Efficient and Mobility-aware Content-based Routing Systems

Sasu Tarkoma

Licentiate Thesis
February 2005
Department of Computer Science
University of Helsinki
## Contents

### I Introduction

1 Introduction ........................................... 3
   1.1 Structure of the Thesis ................................. 6
   1.2 Contributions ........................................ 7
   1.3 Research History .................................... 7

2 Event Routing ........................................... 9
   2.1 Overview ............................................. 9
   2.2 Router Topologies .................................... 12
   2.3 Interest Propagation ................................ 13
   2.4 Definitions ......................................... 15
   2.5 Routing Decision ................................... 16
   2.6 Filtering and Merging ................................. 17
   2.7 Design Patterns ..................................... 19
   2.8 Multicast ............................................ 19

### II Posets and Forests: Towards Efficient Routing  ........................................... 21

3 Posets and Forests ....................................... 23
   3.1 Routing Tables ...................................... 23
   3.2 Siena Filters Poset .................................. 26
      3.2.1 Forwards Sets .................................. 28
      3.2.2 Poset Algorithm ................................. 30
      3.2.3 Useful Properties ............................... 31
   3.3 Poset-derived Forests ................................. 32
      3.3.1 Poset-derived Forest Data Structure .......... 32
      3.3.2 Coloured Poset-derived Forest ................. 35
      3.3.3 Non-redundant Coloured Forest ................. 39
   3.4 Discussion ........................................... 39
   3.5 Equivalence of Forests and Posets ................... 40
## CONTENTS

3.6 Advertisements ................................................. 43  
3.7 Poset-based Matching ......................................... 45  
3.8 Rate-control Using Posets ..................................... 47  

4 Merging  
4.1 Merging and Routing Tables .................................. 49  
4.2 Rules for Merging .............................................. 51   
  4.2.1 Mergeability Rules ........................................... 51  
  4.2.2 Local Merging Rules ......................................... 52  
  4.2.3 Aggregate Merging Rules .................................... 53  
4.3 Example Merging Mechanisms .................................. 53  
  4.3.1 A Local Merging Mechanism: Merging Forest .......... 53  
  4.3.2 A Generic Aggregate Mechanism ......................... 54  

5 Experimentation  
5.1 Workload Generator and the Environment .................... 57  
5.2 Benchmarks .................................................. 58  
5.3 Redundant Forests .......................................... 60  
5.4 Non-Redundant Forests ....................................... 62  
5.5 Forwards Sets ............................................... 63  
5.6 Merging Benchmarks ........................................... 64  
5.7 PosetBrowser ................................................ 66  
5.8 Discussion .................................................. 67  

III Mobility-aware Routing  

6 Mobility and Completeness  
6.1 Introduction ................................................ 73  
6.2 Formal Specification ......................................... 74  
6.3 Mobility-Safety ............................................. 77  
6.4 Background .................................................. 78  
6.5 Generic Mobility Support .................................... 80  
6.6 Acyclic Graphs with Advertisements ......................... 82   
  6.6.1 Overview .................................................. 82  
  6.6.2 Mobile Subscribers ......................................... 83  
  6.6.3 Mobile Publishers ......................................... 88  
6.7 Rendezvous-Point Models .................................... 91  
  6.7.1 Mobility-safety ............................................. 95  
  6.7.2 Incompleteness ............................................ 95  
  6.7.3 Communication Cost ..................................... 95
CONTENTS

6.8 Discussion ........................................... 96
6.9 Summary ............................................. 97

IV Conclusions ........................................... 99

7 Conclusions ........................................... 101

A Filter Merging Mechanism ................................ 119
A.1 Filter Model ........................................ 119
A.2 Covering ........................................... 120
A.3 Overlapping ......................................... 121
A.4 Attribute Filter Merging .............................. 122
A.5 Perfect Merging ...................................... 122
A.6 Imperfect Merging .................................... 124
A.7 Discussion .......................................... 124

B Example Scenario: Smart Office ......................... 125
## List of Figures

3.1 Routing table configurations. .............................................. 25
3.2 Adding support for efficient matchers. ............................... 26
3.3 Example of the forwards set in subscription. ......................... 29
3.4 Example of the forwards set in unsubscription. .................... 29
3.5 Filters poset and poset-derived forest. ............................... 33
3.6 Two cases when adding a filter to a coloured forest. .............. 37
3.7 Inserting a tree into another tree. .................................... 38
3.8 Unnecessary update using the forest. .................................. 41
3.9 Pseudocode for forest-based matching. .............................. 46
4.1 Merging extension for routing tables. ................................. 50
5.1 Add and add/remove scenario tests. ................................... 59
5.2 Add scenario with merging. ............................................. 60
5.3 Comparison of the poset and the forests with a variable number of filters. ................................................. 61
5.4 Comparison of the poset and the forests with a variable number of interfaces. ................................................ 62
5.5 Matching results for the poset and forest structures. ............. 63
5.6 Comparison of the poset and non-redundant forests with variable number of filters. ............................................. 64
5.7 Comparison of the poset and the non-redundant forests with variable number of interfaces. .................................. 65
5.8 Impact of merging on the forest and poset performance. Results for 3 interfaces. ................................................... 66
5.9 Impact of merging on the forest and poset performance. Results for 500 filters. ......................................................... 67
5.10 PosetBrowser with four different data structures. ................ 68
5.11 Filter generation in the PosetBrowser. ............................... 69
6.1 Move-in function in mobility support service  [19]. ............... 81
6.2 Subscription handover with a complete and incomplete topology. Coloured nodes represent the inactive path. . . . . . . . . 84
6.3 Advertisement handover with complete topology. Black arrows indicate subscriptions, grey arrow indicate advertisements. 89
6.4 Three cases for simultaneous pub/sub handovers. . . . . . . . 92
6.5 Example of the Hermes model. . . . . . . . . . . . . . . . . . 94
A.1 Covering relations for integer predicates. . . . . . . . . . . . . 121
B.1 Office demonstration GUI. . . . . . . . . . . . . . . . . . . . . 126
List of Tables

