. This approach requires that clients are able to compute the

covering relations between their filters and explicitly manage their filter sets. In addition, in this model it is not possible to transparently change the routing algorithm semantics without making changes to the client code.

In our proposed model, a separate data structure, the poset-derived forest, is used to manage filters from local clients. In this case, client-side filter set management is simple and efficient. The second benefit is increased performance, because the forest supports faster insertions and deletions than the poset. This model also supports extending the system to support filter merging (also called aggregation and summarization), for example, by starting with the root set of the forest storing the local clients.

A BDD-based routing and matching mechanism was presented in [79]. They use a global predicate index and *Modified Binary Decision Diagrams* (MBDs), which are abstract representations of boolean functions. A MBD is used to represent a subscription. They assume typed tuples and the variables are based on a predefined order.

In a MBD-based routing table, publication matching involves iterating the name/value pairs of the event and computing the truth values for the corresponding attribute filters. The attribute filters are located using the predicate index. After this, the MBDs are evaluated using the computed truth values. The cover algorithm involves iterating the MBDs for elements that cover or are covered by the input subscription. If there are no covering elements, the new subscription is a root element. The algorithm stops when a covering element is found. The mechanism may be used to find the root set in a single scanning of the routing table, but the second-level filters are more difficult to locate, and they are not discussed. The MBD-based approach does not leverage the covering relation in its operations, such as matching. Support for frequent updates, deletion and modification, of the routing table was not analyzed in this work.

Our work emphasizes generic operation and compositionality of the routing table by presenting routing table building blocks, such as the forests and the merging mechanism presented later in the thesis, and dividing the table into different parts. The properties and theorems presented in this chapter support the design and optimization of efficient routing table algorithms. We note that techniques such as MBDs and attribute counters may be used to exclude a subset of filters from inspection before a forest or a poset is used for routing. Figure 4.13 presents three useful routing table configurations that combine the FP, PF, and NRF. We briefly describe each configuration in this section and then present the structures in more detail.

The main insight is to separate the routing table into two parts: the external table and the local table. Figure 4.13a presents an example of this

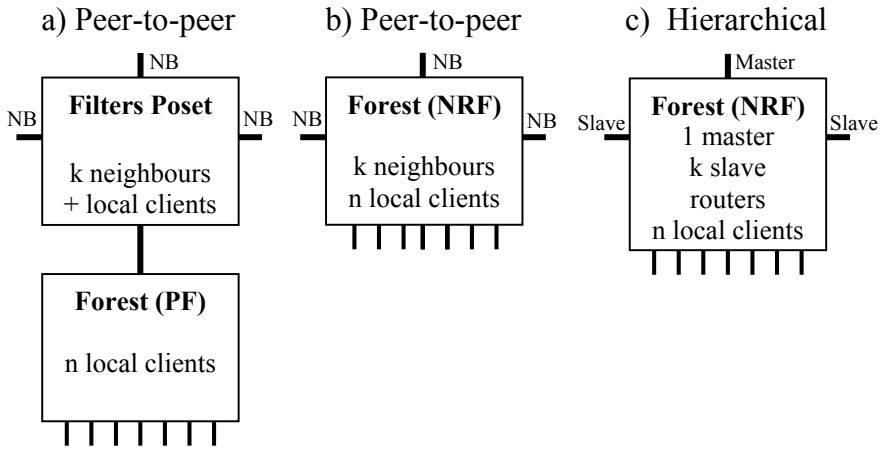

Figure 4.13: Routing table configurations.

by implementing the external table using the FP and the local table using the PF. The term NB in the figure denotes a neighbour interface. The PF is used to maintain client subscriptions (and advertisements), and the FP is updated only when the root set of the PF changes. The Siena system used the FP to also store local filters, which is not efficient based on the experimentation presented in this thesis. Assuming that there exist covering relations between local filters, this separation ensures that the external table is not burdened with frequent updates by local clients. Figure 4.13b illustrates use of the NRF as both local and external routing tables. This configuration is feasible when there are many local clients, but the external forwarding is more complicated than for the FP. Figure 4.13c illustrates how the NRF is used for hierarchical routing with the master and slave interfaces identified. The NRF may also store local clients, but the separation into two parts allows to prioritize operations.

Figure 4.14 illustrates how a more efficient matching data structure may be introduced into the routing core. In this case, any addition (*add*) and deletion (*del*) operations by local clients are processed by the PF and also reflected to the efficient matcher. Only the root set is updated to the FP, the external routing structure. When an incoming event matches the local interface (root filters of the PF), the notification is sent to the efficient matcher.

In addition to hierarchical and peer-to-peer routing, the new data structures may be used to enhance rendezvous-based routing models, such as

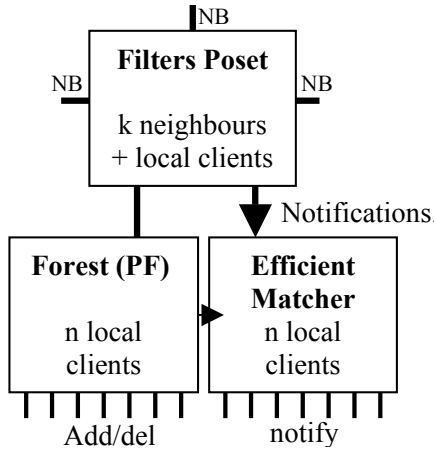

Figure 4.14: Adding support for efficient matchers.

Hermes. In the Hermes model with filters, advertisements are always propagated towards the RP. Subscriptions are propagated towards the RP and towards any overlapping advertisements. Therefore, advertisements may be stored using a non-redundant forest and subscriptions using a non-redundant forest or a poset. In both cases local clients are stored using a forest. For rendezvous-based models, the subscription poset must be extended to support any subscriptions that should be forwarded towards the RP. This is accomplished by using a virtual advertisement from the RP that covers all subscriptions of the designated type.

The main application of the data structure is content-based routing using filter covering. We envisage that the poset-derived forest and its properties may be useful outside routing, because partial orders are also found in, e.g., knowledge-based systems.

Part III

Mobility-aware Routing

Chapter 5

Mobility and Completeness

In this chapter, we formally examine several mobility protocols for different pub/sub topologies. The new results are the cost functions for both subscriber and publisher mobility, establishing mobility-safety of the protocols, and investigation and formulation of completeness of subscriptions and advertisements. The results show that handovers in incomplete topologies are more costly than in complete and the routers involved with mobility have no way of detecting completeness based on local information alone. We conclude this chapter with engineering guidelines for mobility-aware content-based routing.

5.1 Overview

Most research on event systems has focused on event dissemination in the fixed network, where clients are usually stationary and have reliable low-latency high-bandwidth communication links. Recently, mobility and wireless communication have become an active topic in many research projects working with event systems.

The main motivation for a pub/sub mobility protocol is the avoidance of triangle routing with a designated home broker, which may be inefficient. Indeed, experimental results show that home-broker-based approaches do not perform well [17]. Mobility protocols are also needed for load-balancing subscribers and advertisers between brokers, and are thus a necessary functionality that needs to be provided by a scalable event framework.

Content-based routing using subscription and advertisement semantics becomes challenging when the topology needs to be reconfigured with the introduction of mobile components. In advertisement-based pub/sub networks a successful activation of a subscription may require that an adver-

tisement is first propagated through the network, and then a connecting subscription is propagated on the reverse path. In this chapter, we focus on advertisement semantics, because it is more complicated than subscription semantics and it may be used to emulate subscription semantics using trivial advertisements.

We investigate three different pub/sub mobility mechanisms and topologies: generic pub/sub mobility support, acyclic graphs, and rendezvous-based topologies. We show that the mobility protocol for acyclic graphs and rendezvous models with advertisements is mobility-safe when the completeness of the subscriptions and advertisements of the mobile client is assumed. We also discuss techniques for supporting incomplete topologies, in which the subscriptions and advertisements of the mobile client are still being propagated. We present the lower and upper bound messaging costs of subscriber and publisher mobility. The lower-bound cost is discussed in the form of the *covering optimization*, which may be applied when subscriptions are complete in the topology.

In order to understand event-routing we need to have useful metrics to characterize the system. Besides message complexity and computing power, the two most important metrics are the number of false positives and negatives. *False positives* are events that are delivered but were not subscribed, and similarly *false negatives* are events that were subscribed but were not delivered upon publication. Clearly, the presence of false negatives indicates a serious error in any event system. Therefore, we are interested in proving that a candidate event system does not manifest this erroneous behaviour.

Intuitively, given that we first establish a new flow and only after the successful completion of this tear down the old one, there should not be any false negatives, which would satisfy the requirement for mobility-safety. This is our basic hypothesis for the mobility protocols. A perfect topology update protocol may be described using flooding that delivers all events to all brokers. This naive protocol ensures that also mobile components will receive all events that match their filters albeit with a high cost in false positives. A good mobility protocol is mobility-safe, minimizes the number of false positives, and minimizes the signalling cost.

5.2 Formal Specification

5.2.1 Valid Routing Configuration

The valid routing configuration determines that the publish/subscribe system does not manifest illegal traces. A trace is a sequence of operations,

such as subscribe, notify, and unsubscribe. Any valid routing configuration must satisfy the following constraints on traces presented using the operators of the Linear Temporal Logic (LTL). LTL formulas are used to define a specification and a system is correct when it exhibits only traces allowed by the specification. $\Box$ denotes "always", $\Diamond$ "eventually", and $\bigcirc$ "next".

Property 5.1 gives the liveness constraint for the basic publish/subscribe system with subscription semantics. The liveness property defines when a notification should be delivered and ensures that they are eventually delivered. Property 5.2 gives the safety constraint, which ensures that incorrect events are not processed and delivered. The properties are from the definitions in [90] with minor changes in presentation.

Property 5.1 *Liveness:*

$$\Box[Sub(A,F) \Rightarrow [\Diamond\Box(Pub(B,n) \wedge n \in N(F) \Rightarrow \Diamond Notify(A,n))] \vee [\Diamond Unsub(A,F)]],$$

specifies that a subscription with filter F and the publication of an event n that matches the subscription will lead to an eventual notification of subsequent publications of that event unless the subscription is invalidated by unsubscription.

Property 5.2 *Safety:*

$$\Box[Notify(A,n) \Rightarrow [\bigcirc\Box\neg Notify(A,n)\ ] \wedge [n \in Published\ ] \wedge [\exists F \in Subs(A):\ n \in N(F)]],$$

specifies that a notification is delivered only once, it has been published previously, and that the recipient has a matching subscription. Published is the set of published events, and the set Subs gives the subscriptions for each client.

Since it may be difficult to maintain these properties in dynamic pub/sub systems they may be relaxed. A self-stabilizing pub/sub system ensures correctness of the routing algorithm against the specification and convergence [90]. The safety property may be modified to take self-stabilization into account by requiring eventual safety.

5.2.2 Weakly Valid Routing Configuration

The weakly valid routing configuration guarantees only the delivery of notifications to those subscriptions whose update process has terminated. A routing algorithm that uses the weakly valid routing configuration and ensures that every update process terminates satisfies Properties 5.1 and 5.2 (Theorem 3.3 in [90]).

We call all update procedures that have ended successfully complete in the topology, and use completeness to characterize and prove properties of pub/sub mobility. By topology, we mean the logical network among brokers that is used to route messages. Typically, the topology for advertisements consists of the logical connections between the brokers, and for subscriptions it is constrained by advertisements.

The completeness of subscriptions and advertisements is given by Definition 5.3. Advertisements are complete when they have been propagated to every node that may issue an overlapping subscription in the future. Similarly, subscriptions are complete when they have been introduced at every node that has an overlapping advertisement. This formulation is flexible enough to be useful for various routing protocols. Completeness may be used to characterize the whole routing system. In addition, it may also be used to characterize a part of the routing system, such as a *path*.

Definition 5.3 *An advertisement A is complete in a pub/sub system PS if there does not exist a broker r with an overlapping subscription that has not processed A. Similarly, a subscription S is complete in PS if there does not exist a broker r such that r has an advertisement that overlaps with S and S is not active on r.*

5.2.3 Mobility-Safety

In distributed pub/sub systems it is evident that after issuing a subscription it will take some hops before the subscription is activated for all publishers. During this time several notifications may be missed. In the mobility-aware weakly valid routing configuration false negatives that occur during topology reconfiguration caused by subscriptions and advertisements from stationary components are tolerated. We also note that we do not restrict event subscription or publication activity by client systems in any way.

False negatives that occur during client mobility are not tolerated. We define a mobility-safe pub/sub system as follows:

Definition 5.4 *A pub/sub system is mobility-safe if starting from an initial configuration C_0 at time T_0 and ending in a configuration C_e at time T_e handovers (mobile clients) will not cause any false negatives.*

We also relax Definition 5.4 by not considering server failures or assuming that faults may be masked. Duplicate events may be removed by using event identifiers so they are not considered.

5.3 Related Work

Mobility support [66, 67, 110] is a relatively new research topic in event-based computing. Mobility is an important requirement for many application domains, where entities change their physical or logical location. Mobile IP is a layer-3 mobility protocol for supporting clients that roam between IP networks [70, 105]. Higher-level mobility protocols are also needed in order to provide efficient middleware solutions, for example SIP (Session Initiation Protocol) mobility [120] and Wireless CORBA [102]. Event-based systems require their own mobility protocols in order to update the event-routing topology and optimize event flow.

Siena, Rebeca, and Hermes [108] support content-based routing of events using covering relations. To our knowledge, covering relations were first introduced in the Siena project and they support the optimization of event-based communication. Two semantics are generally used: subscription-semantics and advertisement-semantics. In subscription semantics a subscription message is forwarded only if it is not already covered by other subscriptions. Similarly, in advertisement semantics a subscription is forwarded only if it covers an advertisement. In essence, subscriptions are forwarded on the reverse path of advertisements and notifications on the reverse path of subscriptions. Subscription or filter merging was used in the Rebeca project. Filter merging may be perfect or imperfect and uses filter merging rules to remove redundancy from a set of filters.

Recently, mobility extensions have been presented for several well-known distributed event systems, such as Siena and Rebeca. JEDI was one of the early systems to incorporate support for mobile clients with the move-in and move-out commands [46]. JEDI maintains causal ordering of events and is based on a tree-topology, which has a potential performance bottleneck at the root of the tree with subscription semantics. Elvin is an event system that supports disconnected operation using a centralized proxy, but does not support mobility between proxies [127].

Siena is a scalable architecture based on event routing that has been extended to support mobility [22–25]. The extension provides support for

terminal mobility on top of a routed event infrastructure. In addition, the Rebeca event system supports mobility in an acyclical event topology with advertisement semantics [152]. Context-aware subscriptions have also been investigated in the Rebeca project.

Rebeca supports both logical and physical mobility. The basic system is an acyclic routed event network using advertisement semantics. The mobility protocol uses an intermediate node, between the source and target of mobility, called Junction for synchronizing the servers. If the brokers keep track of every subscription the Junction is the first node with a subscription that matches the relocated subscription propagated from the target broker. If covering relations or merging is used this information is lost, and the Junction needs to use content-based flooding to locate the source broker [93].

JECho is a mobility-aware event system that uses opportunistic event channels in order to support mobile clients [37]. The central problem is to support a dynamic event delivery topology, which adapts to mobile clients and different mobility patterns. The requirements are addressed primarily using two mechanisms: proactively locating more suitable brokers and using a mobility protocol between brokers, and using a load-balancing system based on a central load-balancing component that monitors brokers in a domain. The topology update and its mobility-safety are not discussed.

Mobility support in a generic routed event infrastructure, such as Siena and Rebeca, is challenging because of the high cost of the flooding and issues with mobile publishers. The standard state transfer protocol consists of four phases:

1. Subscriptions are moved from broker A to broker B.
2. B subscribes to the events.
3. A sends buffered notifications to B.
4. A unsubscribes if necessary.

The problem with this protocol is that B may not know when the subscriptions have taken effect — especially if the routing topology is large and arbitrary. This is solved by synchronizing A and B using events, which potentially involves flooding the content-based network.

Recent findings on the cost of mobility in hierarchical routed event infrastructures that use unicast include that network capacity must be doubled to manage with the extra load of 10% of mobile clients [17]. Recent findings also present optimizations for client mobility: *prefetching*, *logging*,

home-broker, and *subscriptions-on-device*. Prefetching takes future mobility patterns into account by transferring the state while the user is mobile. With logging, the brokers maintain a log of recent events and only those events not found in the log need to be transferred from the old location. The home-broker approach involves a designated home broker that buffers events on behalf of the client. This approach has extra messaging costs when retrieving buffered events. Subscriptions-on-device stores the subscription status on the client so it is not necessary to contact the old broker. In this study the cost of reconfiguration was dominated by the cost of forwarding stored events (through the event routing network).

The cost of publisher mobility has also been recently addressed [96, 97]. They start with a basic model for publisher mobility that simply tears down the old advertisement and establishes it at the new location after mobility. Thus a specific handover protocol is not needed. They confirm the high cost of publisher mobility and present three optimization techniques, namely *prefetching*, *proxy*, and *delayed*. The first exploits information about future mobility patterns. The second uses special proxy nodes that advertise on behalf of the publisher and maintain the multicast trees. The third delays the unadvertisement at the source to exploit the overlap of advertisements, but does not synchronize the source and target brokers. The publisher mobility support mechanisms used in the study are not necessarily mobility-safe.

5.4 Generic Mobility Support

The Siena event system was extended with generic mobility support, which uses existing pub/sub primitives: publish and subscribe [22, 24]. The mobility-safety of the protocol was formally verified. The benefits of a generic protocol are that it may work on top of various pub/sub systems and requires no changes to the system API. On the other hand, the performance of the mobility support decreases, because mobility-specific optimizations are difficult to realize when the underlying topology is hidden by the API. Indeed, in this section we show that a general API-based pub/sub mobility support may have a very high cost in terms of message exchanges.

The Siena generic mobility support service, the *ping/pong protocol*, is implemented by proxy objects that reside on access routers. Figure 5.1 presents an overview of the process: 1. the client arrives to access point B from A and sends the *move-in* request to the new local proxy. 2. a ping request is sent and a response will be eventually received (3) from the old

proxy. The response can also be called a *pong*. The pong message ensures that subscriptions are fully propagated from B to A. 4. the client sends a *download* request for buffered events and 5. the buffered events are sent to the proxy. Finally, in 6. the client receives the messages and duplicates are removed.

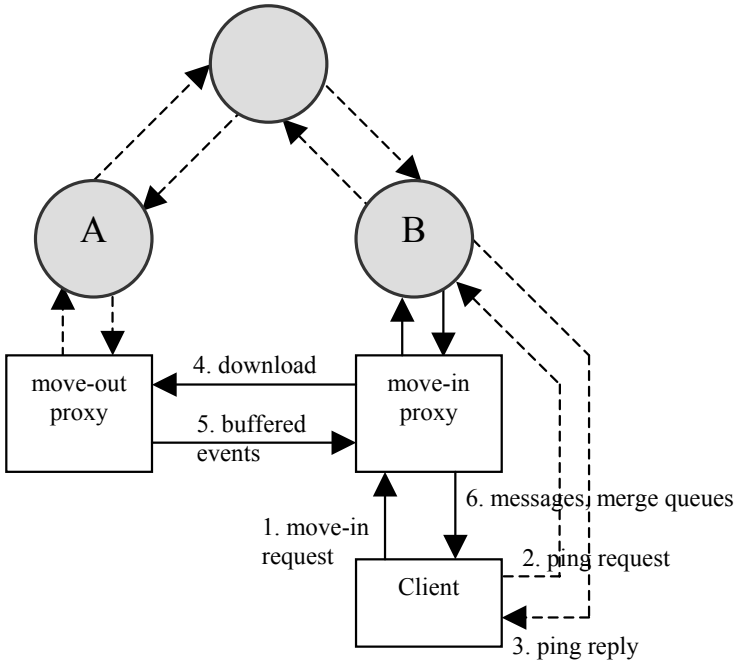

Figure 5.1: Move-in function in mobility support service.

The mobility protocol proceeds in four distinct phases: first, the target subscribes to the relocated events, then the target and source synchronize by sending *ping* and *pong* events in order to ensure that the subscriptions have taken effect, the source unsubscribes, and finally any buffered events are relocated. In addition, there may be further costs triggered by changes in the subscription tables of the intermediate routers. The cost structure of the procedure is given by Table 5.1 as the number of brokers that will be updated during each phase. N denotes the total number of brokers and n the number of brokers on the path from source to destination. For advertisement semantics we assume that the publication of a ping message also includes the advertisement. The unsubscription cost also depends on other active subscriptions on the servers and is a worst-case estimate.

For subscription semantics the basic problem with the ping/pong syn-

Table 5.1: Cost structure for generic mobility.

Phase	Sub semantics	Adv semantics
Source: Subscribe Ping(id)	N	-
Target: Subscribe Filters	$[0, N]$	$[0, N]$
Target: Subscribe Pong(id)	N	-
Target: Publish Ping(id)	n + periodic	N
Source: Publish Pong(id)	n	N
Source: Unsubscribe Ping(id)	N	n
Target: Unsubscribe Pong(id)	N	n
Target: Unadvertise Ping(id)	-	N
Source: Unadvertise Pong(id)	-	N

chronization protocol is that the signalling messages are guaranteed not to be subscribed by other brokers on the network and hence the subscription messages will be introduced at every broker on the network. With advertisements, the protocol works in a similar fashion, but the ping and pong messages need to be advertised, which also requires flooding the network.

If the client relocates faster than the ping message is propagated, it has to wait until the target receives the ping subscription. This requires that the pub/sub API allows to query the subscription status of the broker. The Siena support service does not require specific API support, because the ping messages are continuously resent [24], but this kind of behaviour further burdens the network. The ping is published by the target along with the pong subscription and they will reach the source, which replies with the pong event. It is clear that when the pong reaches the target the subscription is established between the source and target.

Publisher mobility for subscription semantics does not require additional handover functionality, since it is supported directly by the pub/sub system. Publisher mobility for advertisement semantics follows the above model and has a similar cost structure. If changes are allowed to the routing behaviour the ping/pong phase may be optimized by propagating an update message from source to target on the reverse path of the target's advertisement.

5.5 Acyclic Graphs with Advertisements

5.5.1 Overview

An outline of the protocol we describe was presented in [54, 93, 152] with mobility restricted to between border brokers. A mobility protocol for a hierarchical routing topology with more assumptions was examined in [17]. Formally, the network of application-level event brokers is an undirected acyclic graph $G = (V, E)$, where V is the set of vertices and E is the set of edges of the graph. We are interested in finding the theoretical cost for the handover protocol in terms of message exchanges and establishing that handovers are mobility-safe. Some systems allow mobility only between border brokers, the leaves of the routing tree, which limits mobility. We allow clients to roam between any two brokers.

In subsequent examination we assume that the servers use a reliable communication mechanism, G is connected, and messages are delivered in order on each link. Some handover protocols use pub/sub routing to find the source broker; however, this approach may require the use of flooding to find the source broker, for example, in scenarios where the relocated subscriptions are not advertised. In this case, the protocol breaks down, because there is no active path between the source and destination in the pub/sub network. It is not possible to perform the handover only using publish/subscribe routing information. Hence *out-of-band* communication is needed when advertisement semantics is used.

We also assume that buffered events are delivered outside the event-routing framework, since they have already been routed and it is not efficient to reroute them. In some existing systems buffered events are delivered using the event-routing system. While this approach adheres to the pub/sub communication, it has been shown that this cost dominates the pub/sub signalling cost [17] in mobility scenarios. We consider the number of exchanged messages because update processing and propagation causes the most stress for the pub/sub system.

A handover protocol typically supports either *make-before-break* or *break-before-make*. In the former, the source and destination brokers negotiate the handover before the client establishes a connection with the destination broker. This has also been called *prefetching* [17, 97]. In the latter, the destination is not necessarily known and the handover is negotiated after the client establishes connection with the destination broker. In subsequent analysis, we focus on the latter case, but discuss also the former strategy. In principle, the two are very similar for pub/sub mobility and they face the same challenges. The main difference is that the network-initiated make-

before-break has lower latency as perceived by the clients and the covering optimization cannot be performed, because the update message is sent to the destination in any case.

5.5.2 Mobile Subscribers

Handovers on a Complete Topology

Handover is performed between two brokers in the network: $a, b \in V$. a is the *source broker* and b is the *destination broker*. Let α denote the path from a to b. The path is unique, since G is acyclic. We view the path α as consisting of the edges on the path, since that makes it more convenient to handle split paths. An edge between a and b is called *active* with respect to a subscription S, if S has been sent on that edge. A path is active if all of its edges are active. An inactive path is one that is not active, i.e., contains at least one inactive edge. Now, let α_a be the set of active edges on the path, and α_i the set of inactive edges on the path. The topology update cost is the cost of updating α_i.

Figure 5.2 presents an overview of the subscription handover for a complete topology. P denotes a publisher and c and d are new brokers. The handover starts when b receives a message, typically from the client, that the subscription S should be relocated from a to b. In subsequent analysis, we focus on a single subscription, but the analysis generalizes also to a set of subscriptions.

Now, b may optimize the operation if S has already been subscribed to by b. In this case, b simply starts to buffer notifications for S and retrieves the notifications buffered at a for the client. If the optimization cannot be applied, b issues a subscription message, which is propagated by the event system. The subscription message must include those parts that are not covered by existing subscriptions.

Figure 5.3 presents a similar handover scenario, but now with a publisher between a and b. In the figure the complete subscription topology update proceeds as follows: 1. b sends the update message, 2. the update message is propagated towards a on the reverse path of advertisements and will meet a's subscriptions at node M (which can also be a itself), 3. M sends the update message towards a, 4. the session is transferred and a may unsubscribe, if necessary. It is clear S must be covered at M.

Figure 5.4 illustrates the covering optimization, which may be performed at b if S has already been subscribed and it is known that S is complete on the path from a to b. We know that a and b are connected since G is connected. Also the path between a and b is unique, since G is

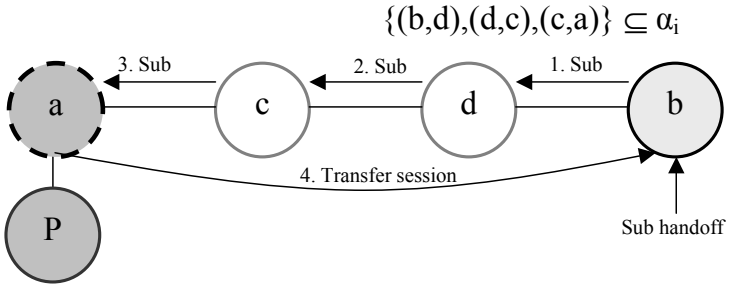

Figure 5.2: Subscription handover with a complete topology when the path from the source to the destination is inactive

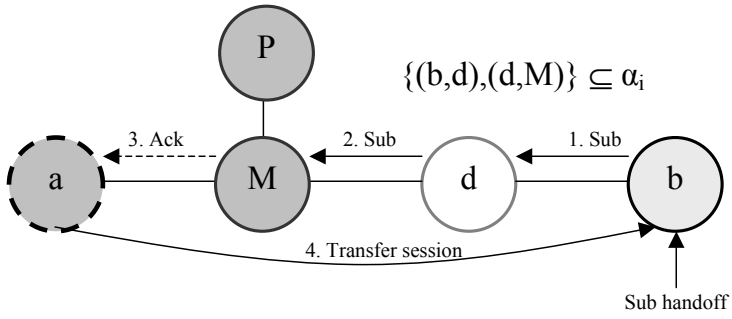

Figure 5.3: Subscription handover with a complete topology when the publisher is between the source and the destination.

acyclic. If the covering optimization is not applicable we know that S is not active on all edges on the path from a to b.

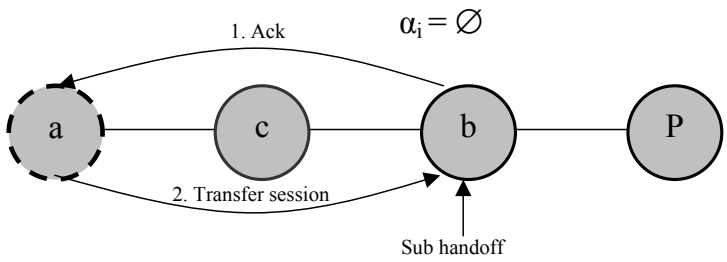

Figure 5.4: Subscription handover with the covering optimization.

Proposition 5.5 *There exists a node M on the path $a \rightsquigarrow b$ such that the path $a \rightsquigarrow M$ consists of exactly the edges in α_a (and then α_i is the path $b \rightsquigarrow M$).*

Proof. Let P be the publisher related to S. Since we are assuming a complete topology, the path $x \rightsquigarrow P$ is active for any node x that has sent S. Specifically, the path $a \rightsquigarrow P$ is active.