2.1 Infrastructure interface operations. . . . . . . . . . . . . . . . . 11
5.1 Data structures used in the experimentation. . . . . . . . . . . . 58
6.1 Cost structure for generic mobility. . . . . . . . . . . . . . . . . . 82
A.1 Predicate list for the filter clause. . . . . . . . . . . . . . . . . . 120
A.2 The rules for number existence test. . . . . . . . . . . . . . . . . 123
LIST OF TABLES
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 in mobile environments. Example applications are news, stock market [7] 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 [57], cyber battlefield awareness [79], Internet games [8], software agent communication [93], and automatic hyperlink creation [34].

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 [2] 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 [42, 84, 113]. 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 [14, 25, 41, 68, 72, 96, 118] are seen as good candidates for supporting distributed applications in dynamic and ubiquitous environments because they support decoupled, anonymous, and asynchronous one-to-many information dissemination [38, 83]. Event systems are widely used, because asynchronous messaging provides a flexible alternative to RPC (Remote Procedure Call) [32, 42]. In the general model of event notification, subscribers subscribe events by specifying their interests using filters. Event
producers publish events, or notifications, which are matched against active subscriptions. Event filtering or matching is used to deliver information to the proper set of subscribers [1, 16, 22, 24, 27, 28, 43, 47, 65, 93, 102].

Filtering is a central core functionality for realizing event-based systems and accurate content-delivery. The main motivation for filtering is to improve accuracy in information delivery by delivering only those messages that are interesting for a client of the system — the delivered messages must match a priori filters defined by the client. 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 [37, 53, 82, 83] working with event systems, such as Siena [25] and Rebeca [44, 66]. 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 that 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 [39] and the routing of events through dynamic networks [108] are emerging research topics. In addition, ad hoc environments require novel solutions for event dissemination. Sensor networks [31] and proximity-based notification [62] 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 [40]. 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 [11, 61, 86, 107]. The Context Toolkit defines a distributed infrastructure for hosting context widgets. The toolkit is used to provide applications with contextual information [89]. The GAIA system is used to manage heterogeneous sensors and support context reasoning [86]. Almost all context-aware
systems employ some kind of asynchronous communication abstraction, typically asynchronous events or tuple spaces [13, 20, 59, 69, 70]. Events support context-triggered actions [11], 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 [61]. Current research prototypes, such as Siena, Rebeca, and Elvin [92, 97], 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 [23]. 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 [26].

The main research questions in this thesis for efficient content-based routing 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?

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. Forest-based structures do not offer significantly better matching performance and additional data structures are needed for efficient 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 [6, 46, 65]. 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
CHAPTER 1. INTRODUCTION

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 [36]. The Siena mobility support service was formally verified to maintain safety and liveness [19]. 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 [45, 68]. 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 incompleteness of the routing topology affects these costs?

1.1 Structure of the Thesis

The thesis is structured into four 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.

The last part presents the conclusions.

1.2 Contributions

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

- The coloured 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.

- Characterization of pub/sub mobility using incompleteness 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. We also examine and analyze publisher mobility, which has not, at the time of writing, been addressed in event literature.

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 [101]. The general challenges of the wireless and mobile environment, especially for software agents, are discussed in [105].

The main influences and starting points for the research were Antonio Carzaniga’s and Gero Mühl’s Ph.D. dissertations [21, 65]. 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 frame-
work 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 [104]. 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 [99]. The filter covering, matching, and merging mechanisms were developed as basic building blocks for event systems. An outline of the mechanisms was presented in [100]. The poset-derived forest data structure is presented in [102]. An overview of chapter 6 of this thesis is presented in [103]. The author also contributed to the Wireless World Research Forum’s vision on adaptive computing [113].

The filter mechanisms presented in this thesis were used in the Fuego event system [100]. 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. Appendix B 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 and coloured forest were joint work with Jaakko Kangasharju from HIIT and Theorem 3.9 is by him. In addition, Proposition 6.6, Lemmas 6.7 and 6.8, and Theorem 6.9 were proposed by the author, but the final proofs are by Kangasharju.
Chapter 2

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 sub-
scribers and mediates event messages between them. Typically, an event router consists of two parts: a set of 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. 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 overhead 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
### 2.1. OVERVIEW

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 the event messages. The ultimate end-point of all event messages is a local subscriber resident on one router. 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 [23]. 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Sub(X,F) )</td>
<td>( X ) subscribes filter ( F )</td>
<td>Sub/Adv</td>
</tr>
<tr>
<td>( Pub(X,n) )</td>
<td>( X ) publishes notification ( n )</td>
<td>Sub/Adv</td>
</tr>
<tr>
<td>( Notify(X,n) )</td>
<td>( X ) is notified about notification ( n )</td>
<td>Sub/Adv</td>
</tr>
<tr>
<td>( Unsub(X,F) )</td>
<td>( X ) unsubscribes filter ( F )</td>
<td>Sub/Adv</td>
</tr>
<tr>
<td>( Adv(X,F) )</td>
<td>( X ) advertises filter ( F )</td>
<td>Adv</td>
</tr>
<tr>
<td>( Unadv(X,F) )</td>
<td>( X ) unadvertises filter ( F )</td>
<td>Adv</td>
</tr>
</tbody>
</table>
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 and they typically employ the same client-server protocol as the local clients. 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. 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 minimum spanning trees to be computed for distributing notifications 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 node, 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 6.7. 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.
The hierarchical topology was used in the JEDI system [10, 36], and an acyclic topology with advertisements in Rebeca [45, 65, 68]. The Siena project investigated and evaluated the topologies with different interest propagation mechanisms [21, 25]. In general, the acyclic and cyclic topologies have been found to be superior to hierarchical topologies [10, 21, 67]. The router topology in Gryphon [54, 95] 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 [39] and [108].

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 nodes and using a variant of prefix-routing to find the proper data node for a given data item. Hermes [81] and Scribe [88] 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 perform better than the acyclic topology [81].

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 [67], 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 [67]. 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 [10, 21, 67].

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 [68] and Hermes [80, 81]. 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 [26]. 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 [23] and later refined and extended in Rebeca [65]. 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. Typically, filtering languages are compositional in the sense that, for example, a filter is composed from subfilters, which are defined using predicates. Predicates are called atomic when they are the smallest possible unit of composition. 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 the filter $F_2$, denoted by $F_1 \supseteq F_2$, if and only if all notifications that are matched by $F_2$ are also matched by $F_1$, i.e. $N(F_1) \supseteq N(F_2)$. The $\supseteq$ relation is reflexive and transitive — and defines a partial order. The filter $F_1$ is equivalent to $F_2$, written $F_1 \equiv F_2$, if $F_1 \supseteq F_2$ and $F_2 \supseteq F_1$.

A set of $n$ filters $S_F = \{F_1, \ldots, F_n\}$ covers a filter $F_k$ if and only if $N(S_F) \supseteq N(F_k) \iff \{\bigcup_i^n N(F_i)\} \supseteq N(F_k)$. 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 $F_1 \supseteq F_2$, since the range $[5, 9]$ is contained in $x < 10$. We have $F_1 \not\supseteq F_3$, 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 [101]. 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 [112] supports header-based routing of XML-messages. Finally, content-based systems use the whole content of the message in making the decision [24].

**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 [76]. 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 [27, 38] and context-sensitive messaging [61]. The latter mechanism formulates the current and future context of entities as event filters and subscribes to them. The Elvin event broker [92] 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 [61].

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) [72]. The Java Messaging Service (JMS) supports a subset of SQL-92 for event filtering [96]. 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 [36], Elvin [97], Rebeca [66], Gryphon [54], and Siena [25] 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 mergeable filters were presented in [65]. 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 [28]. In general, filter matching is done by counting attributes using the counting algorithm [22, 28, 63, 65, 77], counting and clustering [55], using a tree-based data structure [1], or a binary decision diagram [15]. Fast matching algorithms combine client-side processing, caching, and filter clustering [43].

Many matching mechanisms do not take the distribution and selectivity of filters into account. Efficient selectivity-based filtering has been examined in [52]. 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 [60]. The Elvin [97] 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 [109]
and XQuery [110], 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 [49]. 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 [29].

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 [65]. 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 [65, 67]. 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 [35]. Subscription partitioning and routing in content-based systems have been investigated in [111, 114] using Bloom filters [9] 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 [106]. 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.
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 [90]. 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 [71]. 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 [90].

The three well-known patterns for event notification are: the observer pattern [48], the event-channel pattern, and the notifier pattern [50]. 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 [85]. The publish-register-notify, a pattern similar to the observer pattern, is used in the Cambridge Event Architecture [5, 51].

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 [115]. The event channel pattern is used in the CORBA Event Service [71] and Notification Service [72]. A separate specification defines how CORBA event channels are connected to form communication topologies [73].

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 [76]. 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 [54].
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 [36] and Hermes [81] keep filters in a simple table, and Rebeca uses sets and a counting algorithm for finding covering filters and mergeable filters [65].

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 [26].

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. PF may have redundant filters. Emphasis is on very fast additions, deletions, and computation of the non-covered set. The main usage scenario for PF is the management of filters from local clients and border/edge routers.

**Coloured-poset Derived Forest (CF)** Similar to PF, but prunes covered filters from the same interface locally. The subscribing interface is called a *colour*. The main usage scenario for CF is the management of filters from local clients and border/edge routers.

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

**Balancing** Both CF and NRCF may be extended using an index structure to optimize processing. When this extension is used the structure is called *balanced*.

In Chapter 4 we also discuss how filter merging techniques are integrated with the routing table structures to further improve processing efficiency. Figure 3.1 presents three useful routing table configurations that combine FP, CF, and NRCF. 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. Subfigure I of Figure 3.1 presents an example of this by implementing the external table using FP and the local table using CF. The term NB in the figure denotes a neighbour interface. The CF is used to maintain client subscriptions (and advertisements) and update FP only
3.1. ROUTING TABLES

when the root set of the CF changes. The Siena system used FP to store also 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. Subfigure II illustrates the use of NRCF 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 FP. Subfigure III illustrates how NRCF is used for hierarchical routing with the master and slave interfaces identified.

Figure 3.1: Routing table configurations.

Figure 3.2 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 CF and also reflected to the efficient matcher. Only the root set is updated to FP, the external routing structure. When an incoming event matches the local interface (root filters of CF), the notification is sent to the efficient matcher. This design may be used in various configurations and it is also compatible with the merging mechanisms discussed later in the thesis.

In addition to hierarchical and peer-to-peer routing, the new data structures may be used to enhance rendezvous-based routing models, such as Hermes. The Hermes routing model is presented in more detail in Section 6.7. 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
IV. Peer-to-peer with matching

Figure 3.2: Adding support for efficient matchers.

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 data structure configuration must support this.