If there are no active edges on the path $a \rightsquigarrow b$, then a is clearly the desired M. Otherwise there is a last active edge on that path; let this edge be (x, y). Due to the uniqueness of paths in G either the path $a \rightsquigarrow P$ passes through y, the path $y \rightsquigarrow P$ passes through a or there is a node z on the path $a \rightsquigarrow y$ through which both paths $a \rightsquigarrow P$ and $y \rightsquigarrow P$ pass. Clearly in all of these cases the whole path $a \rightsquigarrow y$ is active, and by the choice of y no edges on the path $y \rightsquigarrow b$ are active. Hence y is the desired M. □

It is clear that the M in the proposition is unique. We can distinguish four separate cases depending on the relative positions of a, b, and P:

1. a is on the path $P \rightsquigarrow b$ (Figure 5.2)
2. P is on the path $a \rightsquigarrow b$
3. b is on the path $a \rightsquigarrow P$ (Figure 5.4)
4. A node M on the path $a \rightsquigarrow b$ is on both paths $a \rightsquigarrow P$ and $b \rightsquigarrow P$ (Figure 5.3)

Of these cases 4 is clearly equivalent to 2, since the update is complete when S from b reaches M (we refer to this as *collapsing* P to M). Similarly

b cannot distinguish between cases 1 and 2. So we are left effectively with 1 and 3, and in the latter case the path $b \rightsquigarrow P$ is already active, so no update is necessary. It follows that if an update is required it is sent by b to exactly one output interface.

Handovers on an Incomplete Topology

There are two main forms of routing topology incompleteness that introduce additional overhead to the protocol. First, given that subscriptions are initially complete, simultaneous routing-configuration changes by other clients may make the propagation of the update message more difficult. Second, the subscriptions may not be initially complete.

Simultaneous advertisements and subscription activity are not problematic; however, given that there are simultaneous unadvertisements on the subpath from a to b, Proposition 5.5 is not necessarily valid. Consider a scenario in which during the handover of a subscriber, the advertisement is removed from the system. In this case, M may not exist or may disappear during the handover. The propagation of the update message from b will reach neither a nor M. The protocol must be able to cope with this and signal to a that the update is complete. This incompleteness of the subpath is detectable at both a and b when the unadvertisement is received.

The most difficult scenario happens when the advertisement is removed and re-established between and after the removed advertisement on the path so that the update is lost, but neither a nor b will receive the unadvertisement. This scenario is rare and does not occur when advertisements are relocated using the publisher handover protocol presented later in this chapter. The protocol may cope with this scenario by using timeouts, periodic update messages, or by propagating an update packet to the source using a lower-layer routing protocol. Simultaneous advertisements do not pose such a problem.

It may also be the case that the set of subscriptions is still being propagated and the subscription path is not totally active at the time of the handover. This happens when the handover occurs while the system propagates and processes subscription messages or update messages. In this case Proposition 5.5 is not necessarily valid. This kind of behaviour may occur because the handover is triggered by out-of-band mechanisms, for example terminal mobility. If mobility is activated by sending a message using the pub/sub system this problem may be avoided.

Out-of-band signalling may cause a number of false negatives. This happens when the client rapidly relocates and the covering optimization is performed at b. False negatives may occur if the out-of-band signalling

transfers buffered messages before the path from a to b is complete. Essentially, with the current assumptions, the covering optimization cannot be performed at b if mobility-safety is to be ensured. This means that b has to ping a by propagating a message through the path. Another option would be to send an update message from a to b when the client moves out, but this also requires that the path is tested. The destination server does not know where the source is located and may have to use *content-based flooding* to find M and a. This flooding happens on the reverse path of covering subscriptions or overlapping advertisements.

Incompleteness of advertisements or the relocated subscriptions creates overhead for the protocol. This may happen because the source may not be locatable at all if the advertisements/subscriptions have not reached the destination or the current advertisements are in the wrong direction. The most challenging scenario is the latter one with a number of overlapping advertisements that are not in the direction of the source. In this case, the destination server cannot locate the source using advertisement or subscription information. This is typically solved using periodic updates, which is not practical. Later in this chapter, we present several possible solutions to this problem, namely overlay routing and rendezvous points.

Fast Handovers

Fast handovers are one source of incompleteness. Fast handovers occur when a client relocates so fast that the update messages are still propagating. False negatives that occur during the propagation of the original subscription are not counted as false negatives according to our model. However, since the completeness of the path is ensured for each handover, all messages that would have been received at the source are buffered for the client. In this sense, as long as completeness of the path is ensured, the system is mobility-safe even in the presence of fast handovers. The challenge is to minimize the cost in ensuring this completeness and prevent a cascade of handovers.

Mobility-Safety

Lemma 5.6 states that an initially complete pub/sub system is mobility-safe. The result also holds when initial completeness, for example, in the case of fast handovers, is not assumed given that it is not possible for the update message to be lost on the path from a to b. If the message may be lost due to reconfiguration, techniques such as periodic updates, timeouts, overlay routing and addressing, or rendezvous points, are needed for the

lemma to hold. The lemma applies to both make-before-break and break-before-make.

Lemma 5.6 *A pub/sub system with only subscriber mobility, initially complete subscription and advertisement configuration, and simultaneous subscription and advertisement activity is mobility-safe.*

Proof. Since we are using the relaxed model where advertisements are assumed to be complete, any advertisements happening during mobility are not relevant. The discussion in Section 5.5.2 indicates that the protocol copes with simultaneous unadvertisements. Hence advertisement activity does not affect mobility-safeness.

For simultaneous subscription activity it is sufficient to consider only the cases 1 and 2 from Section 5.5.2. In case 2 the path $a \rightsquigarrow P$ will be kept active until the update is complete by the protocol, and when the update is complete, the path $P \rightsquigarrow b$ is active. Similarly in case 1 the path $b \rightsquigarrow a$ will be made active, and in this case no edge can be made inactive by the update. So in both cases the path $P \rightsquigarrow a$ cannot become inactive before the path $P \rightsquigarrow b$ is made active, so there can be no false negatives. □

Border Broker Restricted Mobility

Mobility becomes simpler if we restrict mobility support only to border brokers, which are the leaves of the routing tree. In this case the border broker always forwards at most one update message. Also in this restricted scenario it is not possible to detect the completeness of the topology at b. If S is covered at b, we do not know if S sent by a has taken effect. If S is not covered at b, M exists, and there is only one direction for the update message.

5.5.3 Mobile Publishers

Related work has typically considered only the protocol for relocating subscriptions. Since it is probable that subscriptions are relocated more often this is reasonable. In addition, publisher mobility support is not needed with subscription semantics. A separate publisher mobility protocol is needed for advertisement semantics. The problem with mobile publishers is that advertisements are propagated throughout the routing network. This means that the removal of an advertisement may have a high cost. We follow a similar approach as with mobile subscribers.

Figure 5.5 presents an overview of the process for break-before-make. The protocol proceeds in four phases: 1. b sends an update message towards

a and overlapping subscriptions are sent towards b. 2. Existing advertisements and subscriptions meet the advertisement sent by b and a is notified by M (Proposition 5.5) that the topology has been updated. If completeness of the advertisement is not known M simply forwards the update message towards a. M is a in the figure. 3. a sends an update message to b to ensure completeness. Finally, a unadvertises if necessary.

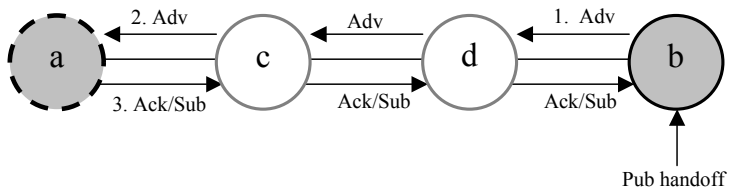

Figure 5.5: Advertisement handover with a complete topology.

Publisher mobility differs from subscriber mobility, because state is not transferred. Rather, the path between a and b must first be tested that the new advertisement issued by b has reached a. Then the path must be tested again to ensure that any subscription from a has reached b. After this, the publisher can be certain that any published events are not missed by the subscribers.

Publisher mobility is easy when the topology is complete. In this case, the covering optimization may also be performed. However, in the typical case when completeness cannot be assumed the path must be tested in both directions. We also have problem of locating nodes in the pub/sub network that was discussed in Section 5.5.2.

In some cases, subscribers are already connected after the first advertisement update sent by b. Since b does not know if there are other subscriptions that overlap with the advertisement later in the path, the whole path from b to a needs to be checked. It is important that the protocol is ended properly in order to ensure that published events are properly disseminated to subscribers, and that the next handover may be started from a complete configuration.

If the advertisements are not successfully terminated in G the publisher handover protocol ensures that upon completion of the handover the subgraph defined by the path from a to b is complete for both advertisements and subscriptions.

The protocol for network-initiated make-before-break is similar to the presented break-before-make protocol, but in this case the update message

is forwarded before the client relocates and it is sufficient to pass it in only one direction. The update message adds an advertisement from the direction of b and connects any subscriptions in this direction. Therefore, the make-before-break case is more efficient given that b can be efficiently located in the pub/sub network.

Mobility-Safety

In this section we establish the mobility-safety of a pub/sub system with both subscriber and publisher mobility. The discussion with Lemma 5.6 applies here also. Given that it is not possible for signalling messages to be lost and that a given broker can be located using other means than pub/sub routing information, initial completeness is not necessary. In addition, the results hold for both make-before-break and break-before-make.

Lemma 5.7 *A pub/sub system with only publisher mobility, initially complete subscription and advertisement configuration, and simultaneous subscription or advertisement activity is mobility-safe.*

Proof. The discussion following Proposition 5.5 applies here too (but we will call the nodes a, b, and W here for clarity). Similarly we may collapse W to M, since the active path $W \rightsquigarrow M$ is not affected by the publisher moving. Also case 3 is trivial, since the subscription then already exists at b, and so the path $b \rightsquigarrow W$ is already active.

In case 2, the advertisement sent by b is seen by W before a, so W's subscription makes it to b before a's ping, so the path $W \rightsquigarrow b$ is made active before $W \rightsquigarrow a$ can be inactivated.

The final case to consider is 1. Here if a couples the subscription with its ping, the path $b \rightsquigarrow a$ is made active simultaneously as the producing responsibility is transferred to b, so no false negatives can happen here either.

Simultaneous advertisement and unadvertisement activity can be treated as in Lemma 5.6. □

Theorem 5.8 *A pub/sub system with both subscriber and publisher mobility and initially complete advertisement and subscription configuration is mobility-safe.*

Proof. We will denote by subscript O the original position of subscribers and publishers, and by subscript F their final position. The possible cases can be reduced to the three shown in Figure 5.6, where only relevant brokers are shown and the edges between brokers are actual paths. In case I,

since the path $M_2 \rightsquigarrow M_1$ is active and not affected by mobility, the situation is equivalent to the subscriber and publisher moving serially, which is mobility-safe by Lemmas 5.6 and 5.7.

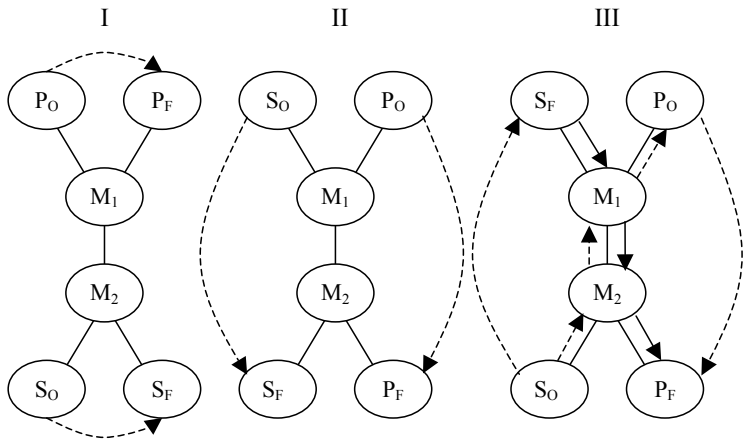

Figure 5.6: Three cases for simultaneous pub/sub handovers.

The investigation for the other cases now splits according to whether the subscription sent by S_F reaches the M nodes before the advertisement sent by P_F, after that but before the acknowledgement sent by P_O, or after the acknowledgement. We refer to these as *before*, *during*, and *after*, respectively.

In case II we need to consider node M_2. In the before scenario the subscription makes its way to P_O, making this scenario equivalent to subscriber mobility followed by publisher mobility. In the during and after scenarios, the subscription is sent towards P_F, so these scenarios are equivalent to publisher mobility followed by subscriber mobility. All of these are therefore mobility-safe by Lemmas 5.6 and 5.7.

In case III the relevant M node is M_1. By similar arguments as in case II we see that the before and after scenarios are equivalent to serial mobility. In the during scenario the advertisement has been seen by M_1, but not the unadvertisement from P_O, so the path $M_1 \rightsquigarrow P_O$ remains active. Due to the reliability of the network, the subscription reaches P_F before the acknowledgement, and therefore is updated before P's mobility procedure is completed. Hence the protocol is mobility-safe even in this last case. □

5.6 Rendezvous Point Models

5.6.1 Overview

A *rendezvous point (RP)* is a special node in the routing network that is used to coordinate signalling. Rendezvous points are used in many overlay routing systems [115, 124, 153] to reduce communication costs and realize non-fixed indirection points.

Hermes [108, 109] is a peer-to-peer event system based on an overlay called Pan that supports a variant of the advertisement semantics. Hermes leverages the features of the underlying overlay system for message routing, scalability, and improved fault-tolerance. Hermes supports the basic pub/sub operations introduced previously. Rendezvous points are used to coordinate advertisement and subscription propagation. The RP manages an event type and Hermes supports chaining RPs into type hierarchies. The RP of an event type is obtained by hashing the event type to the flat addressing space of the overlay [109].

Figure 5.7 illustrates rendezvous-point-based operation using 6 phases: 1. A publisher advertises an event type (and a filter in type/attribute-based routing). 2. The advertisement is forwarded to the rendezvous point. 3. A subscriber subscribes to an event of the same type (and a filter in type/attribute-based routing). 4. The subscription message is not covered (type or filter) at any intermediate broker and is forwarded to the rendezvous point. 5. Another subscriber subscribes. 6. The subscription message is propagated towards the RP.

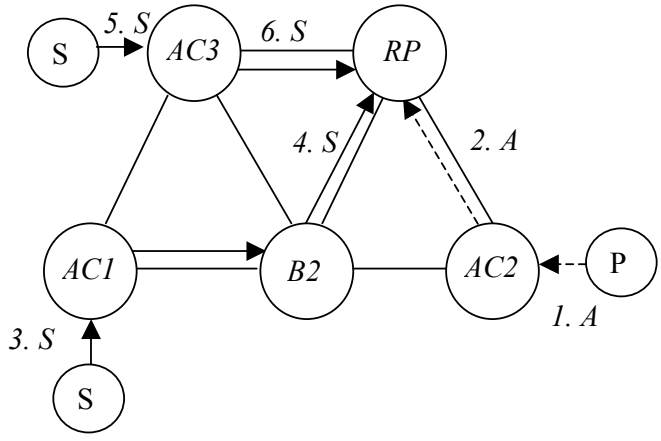

Figure 5.7: Forward path establishment in the Hermes model.

After the last phase, the two routing models supported by Hermes differ. In type-based routing, any events conforming to the advertisement from the publisher are sent on the forward path of the advertisement to the RP, which then forwards the event on the reverse path of any subscriptions. In type/attribute-based routing, the RP sends the subscriptions on the reverse path of advertisements. Any events conforming to the advertisement from the publisher are sent on the reverse path of subscriptions.

The model used by the Hermes system is the familiar advertisement semantics model with three key differences:

- All messages (type-based routing) or advertisements and subscriptions (type/attribute-based routing) are sent towards the RP. Thus routing topology is constrained by the RP.
- Advertisements are introduced only on the path from the advertiser to the RP.
- Subscriptions are introduced on the path from the subscriber to the RP. In addition, for type/attribute-based routing subscriptions are sent on the reverse path of any overlapping advertisements.

These differences are interesting because advertisement becomes a local property of a branch of the multicast tree rooted at an RP. This may be modelled using virtual advertisements. In this case, an RP has virtual advertisements for all events of the event type managed by the RP and hence subscriptions are sent towards it. In the following examination we assume that the overlay topology is static; a dynamic topology would require a more complex investigation.

Figure 5.8 illustrates a subscription handover using a rendezvous point. The handover proceeds as follows: 1. The client ensures that the subscription is complete to the RP before mobility. 2. An update message is sent with a subscription towards the RP at the destination server. 3. The update reaches the RP, a message is sent to a that triggers the session transfer (4).

Figure 5.9 illustrates an advertisement handover using a rendezvous point. First, the client ensures that the advertisement is complete to the RP (1). Then, the client relocates and issues an advertisement/update message at the new server (2). The advertisement is propagated towards the RP (3). The RP sends an update message to a (4) and propagates an acknowledgement along with any overlapping subscriptions towards b (5). The handover terminates when b receives this update (6). This model does not guarantee the safety of any subscriptions not complete to the RP.

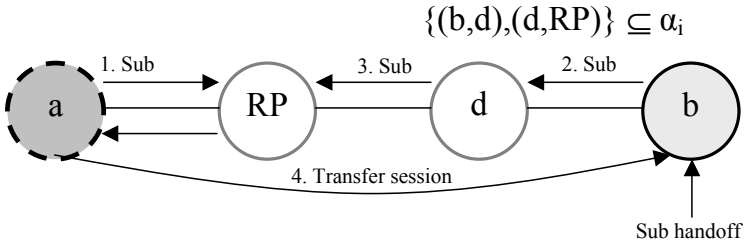

Figure 5.8: Subscription handover with a rendezvous point.

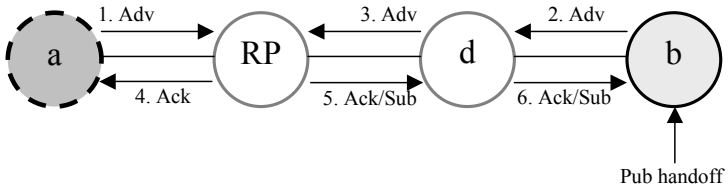

Figure 5.9: Advertisement handover with a rendezvous point.

The support for type hierarchies requires that any relevant mobility update messages for a subtype are propagated to its supertypes. The efficiency of this depends on the relative positions of the subtype brokers with respect to the supertype brokers in the routing topology. If the brokers that are responsible for supertypes are placed near the subtype brokers this update may be performed efficiently.

5.6.2 Mobility-safety

The rendezvous-point-based model can be seen as a special case of the general acyclic graph model. Mobility-safety of the RP model is established by using virtual advertisements at rendezvous-points (Theorem 5.10).

Lemma 5.9 *Any system PS_S with subscription semantics may be modelled in terms of an equivalent system PS_A with advertisement semantics by defining virtual advertisements that overlap with all subscriptions.*

Proof sketch: By extending PS_S with virtual advertisements that always overlap with subscriptions we can implement subscription semantics using advertisement semantics and thus the behaviour of PS_S is not changed. □

Theorem 5.10 *An initially complete rendezvous type/attribute-based routing model is mobility-safe.*

Proof sketch: Follows from Theorem 5.8 and Lemma 5.9 by establishing virtual advertisements at each RP for the corresponding event type. □

5.6.3 Incompleteness

The RP may be used to guarantee completeness of advertisements and subscriptions by requiring an acknowledgement from the RP. We propose to solve the problems posed by incomplete topologies using two mechanisms:

- RP completeness checking both at the source and destination of mobility.
- Preventing content-based flooding using the overlay-address, which allows to determine in which direction a node is located. This property may also be used to prevent the content-based flooding at M when forwarding the mobility update messages within the event topology.

The problems of incompleteness with the regular sub/adv semantics can be avoided with the RP model by ensuring that each subscription and advertisement is complete to the RP. This incurs additional cost, but the number of acknowledgement messages may be minimized by pushing the acknowledgement generation away from the RP to nodes that have covering subscriptions or advertisements that are complete to the RP. The two central problems that are solved are as follows. First, false negatives due to incompleteness of the path and a ping between source and destination is not needed. Second, by using the overlay address to locate the source, content-based flooding does not need to be used.

5.7 Upper and Lower Bounds

Table 5.2 presents the complete topology update costs of the break-before-make protocols as the number of intermediate nodes, excluding source and destination, that need to be updated. Table 5.3 presents the same costs for the incomplete case. Here n denotes the total number of nodes in the system and k the number of nodes on the path from a to b. The generic ping/pong protocol and acyclic graph protocol with the incompleteness assumption also require periodic pinging to ensure correct operation, denoted by the term P, $P \geq 0$. If lower-layer addressing is used to find brokers, periodic

Table 5.2: Topology update costs for a single handover for a complete subpath.

Protocol	Upper bound	Lower bound
Generic API Pub/Sub	$4(n-2)+(P+2)\cdot(n-2)$	$2(n-2)+4k$
Generic API Pub/Sub RP	$2\cdot RP_{max}+(P+4)\cdot(n-2)$	$2\cdot RP_{max}+4k$
Acyclic graph Sub	$n-2$	0
Acyclic graph Pub	$2(n-2)$	0
Cyclic RP Sub	$K\cdot RP_{max}$	0
Cyclic RP Pub	$2K\cdot RP_{max}$	0
Acyclic RP Sub	$min(K\cdot RP_{max},(n-2))$	0
Acyclic RP Pub	$min(2K\cdot RP_{max},2(n-2))$	0

pinging is not required since it is not possible that the update message is lost. In this case $P = 0$. Unsubscription cost is not included in the table, and additional signalling is needed to transfer the buffered messages. The value of 0 means that the topology update can be skipped due to the covering optimization. RP_{max} denotes the maximum path length to an RP. $RP_{max} \leq n-2$. K denotes the number of rendezvous points that need to be updated for the client.

The cost breakdown for the ping/pong protocol is presented in [134] in more detail. The protocol first advertises ping and pong events with a worst-case cost of $2(n-2)$ if a and b are excluded. Then it connects the subscriptions with the cost $2k \leq 2(n-2)$. Both ping and pong message takes k hops. In addition, the ping is sent periodically until the pong is received. The generic ping/pong protocol may also be used in rendezvous-based topologies. In this case, the advertisement cost is reduced to $2 \cdot RP_{max}$.

We note that for arbitrary topologies the generic API may not guarantee completeness of the path from source to target. This happens when the control messages take a different path through the network than the relocated subscriptions and the corresponding events. The generic API with RPs is an example of this kind of topology and completeness is guaranteed if the underlying graph is acyclic.

The incomplete acyclic protocol requires periodic updates and that the path is always tested. With rendezvous points periodic updates are not needed, because all updates are sent towards the RP. However, for the incomplete case the paths from a to RP and b to RP need to be tested.

The publisher cost for the generic ping/pong mechanism is the same as the subscriber handover cost. Acyclic-graph-based publisher mobility

Table 5.3: Topology update costs for a single handover for an incomplete subpath.

Protocol	Upper bound	Lower bound
Generic API Pub/Sub	$4(n-2)+(P+2)\cdot(n-2)$	$2(n-2)+4k$
Generic API Pub/Sub RP	$2\cdot RP_{max}+(P+4)\cdot(n-2)$	$2\cdot RP_{max}+4k$
Acyclic graph Sub	$(P+1)\cdot(n-2)$	k
Acyclic graph Pub	$2(n-2)$	$2k$
Cyclic RP Sub	$2K\cdot RP_{max}$	0
Cyclic RP Pub	$3K\cdot RP_{max}$	0
Acyclic RP Sub	$min(2K\cdot RP_{max}, 2(n-2))$	0
Acyclic RP Pub	$min(3K\cdot RP_{max}, 3(n-2))$	0

protocols require a ping from the source to the destination in any case, which doubles the cost when compared with subscription mobility.

For the complete cyclic rendezvous point case, each RP needs to be tested for completeness at b giving a worst-case cost of $K\cdot RP_{max}$ for subscriber mobility, and $2K\cdot RP_{max}$ for publisher mobility. For the complete acyclic cases, the costs are similar, but bounded by the size of the network.

For the incomplete cyclic rendezvous point case, K RPs must be tested for completeness both at a and b. Hence, a single RP requires a maximum of $2\cdot RP_{max}$ of nodes to be updated for the subscriber case. The publisher case requires that any subscriptions are propagated from the RPs to b, which requires additional messaging giving a total cost of $3K\cdot RP_{max}$.

The costs are similar for the acyclic case, but again they are bounded by the size of the network. A total of $(n-2)$ intermediate nodes need to be tested to find K RPs at a. This testing is also needed at b, and for publisher mobility b has to wait for a reply from the RPs. The completeness checking allows the use of the covering optimization at b and hence the lower bound cost is zero. Periodic updates are not required for publisher mobility with acyclic graphs or rendezvous points, because the advertisements will eventually meet.

5.8 Experimentation

We developed a Java-based discrete event simulator for investigating different handover protocols and routing topologies[1]. To our knowledge, this is the first graphical mobility simulator for pub/sub and event topologies.

[1] http://www.hiit.fi/fuego/fc/demos

The simulator allows the visual inspection of the topology and uses network topologies generated with the BRITE topology generator [85]. We use network delays from the BRITE model, and also include a constant processing overhead at each event broker. Before simulation, the cyclic BRITE topology is converted into an acyclic graph using breadth-first search starting from an arbitrary node.

To simplify the comparison of different mobility protocols, we first generate a mobility script for the given network topology. The destination of a mobile client is selected using a uniform distribution over the set of edge brokers, and the duration of mobility and mobility interval are constant values. If rendezvous points are used, the RP is selected using a uniform distribution over the non-edge brokers.

After the script has been created for the desired number of handovers, each mobility protocol is tested using the script. The simulation counts the number of simulation events caused by the handovers (the total signalling cost) and the average latency of the handovers in simulation time. The latency is measured from the start of the mobility related signalling at the destination to the successful completion of the handover.

Figure 5.10 presents the graphical user interface of the simulator applet and shows the imported BRITE network topology. The layout is based on the graph topology and not the physical location of the nodes. The black dots denote fully connected streams, subscriptions connected with overlapping advertisements, and grey dots denote advertisements. The GUI allows to inspect the routing tables of each node and the mobility script. The GUI also shows the simulation events as they occur and provides *pause* and *step-into* features for inspecting the evolution of the routing topology. The simulator has a warm-up phase during which the initial advertisements and subscriptions are established. After this warm-up, the routing topology is complete and the mobility script is replayed in a deterministic fashion. This allows direct comparison of the mobility protocols. Correctness of operation is ensured by assertions on the completeness of subpaths, routing table integrity, and detecting missing and duplicate messages.

The simulator also calculates the theoretical upper bound cost (theor. in the figure) for the simulation scenario based on the equations presented in Table 5.2. The simulation output is the average cost for the scenario. The simulator supports 8 different variants of the three base protocols presented in this chapter, namely the ping/pong protocol, acyclic graph protocol, and the rendezvous-point-based protocol. The two optimizations supported for the acyclic protocol are the covering optimization if completeness of the relocated subscription is assumed, and the overlay address-based optimiza-

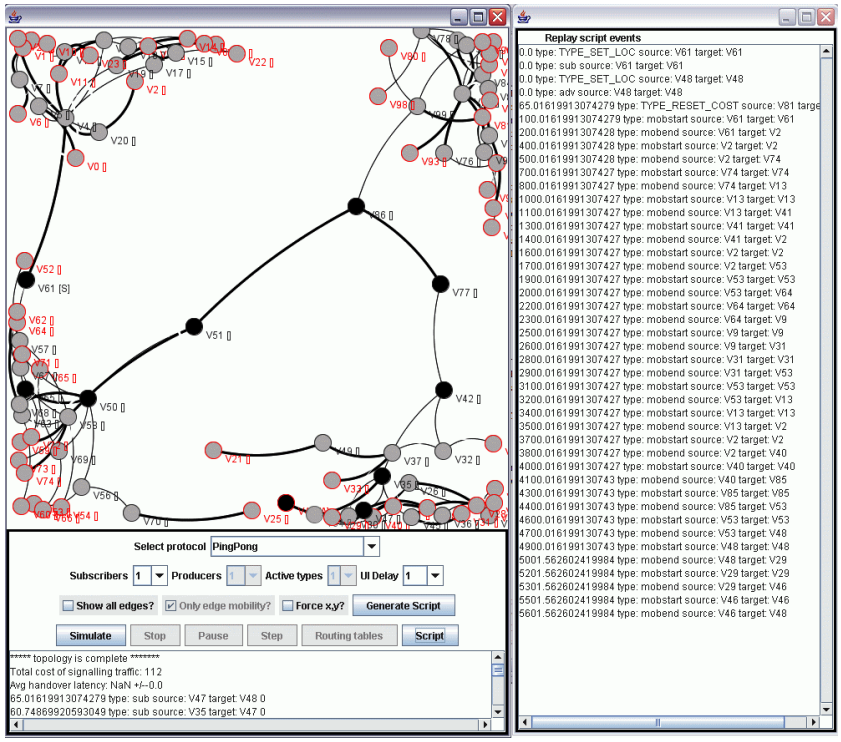

Figure 5.10: Mobility simulator GUI.

tion. If completeness of the relocated subscription is assumed, the covering optimization is used by the simulator. Similarly, if overlay-based addressing is used, content-based flooding does not occur.