3.2 Siena Filters Poset

The filters poset data structure was used in the Siena distributed event system for maintaining covering relations between filters [23]. In Siena peer-to-peer architectures the poset stores additional information for each subscription that is inserted into the poset. The $\text{subscribers}(f)$ set gives the set of subscribers for the given subscription filter $f$, and similarly, $\text{forwards}(f)$ contains the subset of peers to which $f$ needs to be sent. A subscription $\text{subscribe}(X,f)$ where $X$ is the subscriber and $f$ is the filter representing the subscription proceeds as follows [23]:

1. If a filter $f'$ is found for which $f' \sqsupseteq f$ and $X \in \text{subscribers}(f')$ then the procedure terminates, because $f$ for $X$ has already been subscribed by...
2. If a filter $f'$ is found for which $f' \equiv f$ and $X \notin \text{subscribers}(f')$ then $X$ is 
added to $\text{subscribers}(f')$. 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 $\text{subscribers}(f)$. The server 
removes $X$ from all subscriptions covered by $f$ and subscriptions with 
no subscribers are also removed.

In distributed operation the Siena server defines the set $\text{forwards}(f)$ as 
presented in the equation

$$\text{forwards}(f) = \text{neighbours} - \text{NST}(f) - \bigcup_{f' \in P_s \land f' \sqsupseteq f} \text{forwards}(f'). \quad (3.1)$$

The $\text{neighbours}$ set contains the event brokers connected to the current 
broker (one application-level hop distance). $P_s$ denotes the subscription 
poset. The functor $\text{NST}$ (Not on any Spanning Tree) means that the prop-
gagation of $f$ must follow the computed spanning trees rooted at the original 
subscribers of $f$. With acyclic topologies $\text{NST}$ contains the neighbour that 
sent $f$. This means that $f$ is never forwarded towards the originator of $f$.

Cyclic topologies require the functor $\text{NST}$ to be computed. The $\text{NST}$ 
term is needed because there may be multiple paths connecting a subscriber 
to potential publishers. The propagation of a subscription $f$ must follow only 
the computed spanning trees rooted at the original subscribers of $f$. Well-
known techniques such as link-state or distance-vector routing algorithms 
are needed to compute the minimal spanning trees. The Siena papers do not 
give an algorithm for computing or using $\text{NST}$ in cyclic topologies. In this 
thesis, we focus on acyclic routing topologies. 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 interme-
diary 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 ap-
plied atomically by other servers. The $\text{unsubscribe}(X, f)$ removes $X$ from 
the subscribers set of all subscriptions that are covered by $f$. Filters with empty 
subscriber sets are removed. Algorithm 3.2 gives an outline of subscription 
processing.
Algorithm 3.2 Subscription semantics:

IncomingSub\((f, \text{source})\)

1. Add \((f, \text{source})\) to \(P_s\).
2. Forward subscription message using forwards\((f)\).

IncomingUnsub\((f, \text{source})\)

1. Remove \((f, \text{source})\) from \(P_s\).
2. Forward unsubscription using forwards\((f)\). 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 3.3 illustrates the use of the forwards set in subscription with three neighbour servers: \(I_1, I_2, \) and \(I_3\). 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. \(I_1\) sent filter \(a\) and therefore \(a\) is forwarded to \(I_2\) and \(I_3\) but not to \(I_1\). The third and fourth subscriptions need to be forwarded to \(I_1\) in order to avoid false negatives. Finally, the forwards set for the last subscription is empty, so it is not forwarded.

Figure 3.4 gives an example of an unsubscription operation. The first subscription of the previous example is removed. The subscription \((C, I_2)\) is uncovered and since it has only been forwarded to \(I_1\) it has to be sent also to \(I_3\) but not to \(I_2\) that originally sent it. The forwards set of the direct
3.2. SIENA FILTERS POSET

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.
correctness of set-based content-based pub/sub is discussed more formally in Section 6.2.

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 3.3** 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 [75]. The implementation supports hierarchical operation. The insert operation follows the rules presented in the beginning of this section. The relations between filters are maintained using two lookup structures: \( \text{predecessor}(f) \) and \( \text{successor}(f) \), where the former maintains a list of immediate predecessors of \( f \) and similarly the latter maintains the immediate successors of \( f \). The \( \text{predecessor}(f) \) covers \( f \), and similarly \( \text{successor}(f) \) is covered by \( f \).

The \text{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 \textit{add} operation is computationally heavy, because these sets need to be located. The set \( \text{predecessor}(f) \) 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 \textit{successor} set starts from the \( \text{predecessor}(f) \) 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 \( \text{predecessor}(f) \) and \( \text{successor}(f) \). 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 table.

### 3.2.3 Useful Properties

In this section we present and give proof for useful properties of the FP. We assume subscription semantics, but similar proofs may be constructed also for advertisement semantics due to Theorem 3.12. 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 \( I \). For example, the neighbours in Figure 3.1 are interfaces. We also say that the colour of \( F \) is \( I \). 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 3.4 states that no colour appears twice in any depth-first sequence in a poset. The algorithms for FP ensure that this property is maintained. Theorem 3.6 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. Theorem 3.7 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.

**Property 3.4 Minimum colour:** Given a node \( x \) with colour \( c \) in a poset or forest, \( x \) has no descendants with colour \( c \) and \( x \) has no colourless descendants.