Border broker mobility Figure 5.11 presents the total signalling cost as the number of messages sent during the simulation for a variable number of handovers for a BRITE topology with 4 autonomous areas, 100 routers, 50 subscribers, and a single producer. The y-axis is logarithmic. The mobility interval and duration were set to 1 hour and 1/2 hour, respectively. The periodic ping for the generic API was set to 100 milliseconds. The destination of mobility is selected using a uniform distribution over all nodes. Out-of-band delay consists of the network-level delay only. Overlay-based communication has a constant forwarding overhead added to network-level delays. The maximum network-level delay between any two nodes was 25.7 ms, and the processing overhead of pub/sub routing was set to 2 ms at each node.

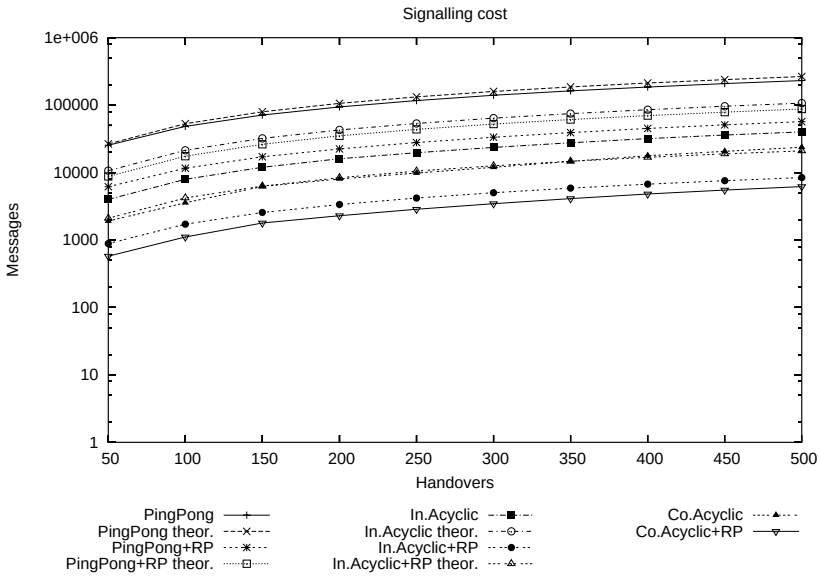

Figure 5.11: Simulation results for a variable number of handovers with border broker mobility.

The average signalling costs of the protocols are below the estimated theoretical upper bounds. The ping/pong protocol is the most inefficient. The use of a rendezvous point reduces the cost significantly, but it still has a higher cost than the acyclic protocols. The complete acyclic protocol has a lower cost than the incomplete protocol, because of the covering optimization.

We measured the average handover latency for the protocols. The latency consists of the processing delay between the source and destination of mobility using the given mobility protocol. Queuing delays are not included in the delay. The latencies are presented in Table 5.4. The high latency of the ping/pong protocol is due to the periodic ping, which is needed for mobility-safety.

Full Mobility Figure 5.12 presents the results for a variable number of handovers for the full mobility scenario and compares it with the border broker scenario. In full mobility, the destination of mobility is selected using a uniform distribution on the set of servers (excluding the current server). The y-axis is logarithmic. We focus on the complete and incomplete acyclic graph protocols. The results are similar to the other mobility scenarios.

Table 5.4: Average handover latency.

Protocol	Latency (ms)
Ping/Pong	170 ± 26
Ping/Pong RP	168 ± 22
Incomplete acyclic	49 ± 16
Incomplete acyclic RP	48 ± 17
Complete acyclic	39 ± 17
Complete acyclic RP	39 ± 17

The main difference is that the border broker scenario has longer path lengths between the source and destination of mobility.

A Variable Number of Subscribers and Publishers We also experimented with a variable number of subscribers and publishers. The cost for the generic ping/pong protocol is independent of these numbers. Hence, it is not necessary to consider the protocol in these scenarios. The incomplete and complete acyclic protocols are affected by the subscription topology due to two phenomena. First, the cost of content-based flooding grows as the number of subscribers increases. Second, the probability that the covering optimization can be performed for the complete protocols increases, because the number of complete paths grows.

In the variable number of subscribers scenario we used a single publisher and a total of 100 subscriber handovers were made for each measurement. We experimented with both border broker restricted mobility (edge mobility) and full mobility. Figure 5.13 presents the results for the subscriber case. The cost of the incomplete acyclic case grows as the number of subscribers grows. This is due to content-based flooding. The complete protocol has a peak at 20 subscribers and then diminishes as the probability for the covering optimization increases. The border broker mobility and full mobility results are similar. The path lengths in mobility are longer for the border broker case, so it has a higher cost for the incomplete protocol. For the complete protocols, the border broker case has mobility only between edge brokers and thus the covering optimization is performed more often. The full mobility case has a larger server set and thus is not as efficient.

Figure 5.14 presents the simulation results for a variable number of publishers for both border broker and full mobility. In this scenario, 100 handovers were simulated with 50 subscribers for each measurement. The cost of the incomplete and complete protocols grows when the number of

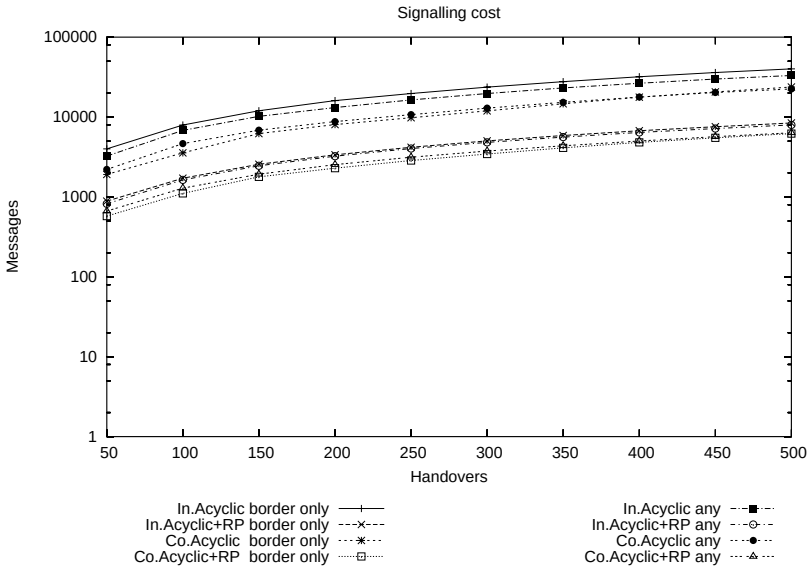

Figure 5.12: Simulation results for a variable number of handovers with full mobility.

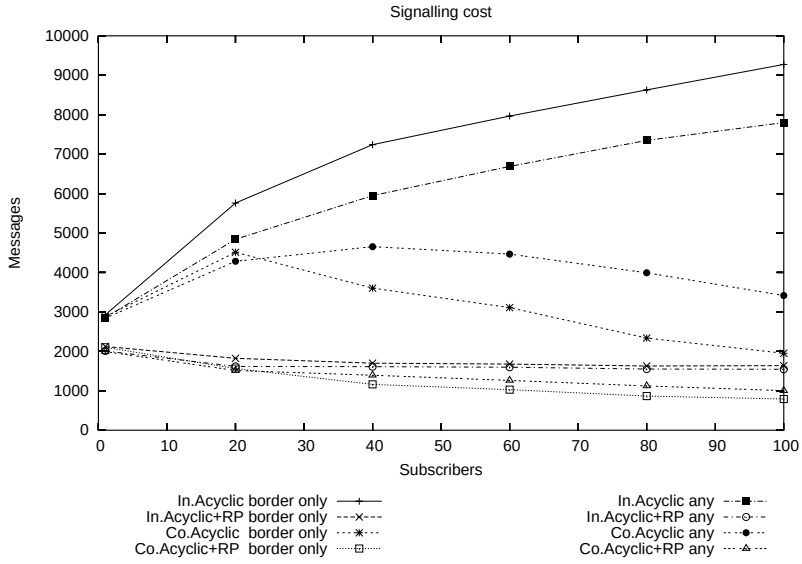

Figure 5.13: Simulation results for a variable number of subscribers.

publishers increases. The new advertisements require a moderate amount of additional messaging. The number of publishers did not have a significant effect on the rendezvous point-based protocols. This was expected, because the RP constrains the topology and flooding does not occur.

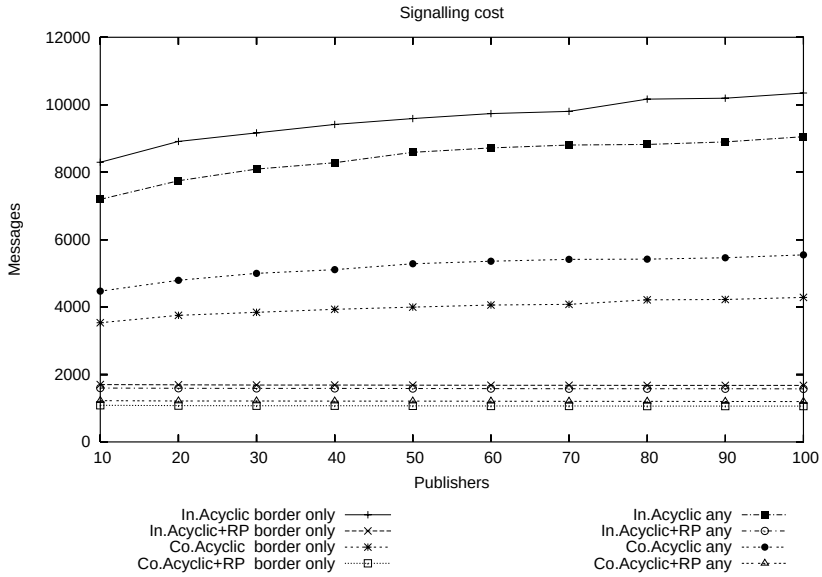

Figure 5.14: Simulation results for a variable number of publishers.

5.9 Engineering Implications

The presented discrete formal model for both subscriber and publisher mobility allows us to gain valuable insight into the engineering of efficient and mobility-safe publish/subscribe protocols. Typically the mobility-safety of pub/sub protocols has not been established, and indeed we have observed that some implementations do not guarantee this. Yet, receiving every message is important for many signalling applications. We have presented a formal foundation for pub/sub mobility, which may also be used with the subscriber and publisher mobility strategies, such as the *prefetching* and *proxy* protocols proposed in [17, 97].

From the analysis presented in this chapter we can draw the following design principles, which are important for engineering efficient mobility-safe pub/sub systems:

- The generic protocol is mobility-safe and applicable to various underlying pub/subs systems, but it is very inefficient and does not allow pub/sub system or topology-specific optimizations. We note that the mobility-safety of this mechanism requires that the ping/pong interaction is sufficient to establish the completeness of the path or paths.

- The general acyclic graph-routing topology is more efficient than the generic protocol, but suffers from the problem that the source broker needs to be located using event routing. Since covering and merging do not preserve information pertaining to the original broker that issued a subscription or advertisement, the use of content-based flooding may be required. This routing topology also suffers from the incompleteness of subscriptions, and thus the covering optimization that uses out-of-band communication cannot be performed if mobility-safety is required. Incompleteness may also cause a broker to flood subscriptions to several exit interfaces. Incompleteness of the subpath from the source broker to the destination broker may be corrected, but it has a high cost due to potential content-based flooding.

- Rendezvous point models with cyclic overlay routing support better coordination of mobility. With rendezvous points, advertisements are no longer flooded throughout the network, which improves update latency and performance. Moreover, rendezvous points may be used for fast completeness checks. The covering optimization may be used with completeness checking. Furthermore, the overlay address may be used to prevent content-based flooding by consulting the overlay routing tables and finding the proper next hop. On the other hand, the upper bound cost for cyclic topologies may be higher than for general acyclic graphs if the moving subscriber has subscribed to multiple rendezvous points that have to be updated.

- Rendezvous point models with acyclic overlay routing have the simplifying features mentioned above and the upper bound cost cannot be greater than for the general acyclic graphs.

Therefore we propose the following three techniques for improving mobility support in pub/sub systems: overlay-based routing, rendezvous points, and completeness checking. Overlay addresses prevent the content-based flooding problem. This abstracts the communication used by the pub/sub system from the underlying network-level routing and allows the system to cope with network-level routing errors and node failures. Rendezvous

Table 5.5: Optimizations for pub/sub handovers.

Optimization	Description
None	Flooding of update messages.
Overlay addresses	No flooding of update messages.
Rendezvous points	Completeness checking to RP.
Covering optimization	If update received (make-before-break) or complete (break-before-make).

points simplify mobility by allowing better coordination of topology updates. There is only one direction where to propagate updates for a single rendezvous point. Completeness checking ensures that subscriptions and advertisements are fully established (complete) in the topology. This is needed to perform the covering optimization. Table 5.5 presents a summary of the optimizations and how they may be applied for the two handover types: network-initiated make-before-break and break-before-make.

We give an outline of the subscriber handover algorithms both at the source (Algorithm 7) and destination (Algorithm 8). We assume that an overlay addressing system is used that allows each broker to deduce in which direction (outgoing interface) a given server is located. This prevents the flooding of control messages. The algorithms support both make-before-break and break-before-make. The algorithms take three parameters: C is a structure that stores information pertaining to the client, *src* is the identifier of the source broker, and *dst* is the identifier of the destination broker.

Algorithm 7 support different modes of operation. First, it supports the RP-based operation and checks the completeness to the RP using the *check-RP-CO* procedure. Second, if make-before-break is used, the algorithm uses the procedure *send-path-update* to send a path update towards the destination. Third, if the basic break-before-make protocol is used, the algorithm simply disconnects the client. Fast handovers are detected using the *waiting-for-update* procedure, which returns true if the given broker is waiting for the update message.

Algorithm 8 presents *move-in* functionality for these three modes of operation. The *skip* variable indicates whether or not the path between source and destination needs to be updated. If make-before-break is used, the algorithm checks if the update message has been received using the procedure *rcv-update*. If the update has been received and the state has been already transferred, the protocol may terminate. Otherwise, the al-

gorithm skips the topology update and uses the *send-transfer-request* to send a state transfer message to the source. If the update has not yet been received, the protocol has to wait until the message is received. The procedure *wait-for-update* adds the current client into a list of unfinished updates.

If RPs are used in Algorithm 8, the update message is sent only to the RP in question using the procedure *send-update-RP*. When receiving the update, the RP notifies the source that the path is complete. If RPs are not used, the update is sent to the source using the procedure *send-update*. For break-before-make, the procedure *sub-complete* is used to check if the relocated subscription is already complete at the destination.

Algorithm 7 The *move-out-sub* algorithm.

```
MOVE-OUT-SUB(C, src, dst)
 1  if WAITING-FOR-UPDATE(src,dst,C)
 2     then
 3          Fast handover detected, prevent transfer of session
 4          return
 5  if dst = ∅ and RPs are used
 6     then
 7          Wait until CHECK-RP-CO(src,dst,C) is true
 8  elseif dst ≠ ∅
 9     then SEND-PATH-UPDATE(src,dst,C)
10          ▷ Wait for response for client-initiated make-before-break
11  C.dst ← dst
12  C.src ← src
13  Disconnect client.
```

The publisher mobility algorithms are similar to the subscriber mobility algorithms. The *move-out-pub* algorithm follows Algorithm 7 with the exception that no state is transferred. If RPs are used, completeness to the RP is checked before relocation. If make-before-break is used, the update message is sent by the source broker. In this case, the path is updated by the message and only the direction from source to the destination needs to be tested.

The *move-in-pub* follows Algorithm 8. If it is known that the path is complete, the protocol terminates. If make-before-break is used, the reception of the update message indicates that the path is complete and

Algorithm 8 The *move-in-sub* algorithm.

```
MOVE-IN-SUB(C, src, dst)
 1  let skip = FALSE
 2  if C.dst = ∅ ▷ break-before-make
 3     then
 4          if SUB-COMPLETE(C)
 5             then skip ← TRUE
 6     else
 7          if RCV-UPDATE(src,dst,C)
 8             then skip ← TRUE
 9                  if state has already been transferred
10                     then return
11             else
12                  WAIT-FOR-UPDATE(src,dst,C) and return
13  if (¬skip)
14     then
15          ▷ Send update or out-of-band transfer request
16          ▷ Update will result in a state transfer
17          if RPs are used
18             then SEND-UPDATE-RP(src,dst,C)
19             else  SEND-UPDATE(src,dst,C)
20     else  SEND-TRANSFER-REQUEST(src,C)
```

the protocol terminates. If RPs are used, the protocol waits until the RP has confirmed that the path is complete. When the message from the RP arrives, the destination broker knows that any active subscriptions have been connected. If RPs and make-before-break are not used, the protocol sends an update message to the source broker. The protocol terminates when a reply is received from the source broker.

5.10 Summary

In this chapter we presented a discrete model for publish/subscribe mobility support. We examined the cost of pub/sub mobility using three mobility mechanisms and topologies: generic mobility support, acyclic graphs, and rendezvous-based topologies. We also discussed the impact of completeness and incompleteness of the pub/sub topology on the cost of mobility. We identified two important optimizations, overlay-based routing and rendezvous points. The generic mechanism has a high cost for mobility. The other two mobility mechanisms have a considerably smaller cost.

Mobility-safety cannot be guaranteed if protocols engineered with the completeness assumption are used for incomplete topologies. Based on both the theoretical model and the simulation results, we propose three techniques for improving mobility-aware pub/sub systems: overlay-based routing, rendezvous points, and completeness checking.

We presented a pub/sub mobility simulator and experimental results with the protocols and compared the theoretical upper bound costs with average costs from a simulation scenario. Based on the simulation experiment, rendezvous points offer significant performance benefits for mobility protocols. The acyclic protocols have lower cost and latency than the generic API ping/pong protocol. The lowest cost and latency were achieved using the complete acyclic protocol with a rendezvous point.

Part IV

Advanced Data Structures and Techniques

Chapter 6

DoubleForest for Temporal Subspace Matching

In this chapter we show how the forest data structure may be used to create more advanced structures for matching and comparing various profiles, such as interest or context profiles. Profiles are defined in a multi-dimensional content space using filters and they support both discrete and interval values. We present the DoubleForest and a graphical browser tool for visualization. The main application areas of the DoubleForest structure are profile and context-based matching and temporal content-based routing. We compare the results to a set-based algorithm.

6.1 Overview

We observed previously that most event and context processing systems do not feature optimized data structures for routing events and supporting dynamic profile-based queries. However, there are many optimized matchers for static queries. Support for rapid updates is required, because context descriptions and interests may change rapidly when circumstances change. In this chapter we address two central limitations of current event systems: temporal notifications and subspace queries. Temporal notifications (or profiles) have a duration instead of being instantaneous. The subspace-matching model follows the previously presented content-based routing model with the difference that notifications are defined as subspaces of the content space instead of as points. We use the term *profile* instead of notification to highlight the difference of the proposed model compared to the pub/sub model.

We represent both profiles and profile queries using generic filters. Our

notion of a filter is derived from event-based systems. The motivation for this is that by leveraging the properties of filters, namely *covering* and *overlapping* between them, we can optimize the data structures in a generic, filter-language-independent fashion, and also support more expressive subset selection and matching operations. Moreover, there are also techniques for performing *filter merging* to remove redundancy from filter sets by modifying them.

We propose that the poset-derived forest is used to store both profiles and queries based on the covering relation. The forest supports frequent updates to the data structure and it has approximately the same matching performance as the filters poset.

DoubleForest is a new data structure for managing profiles and queries based on covering relations. The assumption is that there are covering relations in both sets. This assumption is realistic for many scenarios, for example, covering is useful for geographic queries and profiles. Filter covering may be determined efficiently for simple predicate-based filters [31] and attribute filters with disjunctions [132]. Algorithms exist for arbitrary conjunctive filters [74], and also conjunctive tree queries [35].

We support two different matching operators for subspace matching, namely covering and overlapping. The proposed data structure combines these features and supports what we call *temporal subspace matching*. Matching semantics, such as *exact*, *subsume*, *intersection*, and *disjoint* have been previously proposed in the context of matchmaking with a Description Logic reasoner [80]. However, we are not aware of any optimized data structures for matching with these operators.

DoubleForest consists of two poset-derived forest data structures that have associated mappings from the elements of one structure to the other. We have demonstrated that the forest structure supports frequent data element insertions and removals. We present a new optimization technique for the DoubleForest, which involves the determination of upper and lower bounds. The bounds are used to inspect only the *boundary* — a set that contains the candidate nodes.

We define the *profile-matching problem* as follows given a set of profiles and a set of queries:

- for a new profile, find the set of associated (matching) queries,
- for a new query, find the set of associated (matching) profiles,
- when removing an existing profile, find the set of associated queries,
- when removing an existing query, find the set of associated profiles.

Figure 6.1 illustrates how two forests, based on filter sets P and Q, are combined in DoubleForest to support the matching of profiles and queries with mappings between the elements of each forest. The basic idea is to maintain mappings M_{PQ} and M_{QP} in different directions between the structures. M_{PQ} determines the set of *covering queries* given a profile, and similarly, M_{QP} determines the set of *covered profiles* given a query.

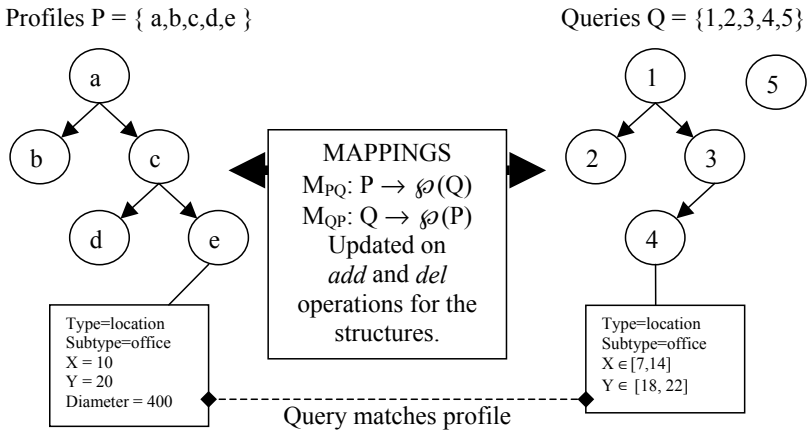

Figure 6.1: Storing and matching of profiles and queries.

The DoubleForest data structure is a building block for more complex routing and matching structures. The main application areas for the structure are the following:

- matching profiles and queries,
- change detection, i.e., finding the set of changed profiles and queries efficiently,
- content-based routing,
- distributed service directories with continuous queries.

We observe that the DoubleForest structure generates a taxonomy for both profiles and queries based on the covering relation. This automatic *taxonomy creation* is envisaged to be useful for applications, such as directory services and context or metadata-based directory browsing.

Context-based service directory querying and directory merging was investigated in [48]. This mechanism assumes that context data is represented using multi-dimensional discrete values. Each service category is

represented by a graph in which the leaves are the services and other nodes represent different context classifications. Distribution of the service directories is addressed by specifying a merging algorithm that merges the input directories. This system is similar to our method, but lacks the generality of covering and overlapping relations and the optimizations used by the forest data structures. Our approach also allows subspace queries. The proposed method generates the taxonomy automatically based on covering relations. Hence, the merging of two service directories is trivial using the DoubleForest.

We also note that filters may contain custom predicates, for example, semantic predicates such as the "IS-A" relation. In this case, the predicate would be introduced into the filtering language with the corresponding ontology. Thus the proposed data structure may be used with ontologies and external taxonomies.

6.2 Formal Definition

We define two sets, P and Q, which denote profiles and queries, respectively. P and Q are the base sets for the two forests. We define a *matching function* $\odot$ between the two forests that can have several different semantics. Given an element of one set, $\odot$ returns a subset of the other set. We use the matching function to formally define and determine the mappings M_{PQ} and M_{QP} that were introduced earlier. In this thesis, we define two semantics, namely covering, "$\sqsupseteq$", and overlapping "$\simeq$". Covering is more complicated, because it is not symmetric, whereas overlap is symmetric. Cover therefore requires two different functions, one for covering queries "$\sqsupseteq$" and one for covered profiles "$\sqsubseteq$". The four required matching functions are presented by Equations 6.1a, 6.1b, 6.1c, and 6.1d. In Equations 6.1a and 6.1c X is a profile ($X \in P$), and in Equations 6.1b and 6.1d X is a query ($X \in Q$).

$$\odot_{\sqsupseteq}(X) = \{y \in Q \mid y \sqsupseteq X\}, \tag{6.1a}$$

$$\odot_{\sqsubseteq}(X) = \{y \in P \mid X \sqsupseteq y\}, \tag{6.1b}$$

$$\odot_{\simeq,Q}(X) = \{y \in Q \mid X \simeq y\}, \tag{6.1c}$$

$$\odot_{\simeq,P}(X) = \{y \in P \mid X \simeq y\}. \tag{6.1d}$$

We define the $parent_F(X)$ function to return X's parent in the forest corresponding to the input set F. If X has no parent, the virtual root node of the forest is returned that has the root nodes of the forest as children. The function $children_F(X)$ is similarly defined, and simply returns the empty set if X has no children. Lemma 6.1 states the set containment

relationships of the matching sets determined using $\odot$ for two input filters A and B. Corollary 6.2 states that the matching function $\odot$ preserves covering relations for the two semantics.

Lemma 6.1 *Given filters $A, B \in P$ we have $A \sqsupseteq B \Rightarrow \odot_{\sqsupseteq}(A) \subseteq \odot_{\sqsupseteq}(B)$. Given that $A, B \in Q$ we have $A \sqsupseteq B \Rightarrow \odot_{\sqsubseteq}(B) \subseteq \odot_{\sqsubseteq}(A)$. Assuming $A, B \in Q$ without loss of generality, we also have $A \sqsupseteq B \Rightarrow \odot_{\simeq,P}(B) \subseteq \odot_{\simeq,P}(A)$.*

Proof. Assume the left side. We consider the three different cases.

1. "$\sqsupseteq$". It must be the case that each element in $\odot_{\sqsupseteq}(A)$ is contained in $\odot_{\sqsupseteq}(B)$, because when B has equal or lesser selectivity than A in P, $\odot_{\sqsupseteq}(B)$ has equal or greater selectivity in Q. Assume to the contrary that there exists an element, $e \in \odot_{\sqsupseteq}(A)$, for which $e \notin \odot_{\sqsupseteq}(B)$. It follows that A cannot have greater selectivity than B in P and therefore $A \sqsupseteq B$ cannot hold. This contradiction proves this case.

2. "$\sqsubseteq$". $\odot_{\sqsubseteq}(A)$ selects all covered nodes in P and it follows from the left side $A \sqsupseteq B$ that $\odot_{\sqsubseteq}(B)$ has equal or lesser selectivity. By assuming to the contrary that there exists an element $e \in \odot_{\sqsubseteq}(B)$ not in $\odot_{\sqsubseteq}(A)$, we have again a contradiction to the assumption. It follows that $\odot_{\sqsubseteq}(B) \subseteq \odot_{\sqsubseteq}(A)$.

3. "$\simeq$". This case is similar to the previous cases. It must be the case that those elements that overlap with B in P must be elements in the overlap of A in P, because $A \sqsupseteq B$. Assume to the contrary that there exists an element $e \in \odot_{\simeq,P}(B)$ but $e \notin \odot_{\simeq,P}(A)$, but now it cannot be the case that $A \sqsupseteq B$, which contradicts the assumption. □

Corollary 6.2 *Given filters $A, B \in P$ we have $A \sqsupseteq B \Rightarrow \odot_{\sqsupseteq}(B) \sqsupseteq \odot_{\sqsupseteq}(A)$. Given that $A, B \in Q$ we have $A \sqsupseteq B \Rightarrow \odot_{\sqsubseteq}(A) \sqsupseteq \odot_{\sqsubseteq}(B)$. Assuming without loss of generality that $A, B \in Q$ also $A \sqsupseteq B \Rightarrow \odot_{\simeq,P}(A) \sqsupseteq \odot_{\simeq,P}(B)$.*

Proof. Assume the left side. Applying Lemma 6.1 in each case we have "$\sqsupseteq$": $\odot_{\sqsupseteq}(A) \subseteq \odot_{\sqsupseteq}(B)$, "$\sqsubseteq$": $\odot_{\sqsubseteq}(B) \subseteq \odot_{\sqsubseteq}(A)$, and "$\simeq$": $\odot_{\simeq,P}(B) \subseteq \odot_{\simeq,P}(A)$. It follows from the definition of the covering relation that "$\sqsupseteq$": $\odot_{\sqsupseteq}(B) \sqsupseteq \odot_{\sqsupseteq}(A)$, "$\sqsubseteq$": $\odot_{\sqsubseteq}(A) \sqsupseteq \odot_{\sqsubseteq}(B)$, and "$\simeq$": $\odot_{\simeq,P}(A) \sqsupseteq \odot_{\simeq,P}(B)$. □

The data structure provides two operations for adding and removing elements, namely the add_{DF} and del_{DF} operations. Algorithm 9 presents the add_{DF} operation in more detail and Algorithm 10 the del_{DF} operation. $\prec$ denotes the semantics used by $\odot$, F denotes the set P or Q, and I denotes the interface or identifier of the object in question.