**Proof sketch:** This follows from the definition of the filters poset in Section 3.2. The minimum colour property is maintained during additions by removing the new colour from all descendants of the new node. \(\square\)
Lemma 3.5 \textit{Covering relations exist only for filters from different interfaces in peer-to-peer routing when local clients are represented using a single interface (external client assumption).}

Proof: By definition, a router sends the filter $F_i$ to an interface $i$ only when it is not covered by an existing filter. Similarly, a filter is removed only when it has been previously sent. Therefore, the filters installed at a router represent the non-covered set of the router that sent them, and the filters cannot have covering relations between them. This is ensured by the minimum colour property (which holds both for coloured forests and posets). We have shown that there cannot be covering relations between filters from the same interface. Existence of covering relations between filters from different interfaces $i \neq j$ follows from the definitions of the poset and forest. \hfill \Box

Theorem 3.6 \textit{The poset or forest has a maximum depth of $k$ when local clients are represented using a single interface (external client assumption).}

Proof: According to the minimum colour property, each depth-first sequence of nodes starting from the root node has unique colours. Lemma 3.5 states that relations may only exist in filters from different interfaces. The maximum depth is achieved for linear order, and the depth is exactly $k$ when each node has a different colour. As an example, consider the sequence $(x > 0, i_1), (x > 5, i_2), (x > 10, i_3), (x > 12, i_4)$. \hfill \Box

Theorem 3.7 \textit{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 greater than or equal to $|\text{neighbours}| - 1$. 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 colour than the root filter because of Property 3.4. \hfill \Box

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.
3.3. POSET-DERIVED FORESTS

presented in the previous section with the exception that only one-to-one relationships are maintained (Figure 3.5). The forest is a generic data structure and may be extended with the sets subscribers(f), forwards(f), advertisers(a), and forwards(a).

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

**Definition 3.8** A pair \((F, \succ)\) is a poset-derived forest with base set \(F\), if

1. \(F\) is a finite set of filters and \(\succ\) is a relation between filters in \(F\).

2. For each \(a \in F\) there is at most one \(b \in F\) for which \(b \succ a\), i.e. \((F, \succ)\) is a forest with the relation \(\succ\) going from parent to child.

3. If \(a, b \in F\) and \(b \succ a\), then \(b \sqsupseteq a\).

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

\((F, \succ)\) is called *maximal in \(F\)* if there do not exist \(a, b \in F\) for which \((F, \succ \cup \{(a, b)\})\) is a poset-derived forest. \(\succ \cup \{(a, b)\}\) denotes a new set that contains both \((a, b)\) and \(\succ\). There may exist several maximal forests for the same base set, but these all contain the same number of trees, as Theorem 3.9 shows.

The forest can be used to easily compute the minimal cover for \(F\) (Theorem 3.11). Theorem 3.12 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.
Theorem 3.9  Every poset-derived forest maximal in $F$ contains the same number of trees. Furthermore, any poset-derived forest with base set $F$ and the same number of trees as a forest maximal in $F$ is itself maximal in $F$.

Proof:  Let $(F, \succ)$ and $(F, \succ')$ be poset-derived forests and let $(F, \succ)$ have more trees than $(F, \succ')$. Then there exists a $b \in F$ which is a root of a tree in $(F, \succ)$ but not in $(F, \succ')$, so there is an $a \in F$ for which $a \succ' b$. Since the forest edges are derived from a partial ordering, it is impossible for $a$ to be in the same tree as $b$ in $(F, \succ)$. Therefore we easily see that $(F, \succ \cup \{(a, b)\})$ is a poset-derived forest, and hence $(F, \succ)$ is not maximal in $F$.

Assume now that $(F, \succ)$ is not maximal in $F$ but contains the same number of trees as another forest maximal in $F$. There exist therefore $a, b \in F$ for which $(F, \succ \cup \{(a, b)\})$ is a poset-derived forest. But to preserve condition 2 it is necessary for $b$ to be a root of a tree and $a$ to be located in a different tree from $b$ in $(F, \succ)$. Therefore $(F, \succ \cup \{(a, b)\})$ is a poset-derived forest containing fewer trees than another forest maximal in $F$, which contradicts the first part of the theorem, since there exists a forest maximal in $F$, which is an extension of $(F, \succ \cup \{(a, b)\})$ and therefore contains at most the same number of trees as it.

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. In subsequent examination we assume that the forest is sibling-pure.

Definition 3.10  A poset-derived forest $(F, \succ)$ is sibling-pure at node $a$ (a may be the imaginary root) if there do not exist $b, c \in F$ 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.

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

Theorem 3.11  The set of root nodes of a maximal poset-derived forest with base set $F$ is the minimal cover for $F$.

Proof:  Let $G$ be the set of root nodes of $F$. This set certainly covers $F$, and by the definition of maximality no node can be removed without destroying this property. Hence the set of root nodes is the minimal cover for $F$.

For two sets of filters $F, G$ the overlap of $F$ with $G$ is defined to be the subset of those elements in $G$ which overlap with some element of $F$.  \[\square\]
Theorem 3.12 For any sets of filters $F, G$ the overlap of $F$ with $G$ is the same as the overlap of the minimal cover of $F$ with $G$.

Proof: Obviously the overlap of the minimal cover is contained in the overlap of the full set. Now let $g \in G$ belong to the overlap of $F$ with $G$. Then there exists an $f \in F$ for which $f \simeq g$, or $N(f) \cap N(g) \neq \emptyset$. Now there exists an $f'$ in the minimal cover of $F$ 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 $F$ with $G$. □

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 3.13.

Algorithm 3.13 Let $(F, \succ)$ be a poset-derived forest. It is assumed that there is an efficient way to find a node in $F$ based on its identifier. In subsequent examination, references to “larger” and “smaller” are to be taken with respect to the relation $\sqsubseteq$. We define the following algorithms with inputs $F$ and a filter $x$ and output a poset-derived forest:

add Set the current root to the imaginary root of $F$

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

del 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 with $r$ as the current root. In this an element of $C$ carries the whole subtree rooted at it with the addition.

3.3.2 Coloured Poset-derived Forest

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

**Definition 3.14** A triple \((\mathcal{F}, \succ, G)\) is a poset-derived coloured forest, 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 \(G(x) \neq \emptyset\) iff \(x \in \mathcal{F}\).

3. If \(x \in \mathcal{F}\) and \(k \in G(x)\), then \(k \notin G(y)\) holds for all descendants of \(x\) in the relation \(\succ\).

To satisfy Property 3.4 we extend the add and del operations accordingly. First, all the nodes with the same colour are examined for covering. Second, the colour of the inserted node is removed from the node’s descendants, and nodes that become colourless because of this are removed.

**Operations**

The add operation inserts a new filter \(x\) into the forest and if the colour \(c\) of the input filter is also new it is inserted to the set of colours. 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 \(G'(y) = G(y), y \neq x\). The basic function of add is the same as before, but the subtree to descend into is selected based on the colour.

The add operation is extended with the eliminate-colour procedure shown as Algorithm 3.15, which removes the colour of \(x\) from its descendants and prunes colourless nodes in order to maintain the minimum colour property.

The del operation is based on one or more add operations so it is sufficient to consider only add. In del the add operation may need to be performed for a subtree. This requires that the eliminate-colour has to remove a set of colours — from the root to the point of insertion. This eliminate-colour-set is a simple extension of the basic algorithm. The running time of add and del may be improved by keeping an index of colours under each node.

**Algorithm 3.15** eliminate-colour\((x, c)\):

1. If \(c \in G(x)\)

   Remove colour \(c\) from \(x\): \(G(x) \setminus \{c\}\).

   If \(G(x) = \emptyset\) remove \(x\) from \(\mathcal{F}\), add the children of \(x\) to \(x\)’s parent and flag them as processed.
3.3. **POSET-DERIVED FORESTS**

Return.

2. \( \forall u \in \text{children}(x): \text{if } \neg \text{flag}(u) \text{ call } \text{eliminate-colour}(u, c). \)

Figure 3.6 presents the two cases when inserting a filter to a coloured forest: first, when the input filter is covered by a node of the same colour, and when second the input filter is covered by a different colour and covers a filter of the same colour. In the latter case, the \( \text{eliminate-colour} \) is performed.

![Diagram of two cases when adding a filter to a coloured forest.](image)

**Figure 3.6:** Two cases when adding a filter to a coloured forest.

Figure 3.7 illustrates the insertion of a tree into another tree. This procedure is required for the set version of \( \text{add} \), which is used by the \( \text{del} \) procedure. Any subtree of an existing tree satisfies the minimum colour property. The algorithm records the colours of nodes that cover the inserted root-node \( D \). In this case, the colours of \( A \) and \( C \) must not appear in the subtree rooted at \( D \). The \( \text{eliminate-colour-set} \) removes nodes \( D \) and \( E \) to maintain the minimum colour property.

**Colour-based Balancing**

We are interested in balancing the coloured poset-derived forest in such a way that redundancy is removed. The forest may keep more nodes in memory than the Siena filters poset, because only one-to-one relations are stored. Colour-based balancing is a technique for optimizing the insertion cost of a filter. Each node maintains a set of colours used by its descendants. The \( \text{add} \) operation sorts the set representing the current level of the forest using the colour of the node to be inserted. Nodes that have descendants of the same colour are processed first. The idea is to cluster nodes with the same
colour to the same subtree. This strategy increases the probability that the new filter covers other filters with the same colour, or that the new filter is covered by an existing filter with the same colour.

We use an index presented in Definition 3.16 to optimize the performance of the data structure. For each filter in the structure the index keeps a record of the colours of its descendants. The index is updated for every addition and deletion. The add operation uses the index in deciding which subtree to traverse. A colour index entry is not necessarily needed for leaf nodes. The index requires at most \( n - 1 \) indices where \( n \) is the number of nodes in the forest and \( n \) bits per entry if each node has unique colour. 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 \( n \) bits for representing the colours, where \( n \) is the maximum number of colours in the forest.

**Definition 3.16** Colour-index(\( x \)): the input is the node \( x \) and the output is the set of colours of \( x \)'s siblings.

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. The eliminate-colour procedure must also update the index when colours are removed, but the index also allows to terminate the colour elimination for a subtree if the given colour is not found among the descendants. The algorithm for eliminate-colour is extended to support the colour index by adding the following line to the beginning of Algorithm 3.15 (before 1.): If \( c \not\in G(x) \) and \( c \not\in \text{colour-index}(x) \) then return.
We envisage that in the typical scenario the number of colours is significantly lower than the number of nodes, which improves the efficiency of the index. This is the case when every node is a root node with no descendants.

### 3.3.3 Non-redundant Coloured Forest

The non-redundant coloured poset-derived forest (NRCF) extends the CF with an *eliminate-colour-all* procedure during *add*. This procedure ensures that there are no redundant filters in the data structure. A redundant filter is such a \((F_1, i)\) that there exists a \((F_2, i)\) for which \(F_2 \sqsubseteq F_1\).

This extension results in performance loss, but may be optimized by using the colour index in balanced coloured forests. In this case, the *eliminate-colour-all* procedure checks only those parts of the forest that advertise the eliminated colour. Cover check needs to be performed only for filters with the input colour.

### 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 non-coloured 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. For balanced forests, the colour-index is used to order nodes based on the input colour.

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 coloured 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 to remove 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 CF, 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 CF: 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 colour.

NRCF: Redundant filters are removed during add.
3.5. EQUIVALENCE OF FORESTS AND POSETS

Peer-to-peer Routing  We employ a simplified model here, in which we assume that the first term of Equation 3.1 is the set *neighbours* and that the second NST term 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 coloured forest has two limitations when compared with the filters poset. First, it may have a number of redundant filters. This is solved by using NRCF. 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 3.8 gives an example of an unnecessary update. In this case, there are two possible places for inserting a subscription and the other is more favourable. This may result in an additional element in the *forwards* set. Example 3.17 deals with the problem of finding the set of subscriptions that are uncovered by an unsubscription.