In algorithm 9 the result sets $M_{PQ}(X)$ and $M_{QP}(X)$ need to be computed efficiently. Initially no relations are known for X in Q and P. When adding a profile X, a simple algorithm adds X to the set P and then tests it against each element in Q for covering or overlapping. The idea of the DF structure is to utilize the two forests in this computation. In subsequent sections we discuss a number of optimizations for determining the $M_{PQ}(X)$ and $M_{QP}(X)$ sets.

Algorithm 9 The add_{DF} algorithm.

$add_{DF}(\prec, X, F, I)$

1. *add(F,X,I)* (add X to forest identified by F).
2. If $F = P$ then
3. $M_{PQ}(X)$ = result set for X (computed using $\odot$).
4. Update $\forall y \in M_{PQ}(X) : M_{QP}(y) \leftarrow M_{QP}(y) \cup \{X\}$.
5. Else if $F = Q$ then
6. $M_{QP}(X)$ = result set for X (computed using $\odot$).
7. Update $\forall y \in M_{QP}(X) : M_{PQ}(y) \leftarrow M_{PQ}(y) \cup \{X\}$.

6.3 Determining the Result Set Efficiently

Algorithm 9 requires that the result set is computed using $\odot$. The computation of this set needs to be optimized. One immediate optimization is to traverse the data structure in question starting from the root nodes and traversing to only matching nodes. This optimization was proposed in the Siena filters poset data structure that was used for event routing. The idea for this optimization is presented in Algorithm 6.

For the DoubleForest, we need to consider four different cases, the addition of a profile or a query for the two semantics. The case of $\sqsupseteq$ is more complicated than $\simeq$. The former case is more difficult, because the mappings between the two forests are not symmetric.

"$\sqsupseteq$" **1. Add profile** All elements that cover the input profile in Q are in the result set determined by $\odot$. The forest defined by the set Q is traversed starting from the root nodes towards covering filters.

Algorithm 10 The del_{DF} algorithm.

$del_{DF}(\prec, X, F, I)$

1. If $F = P$ then
2. $M_{PQ}(X)$ = remove result set for X (from hashtable).
3. Update $\forall y \in M_{PQ}(X) : M_{QP}(y) \leftarrow M_{QP}(y) \setminus \{X\}$.
4. Else if $F = Q$ then
5. $M_{QP}(X)$ = remove result set for X (from hashtable).
6. Update $\forall y \in M_{QP}(X) : M_{PQ}(y) \leftarrow M_{PQ}(y) \setminus \{X\}$.
7. *del(F,X,I)* (remove from poset-derived forest identified by F).

2. Add query All covered elements in P are in the result set determined by $\odot$. The forest defined by the set P is traversed starting from the root nodes. If an element covered by the input filter is detected, the traversal of that subtree is stopped, and the current node and all the descendants of the current node are added to the result set.

"$\simeq$" **3. Add profile** All elements that overlap with the input profile in Q are in the result set determined by $\odot$. The forest defined by the set Q is traversed starting from the root nodes towards overlapping filters.

4. Add query All overlapping elements in P are in the result set determined by $\odot$. The forest defined by the set P is traversed starting from the root nodes towards overlapping filters.

Another immediate optimization is to *partition* filters into disjoint sets, for example, by their type. Type-based partitioning is easily implemented by using a hashtable. In this case, each type has an associated set that contains the root filters of that type. A hashtable is then used to quickly lookup the set corresponding to the given type.

6.4 Optimization using Upper and Lower Bounds

The basic optimizations presented in the previous section can be extended to further reduce the computation cost of the result set. These optimiza-

tions pertain to restricting the set of tested filters by finding a candidate set. The upper and lower bound sets in the other forest are used to find the parts in the data structure where potential matching nodes are located. In essence, this allows to exclude parts of the forest from examination.

Figure 6.2 illustrates the case when profiles are added to P and the covering queries need to be located in Q. The node x has been added as a direct successor to p. Node x has a parent and two children. The mapping from x's parent provides the lower bound — those filters that must be in the result set. The lower bound contains all covering root queries if such exist. The mappings from x's children provide the candidate set — those nodes that may be in the result set and must be tested.

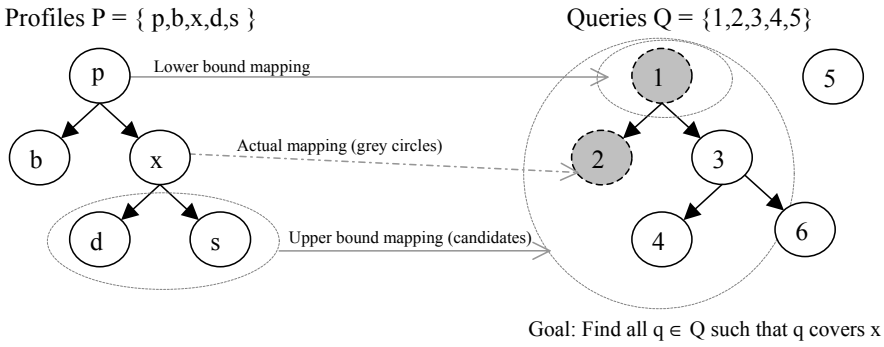

Figure 6.2: Using upper and lower bound optimizations when profiles are added.

Figure 6.3 illustrates the case when a query is added to Q and the covered profiles need to be located in P. In this case, the mapping from x's parent provides the upper bound set — the candidate set of nodes that must be tested for inclusion into the result set. The mappings from x's children provide the lower bound mapping — the set that must be contained in the result set. Nodes that are not in the former set are not tested for inclusion.

The following list presents the upper and lower bound optimizations for adding either a profile or query with the two semantics.

"$\sqsupseteq$" **1. Add profile** All elements that cover the input profile in Q are in the result set determined by $\odot$.
Upper-bound optimization: Since elements in $\odot_{\sqsupseteq}(X)$ are covered by elements in $\odot_{\sqsupseteq}(children_P(X))$ by Corollary 6.2, elements in this set are tested for inclusion in the result set. Only elements in this set are evaluated.

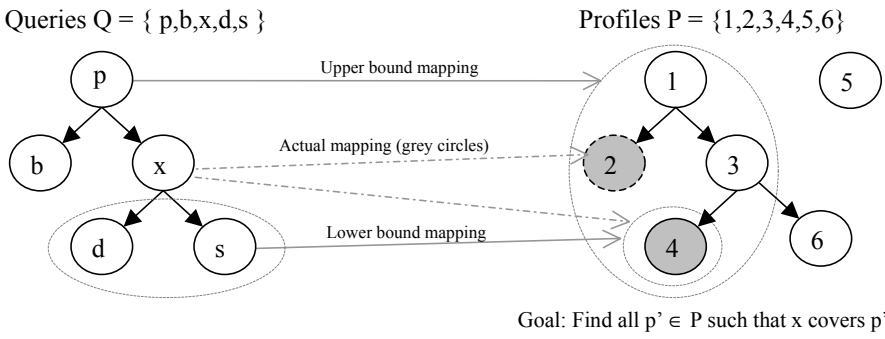

Figure 6.3: Using upper and lower bound optimizations when queries are added.

Lower-bound optimization: Elements in $\odot_{\sqsupseteq}(parent_P(X))$ are contained in the results set. These elements are added to the results set without testing them.

2. **Add query** All covered elements in P are in the result set determined by $\odot$.
   Upper-bound optimization: Since elements in $\odot_{\sqsubseteq}(parent_Q(X))$ cover elements in $\odot_{\sqsubseteq}(X)$ by Corollary 6.2, they are tested for inclusion in the result set.
   Lower-bound optimization: Descent into the forest defined by Q is stopped when an element in $\odot_{\sqsubseteq}(children_Q(X))$ is detected. Descendants of elements in this set are covered by the input filter (Corollary 6.2).

"$\simeq$" 3. **Add profile** All elements that overlap with the input profile in Q are in the result set determined by $\odot$.
   Upper-bound optimization: $\odot_{\simeq,Q}(parent_P(X))$ covers $\odot_{\simeq,Q}(X)$ by Corollary 6.2. Elements in this set are tested for inclusion in the result set.
   Lower-bound optimization: Elements in $\odot_{\simeq,Q}(children_P(X))$ are contained in the results set. These elements are added to the results set without testing them.

4. **Add query** All overlapping elements in P are in the result set determined by $\odot$.
   Upper-bound optimization: $\odot_{\simeq,P}(parent_Q(X))$ covers $\odot_{\simeq,P}(X)$ by Corollary 6.2. Elements in this set are tested for inclusion in the result set.

Table 6.1: Optimizations for the two semantics.

Description		Optimizations
Upper bounds	"$\sqsupseteq$". $X \in P$	$\odot_{\sqsupseteq}(X) \subseteq \odot_{\sqsupseteq}(children_P(X))$
	"$\sqsubseteq$". $X \in Q$	$\odot_{\sqsubseteq}(X) \subseteq \odot_{\sqsubseteq}(parent_Q(X))$
	"$\simeq$". $X \in P$	$\odot_{\simeq,Q}(X) \subseteq \odot_{\simeq,Q}(parent_P(X))$
	"$\simeq$". $X \in Q$	$\odot_{\simeq,P}(X) \subseteq \odot_{\simeq,P}(parent_Q(X))$
Lower bounds	"$\sqsupseteq$". $X \in P$	$\odot_{\sqsupseteq}(parent_P(X)) \subseteq \odot_{\sqsupseteq}(X)$
	"$\sqsubseteq$". $X \in Q$	$\odot_{\sqsubseteq}(children_Q(X)) \subseteq \odot_{\sqsubseteq}(X)$
	"$\simeq$". $X \in P$	$\odot_{\simeq,Q}(children_P(X)) \subseteq \odot_{\simeq,Q}(X)$
	"$\simeq$". $X \in Q$	$\odot_{\simeq,P}(children_Q(X)) \subseteq \odot_{\simeq,P}(X)$

Lower-bound optimization: Elements in $\odot_{\simeq,P}(children_Q(X))$ are contained in the results set. These elements are added to the results set without testing them.

Table 6.1 summarizes the upper and lower bound formulas that are used in optimizing matching in the DoubleForest. We have to consider two special cases. First, if X has no parent, the virtual root node is returned by $parent_F$, which maps to the full node set. Second, if X has no children, the bounds that require this information cannot be determined.

Algorithm 11 presents the optimized matching algorithm for the covering semantics. The algorithm finds queries matching the given profile X when $Op =' \sqsupseteq'$ is given. When $Op =' \sqsubseteq'$ is given, the algorithm finds profiles matching the given query X.

Algorithm 12 presents the matching algorithm for the overlapping semantics. The algorithm finds all elements in the current base set that overlap with the given X. The upper and lower bounds are used to optimize this procedure.

Let U denote the upper bound and L the lower bound. Then $\Delta = U \setminus L$. If U exists and is empty, the matching terminates, otherwise the corresponding algorithm is used. We note when adding a profile it is sufficient to iterate only the Δ set if it exists. If it cannot be computed then Algorithms 11 or 12 must be used. This optimization is used in our implementation of the data structure.

Algorithm 11 Pseudocode for matching with the DoubleForest with $\sqsupseteq$ semantics.

```
MATCH-DOUBLEFOREST(X, Op, upper, lower)
 1  let S be an empty sequence and RS an initially empty set
 2
 3  R = GET-ROOTS(X.type)
 4  let Im be an imaginary root of a tree
 5  Im.children = R
 6  ADDLAST(S, Im)
 7
 8  while S is non-empty
 9        do
10           o = REMOVEFIRST(S)
11           while o has unprocessed children
12                 do
13                    let f_i, f_m be Boolean flags (initially FALSE)
14                    c = NEXTCHILD(o)
15                    if lower ≠ null and c ∈ lower
16                       then f_m = TRUE
17                    elseif upper ≠ null and c ∉ upper
18                       then
19                             ▷ continue descent if query to profile
20                             if Op = ' ⊑'
21                                then f_i = TRUE
22                    elseif Op = ' ⊑'
23                       then
24                             ▷ query to profile
25                             if X ⊒ c
26                                then f_m = TRUE
27                                else  f_i = TRUE
28                    elseif Op = ' ⊒' and c ⊒ X
29                       then
30                             ▷ profile to query
31                             f_m = TRUE
32                    DODESCENT(Op, RS, S, c, o, f_m, f_i)
33 return RS
```

Algorithm 12 Pseudocode for matching with the DoubleForest with $\simeq$ semantics.

```
MATCH-DOUBLEFOREST-OVERLAP(X, upper, lower)
 1  let S be an empty sequence and RS an initially empty set
 2
 3  R = GET-ROOTS(X.type)
 4  let Im be an imaginary root of a tree
 5  Im.children = R
 6  ADDLAST(S, Im)
 7
 8  while S is non-empty
 9        do
10           o = REMOVEFIRST(S)
11           while o has unprocessed children
12                 do
13                    let f_i and f_m be Boolean flags (initially FALSE)
14                    c = NEXTCHILD(o)
15                    if upper ≠ null and c ∉ upper
16                       then Do Nothing
17                    elseif lower ≠ null and c ∈ lower
18                       then f_m = TRUE
19                    elseif X ≃ c
20                       then f_m = TRUE
21                    DODESCENT(≃, RS, S, c, o, f_m, f_i)
22 return RS
```

Algorithm 13 Pseudocode for the auxiliary doDescent function.

```
doDescent(Op, RS, S, c, o, f_m, f_i)
 1  if f_m
 2     then
 3          if Op = '⊑'
 4             then
 5                  ▷ query to profile
 6                  addToSet(RS, c)
 7                  addDescendantsToSet(RS, c)
 8                  return ▷ stop descent
 9             else
10                  addToSet(RS, c)
11  if f_m or f_i
12     then addLast(S, c)
```

6.5 Correctness

Correctness of the DoubleForest data structure follows from the correctness of the result set functions. For *del*, it is sufficient that the *add* operation is correct and the mappings have been updated properly. The correctness of the upper and lower bound optimization requires further examination.

Theorem 6.3 *Assuming* $children_P(X) \neq \emptyset$*, the upper bounds are correct:*
1. $X \in P$ *and* $\odot_{\sqsupseteq}(X) \subseteq \odot_{\sqsupseteq}(children_P(X))$.
2. $X \in Q$ *and* $\odot_{\sqsubseteq}(X) \subseteq \odot_{\sqsubseteq}(parent_Q(X))$.
3. $X \in P$ *and* $\odot_{\simeq,Q}(X) \subseteq \odot_{\simeq,Q}(parent_P(X))$.
4. $X \in Q$ *and* $\odot_{\simeq,P}(X) \subseteq \odot_{\simeq,P}(parent_Q(X))$.

Proof. We consider the four cases and show that there cannot exist an element in the set returned by $\odot$ for X that is not in the right-side upper bound set. We observe that cases 3. and 4. are symmetrical. As previously discussed $parent_F(X)$ is never empty.

1. It holds that for any child A of X we have $X \sqsupseteq A$. From Lemma 6.1 it follows that $\odot_{\sqsupseteq}(X) \subseteq \odot_{\sqsupseteq}(A)$. By taking union over all X's children we have $\odot_{\sqsupseteq}(X) \subseteq \odot_{\sqsupseteq}(children_P(X))$.

2. It holds that for a parent A of X we have $A \sqsupseteq X$. By Lemma 6.1 we have $\odot_{\sqsubseteq}(X) \subseteq \odot_{\sqsubseteq}(A)$.

3. This case is similar to the previous cases. Again, it holds that for any parent A of X we have $A \sqsupseteq X$. By Lemma 6.1 we have $\odot_{\simeq,P}(X) \subseteq \odot_{\simeq,P}(A)$. □

Theorem 6.4 *Assuming* $children_P(X) \neq \emptyset$*, the lower bounds are correct:*
1. $X \in P$ *and* $\odot_{\sqsupseteq}(parent_P(X)) \subseteq \odot_{\sqsupseteq}(X)$.
2. $X \in Q$ *and* $\odot_{\sqsubseteq}(children_Q(X)) \subseteq \odot_{\sqsubseteq}(X)$.
3. $X \in P$ *and* $\odot_{\simeq,Q}(children_P(X)) \subseteq \odot_{\simeq,Q}(X)$.
4. $X \in Q$ *and* $\odot_{\simeq,P}(children_Q(X)) \subseteq \odot_{\simeq,P}(X)$.

Proof. We consider the four cases and show that there cannot exist an element in the left-side lower bound set that is not in the set returned by $\odot$ for X. We observe that cases 3. and 4. are symmetrical. As previously discussed $parent_F(X)$ is never empty.

1. It holds that for a parent A of X, $A \sqsupseteq X$. Hence from Lemma 6.1 $\odot_{\sqsupseteq}(A) \subseteq \odot_{\sqsupseteq}(X)$.

2. It holds that for any child A of X, $X \sqsupseteq A$. Hence from Lemma 6.1 $\odot_{\sqsubseteq}(A) \subseteq \odot_{\sqsubseteq}(X)$.

3. This case is similar to the two other cases. It holds that for any child A of X, $X \sqsupseteq A$. From Lemma 6.1 we have $\odot_{\simeq,P}(A) \subseteq \odot_{\simeq,P}(X)$. □

Theorem 6.5 *The DoubleForest structure with upper and lower bound optimizations is correct.*

Proof sketch: We establish that the optimizations narrow down the candidate set for $\odot$ and cannot exclude elements from the results set. The upper bound sets are candidate sets and cannot exclude elements (Theorem 6.3). The lower bound sets are contained in the result set and cannot exclude elements (Theorem 6.4). It follows that the structure with optimizations is correct. □

6.6 Computational Complexity

In DoubleForest, each node in P maps to at most $|Q|$ nodes, and similarly, each node in Q maps to at most $|P|$ nodes. This gives a space complexity of $O(|P| \cdot |Q|)$ data entries in the structure in the worst case.

The average time complexity for the structure depends on the underlying data set. The worst-case happens for linear orders. If we consider the matching of a query to profiles, we have $O(|Q|)$ insertion cost of the query, $O(|P|)$ cost of finding the matching profiles. This results in linear complexity $O(|Q| + |P|)$.

6.7 Temporal Subspace Matching

The DoubleForest is used to perform temporal subspace matching. The query forest supports continuous queries. Similarly, the profile forest supports profiles that persist in time. The system supports subspace matching using the two specified operators, namely covering and overlapping.

Figure 6.4 presents an example of temporal subspace matching. In the figure, we have five range queries and five range profiles. The mappings are presented below the forests.

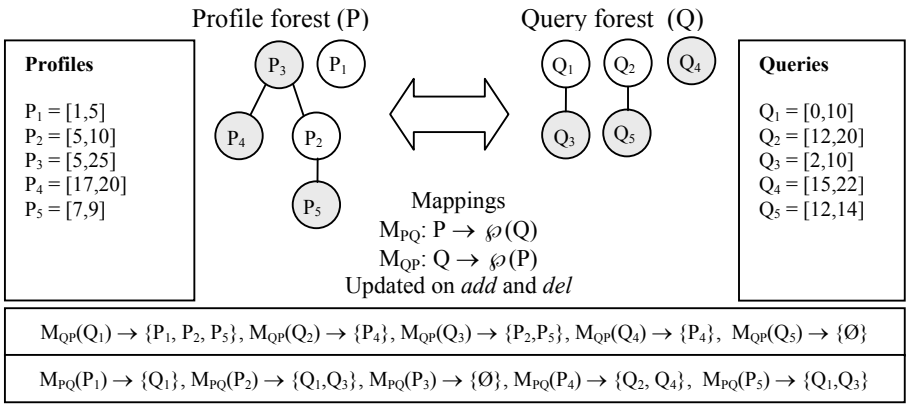

Figure 6.4: DoubleForest with range queries and profiles.

6.8 Experimentation

6.8.1 Overview

We experimented with the *add* scenario presented in Chapter 4. We used the same workload generator and parameters, namely 2-dimensional range filters. Only a single type is used, because this is the worst-case scenario for performance, because the type cannot be used for optimization. The scenario differs from the previous benchmark scenario, because interfaces are not taken into account.

We use the redundant non-balanced poset-derived forest implementation. This means that each added filter is retained in the structure and interface elimination has no effect. Figure 6.5 presents an overview of the benchmark. The workload generator is used to create two sets of filters, the

queries and profiles, which are then inserted into the data structure. Each measurement is replicated 5 times and the standard deviation is shown in the diagrams.

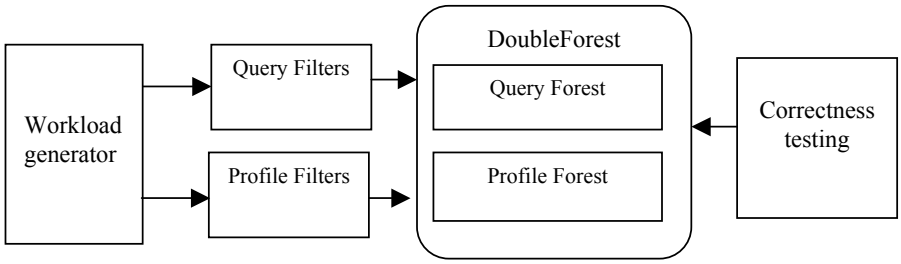

Figure 6.5: DoubleForest benchmark overview.

We developed the following two benchmarks for both $\sqsupseteq$ and $\simeq$ scenarios:

- A variable number of profiles and a constant number of queries.
- A variable number of queries and a constant number of profiles.
- A variable number of both queries and profiles. Queries and profiles are added in random order.

The purpose of these benchmarks is to measure the cost in both *computation time* and *the number of covering operations* of the addition of a new profile or a query, to understand the impact of the two different semantics for performance, and to understand the impact of the optimizations presented in Section 6.4. Three different cases are measured for each scenario:

- unoptimized DoubleForest,
- optimized DoubleForest (upper and lower bound optimizations), and
- set-based operation.

The last case represents the trivial implementation method and serves as a general performance benchmark. The set-based case is also used to ensure that the DoubleForest operates correctly. It should be noted that the set-based operation finds the correct mappings, but does not determine other covering relations between the input filters. This means that the set-based mechanism cannot be used for content-based routing without modifying the algorithm to find the root set and the relations between the filters.

We focus on adding filters using the add_{DF} operation to the DoubleForest structure, because the del_{DF} operation only involves the *add* operation for the poset-derived forest in question, which was shown to have good performance in Chapter 4. In this sense, the *add* used in del_{DF} has only local implications, whereas the add_{DF} has global implications, because the mappings may change. Therefore, it is not necessary to consider the del_{DF} operation in order to understand the performance of the two poset-derived forests combined into a DoubleForest structure. Furthermore, the del_{DF} operation is not necessary for those filters that are frequent in the workload as the results with preloading in Chapter 7 indicate.

6.8.2 Results

$\sqsupseteq$ Semantics. Figure 6.6 shows the results for a variable number of profiles and queries for the range queries with two attribute filters. In each experiment either the number of profiles or queries is constant (1000 filters) while the other is variable. The top diagrams show the number of covering operations and the bottom diagrams show the computation time. The optimized DoubleForest has the best performance both in terms of operations and time.

$\simeq$ Semantics. Figure 6.7 shows the results for a variable number of profiles and queries for the range queries with two attribute filters. In each experiment either the number of profiles or queries is constant (1000 filters) while the other is variable. The optimized DoubleForest is more efficient than the set-based benchmark algorithm, but the difference is still small. The results indicate that overlap is more difficult to optimize using the forest-based mechanism that derives its structure from the covering relation.

Adding both Profiles and Queries. Figure 6.8 presents the results for both variable profiles and queries. In this case, both sets are combined into a larger set and then elements are randomly inserted to the data structure. We used 1000 queries and profiles for a total structure size of 2000 filters. The results indicate that the DoubleForest has considerably better performance in this scenario for cover ("$\sqsupseteq$"), but the performance for overlap ("$\simeq$") is better than the set-based benchmark case, but not very good.

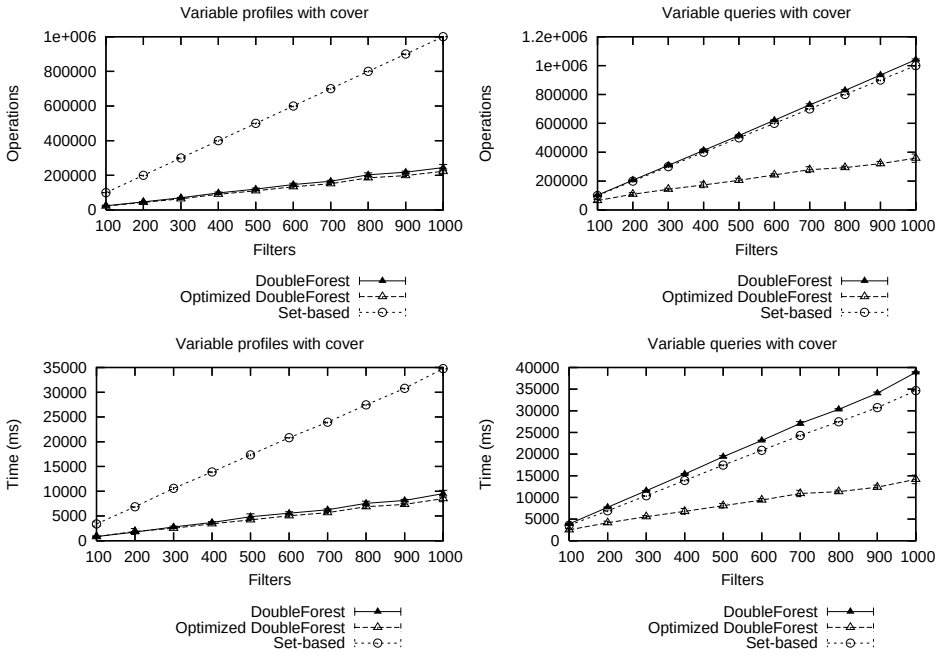

Figure 6.6: DoubleForest cover for 2D range filters.

6.8.3 Context Browser

In order to experiment with the DoubleForest data structure, we developed a graphical browser tool, called *ContextBrowser*[1], which allows users to inspect the data structure. Context-matching is one of the application areas of the data structure, which has motivated the name of the tool. Figure 6.9 presents the user interface of the program. The two forests are displayed and the mappings between the forests are shown in tables. The tool also tests that the mappings are correct using the set-based algorithm. The filter set may be changed and the number of filters in the structure can also be adjusted.

ContextBrowser demonstrates the use of the structure in real-time collection synchronization by defining the queries and profiles and then updating a query-defined collection as profiles are added. We investigate collection synchronization in more detail in Chapter 9.

[1]Available at www.hiit.fi/fuego/fc/demos

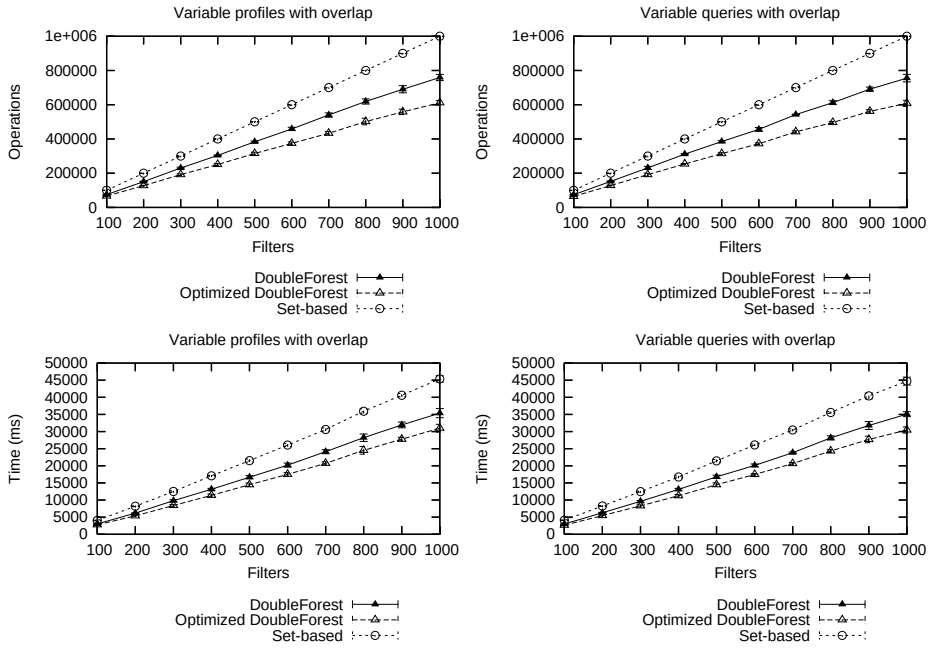

Figure 6.7: DoubleForest overlap results for 2D range filters.