![Figure 3.8: Unnecessary update using the forest.](image)

The situation is the same when advertisements are used, because advertisement forwarding is essentially the same and subscription forwarding uses the constrained *neighbours*, set. Since a root filter has the most overlap with any advertisements (Theorem 3.12) it will have the largest *neighbours*, set and hence the largest *forwards* set.

**Example 3.17** 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. $F^a_1$ and $F^c_2$ are root filters. $F^c_1$ and $F^a_2$ cover $F^a_3$ and $F^a_3$ covers...
When \( F^a_3 \) is unsubscribed, the forwards set of \( F^b_4 \) 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: \( F^c_1 \) covers \( F^a_3 \) and \( F^c_2 \) covers \( F^b_4 \). When \( F^c_3 \) is unsubscribed there are no covered elements and thus \( F^b_4 \) 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 NRCF 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 \(|\text{neighbours}| \) 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 4 may be used for both subscriptions and advertisements.

In advertisement semantics a second poset $P_a$ is used for advertisements [23]. The sets $\text{advertisers}(a)$ and $\text{forwards}(a)$ 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 $\text{neighbours}$, a set constrained by advertisements is used as presented by the equation

$$\text{neighbours}_s = \bigcup_{a \in T_A, a \in T_A} \text{advertisers}(a) \cap \text{neighbours}. \quad (3.18)$$

In this case, Equation 3.1 uses the $\text{neighbours}_s$ set instead of the $\text{neighbours}$ 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. Algorithm 3.19 gives an outline of advertisement processing. The algorithm is derived from [23] and [65].

**Algorithm 3.19 Advertisement semantics:**

**IncomingSub**(f, source)

1. Add (f, source) to $P_s$.
2. Calculate $\text{neighbours}_s$ using $P_a$ and Equation 3.18.
3. Send subscription message to forwards(f).

**IncomingUnsub(f, source)**

1. Remove (f, source) from $P_s$.

2. Forward unsubscription using forwards(f). 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.

**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 source that overlaps and send them to 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 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 to each neighbour in forwards(a). The 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 source that do not have an associated overlapping advertisement from some other neighbour.
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 [22, 28, 63, 65, 77]. The counter algorithm keeps track of the number of attribute filter matches for each filter. Counting is based on the 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 [33] 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 3.20 and 3.21).

**Property 3.20**  If a node $n_1$ matches a notification then all the predecessors of $n_1$ must also match the notification.

**Property 3.21**  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. The Siena poset-based matcher uses Property 3.21 in order to optimize matching. The pseudocode for the forest is given by Figure 3.7. The poset-based matcher is similar, but requires the testing of nodes that have been already visited.
CHAPTER 3. POSETS AND FORESTS

Match-Forest$(n)$

1. let $S$ be an empty sequence
2. let $FW$ be an initially empty set of forward interfaces
3. let $I_{\text{max}}$ be the # of interfaces for the event type
4. let $q = false$
5. 
6. $R = \text{Get-Roots}(n.\text{type})$
7. let $Im$ be an imaginary root of a tree
8. $Im.\text{children} = R$
9. $\text{ADDLAST}(S, Im)$
10. while $S$ is non-empty and not $q$
11. 
12. do
13. o = \text{REMOVEFIRST}(S)
14. while o has unprocessed children and not $q$
15. 
16. c = \text{NEXTCHILD}(o)
17. if $\text{subscribers}(c) \not\subseteq FW$ and $\text{MATCH}(c, n)$
18. 
19. \text{ADDLAST}(S, c)
20. \text{ADDTOSET}(FW, \text{subscribers}(c))
21. if $|FW| \geq I_{\text{max}}$
22. 
23. then
24. $q = true$
25. return $FW$

Figure 3.9: Pseudocode for forest-based matching.

The filter forest or poset supports also 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.

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 colour index to
prevent the processing of those subtrees in a balanced coloured forest, which
have been already matched. This is easily accomplished by simply checking
whether the colours (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 part of
filters [68]. 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) \((\sqsubseteq\) (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

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 used filtering language and routing data structure and do not depend on the mechanism that is used to perform the filter merging. 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 routing table entries.

4.1 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 \( \text{merge}(F_1,F_2) \) procedure exists that merges input filters \( F_1 \) and \( F_2 \) and returns a single merged filter \( F_M \) for which \( F_M \supseteq F_1 \cap F_M \supseteq F_2 \). 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 nodes minimum. Appendix A presents the filter merging mechanism that we use in experimentation, but the techniques we present in this chapter are not restricted to this mechanism. 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 \( \text{merge}(x,y) \), but after the delete 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.

Figure 4.1 presents two router configurations with filter merging and highlights the modular structure of content-based routers. Subfigure I illustrates filter merging in peer-to-peer routing. 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.

Subfigure II 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 coloured forest, are merged. Forest is superior to the filters poset in hierarchical operation, because the computation of the forwards sets is not needed.
4.2 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.

4.2.1 Mergeability Rules

Filters may be merged in two ways: local merging that is performed within the data structure and aggregate merging that is performed for exiting routing table entries only. The former is given by the local merging rule presented in Definition 4.1. 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 benefits also the local router.

Definition 4.1 Local merging rule: The operation $\text{merge}(F_1, F_2)$ may be performed if $F_1$ and $F_2$ are mergeable and the intersection of their subscribers sets has at least one element, $\text{subscribers}(F_1) \cap \text{subscribers}(F_2) \neq \emptyset$. The subscribers set of the resulting merger must contain only a single element.