6.9 Related Work

Many different indexing algorithms have been developed for matching [14]. The presented work differs from the majority of the current matching algorithms, because it relies only on the covering relation. R-tree [61] and its variants, such as R* and R+-tree, are efficient structures for determining containment, intersection, and overlapping of multi-dimensional data, such as geographical data. They require that minimum bounding rectangles can be computed for the nodes in the trees.

Navigating nets [76] are leveled directed acyclic graphs, in which multiple paths may exist from the root to a lower-level node. The main application is nearest neighbour search for points and range queries. Each consequent level covers the data placed on the lower level and the levels are connected using pointers to allow navigation between scales. This is similar in principle to the filters poset algorithm, but the navigating net algorithm assumes metric spaces and is only applicable for points and ranges.

A *cover tree* [11] is a leveled tree, in which each level covers the level under it. The main application of cover trees is nearest neighbour search

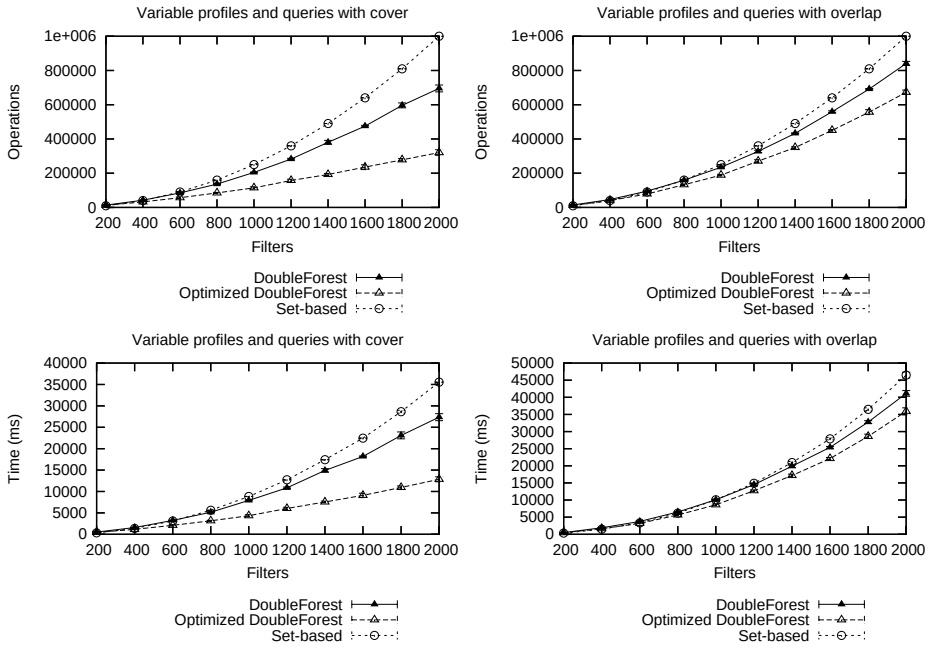

Figure 6.8: DoubleForest results for variable profiles and queries for 2D range filters.

for points. The poset-derived forest is similar in idea to the cover tree, but the underlying assumptions and expressiveness are fundamentally different. The cover tree assumes metric spaces.

Transitive closure [40, 147, 148] is a vital operation for database systems, query systems [36], information management systems and ontologies [2], and most recently context-aware systems [3]. There are many algorithms for transitive closure with different assumptions and performance. Typically, transitive closure algorithms are static and designed for database operations. In this case, efficient disk I/O is important, which has lead to a number of *directed acyclic graph* (DAG) based optimization techniques [98], for example, *range compression* [2] and hybrid algorithms [3]. In addition to static algorithms, several dynamic transitive closure algorithms have been proposed in recent years [75].

The DoubleForest differs from previous transitive closure algorithms, because we focus on a fully dynamic algorithm for binary relations, DAGs, and the edge set introduced by a new node is not known. Moreover, our matching problem is based on two separate sets: the queries and the pro-

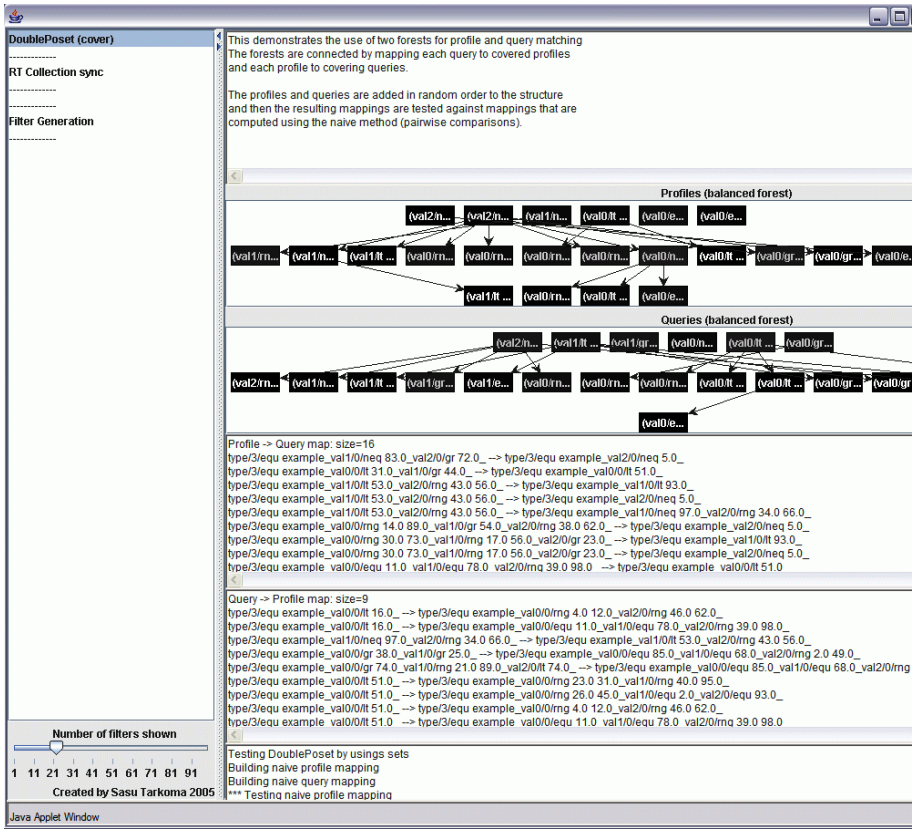

Figure 6.9: The user interface of the ContextBrowser.

files.

The main differences of the poset-derived forest to prior work is that indexing algorithms, such as R-trees, assume metric spaces and that bounding rectangles may be determined. The proposed mechanism only assumes the covering relations, which makes it suitable for *mixed queries* that combine, for example, strings and integers. We are not aware of any data structures for the efficient cover-based matching, what we call subspace matching, of arbitrary filters. A graph-structure approach for matching simple content was proposed in [77]. In this approach, the content is represented by predefined hierarchical categories. The system was optimized by computing the transitive closure for each element in the graph, which allows constant time queries for the predefined types. The algorithm for computing the closure was not elaborated.

6.10 Summary

In this chapter, we have presented a new data structure for filter-based matching with emphasis on temporal operation and content defined using subspaces. We use the properties of filters, namely covering, to optimize the data structure. This strategy requires that covering may be efficiently determined for the input filter set. We gave a formal definition of the DoubleForest structure and proofs for its correctness. Experimental results with range queries, which represent input workload with a moderate amount of covering relations, indicate that the proposed optimizations are useful, especially for cover-based matching. However, overlap-based matching was observed to have more overhead than cover-based matching.

The envisaged application areas of the structure are context-aware systems, information routing, information exchange in peer-to-peer systems, change-set computation, and matching profiles and queries. We noted that the data structure automatically computes a taxonomy for both profiles and queries, which is envisaged to be useful for directory services.

Chapter 7

Constant-time Subspace Matching with Preloading

In this chapter, we present a constant-time technique for temporal subspace matching by extending the DoubleForest with full preloading. This technique supports constant-time insertions and deletions for those queries and profiles that have been preloaded. This technique requires *a priori* knowledge of the filters.

7.1 Preloading

The performance of the DoubleForest data structure may be enhanced by preloading queries and profiles in advance. A special dummy identifier is associated with the preloaded objects to indicate that they are placeholders.

The insertion and deletion of an object to the data structure that has been loaded previously is a constant-time operation. Theorem 7.1 establishes that the mappings are determined in constant-time.

Theorem 7.1 *The cost of computing the mappings for a preloaded object is O(1).*

Proof. The mappings have been computed beforehand for a preloaded query or profile. Hence, for an insert or delete operation, the mappings are retrieved in O(1) time using a hashtable. □

If all or part of the profiles and queries are known beforehand, preloading may be used to significantly improve temporal subspace matching performance. The optimizations for DoubleForest indicate that preloading may be performed efficiently. Since the DoubleForest functions irrespec-

tive of preloading, the system degrades well for those objects that were not preloaded.

7.2 Experimentation

We have experimented with two benchmark scenarios with preloading. The benchmark workload is identical to the 2-dimensional range workload discussed in Section 6.8, and both profiles and queries are inserted into the structure. The evaluated data structures are: DoubleForest, optimized DoubleForest, and the set-based algorithm. The set-based algorithm is not preloaded. Each measurement was replicated 5 times.

In the first preload benchmark, we preload a variable percentage of profiles and queries. Figure 7.1 presents this scenario and illustrates the preload set creation. The preloaded set is a randomly selected subset of the input set. The size of this preload set is determined by the preload percentage.

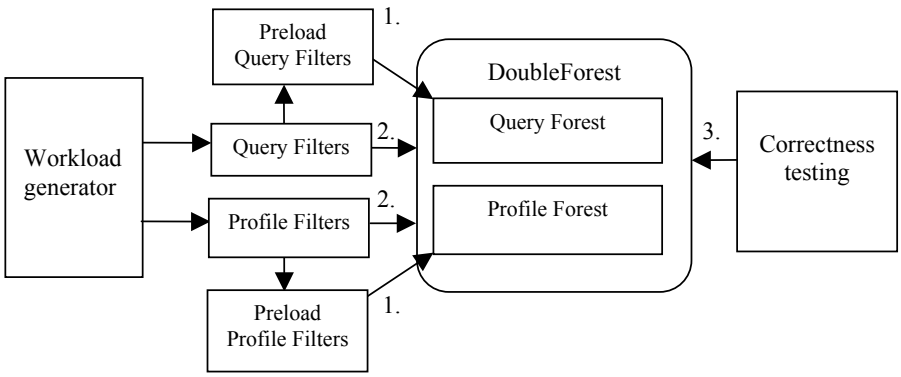

Figure 7.1: The full preload benchmark.

It may also be the case that preloading does not succeed and the preloaded objects represent additional overhead. We have also experimented with this scenario, in which the preloaded sets are randomly generated and not subsets of the input sets. This happens when the preloading fails.

Preloading. In each experiment the size of the query and profile sets is constant (1000 filters) and the preload level is variable. The results indicate that preloading is very effective and with 100% preloading, constant

matching time was achieved. In this optimum case, the number of covering operations required was zero. Figure 7.2 presents the results when both queries and profiles are preloaded. After preloading they are inserted into the structure in random order. Preloading is effective in this scenario for both cover and overlap.

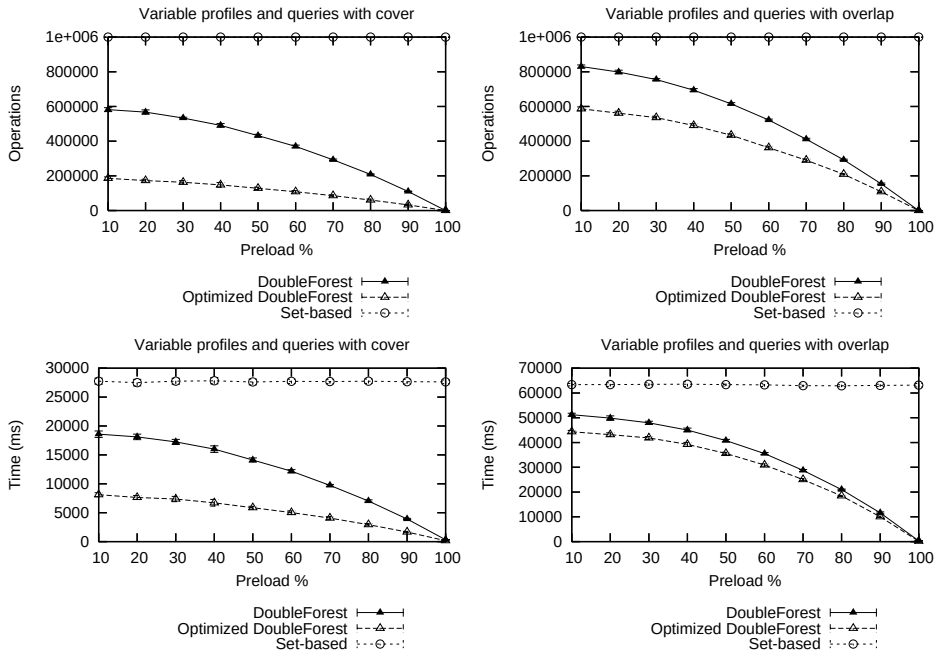

Figure 7.2: Inserting both queries and profiles with a variable preload level.

Preloading Random Filters. The random preload scenario is similar to the previous benchmark scenario, but in this case the preloaded sets are independent of the queries and profiles and are randomly generated for each replication. Figure 7.3 presents the results when random preload is performed for both queries and profiles. After preloading they are inserted into the structure in random order. Random preloading does not introduce significant overhead for covering when compared with the set-based algorithm that does not use preloading. However, the results indicate significant overhead for overlap-based matching.

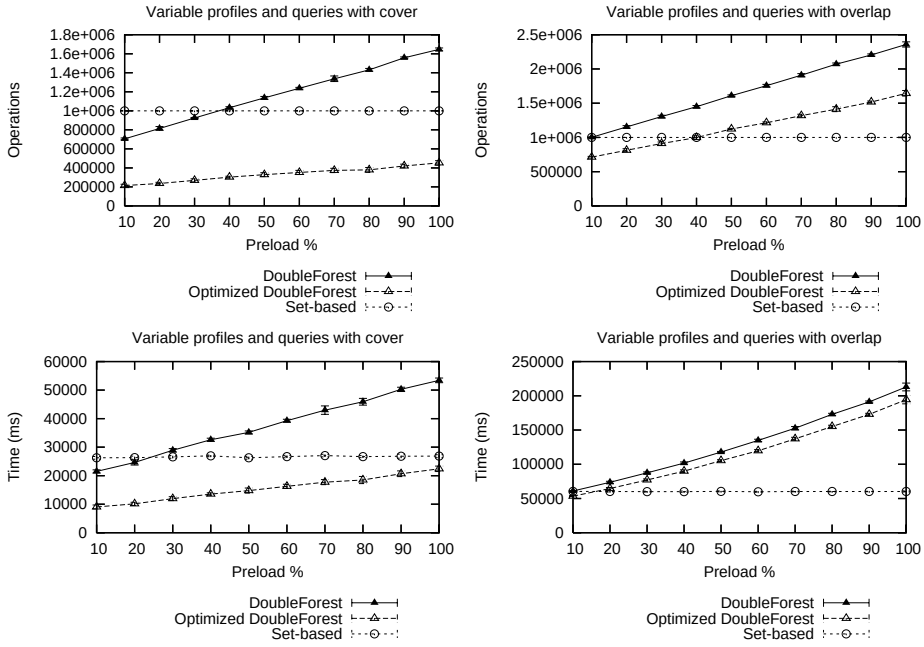

Figure 7.3: Inserting both queries and profiles with a variable random preload level.

Chapter 8

Filter Merging

In this chapter we present techniques for incorporating filter merging into content-based routers in a transparent fashion. The techniques are independent of the filtering language and routing data structure used, and do not depend on the mechanism that is used to merge two input filters. We present two distinct ways to merge filters: local merging and remote merging. In the former, merged filters are placed into the data structure. In the latter, merged filters are stored separately from the data structure and only for exiting (outgoing) routing table entries.

8.1 Overview

Filter merging is potentially useful for event routers, because it allows to remove redundancy from filter sets. Figure 8.1 illustrates the benefits of filter merging. First, we have a set of four filters. There are no covering relations in this set, so they need to be propagated by the current router. The filters may be merged by applying an existence test (top filters) and combining two ranges (lower filters), resulting in two merged filters. There are covering relations between the merged filters, and only the top filter is propagated. In this example, the size of the propagated set was reduced from four filters to one filter.

8.2 Merging and Routing Tables

We propose a merging extension to the generic content-based routing table. The desired characteristics for this merging mechanism are simplicity in implementation, efficiency, and minimal requirements for the underlying routing table. We assume that a *merge(F_1,F_2)* procedure exists that merges

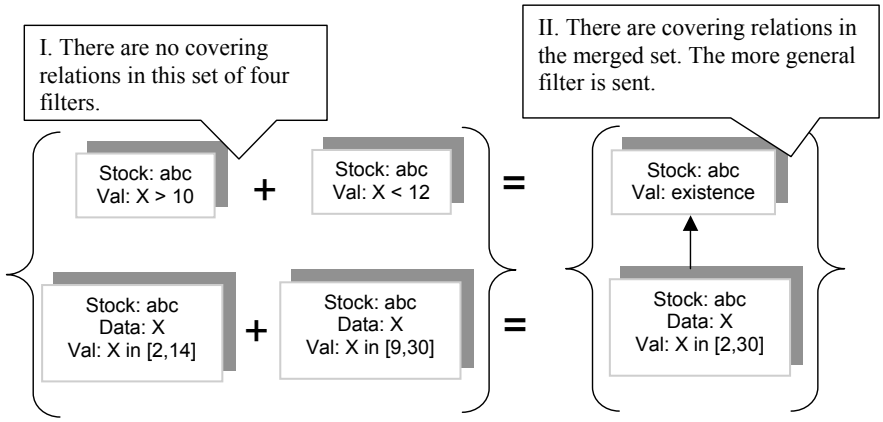

Figure 8.1: Example of filter merging.

input filters F_1 and F_2 and returns a single merged filter F_M for which $F_M \sqsupseteq F_1$ and $F_M \sqsupseteq F_2$. A merge of two or more filters is called a *merger*. Filter merging is useful, because it allows to further remove redundancy and keep the number of elements minimal.

A merger is either *perfect* or *imperfect*. A perfect merger does not result in false positives or negatives, whereas an imperfect merger may result in false positives. In addition to accuracy, we have additional requirements with filter merging:

- Merging must be transparent for applications and routers.
- Merging must maintain the set of inserted nodes. An insert of x may result in a new merged node *merge(x,y)*, but after the deletion of x the resulting node must cover y.

Filter merging may be applied in different places in the event router. We distinguish between three different merging scenarios and techniques: *local merging*, *root-merging*, and *aggregate merging*. In the first scenario, filter merging is performed within a data structure. In the second scenario, filter merging is performed on the root sets of local filters, edge/border routers, and hierarchical routers. In the third scenario, filter merging is performed on the two first levels of a peer-to-peer data structure, such as the filters poset. The latter two scenarios are exampled of remote merging.

Figure 8.2 presents two router configurations with filter merging and highlights the modular structure of content-based routers. Figure 8.2a illustrates filter merging in peer-to-peer routing. The filters poset is an

example data structure for peer-to-peer routing. Local clients are stored by the redundant forest data structure, which is the preferred structure for storing filters from local clients.

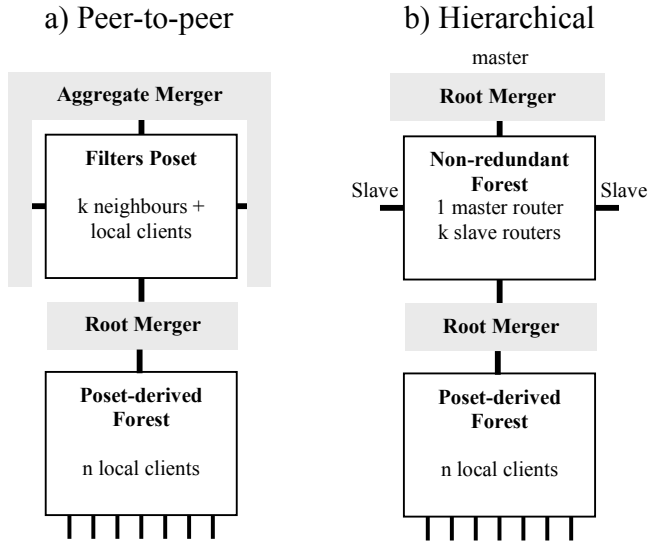

Figure 8.2: Merging extension for routing tables.

Two different merging techniques are used in the figure: root-merging for local clients and aggregate merging for remote operation. The merging of the local filters is easy and efficient, because it is performed only on the root-set and re-merging is needed only when this set changes.

Figure 8.2b shows the use of filter merging in the hierarchical environment using the forest data structure. The figure illustrates the use of root-merging for both local clients and the master router. Filter merging is easy for both local clients and the master router. Only the root sets of the local routing table and external routing table, a non-redundant forest, are merged. The forest is superior to the filters poset in hierarchical operation, because the computation of the *forwards* sets is not needed.

8.3 Rules for Merging

Two rule sets are needed in order to ensure that data structures that have been extended with merging are equivalent to the same data structures without merging. First, a set of *mergeability rules* are defined that specify

when two filters may be merged. Then, we define a set of *merging rules* for preserving equivalence for insertions and deletions between routing data structures and their counterparts that have been extended with filter merging. If deletions are allowed, filter merging requires that the merging system keeps track of both the mergers and their components.

8.3.1 Mergeability Rules

Given that it is possible to merge two input filters, for example using techniques presented in Appendix A, the merging system must decide whether or not merging is possible based on information in the routing table.

Filters may be merged in two ways: local merging that is performed within the data structure and remote merging that is performed for exiting (outgoing) routing table entries only. The former is given by the local mergeability rule presented in Definition 8.1. This rule says that only those filters that are mergeable and share an element in the *subscribers* set may be merged. This requires that the *subscribers* sets of the input filters are updated accordingly. Local merging means that mergeable filters from the same interface are merged. This allows merged filters to be stored within the data structure. This approach puts more complexity into the data structure, but also benefits the local router.

Definition 8.1 *Local mergeability rule: The operation* $merge(F_1, F_2)$ *may be performed if* F_1 *and* F_2 *are mergeable and the intersection of their subscribers sets has at least one element,* $subscribers(F_1) \cap subscribers(F_2) \neq \emptyset$. *The subscribers set of the resulting merger must contain only a single element.*

The latter option is given by Definition 8.2. Remote merging merges any filters that have the same or overlapping *forwards* sets. This may be applied only to exiting entries and the mergers should not be placed into the data structure. Remote merging allows the aggregation of multicast traffic, since the *forwards* sets of root nodes are essentially sent to most neighbours. Only the first two levels of nodes need to be considered and in most cases it is enough to inspect only root nodes. On the other hand, this approach does not benefit the local router in matching operations, but the benefit is gained in distributed operation if all neighbours employ this approach as well.

Definition 8.2 *Remote mergeability rule: Given that the forwards sets of the filters are non-empty, the filters are mergeable only when forwards(F_1) $\cap$ forwards(F_2) $\neq \emptyset$. The forwards set of the resulting merger is the intersection of the two forwards sets.*

The rule of Definition 8.1 corresponds to interface-specific merging of filters. The rule of Definition 8.2 takes into account the *forwards* sets and may be used to aggregate multicast traffic. These rules may be applied simultaneously.

8.3.2 Local Merging Rules

Let $\mathcal{M}$ denote the set of merged nodes/filters. Each element $x \in \mathcal{M}$ is a result of a sequence of *merge* operations and has a corresponding set, denoted by $CO(x)$, which contains the components of the merger x. Further, let $CV(x)$ denote nodes that were removed due to covering by x if the merger is placed in the data structure. The sets CO and CV are needed in order to maintain transparent operation.

We present six rules for maintaining equivalence. These rules do not specify the semantics or performance of the merging mechanism. They specify the requirements for equivalence. A merging algorithm needs to follow these rules. For example, rule number five does not imply that remerging of the removed merger should not be done. We assume that the routing data structure provides two operations: *add* and *del*. The *add* inserts a filter to the structure, and *del* removes a filter. Note that the *del* in rule four is applied to a merger, and the *del* in rule five is applied to a component of a merger. The *del* operation for a merger is only invoked internally; the client of the system that sent the components of the merger has no knowledge of its existence. When a *del* is performed to a node that is part of a merger's CV set, the deleted node is also removed from that set.

We also define two auxiliary operations *addComponent* and *addComponents*. The *addComponent(S, F)* operation takes a set S and a filter F as arguments and adds F to S if there does not exist a filter in S that covers F. Similarly, any filters in S covered by F are removed from S. The *addComponents(S, P)* operation is similar to *addComponent*, but the second argument, P, is a set.

The following rules assume that *subscribers(F_1)* $\supseteq$ *subscribers(F_2)* and that identical filters, $F_1 \equiv F_2$, are detected and processed before any merging rules are applied. The rules pertain to two arbitrary input filters F_1 and F_2. The rules are presented as tautologies, they have to be always

true, and we assume that each operation on the right side returns true. We assume that elements in a conjunction are evaluated from left to right.

1. $F_1 \sqsupseteq F_2 \wedge F_1 \notin \mathcal{M} \wedge F_2 \notin \mathcal{M} \Rightarrow del(F_2)$. This rule says that when a non-merged node covers another non-merged node, the covered node is removed using the *del* operation. The *del* operation performs necessary data structure changes pertaining to possible interface elimination.

2. $F_1 \sqsupseteq F_2 \wedge F_1 \notin \mathcal{M} \wedge F_2 \in \mathcal{M} \Rightarrow del(F_2)$. This rule states that when a merger is covered by a non-merger, the merger is removed and all of its components are also removed (Rule 6).

3. $F_1 \sqsupseteq F_2 \wedge F_1 \in \mathcal{M} \wedge F_2 \notin \mathcal{M} \Rightarrow del(F_2) \wedge addComponent(CV(F_1), F_2)$. This rule states that when a merger covers a non-merger, the covered node is removed and added to the merger's set of covered nodes.

4. $F_1 \sqsupseteq F_2 \wedge F_1 \in \mathcal{M} \wedge F_2 \in \mathcal{M} \Rightarrow addComponents(CV(F_1), CO(F_2)) \wedge addComponents(CV(F_1), CV(F_2)) \wedge del(F_2)$. Specifies that when a merger covers another merger, the covered merger is removed (Rule 6) and all components of the merger and nodes covered by the merger are added to the respective sets of the covering merger.

5. $del(F_1) \wedge (\exists x \in \mathcal{M} : F_1 \in CO(x))(\forall x \in CO(F_1) \setminus \{F_1\} : add(x)) \wedge (\forall x \in CV(F_1) : add(x))$. This rule says that when a component of a merger is removed, all the components and covered nodes should be returned to the data structure. After this, the merger should be removed.

6. $del(F_1) \wedge F_1 \in \mathcal{M} \Rightarrow (\mathcal{M}' = \mathcal{M} \setminus \{F_1\}) \wedge (\forall x \in CO(F_1) : del(x)) \wedge (\forall x \in CV(F_1) : del(x))$. This rule states that when a merger is removed, all its components must also be removed.

8.3.3 Remote Merging Rules

Remote filter merging rules are similar to the local merging rules with the exception that they do not need to address covered nodes within the data structure, because merged filters are not inserted into the data structure. On the other hand, it is useful to keep track of the nodes covered by a merger, the direct successor set. This may be done by placing covered nodes in CO or by using the CV set. In the former case, the third rule is not needed, and for the latter case the *del* in the third rule is not performed.

For remote merging, instead of the *subscribers* set condition, we have the *forwards* set condition presented in Definition 8.2. Implementations need to update the *forwards* sets of any mergers covered by other mergers.

8.4 A Generic Aggregate Mechanism

We present a simple generic remote merging mechanism based on the remote merging rules. The data structure must provide two information sets: the root set, and the *forwards* sets of root nodes. Both are easy to compute and the computation of the *forwards* set may be performed based on the root set alone. We propose that all mergeable root filters with the same *forwards* sets are merged. The root set is the natural candidate set for merging, because it covers other filters. By merging filters with the same *forwards* sets we simplify the finding of the mergeable set. Also, there is no need to keep track of separate *forwards* set entries.

The proposed technique may be applied to both hierarchical routing and peer-to-peer routing. For hierarchical routing, the *forwards* set of root nodes contains only a single entry, the master router. In peer-to-peer routing, merged sets are always multicast to at least $|neighbours|-1$ external interfaces. This merging technique may be called *weakly merging* for peer-to-peer routing, because it does not merge all mergeable candidates and unicast updates are not considered. It is more efficient to operate on aggregates than on separate entries.

The proposed aggregate merging mechanism is:

Generic It makes minimal assumptions on the underlying data structure. It may be used with both peer-to-peer and hierarchical routing, and also for local clients. Merging a filter in the hierarchical scenario is equivalent to merging filters from local clients, because in both cases there is only one direction where the messages are forwarded.

Efficient It is activated only when the root set changes, and it uses the *forwards* sets to aggregate merger updates. This kind of approach may be used to leverage any multicast mechanisms.

Relatively simple Tracks changes in the root set and merges filters with the same *forwards* sets. This requires management of the merged sets.

The merging mechanism requires that an additional data structure is used to keep track of merged nodes. The sets $\mathcal{M}$, CO, and optionally CV are needed for aggregate merging.

Inserting Filters The insertion of a new filter f is only interesting when it is placed in the root set. If f is not mergeable, the *add* operation is performed. If f is covered by an existing filter or a merger, the corresponding *forwards* set is empty. For the *add* operation each new element f in the root set must be checked for covering by mergers. The new forwards set for f is

$$forwards'(f) = forwards(f) - \bigcup_{f' \in \mathcal{M} \wedge f' \sqsupseteq f} forwards(f'). \tag{8.1}$$

If f has a non-empty *forwards* set and is mergeable with an existing filter or filters, aggregate merging needs to be performed. Merging can be performed only for those filters that have the same *forwards* sets. Any mergers covered by a new root filter f are discarded if they have the same *forwards* sets. This approach may result in unnecessary updates if a merger covers another merger and they have differing *forwards* sets. On the other hand, this simplified approach does not require the complex tracking of the *forwards* sets.

Deleting Filters Deletion of an existing filter f is only interesting if f is part of a merger or in the root set. When a filter that is part of a merger is removed, the merger is either re-evaluated if the size is greater than one, or removed if there is only one remaining filter in the merger. In either case the merger is unsubscribed. The corresponding uncovered set must be computed using the root set and forwarded with the unsubscription. The *forwards* sets of any direct successors to a removed merger must be re-evaluated. The *forwards* set is empty for any element in the successor set that is covered by other mergers.

8.5 Root-set Merging Algorithm

In this section we present the basic root-set merging algorithm that is the building block for the various merging mechanisms: local, hierarchical, and aggregate.

Algorithm 14 presents the basic merging algorithm that tests the mergeability of the given node n against the given set S. R denotes the root set of the routing data structure. If a merge is possible, the mergeability rules are applied and the root set is scanned for covered nodes. The algorithm maintains the CV set that contains the direct successors of a merged node. This set is convenient when deleting mergers. Algorithm 14 is linear to the

Algorithm 14 The *merge-set* algorithm.

```
MERGE-SET(S, R, n)
 1  ▷ Scan set for mergeable nodes
 2  ForAll x ∈ S
 3        do
 4           if MERGEABLE(x, n)
 5              then
 6                   m ← merge(x, n)
 7                   ADDCOMPONENTS(CO(m), CO(x) ∪ {n, x})
 8                   ADDCOMPONENTS(CV(m), CV(x))
 9                   CO(x) ← ∅
10                   CV(x) ← ∅
11                   S ← S \ {x}
12                   ▷ Scan merged set for cover
13                   ForAll x ∈ M
14                        do
15                           if m ⊒ x
16                              then
17                                   ADDCOMPONENTS(CV(m), CO(x))
18                                   ADDCOMPONENTS(CV(m), CV(x))
19                                   M ← M \ {x}
20                   ▷ Scan root set for cover
21                   ForAll y ∈ R
22                        do
23                           if y ∉ CV(m) ∧ y ∉ CO(m) and m ⊒ y
24                              then
25                                   ADDCOMPONENT(CV(m), y)
26                   M ← M ∪ {m}
27                   return TRUE
28  return FALSE
```

size of the input set S, the number of root nodes, and the size of the merger set $\mathcal{M}$. Since S is either R or $\mathcal{M}$, and $|\mathcal{M}| \leq |R|$, we have $O(|R|)$.

Algorithm 15 gives the algorithm for the addition of a new node N. When many nodes are added to the root set after a *del*, this may be invoked sequentially. The algorithm first scans the merger set $\mathcal{M}$ for covering nodes. If one or more are found, it is enough to update the CV sets of the covering mergers by adding the new node to them. If there are no covering mergers, the algorithm invokes *merge-set* first with the set of mergers $\mathcal{M}$ and if that is not successful, the set of non-covered and non-merged root nodes. We note that the algorithm may not find the best possible mergers, but this approximative approach is more favourable when support for frequent updates is required.

Algorithm 15 The *root-set-add* algorithm.

```
ROOT-SET-ADD(n)
 1  Let F be a Boolean flag
 2  F ← FALSE
 3  ▷ Check if the node is covered by a merger
 4  Forall x ∈ M
 5        do
 6           if x ⊒ n
 7              then
 8                   addComponent(CV(x), n)
 9                   F ← TRUE
10           elseif ¬F ∧ n ⊒ x
11              then remove merger x
12  if F
13     then return
14
15  ▷ Try to merge with merged set
16  F ← MERGE-SET(M,R,n)
17  if F
18     then return
19
20  ▷ Try to merge with non-covered and non-merged nodes
21  T ← R \ (⋃_{x∈M} CV(x) ∪ CO(x))
22  F ← MERGE-SET(T,R,n)
```

Algorithm 16 presents the *del* operation. When a root node is removed, the *root-set-del* is invoked first, and after that the actual *del* operation is performed for n. The *root-set-del* simply removes any mergers whose components are removed. A node is also removed from any CV sets. Possible re-merging is performed when elements of the uncovered set, when non-empty, are added to the structure. Finally, the root set is scanned for nodes that are not covered by mergers and that have not yet been forwarded. Actual implementations need also to take into account any changes due to interface elimination and nodes being removed from the root set.

Algorithm 16 The *root-set-del* algorithm.

```
ROOT-SET-DEL(n)
1  if ∃x ∈ M : n ∈ CO(x)
2     then
3          CO(x) ← ∅
4          CV(x) ← ∅
5          M ← M \ {x}
6  Forall x ∈ M: n ∈ CV(x)
7        do
8          CV(x) ← CV(x) \ {n}
```

8.6 Experimentation with One-Shot Merging

Figure 8.3 presents an overview of experimentation with filter merging. We experimented with both perfect and imperfect merging using the add scenario. The main goal of the merging benchmark is to compare the performance of the data structures when using merged filter sets.

The workload generator is used to generate filters and notifications. Each interface-specific filter set is merged using the merging algorithm and the merging time is recorded. The merging time includes only the time spent in the merging of the root set for all interfaces and thus it represents the overhead of all neighbouring routers and not the overhead of a single router. The average processing time for a single router can be obtained by dividing the time by the number of interfaces. The filters are also merged as a one-shot operation in the benchmark and the removal of a merged filter is not considered. The benchmark scenario corresponds to a situation, in which the router receives already merged filter sets. We can

compare the performance of the data structures in this situation with the basic benchmark in which merging is not used.

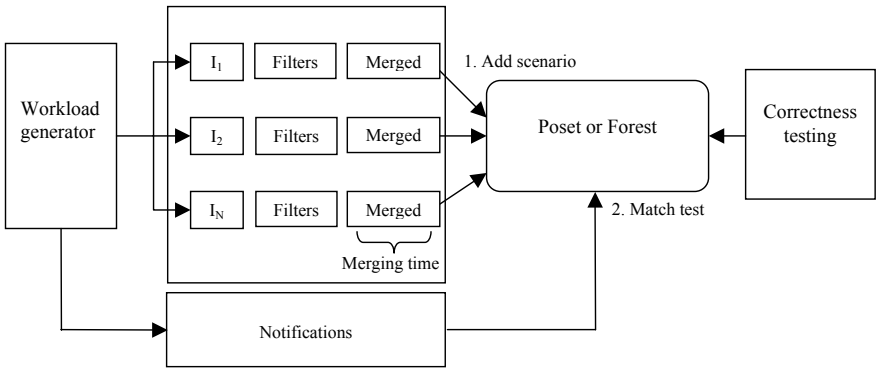

Figure 8.3: Add scenario with merging.

Appendix A presents the perfect filter merging mechanism that we use in experimentation, but the techniques we have presented in this chapter are not restricted to this mechanism. The filters were generated using the structure enforced by a schema. Each attribute filter has a single integer predicate randomly selected from the set $\{<, >, \leq, \geq, =, \neq, [a, b]\}$. The range for integer values was 100. Notifications are generated as follows: each notification has the structure of the schema, integer tuples have a random value from the range $[0, 100]$ and strings are drawn from the constraint name pool. We used the following equipment: an HP laptop with a 2 GHz Pentium III and 512MB of main memory, Windows XP, and Java JDK 1.4.2.

The two important cases in experimentation were a variable number of filters with unique interfaces and a variable number of interfaces with a static number of filters. We used a variable number of attribute filters (2-4) and 1 schema for the results. The motivation for using a variable number of attribute filters is that it seems to be more realistic for multi-router environments than a static number of attribute filters, because user interests vary. Hence, we are using one attribute filter for the type and then 1-3 attribute filters for additional constraints. The single schema situation is the most difficult scenario for matching and merging, because two schemas are by definition independent and a notification may match only one schema (event type) at a time, but filters of the same schema have to be analyzed.

We measured matching time using a matching algorithm that walks only those sub-posets or sub-trees of the structure that match the notification. We compare this algorithm with a naive matcher that tests the notification against each filter.

Perfect Merging Figure 8.4 presents the impact of interface-specific merging for forest and poset performance with a static number of interfaces (3) and a variable number of filters. The merging benchmark compares the insertion and matching performance of interface-specific minimal cover sets with merged sets. 60 replications were used for these results. The merging time represents the worst case, because the input sets were merged using a one-shot procedure and normally this would be performed incrementally.

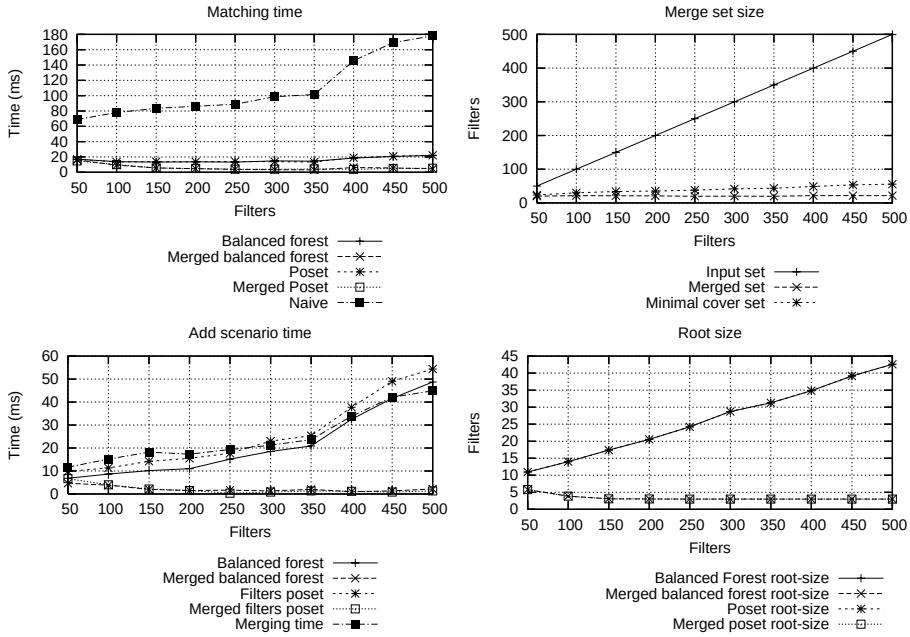

Figure 8.4: Impact of merging on the forest and poset performance. Results for 3 interfaces.

As the number of filters grows the merging algorithm is able to remove redundancy. The root size of the merged forest and the poset are the same and root set sizes in merging scenarios are considerably smaller than in normal operation. The root set sizes are shown for the merged and non-merged sets, respectively. The size of the non-merged set grows with the number of filters, whereas the size of the merged set is constant in

the figure. Based on these results the merging time is reasonable and the merged forest or poset is created quickly. The matching time for the merged set is considerably shorter than for the non-merged set.

Figure 8.5 presents merging results for a variable number of interfaces and a static number of filters (500). The insertion and matching times for the merged sets are also significantly lower in this scenario. The processing performance decreases as the number of interfaces grows, because there are fewer filters per interface. The root sizes are very small for the merged sets whereas the non-merged sets are large. This is due to saturation, where the merged roots become very general when there are many filters.

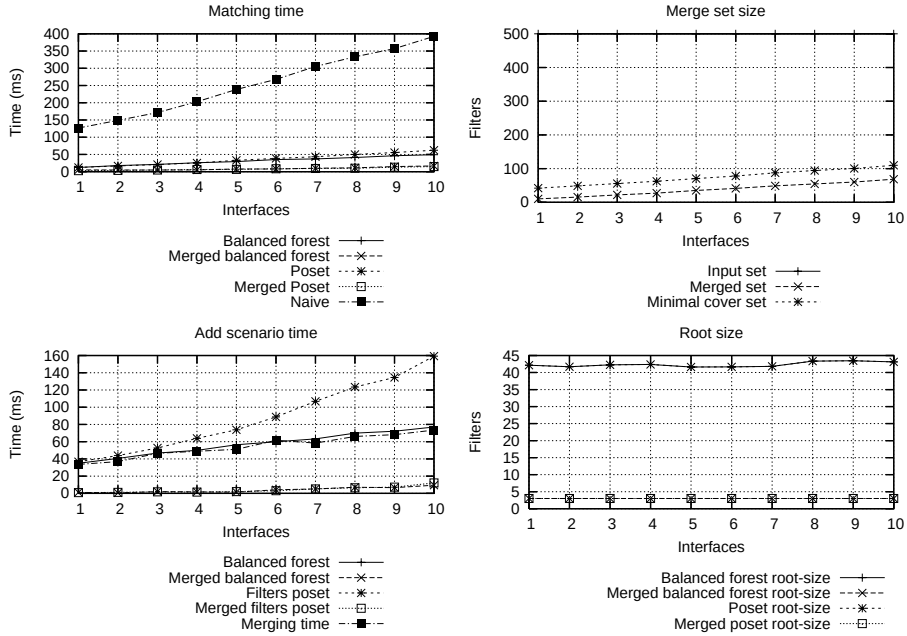

Figure 8.5: Impact of merging on the forest and poset performance. Results for 500 filters.

Imperfect Merging We also experimented with imperfect merging with the same parameters as the perfect merging scenario. Imperfect merging had similar performance to perfect merging. When the number of non-covered filters with the same structure grows, imperfect merging performs considerably better than perfect merging. On the other hand, the mechanism may result in a number of false positives. The false positive rate depends on the parameters.

8.7 Experimentation with Dynamic Root Merging

Experimentation with the dynamic root merging algorithm presented previously is similar to the add and add/remove scenarios presented in Chapter 4. Figure 8.6 illustrates the benchmark. First, a number of filters (up to 500) are generated using the workload generator. Each filter has a unique interface, which corresponds to the local or hierarchical scenario. The filters are added to a forest extended with the root merging algorithm. The merging algorithm maintains a merged root set when filters are added and removed to the structure.

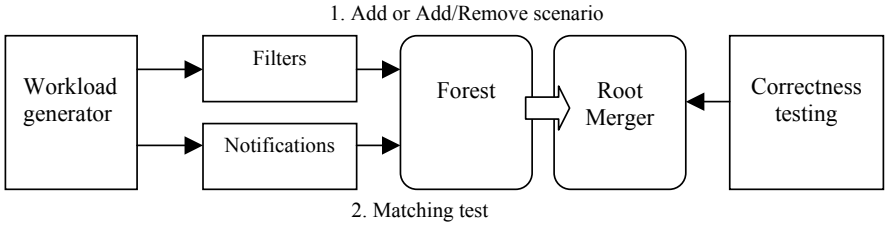

Figure 8.6: Add and add/remove scenario with the root merger algorithm.

Correctness of operation is ensured by testing that the merged set does not result in false negatives and positives, and ensuring that the internal data structures used by the merging algorithm are correct.

Each measurement was replicated 20 times and the predicates were randomly selected form the set of predicates $\{<, >, \leq, \geq, =, \neq, [a, b]\}$ using a uniform distribution. The two benchmarked filter schemas were: a variable number of attribute filters (1-3), and a static number of attribute filters (2 and 3). The number of filters to be added and removed in the add/remove microbenchmark was 100.

Figure 8.7 presents the results for the variable number of attribute filters case. In this scenario, filter merging may be performed without significant overhead, because the root set is small due to covering and root filters have only a few attribute filters. Merging is very useful in this case and a constant or near constant root set size is achieved, whereas the non-merged root set size is linear to the number of input filters.

Figure 8.8 presents the results for two static attribute filters. Filter merging is also beneficial in this case, but has more overhead than for the previous microbenchmark. The merged filter set size has also a near constant size in this case. The root set size is larger in this case compared

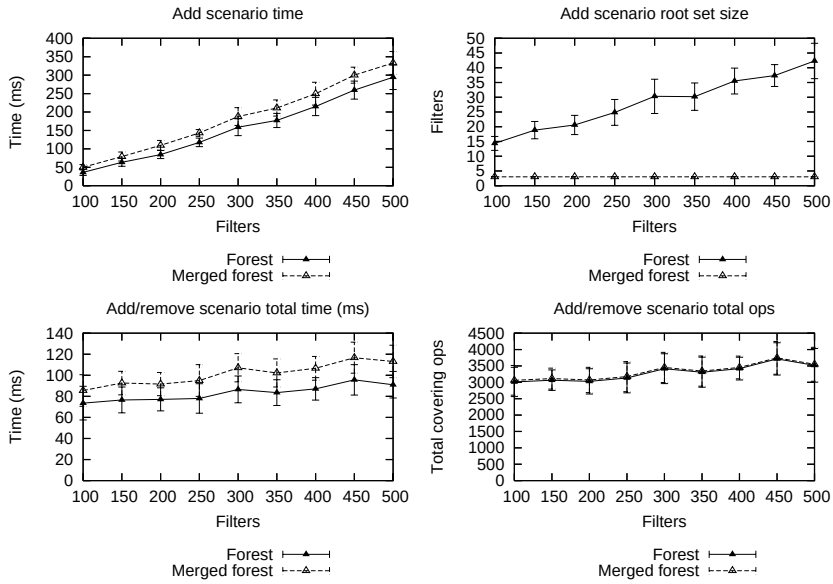

Figure 8.7: Add and add/remove scenario for a variable number of attribute filters.

to the previous case and filter merging cannot be performed as often due to the static number of attribute filters.

Figure 8.9 shows the results for three static attribute filters. In this case, the merging overhead is substantial and the root set size is not reduced. This demonstrates the effects of a non-mergeable workload.

8.8 Summary

The results show that covering and merging are very useful and give significant reduction of the filter set, especially with a variable number of attribute filters, because those filters with fewer attribute filters may cover other filters with more attribute filters. We also experimented with a static scenario, where the number of attribute filters per filter is fixed. The static scenario also gives good results for covering, but perfect merging does not perform well when the number of attribute filters grows.

When the number of filters per schema grows the whole subscription space becomes covered, which we call *subscription saturation*. This motivates high precision filters for a small amount of filters, and more general filters when the subscription space becomes saturated.

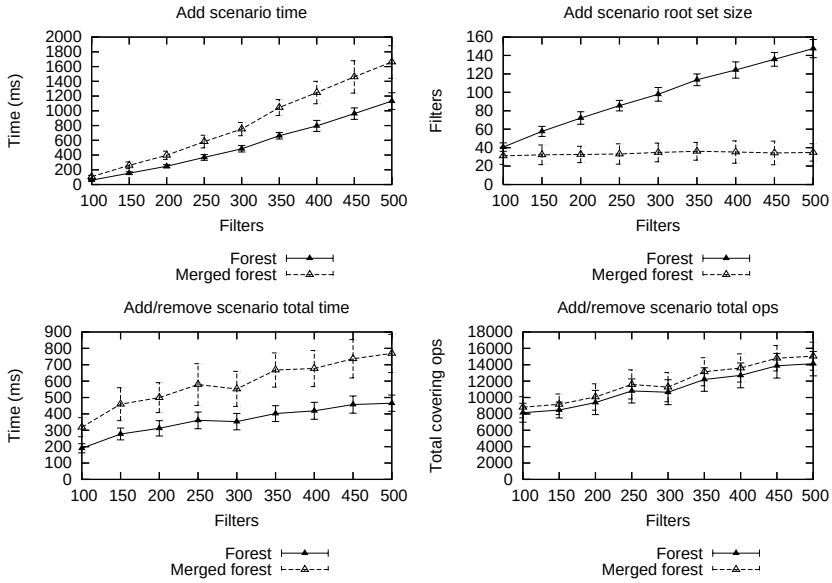

Figure 8.8: Add and add/remove scenario for a static number of attribute filters (2).

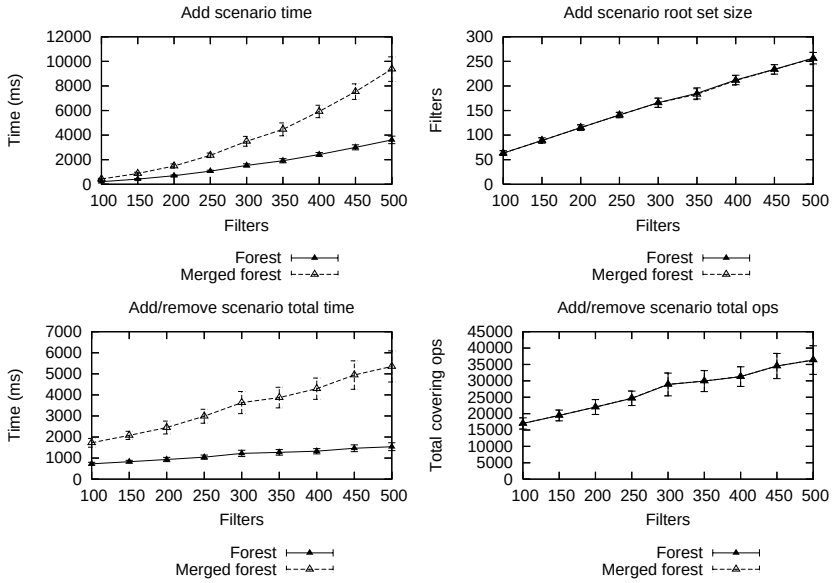

Figure 8.9: Add and add/remove scenario for a static number of attribute filters (3).

The results indicate that filter merging is feasible and beneficial when used in conjunction with a data structure that returns the minimal covering set, and the filter set contains elements with a few attribute filters that cover more complex filters. The cost of unsubscription, or removing filters from the system, can be minimized by merging only elements in the minimal cover or root set. When a part of a merger is removed, the merger needs to be re-evaluated. Any delete or add operation outside the minimal cover does not require merging. For complex filter merging algorithms it is also possible to use a *lazy* strategy for deletions. In lazy operation, the system delays re-merging and counts the number of false positives. When a threshold is reached the minimal cover set or parts of it are merged and sent to relevant neighbours.

Part V

Applications

Chapter 9

Collection and Object Synchronization Based on Context Information

We present a novel mechanism for collection and object synchronization based on context information. The mechanism is based on a distributed event system and uses event filters to represent context and realize context queries. The central operations of the system are storing and retrieving objects by their context. The new feature of the system is context-based synchronization, which allows synchronizing collections of objects continuously based on the given context. The system may also be used for context-based service provisioning. We present mechanisms for both collection and object synchronization. The former uses the publish/subscribe paradigm and the latter builds on an XML-aware file synchronizer. We focus on the first mechanism and present a context-aware photo library as a sample application.

9.1 Introduction

A number of core technologies are needed in order to realize the intelligent and adaptive services of tomorrow. Efficient and intelligent *data synchronization* is a basic property of current and future applications, especially in mobile and ubiquitous environments. Mobile phones, laptops, and PDAs have become commonplace. We are faced with the question of how to locate important data items and keep them synchronized on different devices. Furthermore, the rules for synchronization are dependent not only on the device, but also on the past, current, and future operating context. In this chapter, we present a *middleware* system and an *API* for creating and

tracking object collections based on *context queries*, and then synchronizing the objects using a file *synchronizer*.

Context-awareness is considered to be an important property of future mobile applications [1]. In this chapter, context is represented by a set of dimensions that take either discrete or interval values. We focus on how context information may be used to synchronize objects and do not consider how the actual context information is acquired.

The proposed mechanism is based on three basic middleware services included in the Fuego middleware service set: the *messaging service* based on synchronous and asynchronous SOAP [71], the *event service* [129] that facilitates distributed *publish/subscribe* (pub/sub), and the *XML-aware file synchronizer* [82]. The messaging service is responsible for transporting information between known entities using, for example, explicit addresses or queue-names. The event service is responsible for decoupled, anonymous many-to-many information dissemination [51]. The XML-aware file synchronizer provides facilities for the synchronization of files and directories using XML directory trees and a tree reconciliation mechanism.

The contributions of this chapter are: 1. Using techniques from publish/subscribe systems to match and compare context information. This allows both point and subspace matching in the context/content space. 2. An API for tracking and synchronizing collections using context queries. 3. Using the distributed pub/sub to synchronize collections, and 4. using an XML-aware synchronizer to synchronize files. The system may also be used for context-based service provisioning and context-based personalization of applications.

This chapter is structured as follows: in Section 9.2 we discuss how context information may be represented using event filters. Section 9.3 presents the proposed synchronization system and gives examples of context-based synchronization. In Section 9.4 we examine the sample application and Section 9.5 discusses related work. Finally, Section 9.6 presents the summary.

9.2 Representing Context with Filters

Content-based event routing has been proposed as one of the requirements for advanced applications, in particular for mobile users [33, 44]. Context-sensitive messaging using content-based routing is implemented by subscribing the context the client is interested in, after which all events matching the context will be delivered to the subscriber. The context is represented using a filter. The expressiveness of the subscription language is

one of the main factors that define the power and scalability of the event service.

A generic data structure, such as the poset-derived forest, supports the use of arbitrary filter objects as long as the covering relations are defined for the input set. On the other hand, being generic, they cannot provide the same performance as filter object and language specific matchers. It is more efficient to use a separate data structure for matching. In general, filter matching is done by counting attributes using the counting algorithm [34, 90] or using a tree-based data structure [4]. These more efficient matching mechanisms assume that the filters and notifications have a distinct and often simple structure. Moreover, typically the most efficient matchers do not support frequent changes in the set of queries, which poses challenges for context queries.

We propose that the DoubleForest is used to store both context profiles and context queries. Using this structure, it is easy to produce *change notifications* to relevant entities when an element is added or removed from either structure. This approach is general and supports various filtering languages; however, it does not work well if there are only a few covering relations between the elements. We have developed an online example of this mechanism called the *ContextBrowser*[1]. The ContextBrowser shows context queries and profiles graphically and demonstrates real-time collection tracking.

9.3 Synchronizing Collections

We follow the ideas in [84] and propose to support basic communication between context providers and consumers using a pub/sub event-routing network. In addition, we propose to leverage the covering and overlapping relations between filters to realize context-based object synchronization. The separation of concerns offered by pub/sub systems simplifies the development of higher level components, because mobility transparency and scalability is handled by the lower pub/sub layer.

Figure 9.1 presents an overview of the synchronization process. The environment consists of a number of client systems (terminals) and a number of synchronization service instances. Consider a scenario, in which the client is interested in documents modified in specific project meetings. Relevant metadata and contextual information is added to documents after meetings so that this information is available. The client can formulate a context query that includes those specific meetings and track documents

[1] Available at www.hiit.fi/fuego/fc/demos

that match this query using the service. The pub/sub system delivers change notifications and the collection is updated in real-time. In collection synchronization the client receives notifications that denote changes to the collection, namely addition and removal of entries. At some point when tracking the collection, the client may be interested in retrieving the documents to his current machine. This is accomplished using the file synchronizer, which updates the local directory structure and downloads the requested files. This motivates the different phases of the synchronization process.

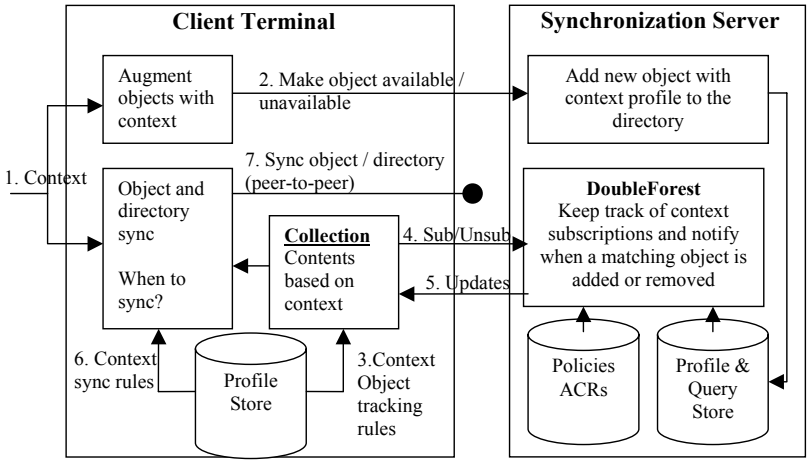

Figure 9.1: Context-based collection and object synchronization.

Today security is also an important requirement and in many cases profiles can be matched to queries only when *access control rules (ACR)* allow it. ACRs can also be represented using filters. A valid context query is covered by at least one access control rule that is applicable for the client that made the query.

Context-based synchronization proceeds in the following 7 phases as depicted in Figure 9.1. We assume that there is an existing mechanism for acquiring the context of the client system (1). The context is associated with objects, files, and directories that clients make available using the system (2). The synchronization service is responsible for maintaining objects and their associated context profiles, which are updated by clients. The client system stores two different kinds of filters: first, the context queries needed for tracking objects based on their context (3), and then a set of rules that are used to synchronize subsets of these tracked documents. The

synchronization rules are motivated by the observation that the client may not always wish to synchronize the whole collection, but only a subset of it.

The service is also responsible for accepting collection-tracking subscriptions and unsubscriptions from clients (4). The service keeps the client notified about the current contents of the tracked collection (5). The service uses the DoubleForest data structure to maintain context profiles and context queries.

The client uses a set of synchronization rules for synchronizing subsets of the tracked documents (6). A subset of the collection is synchronized using the file synchronizer (7). The synchronized objects can reside anywhere in the distributed system, for example on other terminals.

We divide the synchronization of objects and collections by their context into the following three parts:

- Description: How to describe context and how to describe objects. We separate the description of an object from the object itself. This allows the system to keep metadata separate from the actual location of the object. Typically the object description, denoted by O_D, contains a pointer (URI) to the object. The description also includes the context profile.

- Continuous collection synchronization: How to synchronize a collection defined by a context query. A query exactly defines the contents of a collection to be those profiles that match the query. The query may also contain metadata relevant for matching. If the change notifications are ordered using causal or total order the client can apply them in the correct sequence. The communication is performed using asynchronous publish/subscribe.

- Document and directory synchronization: How to synchronize the data associated with the items. We use the file synchronizer to perform one-shot synchronization after periods of disconnection.

Collection synchronization differs from the basic pub/sub semantics, because context profiles have temporal duration, but events typically are instantaneous and published only once. This means that both profiles and queries must be maintained by the service and the mappings introduced earlier are needed between these sets.

The synchronization service is logically centralized and based on the client-server interaction model. We use the service to solve the context-matching problem. The service exports a pub/sub API for adding and

Table 9.1: Basic API operations.

Operation	Description
add(O_D,F_D)	Adds the context profile F_D to object desc. O_D.
del(O_D,F_D)	Removes the context profile F_D from object desc. O_D.
get(F)	Get descriptions (one-shot) for the given context query F.
add(C,F)	Adds the context query F to the collection C.
del(C,F)	Removes the query F from the collection C.
sync(C)	Sync C using the synchronizer.

removing object descriptions and context queries. Clients make requests and receive change notifications using the pub/sub system, which abstracts issues such as disconnections and mobility. The service is implemented using a context-matcher component that stores both context profiles and object descriptions. The component uses the underlying event system for communication and matching filters and notifications. The implementation of the component is simple, because existing data structures and communication primitives are used.

In addition to the client-server model, it is also possible to implement the system using the peer-to-peer model. The peer-to-peer model does not have this centralization of service instances and allows entities to produce and consume context information directly using pub/sub interfaces. This means that clients need to keep track of their objects and the associated context filters, and be able to match them against incoming queries. This approach requires more from clients, but does not require a centralized component. On the other hand, features such as authentication [56] and logging become more difficult to realize.

9.3.1 Operations

The central operations of the system are storing and retrieving objects by their context. Let F_D denote a context profile that represents the current or past context of an object. Figure 6.1 gives an example of a profile and a query. The API consists of two parts: addition and removal of context profiles, and the management of collections. The collection C is determined by a set of context queries. The API allows the client to add and remove these queries (Table 9.1).

A client may modify or remove only objects that belong to it. The synchronization service must keep track of object descriptions sent to clients

and only new descriptions are sent. Similarly, clients are notified when a change or deletion of a description changes their collection. The operations may be extended with access control rules, for example, rules for deciding which entities may access the object or receive a notification about it.

The context profile associated with an object (the object's context) may be a point or a subspace of the content space. Context queries associated with collections are assumed to be subspaces. We distinguish between matching points and subspaces, because there are efficient and sublinear algorithms for matching points [34, 90]. Algorithms for covering and overlapping are computationally more expensive. Covering and overlapping can be computed efficiently for the commonly used filtering model, in which each tuple in a filter contains exactly one predicate (constraint).

The *sync* operation creates local replicas for new objects, updates concurrent changes to distributed copies, and removes files that disappear from the collection. The URI may also map to a directory, in which case the whole directory is synchronized based on the given context. It may also happen that the client wishes to synchronize only a subset of the collection.

9.3.2 Mapping to the Publish/Subscribe Paradigm

The collection synchronization service builds on the Fuego event system and maintains two data sets: the set of object descriptions and the set of context queries. The service subscribes to receive new object descriptions and context queries. The client on the other hand subscribes its event message queue name. This provides a transparent way to send messages to the client irrespective of physical location using the queue name. The service multicasts updates using the recipient queue names. This means that only a single update message is sent by the service for each modification. The default Fuego filtering language supports the basic relational and string comparison operations. The default language may be extended or replaced to support more expressive languages.

One useful feature for collection synchronization is the *replace-identifier* feature of our event system, which allows one notification to replace other notifications in a message queue that have the same source-specified identifier. The replace-identifier in conjunction with *time-to-live* are useful in removing stale and obsolete messages from queues before they are delivered to clients, thus reducing wasted bandwidth and client-side processing. For example, they may be used to ensure that only one notification is sent for a changed object.

The *add*, *del*, and *get* client collection API operations are implemented using the publish operation of the event system. They synchronization

service must subscribe to these operations before they may be used.

Scalability may be improved by separating orthogonal context types (or schemas) into different distributed service modules. It is assumed that a single service instance is responsible for a subspace of the context space and can thus uniquely number collection change notifications for that subspace. If this design is used, the clients must treat change notifications for the same object from different service instances as independent. This means that a remove notification from one service instance does not remove the object from the client's collection if the client has previously received an addition for the same object from a different service instance.

9.3.3 Sequence Diagram

Figure 9.2 presents a sequence diagram of synchronization with pub/sub operations omitted. In 1. the client B stores the object O using any available storage mechanism and either receives or formulates a URI for the object (2). After this the client creates the description O_D of the object that contains the URI and the context profile F_1. For example, the filter could contain information about a workshop: the duration, the location, presenters, participants, and the topics of the workshop. In 3. B adds the description with the associated filter to the synchronization service.

Client A creates a collection (4) with the context filter F_2. The filter could, for example, specify all objects created in technical workshops where Amy, John, and Bob were present. In 5. the filters match, and A is notified about a new member in the collection (6), which is updated (7). After this the collection synchronization is started (8) and the collection or a subset of the collection is selected for synchronization. Finally, in the last step (9) the files are retrieved and updated.

9.4 Sample Application: Context-aware Photo Library

To demonstrate the proposed context and metadata-based synchronization, we extended an existing synchronizing photo-library application [81] to support context-aware operation. This required three modifications: first, the application attaches metadata to each photo. The metadata is acquired from various sources, location information may be obtained using a GPS device or GSM cell-identifier, for example. The second modification updates this information to the synchronization service that is reachable through the pub/sub network and allows the client to receive change no-

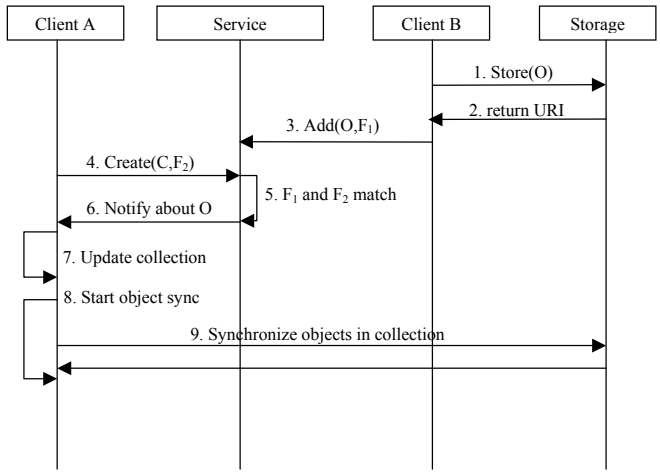

Figure 9.2: Sequence diagram of synchronization without pub/sub operations.

tifications using either push or pull mode. The pull mode is unintrusive and does not incur any communication cost that the user is not aware of. The push mode, on the other hand, allows proactive tracking of shared photos. The third modification uses the collection API for tracking and synchronizing images.

The context-aware photo application uses the proposed mechanism as follows: 1. Context information is attached to new photos. The context information may be input from the user, gathered from various sensors, or a combination of them. In our example, we use pre-defined context descriptions. 2. The photos are published to the synchronization service using the *add* API method. This is implemented in the client application, which makes all photos in the assigned directories available to others using this API call. 3. The user instructs the application on what contexts to follow and synchronize by formulating context queries. An example of a context query could be: { *type* = "photo" $\wedge$ location = "Helsinki" $\wedge$ description *contains* ("concert" $\vee$ "rock") }. 4. The application will receive updates on any objects whose profile matches the active queries. 5. This collection is shown to the user, and depending on the mode of operation, either all or selected elements of this collection are synchronized. The photo library synchronizes all images.

This example illustrates how the proposed mechanism is used in context-

aware applications. This same approach could be used, for example, to follow documents in a collaborative environment.

The DoubleForest data structure is used to store the profiles and queries at the server and to perform temporal subspace matching. When a query or a profile is added or removed, the DoubleForest computes the new mappings between them. This functionality can be directly used to determine where collection update notifications should be sent.

9.5 Related Work

Novel context-aware applications have been proposed, Mobile Media Metadata [118] for example, but typically many applications rely on centralized servers and do not have specific middleware support for tracking and managing objects by their context. The main goal of our proposed system is to provide this middleware support for context-aware applications.

Context-awareness is an active research topic and many systems provide applications with support for context-aware operation. Systems such as the Context Toolkit [117], Gaia [114], and the Context Cube [62] support context-aware applications. A context-based storage system in presented in [73] with an emphasis on group access. Almost all context-aware systems employ some kind of asynchronous communication abstraction, typically asynchronous events. Events support context-triggered actions, and allow run-time binding of components supporting modularity. In addition to push communication semantics many systems, such as Gaia, also support a query interface for receiving (pulling) context information.

Most systems do not have middleware support for synchronizing objects and directories based on context. Gaia has a context file system, which supports the naming and location of files based on the file path identifier. Gaia's context file system allows the attachment of context attributes to files or directories [64]. The proposed system is based on a similar idea, but the systems have several differences. First, the proposed mechanism is based on filters and uses the covering relations for matching. This allows automatic categorization of objects based on their context and the context descriptions may contain predicates. Second, the context file system is more concerned with personal data and making it available in different active spaces than distributed data synchronization and change notification. Tuple spaces, namely Lime [94], support synchronization of the tuple space, but do not offer any specific API support for continuously tracking context changes and then synchronizing a select subset of objects based on the current context.

9.6 Summary

In this chapter we presented a mechanism for context and metadata-based collection and object synchronization. We examined three important parts: the description of context and separation of the actual object data from the metadata, continuous collection synchronization, and object and directory synchronization. We focused on the first two parts and showed how the pub/sub paradigm may be used to implement collection synchronization and how filters may be used to represent both profiles and queries.

The pub/sub paradigm decouples entities from each other and abstracts issues such as disconnections, message buffering, and mobility management. Existing algorithms for matching, covering, and overlapping allow straightforward implementation of the components of the synchronization service. We use the DoubleForest data structure to store the profiles and queries at the server and to perform temporal subspace matching. Distribution of service functionality may be realized by distributing different context types or schemas to different servers.

Chapter 10

Example Scenario: Smart Office

We have developed a smart office scenario that highlights the features of the Fuego Core middleware service set discussed in Chapter 9, and the different ways that information is processed and used in a modern office environment, where people have meetings, do their work, and also change their location. Information sharing is vital in this environment and this sharing is context-sensitive — people require different information depending on time, location, and other contextual parameters. The scenario was demonstrated at the sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004).

The scenario highlights various events and interactions in the environment using a Graphical User Interface (GUI) and physical devices, such as mobile phones. The server and the mobile devices run the Fuego middleware system. Figure 10.1 presents the animated GUI that shows daily events happening and we may observe the activities of one person in the office environment through several physical mobile devices: a J2ME smartphone and a laptop.

Key activities in the scenario are:

1. Reading important news and information while en-route to work. Receiving interesting information proactively (push) to a mobile device.

2. Presence status at work and elsewhere and presence change distribution. Arriving to work changes presence status automatically and interested co-workers are notified.

3. Arriving to office and automatically starting the desktop with current configuration. Changing the end-point of events from one device to another (session mobility), for example transfer Instant Messaging (IM) discussions and news feeds between devices at run-time.

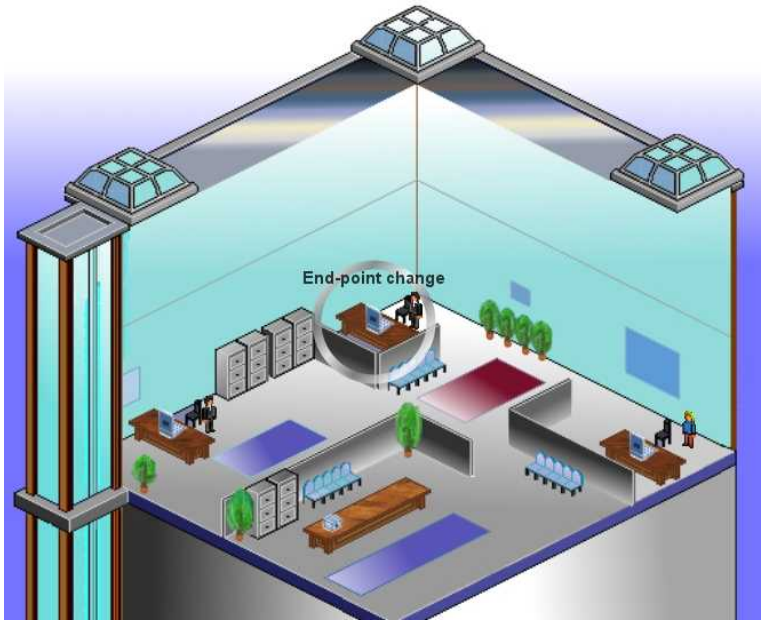

Figure 10.1: Office demonstration GUI.

4. Context-sensitive messaging at the workplace: all who are present at the office and assigned to a project will receive a notification of a meeting starting in 10 minutes.

5. Going to the meeting, the event session is transferred back to the smart-phone. Changing presence characteristics for the meeting.

6. Synchronization in meetings — using the Fuego XML-aware file system to synchronize important documents and calendars. The secretary of the meeting sends the synchronization trigger as a context-sensitive message to the project group. The trigger prompts the desktop to synchronize the document using 3-way XML merging.

7. End of meeting, going to lunch. Sending context-sensitive messages to co-workers: "Anyone interested in going to a nearby pizzeria?"

8. Leaving office, changing presence values automatically.

In this scenario we support decoupled communication between context providers and consumers using the Fuego event service. The event service uses the poset-derived forest for content-based routing and matching. The system is based on a modular content-based router, which supports different

routing configurations and handover protocols by separating local clients and distributed operation. Events are buffered for disconnected clients. The default configuration is based on event channels.

Context-sensitive messaging using content-based routing is implemented by subscribing to the current context. This means that a filter is created and subscribed that represents the current context. All events that match the context are delivered to the proper recipients. Using the event service for context-aware messaging offers separation of concerns that simplifies the development of higher level components, because mobility transparency and scalability are handled by the lower pub/sub layer. The communication infrastructure is naturally divided into two parts: the messaging service, and the event service.

The file synchronizer addresses the information-sharing requirements of the environment and synchronization may take place, for example, after receiving an asynchronous event indicating the ending of a meeting and pointing to modified documents. In order to keep track of documents based on their context, we used the DoubleForest data structure to implement a context-based collection and object tracking service on top of the event service.

Part VI

Conclusions

Chapter 11

Conclusions

In this thesis we have presented and investigated efficient content-based routing for static and mobile environments and considered temporal subspace matching and context-aware operation. In the first part we presented the introduction and an overview of content-based pub/sub. In the second part of the thesis we presented a set of new data structures for efficient content-based routing tables. Useful designs for content-based routing tables based on forests and posets were also presented and examined. In the third part, we examined and analyzed client mobility in different publish/subscribe topologies. In the fourth part we presented advanced structures and techniques, namely the DoubleForest structure for matching profiles and queries, and filter merging for efficient information dissemination. In the fifth part, we presented example applications and scenarios.

The main contribution of the second part is a formal definition of the poset-derived forest data structure and its variants. This work addresses the requirement for frequent updates to routing tables, and really efficient data structures have not been proposed in literature. Typically, event systems define routing tables using sets, with the exception of the filters poset used in Siena.

Experimental results indicate that the proposed data structures are efficient and perform considerably better than the directed acyclic graph-based filters poset when there are many local clients. The forests may be combined with the poset to create efficient routing tables. The runtime cost of the structures depend on the underlying data set and the covering relations between the entities of the data set. The investigated mechanisms are generic and do not require knowledge about the filtering language other than the covering relations between filters.

The main engineering guidelines and observations of the second part are:

Local clients and routing Local clients should be stored using a forest. The forest that stores filters from clients may be connected to other structures, for example: a non-redundant forest in hierarchical routing, or a poset in peer-to-peer routing. If there are no local clients, the poset is more suitable for peer-to-peer operation.

Matching The matching performance of the data structures is not on the same level as with more optimized matchers. An additional matcher data structure should be used to quickly match notifications to the local clients. This is a two-phase process: first notifications are matched for the covering set by the poset or forest, and then they are matched by the matching data structure.

The main contributions of the third part is the characterization of pub/sub mobility using completeness and mobility-safety and deriving the upper and lower bound costs in terms of message exchanges for the handover. The lower bound cost is analyzed in terms of the covering optimization, which prevents unnecessary topology updates when relocated subscriptions are already established at the destination.

We examined the cost of pub/sub mobility using three mobility mechanisms and topologies: generic mobility support, acyclic graphs, and rendezvous-based topologies. The generic mechanism has a high cost for mobility. The other two mobility mechanisms have considerably smaller cost. Rendezvous points were observed to be good for mobility.

If an acyclic graph-based routing topology is incomplete, content-based flooding should be used to complete the handover successfully. This means that the optimizations discussed for complete topologies are not applicable for incomplete topologies. Since it is not possible to detect completeness and many systems are inherently incomplete, the optimizations that avoid content-based flooding may not be applied in practise. Mobility-safety cannot be guaranteed if protocols engineered with the completeness assumption are used for incomplete topologies.

The rendezvous-based model used in the Hermes overlay pub/sub system was observed to be a good candidate topology for pub/sub mobility support, because it may be extended to guarantee the completeness of subscriptions and advertisements. The model does not require the flooding of the whole network with advertisement messages. We proposed a mobility-friendly topology by limiting the rendezvous-based model to acyclic overlay graphs. This limitation guarantees that the rendezvous-based mobility protocol cannot have greater cost than the general acyclic graph protocol.

We proposed the following techniques for improving mobility-aware pub/sub systems:

Overlay-based routing. Overlay addresses prevent the content-based flooding problem and allow better error recovery than using lower level protocols.

Rendezvous points. Rendezvous points simplify mobility by allowing better coordination of topology updates.

Completeness checking. Completeness checking ensures that the subscriptions and advertisements are fully established in the topology. This is needed to perform the covering optimization.

The main contributions of the fourth part are the DoubleForest structure with optimizations and the filter merging technique. We summarize the main contributions and guidelines as follows:

Temporal subspace matching The DoubleForest with optimizations performed considerably better than the set-based benchmark algorithm for the cover-based matching. The performance of overlap-based matching did not improve substantially. The storage cost of transitive closure due to cover may be reduced by storing only the immediate successors in the structure. To our knowledge, this is the first published data structure for generic temporal subspace matching.

Preloading Filter preloading was observed to be beneficial for subspace matching (both cover and overlap). Constant or near constant matching time is achieved by preloading filters into the data structure. Random preloading was used to experiment with effects of failure to anticipate queries and profiles. Random preloading did not introduce significant overhead for covering, but the cost for overlap-based matching increased linearly with the number of preloaded objects.

Merging Hierarchical routing systems are easy to extend with filter merging. The merging of local filters is easy for both hierarchical and peer-to-peer routing. On the other hand, merging of outgoing filters for peer-to-peer routing tables is more difficult especially for the *del* operation. We observed significant performance benefits from the merging of interface-specific filter sets. We also presented a dynamic algorithm for filter merging and showed that, given a mergeable workload, dynamic filter merging is feasible.

The main contributions of the fifth part are the context-aware object and collection synchronization, and the Smart Office scenario. The DoubleForest data structure was used to store the profiles and queries at the

server and to perform temporal subspace matching. The structure can be directly used to determine where collection update notifications should be sent. The applications and scenarios suggest that the results of this thesis are generic and might be applied in different areas of computing — also outside basic content-based routing.

Figure 11.1 summarizes the data structures and techniques presented in this thesis. The basic data structures, such as the poset-derived forest, for routing and matching, are shown on the left. The middle column presents advanced data structures, such as the DoubleForest. The right column presents techniques for improving system behaviour, such as preloading, mobility support, and filter merging.

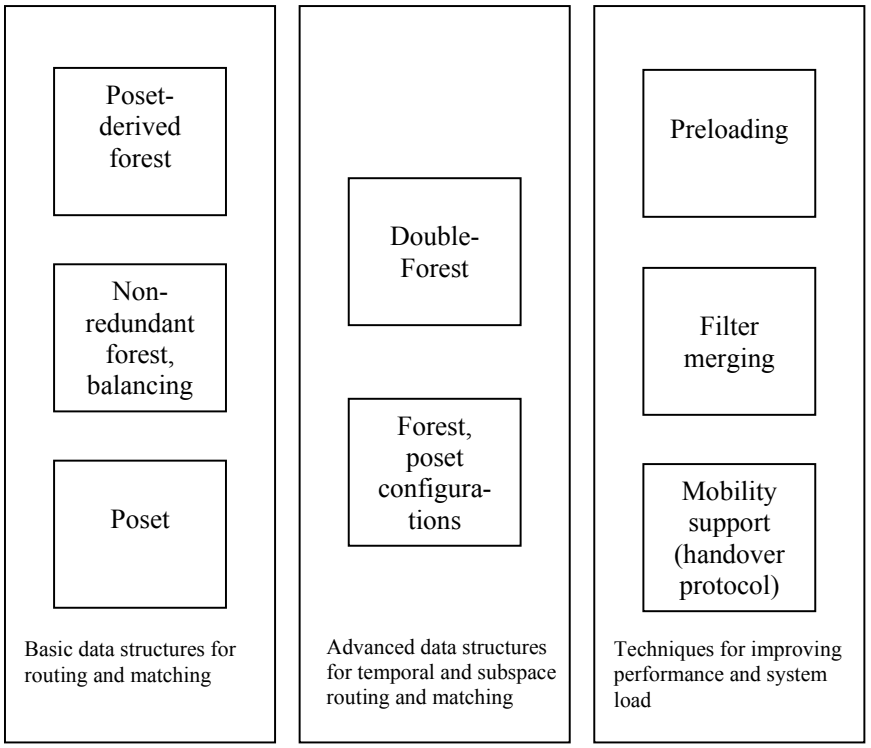

Figure 11.1: Summary of data structures and techniques.

References

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 253–262, New York, NY, USA, 1989. ACM Press.

[3] R. Agrawal and H. V. Jagadish. Hybrid transitive closure algorithms. In *Proceedings of the sixteenth international conference on Very large databases*, pages 326–334, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[4] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of Distributed Computing*, pages 53–61, New York, NY, USA, 1999. ACM Press.

[5] B. Aiken, J. Strassner, B. E. Carpenter, I. Foster, C. Lynch, J. Mambretti, R. Moore, and B. Teitelbaum. *RFC 2768: Network Policy and Services: A Report of a Workshop on Middleware*. IETF, Feb. 2000.

[6] G. Alder. *Design and Implementation of the JGraph Swing Component*, 1.0.6 edition, Feb. 2003. Available at: http://jgraph.sourceforge.net/doc/paper/.

[7] J. Antollini, M. Antollini, P. Guerrero, and M. Cilia. Extending Rebeca to support concept-based addressing. In *First Argentine Symposium on Information Systems (ASIS 2004)*, Sept. 2004.

[8] J. Bacon et al. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, Mar. 2000.

[9] R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito. Modelling publish/subscribe communication systems: towards a formal approach. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 304–311. IEEE, 2003.

[10] K. Betz. A scalable stock web service. In *Proceedings of the 2000 International Conference on Parallel Processing, Workshop on Scalable Web Services*, pages 145–150, Toronto, Canada, 2000. IEEE Computer Society.

[11] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor, 2004. Preprint. Available at: http://www.cs.rochester.edu/u/beygel/publications.html.

[12] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: A scalable publish-subscribe system for Internet games. In *Proceedings of the 1st Workshop on Network and System Support for Games*, pages 3–9, Braunschweig, Germany, 2002. ACM Press.

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

[14] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[15] G. Bricconi, E. Tracanella, E. D. Nitto, and A. Fuggetta. Analyzing the behavior of event dispatching systems through simulation. In M. Valero, V. K. Prasanna, and S. Vajapeyam, editors, *HiPC*, volume 1970 of *Lecture Notes in Computer Science*, pages 131–140. Springer, 2000.

[16] P. J. Brown. Triggering information by context. *Personal Technologies*, 2(1):1–9, September 1998.

[17] I. Burcea, H.-A. Jacobsen, E. de Lara, V. Muthusamy, and M. Petrovic. Disconnected operation in publish/subscribe middleware. In *Mobile Data Management*. IEEE Computer Society, 2004.

[18] N. Busi and G. Zavattaro. Publish/subscribe vs. shared dataspace coordination infrastructures. In *Proc. of IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'01)*, pages 328–333. IEEE Press, 2001.

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

[20] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[21] F. Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, Mar. 2004. IEEE.

[22] M. Caporuscio, A. Carzaniga, and A. Wolf. An experience in evaluating publish/subscribe services in a wireless network. In *WOSP '02: Proceedings of the 3rd International Workshop on Software and Performance*, pages 128–133, New York, NY, USA, 2002. ACM Press.

[23] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. Technical Report CU-CS-944-03, Department of Computer Science, University of Colorado, Jan. 2003.

[24] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, Dec. 2003.

[25] M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of clients mobility in the Siena publish/subscribe middleware. Technical report, Department of Computer Science, University of Colorado, Oct. 2002.

[26] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[27] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.

[28] A. Carzaniga, J. Deng, and A. L. Wolf. Fast forwarding for content-based networking. Technical Report CU-CS-922-01, Department of Computer Science, University of Colorado, Nov. 2001.

[29] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, Oct. 1999. Revised May 2000.

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

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

[32] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, Mar. 2004. IEEE.

[33] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In B. König-Ries, K. Makki, S. A. M. Makki, N. Pissinou, and P. Scheuermann, editors, *Infrastructure for Mobile and Wireless Systems*, volume 2538 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2001.

[34] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proceedings of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, Aug. 2003.

[35] C. Y. Chan, W. Fan, P. Felber, M. N. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *VLDB*, pages 826–837, 2002.

[36] L. Chen, A. Gupta, and M. E. Kurul. Efficient algorithms for pattern matching on directed acyclic graphs. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 384–385, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Y. Chen, K. Schwan, and D. Zhou. Opportunistic channels: Mobility-aware event delivery. In Endler and Schmidt [50], pages 182–201.

[38] S. Chugh, S. Dharia, and D. P. Agrawal. An energy efficient collaborative framework for event notification in wireless sensor networks. In *28th Annual IEEE International Conference on Local Computer Networks*, pages 430–439, 2003.

[39] G. Colouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, Boston, Massachusetts, 2nd edition, 1994.

[40] T. H. Cormen, C. E. Leiserson, and T. L. Rivest. *Introduction to Algorithms*. The MIT Press, 2001.

[41] S. Courtenage and S. Williams. Automatic hyperlink creation using P2P and publish/subscribe. In *Workshop on Peer-to-Peer and Agent Infrastructures for Knowledge Management (PAIKM)*, Kaiserslautern, Germany, Apr. 2005.

[42] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE Trans. Knowl. Data Eng.*, 15(1):174–191, 2003.

[43] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th international conference on Software engineering*, pages 261–270. IEEE Computer Society, 1998.

[44] G. Cugola, E. Di Nitto, and G. P. Picco. Content-based dispatching in a mobile environment. In *Workshop su Sistemi Distribuiti: Algorithmi, Architectture e Linguaggi*, Sept. 2000.

[45] G. Cugola, D. Frey, A. L. Murphy, and G. P. Picco. Minimizing the reconfiguration overhead in content-based publish-subscribe. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1134–1140. ACM Press, 2004.

[46] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4), Oct. 2002.

[47] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, Feb. 2001.

[48] C. Doulkeridis and M. Vazirgiannis. Querying and updating a context-aware service directory in mobile environments. In *Web Intelligence*, pages 562–565. IEEE Computer Society, 2004.

[49] S. Duarte, J. L. Martins, H. J. Domingos, and N. Preguia. DEEDS - a Distributed and Extensible Event Dissemination Service. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS)*, Forli, Italy, 2001.

[50] M. Endler and D. C. Schmidt, editors. *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*, volume 2672 of *Lecture Notes in Computer Science*. Springer, 2003.

[51] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[52] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In T. Sellis and S. Mehrotra, editors, *Proceedings of the 20th Intl. Conference on Management of Data (SIGMOD 2001)*, pages 115–126, Santa Barbara, CA, USA, 2001.

[53] L. Fiege, F. C. Gärtner, S. B. Handurukande, and A. Zeidler. Dealing with uncertainty in mobile publish/subscribe middleware. In *1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 03)*, Rio de Janeiro, Brazil, 2003.

[54] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In Endler and Schmidt [50], pages 103–122.

[55] L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.

[56] L. Fiege, A. Zeidler, A. Buchmann, R. Kilian-Kehr, and G. Mühl. Security aspects in publish/subscribe systems. In *Third Intl. Workshop on Distributed Event-based Systems (DEBS'04)*, Edinburgh, Scotland, UK, May 2004.

[57] G. Fox and S. Pallickara. The Narada Event Brokering System: Overview and Extensions. In H. Arabnia, editor, *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, pages 353–359. CSREA Press, 2002.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[59] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 179–190. ACM Press, 2003.

[60] S. Gupta, J. Hartkopf, and S. Ramaswamy. Event notifier, a pattern for event notification. *Java Report*, 3(7):19–36, July 1998.

[61] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[62] L. D. Harvel, L. Liu, G. D. Abowd, Y.-X. Lim, C. Scheibe, and C. Chatham. Context cube: Flexible and effective manipulation of sensed context data. In A. Ferscha and F. Mattern, editors, *Pervasive*, volume 3001 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2004.

[63] R. Hayton, J. Bacon, J. Bates, and K. Moody. Using events to build large scale distributed applications. In *Proceedings of the 7th ACM SIGOPS European Workshop on Systems support for worldwide applications*, Sept. 1996.

[64] C. K. Hess and R. H. Campbell. A context-aware data management system for ubiquitous computing applications. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA, 2003. IEEE Computer Society.

[65] A. Hinze and S. Bittner. Efficient distribution-based event filtering. In J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, and G. Mühl, editors, *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2002.

[66] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile enviroment. In *Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 27–34. ACM Press, 2001.

[67] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10(6):643–652, 2004.

[68] IBM. *Gryphon: Publish/subscribe over public networks*, Dec. 2002. (White paper) `http://researchweb.watson.ibm.com/distributedmessaging/gryphon.html`.

[69] H. A. Jacobsen, G. Ashayer, and H. Leung. Predicate matching and subscription matching in publish/subscribe systems. In J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, and G. Mühl, editors, *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2002.

[70] D. Johnson, C. Perkins, and J. Arkko. *Mobility Support in IPv6*. IETF, June 2004. [Standards Track RFC 3775].

[71] J. Kangasharju, T. Lindholm, and S. Tarkoma. Requirements and design for XML messaging in the mobile environment. In N. Anerousis and G. Kormentzas, editors, *Second International Workshop on Next Generation Networking Middleware*, pages 29–36, May 2005.

[72] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A publish & subscribe architecture for distributed metadata management. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, 2002.

[73] S. Khungar and J. Riekki. A context based storage system for mobile computing applications. *Mobile Computing and Communications Review*, 9(1):64–68, 2005.

[74] A. Kiani and N. Shiri. Containment of conjunctive queries with arithmetic expressions. In Meersman et al. [86], pages 439–452.

[75] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.*, 65(1):150–167, 2002.

[76] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 798–807, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[77] G. J. Lee, T. Naganuma, and S. Kurakake. Efficient matching in a context-aware event notification system for mobile users. In *Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS'05)*, 2005.

[78] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. Hitting the distributed computing sweet spot with TSpaces. *Comput. Networks*, 35(4):457–472, 2001.

[79] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS*, pages 447–457. IEEE Computer Society, 2005.

[80] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 331–339, New York, NY, USA, 2003. ACM Press.

[81] T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 93–97. ACM Press, 2003.

[82] T. Lindholm, J. Kangasharju, and S. Tarkoma. A hybrid approach to optimistic file system directory tree synchronization. In V. Kumar, A. B. Zaslavsky, U. Çetintemel, and A. Labrinidis, editors, *MobiDE*, pages 49–56. ACM, 2005.

[83] H. Liu and H.-A. Jacobsen. A-TOPSS – a publish/subscribe system supporting approximate matching. In *Procedings of the 28th VLDB Conference*, Hong Kong, China, 2002.

[84] S. W. Loke, A. Padovitz, and A. B. Zaslavsky. Context-based addressing: The concept and an implementation for large-scale mobile agent systems. In Stefani et al. [123], pages 274–284.

[85] A. Medina, A. Lakhina, I. Matta, and J. W. Byers. Brite: An approach to universal topology generation. In *MASCOTS*. IEEE Computer Society, 2001.

[86] R. Meersman, Z. Tari, M.-S. Hacid, J. Mylopoulos, B. Pernici, Ö. Babaoglu, H.-A. Jacobsen, J. P. Loyall, M. Kifer, and S. Spaccapietra, editors. *On the Move to Meaningful Internet Systems 2005:*

CoopIS, DOA, and ODBASE, OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings, Part I, volume 3760 of *Lecture Notes in Computer Science*. Springer, 2005.

[87] R. Meier and V. Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In Stefani et al. [123], pages 285–296.

[88] C. Mitidieri and J. Kaiser. Attribute-based filtering: Improving the expressiveness while keeping the predictability in P/S systems. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003.

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

[90] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, September 2002.

[91] G. Mühl and L. Fiege. Supporting covering and merging in content-based publish/subscribe systems: Beyond name/value pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.

[92] G. Mühl, L. Fiege, F. C. Gärtner, and A. P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In A. Boukerche, S. K. Das, and S. Majumdar, editors, *The Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, pages 167–176, Fort Worth, TX, USA, October 2002. IEEE Press.

[93] G. Mühl, A. Ulbrich, K. Herrmann, and T. Weis. Disseminating information to mobile clients using publish/subscribe. *IEEE Internet Computing*, pages 46–53, May 2004.

[94] A. Murphy, G. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.

[95] A. L. Murphy and G. P. Picco. Using coordination middleware for location-aware computing: A lime case study. In R. D. Nicola, G. L. Ferrari, and G. Meredith, editors, *COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2004.

[96] V. Muthusamy, M. Petrovic, D. Gao, and H.-A. Jacobsen. Publisher mobility in distributed publish/subscribe systems. In J. Dingel and R. Strom, editors, *4th Intl. Workshop on Distributed Event-Based Systems (DEBS'05)*, pages 421–427, Columbus, Ohio, USA, June 2005. IEEE Press.

[97] V. Muthusamy, M. Petrovic, and H.-A. Jacobsen. Effects of routing computations in content-based routing networks with mobile data sources. In *MobiCom '05: Proceedings of the 11th annual international conference on Mobile computing and networking*, pages 103–116, New York, NY, USA, 2005. ACM Press.

[98] E. Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Inf. Process. Lett.*, 52(4):207–213, 1994.

[99] Object Management Group. *CORBA Event Service Specification v.1.1*, Mar. 2001.

[100] Object Management Group. *CORBA Notification Service Specification v.1.0*, Mar. 2001.

[101] Object Management Group. *Management of Event Domains Specification*, June 2001. `http://www.omg.org/cgi-bin/doc?formal/2001-06-03`.

[102] Object Management Group. *Wireless Access and Terminal Mobility in CORBA v.1.1*, Apr. 2004.

[103] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 185–207, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[104] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In O. Etzion and P. Scheuermann, editors, *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*, volume 1901 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 2000.

[105] C. Perkins. *IP Mobility Support for IPv4*. IETF, Aug. 2002. [Standards Track RFC 3344].

[106] L. Perrochon, E. Jang, S. Kasriel, and D. C. Luckham. Enlisting event patterns for cyber battlefield awareness. In *DARPA Information Survivability Conference and Exposition (DISCEX'00)*. IEEE Computer Society Press, Jan. 2000.

[107] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPSS: fast filtering of graph-based metadata. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 539–547, New York, NY, USA, 2005. ACM Press.

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

[109] P. R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.

[110] I. Podnar, M. Hauswirth, and M. Jazayeri. Mobile push: Delivering content to mobile users. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 563–570. IEEE Computer Society, 2002.

[111] I. Podnar and I. Lovrek. Supporting mobility with persistent notifications in publish/subscribe systems. In *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, Edinburgh, Scotland, UK, May 2004.

[112] K. Raatikainen, H. B. Christensen, and T. Nakajima. Application requirements for middleware for mobile and pervasive systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):16–24, oct 2002.

[113] S. Roberts and J. Byous. Distributed events in Jini technology, 2001. http://java.sun.com/developer/technicalArticles/jini/JiniEvents/.

[114] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[115] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[116] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In J. Crowcroft and M. Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2001.

[117] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 434–441, New York, NY, USA, 1999. ACM Press.

[118] R. Sarvas, E. Herrarte, A. Wilhelm, and M. Davis. Metadata creation system for mobile images. In *MobiSys*. USENIX, 2004.

[119] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[120] H. Schulzrinne and E. Wedlund. Application-layer mobility using SIP. *SIGMOBILE Mob. Comput. Commun. Rev.*, 4(3):47–57, 2000.

[121] W. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia*, 1997. http://elvin.dstc.edu.au/doc/papers/auug97/AUUG97.html.

[122] N. Skarmeas and K. Clark. Content-based routing as the basis for intra-agent communication. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555 of *LNAI*, pages 345–362, Berlin, July 04–07 1999. Springer.

[123] J.-B. Stefani, I. M. Demeure, and D. Hagimont, editors. *Distributed Applications and Interoperable Systems, 4th IFIP WG6.1 International Conference, DAIS 2003, Paris, France, November 17-21, 2003, Proceedings*, volume 2893 of *Lecture Notes in Computer Science*. Springer, 2003.

[124] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Computer Communication Review*, 31(4):149–160, Oct. 2001.

[125] R. E. Strom, G. Banavar, T. D. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. C. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. *CoRR*, cs.DC/9810019, 1998.

[126] Sun Microsystems. *Java Message Service Specification*, June 2001.

[127] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness-transparent information delivery for mobile and invisible computing. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 277, Washington, DC, USA, 2001. IEEE Computer Society.

[128] S. Tarkoma. Distributed event dissemination for ubiquitous agents. In *Tenth ISPE International Conference on Concurrent Engineering*, pages 105–110, July 2003.

[129] S. Tarkoma. Event dissemination service for pervasive computing. In *Pervasive 2004 Doctoral Colloquium*, pages 155–160, Apr. 2004.

[130] S. Tarkoma, R. Balu, J. Kangasharju, M. Komu, M. Kousa, T. Lindholm, M. Mäkelä, M. Saaresto, K. Slavov, and K. Raatikainen. State of the art in enablers for applications in future mobile wireless internet. HIIT Publication 2004-2, Helsinki Institute for Information Technology, Helsinki, Finland, Sept. 2004.

[131] S. Tarkoma and J. Kangasharju. A data structure for content-based routing. In M. H. Hamza, editor, *Ninth IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 95–100. ACTA Press, Feb. 2005.

[132] S. Tarkoma and J. Kangasharju. Filter merging for efficient information dissemination. In Meersman et al. [86], pages 274–291.

[133] S. Tarkoma and J. Kangasharju. Handover cost and mobility-safety of content streams. In *Eighth ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Oct. 2005.

[134] S. Tarkoma and J. Kangasharju. Mobility and completeness in publish/subscribe topologies. In M. H. Hamza, P. Prapinmonkolkarn, and T. Angkaew, editors, *IASTED International Conference on Networks and Communication Systems*. ACTA Press, Apr. 2005.

[135] S. Tarkoma, J. Kangasharju, and K. Raatikainen. Client mobility in rendezvous-notify. In *Intl. Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[136] S. Tarkoma, M. Laukkanen, and K. Raatikainen. Software agents for ubiquitous computing. In R. Khosla, N. Ichalkaranje, and L. Jain, editors, *Design of Intelligent Multi-Agent Systems: Human-Centredness, Architectures, Learning and Adaptation*, volume 162 of *Studies in Fuzziness and Soft Computing*, chapter 2, pages 31–62. Springer-Verlag, 2005.

[137] S. Tarkoma, T. Lindholm, and J. Kangasharju. Collection and object synchronization based on context information. In T. Magedanz, A. Karmouch, S. Pierre, and I. S. Venieris, editors, *MATA*, volume 3744 of *Lecture Notes in Computer Science*, pages 240–251. Springer, 2005.

[138] P. Triantafillou and A. Economides. Subscription summaries for scalability and efficiency in publish/subscribe systems. In J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, and G. Mühl, editors, *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2002.

[139] P. Triantafillou and A. Economides. Subscription summarization: A new paradigm for efficient publish/subscribe systems. In *ICDCS*, pages 562–571. IEEE Computer Society, 2004.

[140] University of Colorado. Siena Java language API and server code, 2005.

[141] T. Urnes, A. S. Hatlen, P. S. Malm, and Ø. Myhre. Building distributed context-aware applications. *Personal and Ubiquitous Computing*, 5(1):38–41, 2001.

[142] A. Virgillito, R. Beraldi, and R. Baldoni. On event routing in content-based publish/subscribe through dynamic networks. In *Proceedings of the Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, pages 322–328. IEEE, 2003.

[143] W3C. *XML Path Language (XPath) 1.0*, Nov. 1999. [Recommendation] `http://www.w3.org/TR/xpath`.

[144] W3C. *XQuery 1.0: AN XML Query Language*, 2005. [W3C Working Draft 04 April 2005] http://www.w3.org/TR/xquery/.

[145] Y.-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. Subscription partitioning and routing in content-based publish/subscribe networks. In D.Malkhi, editor, *Distributed algorithms*, volume 2508/2002 of *Lecture Notes in Computer Science*, Oct 2002.

[146] Y.-M. Wang, L. Qiu, C. Verbowski, D. Achlioptas, G. Das, and P. Larson. Summary-based routing for content-based event distribution networks. *SIGCOMM Comput. Commun. Rev.*, 34(5):59–74, 2004.

[147] H. Warren. A modification of warshall's algorithm for the transitive closure of binary relations. *Journal of the ACM*, 18(4):218–220, 1975.

[148] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[149] World Wide Web Consortium. *SOAP Version 1.2*, June 2003. [W3C Recommendation].

[150] *Technologies for the Wireless Future: Wireless World Research Forum*. John-Wiley & Sons, Oct. 2004.

[151] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for internet-scale event services. In *Proceedings of 8th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '99)*, pages 78–83, Palo Alto, CA, USA, 1999.

[152] A. Zeidler and L. Fiege. Mobility support with REBECA. In *ICDCS Workshops*. IEEE Computer Society, 2003.

[153] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: a fault-tolerant wide-area application infrastructure. *SIGCOMM Comput. Commun. Rev.*, 32(1):81–81, 2002.

[154] D. Zhou, Y. Chen, G. Eisenhauer, and K. Schwan. Active brokers and their runtime deployment in the ECho/JECho distributed event systems. In *Active Middleware Services*, pages 67–72. IEEE Computer Society, 2001.

Chapter A

Filter Merging Mechanism

This appendix presents the filter model and merging mechanism used in the experimentation. The merging mechanism source code is part of the publicly available Fuego Core middleware service set[1].

A.1 Filter Model

A filter is represented with a set of attribute filters, which are 3-tuples defined by <name, type, filter clause>. Name is an identifier and type is an element of the set of types. To simplify structural comparison of filters only conjunction may be used in defining the set of attribute filters. While disjunctions would give more power over the structure of the notification, they complicate the algorithms and, in any case, a disjunctive attribute filter set may be represented using a set of conjunctive attribute filter sets. The filter clause defines the constraints imposed by the attribute filter to a corresponding tuple in a notification identified by the name and type.

The attribute filter consists of atomic predicates and it may have various semantics. The simplest attribute filter contains only a single predicate. This kind of format is being used in many event systems, such as Siena and Rebeca. A more complex format supports conjuncts and disjuncts, but they complicate filter operations such as covering, overlapping, and merging.

We define the filter clause to support atomic predicates and disjunctions. In general, covering and overlapping of arbitrary filters in the disjunctive normal form may be determined using expression satisfiability. Expressions written in the disjunctive normal form are satisfiable if and only if at least

[1] Available at `http://hoslab.cs.helsinki.fi/homepages/fuego-core/`

Table A.1: Predicate list for the filter clause.

Predicate	Description	String	Number
Equ	Exact matching of any type	X	X
Neq	Not equal to given value	X	X
In	Substring within string value	X	
Nin	Subtring not within string value	X	
Stw	Starts with the given string	X	
End	Ends with the given string	X	
Exists	Tuple exists in notification	X	X
Gr/Gre	Value $>/\geq$ to given value		X
Lt/Lte	Value $</\leq$ to given value		X
Rng	Range test $[a, b]$		X
$name	Placeholder for the value of the tuple *name*	X	X

one of those disjunctive terms is satisfiable. A term is satisfiable if there are no contradictions [7].

Table A.1 presents the predicates for integer and string types. All predicates are unary except the range predicate, which is binary. The predicates consist of basic string, and number matching, and value selection for supporting placeholders.

A.2 Covering

Covering relations exist between four different components: predicates, disjuncts, attribute filters, and filters. Filter covering is an important part of the framework and it is used by the poset-derived forest data structure for determining the relationship between filters and the merging algorithm to remove redundancy from attribute filters.

Filter covering can be applied to also to filter sets. Given two filter sets, S_1 and S_2, S_1 covers S_2 when each filter in S_2 is covered by a filter in S_1.

The covering test for filters is implemented by counting the number of covered attribute filters. In the following, the precondition for covering for filters and attribute filters is that the names and types of the objects are identical. A filter B is covered by the filter A if and only if for all attribute filters A_i in A exists an attribute filter B_j in B that is covered by A_i [90]:

$$\forall i \exists j A_i \sqsupseteq B_j \Leftrightarrow A \sqsupseteq B. \tag{A.1}$$

Figure A.1 presents the covering relations for integer predicates. Each cell gives the condition for $F1 \sqsupseteq F2$. If a placeholder ($value) is used, it must be equal in both predicates, because the value cannot be determined at the time of performing the covering check. The following notation is used: $F1$ and $F2$ denote the values of unary input predicates. If a predicate is binary, the values are given by $F1.x$ and $F1.y$ and similarly for $F2$. The binary range predicate is also written using $rng(x, y)$. The sentence "$F1$ is in $rng(x, y)$" denotes that the single value of the unary predicate $F1$ is contained in the range $[x, y]$ of the binary predicate.

F2 / F1	Equ	Neq	Gr	Lt	Gre	Lte	Rng	Exists
Equ	F1=F2							
Neq	F1≠F2	F2=F1	F1≤F2	F1≥F2	F1<F2	F1>F2	F1 not in rng(x,y)	
Gr	F1 < F2		F1≤F2		F1<F2		F1 < F2.x	
Lt	F1 > F2			F1≥F2		F1>F2	F1 > F2.y	
Gre	F1≤F2		F1≤F2		F1≤F2		F1 ≤ F2.x	
Lte	F1≥F2			F1≥F2		F1≥F2	F1 ≥ F2.y	
Rng	F2 in rng(x,y)						F2.x ≥ F1.x and F2.y ≤ F1.y	
Exists	True	True	True	True	True	True	True	True

Figure A.1: Covering relations for integer predicates.

The covering test for an attribute filter is similar to the covering test for filters and conjunctive formulas presented in [90]. Theorem A.1 presents covering for disjunctive attribute filters. A pre-requirement is that filter A has the same name and type as B. The other direction requires a more complicated mechanism and proof. It is envisaged that this implication is still useful for simple and efficient operation. By using Theorem A.1 it is possible that not all relations are captured, but if used in a consistent manner it does not alter routing semantics and provides an efficient way to compute covering relations.

Theorem A.1 *Let* $A = \bigvee_{j=1}^{m} A_j$ *and* $B = \bigvee_{i=1}^{n} B_i$*, where the* A_j*'s and* B_i*'s are predicates. If* $\forall i \exists j A_j \sqsupseteq B_i$ *then* $A \sqsupseteq B$*.*

Proof. Assume $\forall i \exists j A_j \sqsupseteq B_i$. Then, when B is true, some B_i is true. But by assumption there then exists some true A_j, which means that A is true. By definition of the covering relation we then have $A \sqsupseteq B$. □

A.3 Overlapping

In order to determine whether two filters overlap, we need to determine the overlap between the parts of the two input filters: predicates, disjuncts, and attribute filters. The overlapping relation, $\simeq$, is reflexive, symmetric, and non-transitive. As in the case of covering relations, the overlap is defined in atomic predicates, which are the building block of more complex filters.

Techniques for determining overlap between single predicate attribute filters are given in [27, 90] and overlap detection for filters in the disjunctive normal form is discussed in [7].

A.4 Attribute Filter Merging

Attribute filter merging is based on two mechanisms: covering relations between disjuncts and perfect merging rules for atomic predicates. Covering relations are used to remove any covered disjuncts. Perfect merging rules are divided into two categories: existence tests and predicate modification. The former combines two input predicates into an existence test when the merging condition is satisfied. The latter combines two predicates into a single predicate when the condition is satisfied. A merged disjunct may cover other disjuncts, which requires that the other disjuncts are checked for covering after a merge.

The *merge(D_1,D_2)* operation for two disjuncts is defined as follows: first, if $D_1 \sqsupseteq D_2$ or $D_2 \sqsupseteq D_1$ the covered disjunct is removed, if $D_1 \equiv D_2$ either one of them is selected, or if they are incomparable the applicability of a merging rule is tested. If a merging rule is applied the merger is the result.

We assume that the length of a disjunct is not important — if a cost function is associated with the selection of the disjuncts to be merged and covered, the computational complexity of merging may not necessarily be polynomial.

The merging rules for the number existence test are presented in Table A.2. The existence test conditions are given for two input filters, F_1 and F_2. The condition must be satisfied in order for the two input filters to merge to an existence test. Each filter has a predicate and an associated constant denoted by a_1 and a_2, respectively. Ranges are denoted by two constants a and b. For example, given $x < a_1$ and $x \neq a_2$ where $a_1 = 10$ and $a_2 = 7$ the condition $a_1 > a_2$ is satisfied and results in an existence test. If a_2 is greater or equal to a_1 the condition is no longer satisfied.

Table A.2: The rules for number existence test.

F_1	F_2	Condition	F_1	F_2	Condition
$x = a_1$	$x \neq a_2$	$a_1 = a_2$	$x \neq a_1$	$x = a_2$	$a_1 = a_2$
$x < a_1$	$x > a_2$	$a_1 > a_2$	$x < a_1$	$x \geq a_2$	$a_1 \geq a_2$
$x \leq a_1$	$x > a_2$	$a_1 \geq a_2$	$x \leq a_1$	$x \geq a_2$	$a_1 \geq a_2$
$x > a_1$	$x < a_2$	$a_1 < a_2$	$x > a_1$	$x \leq a_2$	$a_1 \leq a_2$
$x \geq a_1$	$x < a_2$	$a_1 \leq a_2$	$x \geq a_1$	$x \leq a_2$	$a_1 \leq a_2$
$x \neq a_1$	$x \neq a_2$	$a_1 \neq a_2$	$x \neq a_1$	$x > a_2$	$a_1 > a_2$
$x \neq a_1$	$x \geq a_2$	$a_1 \geq a_2$	$x \neq a_1$	$x \leq a_2$	$a_1 \leq a_2$
$x \neq a_1$	$x < a_2$	$a_1 < a_2$	$x > a_1$	$x \neq a_2$	$a_1 < a_2$
$x \geq a_1$	$x \neq a_2$	$a_1 \leq a_2$	$x < a_1$	$x \neq a_2$	$a_1 > a_2$
$x \leq a_1$	$x \neq a_2$	$a_1 \geq a_2$	$x \neq a_1$	$x \in [a, b]$	$a_1 \in [a, b]$
$x \in [a, b]$	$x \neq a_1$	$a_1 \in [a, b]$			

A.5 Perfect Merging

The perfect merging algorithm merges two filters that have identical attribute filters except for one pair of distinctive attribute filters. These distinctive attribute filters are then merged. Perfect merging is based on Definition A.2. For disjunctive attribute filters the distinctive attribute filters are always mergeable. Conjunctive attribute filters are not necessarily mergeable. For any two mergeable filters F_1 and F_2 the operation $merge(F_1, F_2)$ either merges the distinctive attribute filters or the filters are identical.

Definition A.2 *Perfect merging may be performed if and only if at least* $n - 1$ *conjuncts are identical. Perfect merging is based on the tautology:* $(C_1 \wedge .. \wedge C_n \wedge B) \vee (C_1 \wedge .. \wedge C_n \wedge D) \Leftrightarrow C_1 \wedge .. \wedge C_n \wedge (B \vee D)$*, where all* C_i*,* B*, and* D *are conjuncts. If* B *and* D *are mergeable:* $C_1 \wedge .. \wedge C_n \wedge (B \vee D) \Leftrightarrow C_1 \wedge .. \wedge C_n \wedge merge(B, D)$*.*

This type of merging is called perfect, because covering and perfect merging rules do not lose or add information. More formally, a merger M of filters $\{F_1, \ldots, F_n\}$ is perfect if and only if $N(M) = \{\bigcup_i N(F_i)\}$. Otherwise, the merger is called imperfect [7, 90].

The selection of the best merging candidate is an important part of the merging algorithm. First, the best filter must be located for merging. Second, the best attribute filter or filters within that filter must be selected for merging. Our current implementation selects the first mergeable

candidate; however, a more optimal merging algorithm would select the candidate that has the most general merging result, because in some cases merging may add complexity to a filter in the form of a disjunct. This latter behaviour reduces a filter's probability to merge with other filters in the future. Therefore a good candidate filter for merging is one that either has less predicates or disjuncts after the merge operation or the predicates and disjuncts are more general. In the best case, the merged filter will cover many previously uncovered filters.

A.6 Imperfect Merging

There are many ways to realize imperfect merging. We propose a simple imperfect merging mechanism that has a less strict mergeability condition than perfect merging: all filters that are structurally equivalent may be merged. Attribute filters are merged using the technique discussed in this section. This kind of approach does not require any selection process. Imperfect merging results in a number of false positives.

A.7 Discussion

The single predicate and set-based perfect merging approach presented in [89, 90] requires that all attribute filters are identical except for one pair of distinctive attribute filters. It is not always possible to merge simple constraints: for example the perfect merging of two ranges $[0, 20]$ and $[30, 40]$ is not possible using conjunctive or single predicate attribute filters. The presented approach is more expressive, because it supports disjunctions.