The latter option is given by Definition 4.2. Aggregate 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 if the resulting merger has more than one entry in the subscribers set. Aggregate 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 (Lemma 3.7). 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 4.2 Aggregate merging rule: Given that the forwards sets of the filters are non-empty, the filters are mergeable only when $\text{forwards}(F_1) \cap \text{forwards}(F_2) \neq \emptyset$. The forwards set of the resulting merger is the intersection of the two forwards sets.
The rule of Definition 4.1 corresponds to interface specific merging of filters. The rule of Definition 4.2 takes into account the forwards sets and aggregates multicast traffic. These rules may be applied simultaneously.

A mergeability rule was sketched in [65] by observing that a merged set of filters $M$ created from a set of filters can be forwarded to all neighbours if any notification $n$ that is matched by $M$ is matched by at least one filter of a local client and by two distinct filters $F_i$ and $F_j$ with differing output interfaces. The presented forwards set-based formulation allows a more flexible and elegant way to determine mergeability.

### 4.2.2 Local Merging Rules

Let $MR$ denote the set of merged nodes/filters. Each element $x \in MR$ is a result of a single or 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 re-merging 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 for a merger, and the del in rule five is applied for 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.

We also define two auxiliary operations addComponent and addComponents. The addComponent($S$, $F$) operation takes the set $S$ and the 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.

1. $F_1 \trianglerighteq F_2 \land F_1 \notin MR \land F_2 \notin MR \Rightarrow \text{del}(F_2)$. This rule says that when a non-merged node covers another non-merged node, the covered node
4.3. EXAMPLE MERGING MECHANISMS

should be removed.

2. \( F_1 \supseteq F_2 \land F_1 \not\subseteq MR \land F_2 \in MR \Rightarrow \text{del}(F_2) \). 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. \( F_1 \supseteq F_2 \land F_1 \in MR \land F_2 \not\in MR \Rightarrow \text{del}(F_2) \land \text{addComponent}(\text{CV}(F_1), F_2) \). This rules states that when a merger covers a non-merger, the covered node is added to the components of the merger.

4. \( F_1 \supseteq F_2 \land F_1 \in MR \land F_2 \in MR \Rightarrow \text{addComponents}(\text{CO}(F_1), \text{CO}(F_2)) \land \text{addComponent}(\text{CV}(F_1), F_2) \land \text{del}(F_2) \). Specifies that when a merger covers another merger, the covered merger is removed and all components of the merger and nodes covered by the merger are added to the respective sets of the covering merger.

5. \( \text{del}(F_1) \land (\exists x \in MR : F_1 \in \text{CO}(x)) \Rightarrow (\forall x \in \text{CO}(F_1) \setminus \{F_1\} : \text{add}(x)) \land (\forall x \in \text{CV}(F_1) : \text{add}(x)) \). 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.

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

4.2.3 Aggregate Merging Rules

Aggregate filter merging rules are similar to the local merging rules with the exception that they do not need to address covered nodes, because merged filters are not inserted into the data structure. The merging rules include the following local rules: 1, 2, 4, 5, and 6. The set \( CV \) is empty in this case and instead of the \textit{subscribers} set condition, we have the \textit{forwards} set condition presented in Definition 4.2. Implementations need to update the \textit{forwards} sets of any mergers covered by other mergers.

4.3 Example Merging Mechanisms

4.3.1 A Local Merging Mechanism: Merging Forest

A merging coloured forest is a coloured forest with an additional \textit{merge} operation that merges two input filters. Merging is performed during the \textit{add} operation with the first mergeable candidate. The \textit{del} operation removes one
of the merged filters and the remaining filters are re-merged. The position
and children of the merged node may need to be updated after operations.

It would be more optimal to find better merging candidates than the
first one; however, this selection process is laborious because it requires
the identification of the candidates. Furthermore, merging of the input filter
may require multiple iterations, since a filter may be repeatedly mergeable
with other filters. We restrict the merging procedure by allowing an input
filter to be merged only once. A merging forest that has this restriction is
called a weakly merging coloured forest. The reason for this restriction is that
repeated merging requires more processing power than merging only once.

4.3.2 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
ensure that perfect mergers do not cause false positives and may be also
merge filters from different input interfaces. Also, there is no need to keep
track of separate forwards set entries.

The proposed technique may be applied for 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 $|\text{neighbours}| - 1$ external inter-
faces. This merging technique may be called weakly merging for peer-to-peer
routing, because it does 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, 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 uses the forwards
sets to aggregate merger updates. This kind of approach may be used
to leverage any multicast mechanisms.
4.3. EXAMPLE MERGING MECHANISMS

Relatively simple  Tracks changes in the root set and merges filter 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 $MR$ and $CO$ 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

$$forwards'(f) = forwards(f) - \bigcup_{f' \in MR \land f' \supseteq f} forwards(f').$$

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 false positives 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 elements in the successor set that are covered by other mergers.
Chapter 5

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 and filter merging performance. We present performance results for the following data structures: filters poset (FP), redundant coloured forest (CF), non-redundant coloured forest (NRCF), and also colour-based balancing with the coloured forests.

5.1 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 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 random type and a name with the restriction that the \(<\text{name},\text{type}>\) pairs must be unique. Each attribute filter has a single predicate randomly selected from the set \{\text{<, >, ≤, ≥, =, ≠, [a, b]}\}. Appendix A presents the filtering language used in experimentation.

Random and unique field names, schema names, and constraint names were generated using uniform distribution over the set of lower-case alphabets with a uniformly distributed length of $U(3, 10)$. The range for integer values was 1 to 100. Notifications are generated as follows: each notification has the
Table 5.1: Data structures used in the experimentation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coloured Forest (CF)</td>
<td>The basic coloured forest without balancing.</td>
</tr>
<tr>
<td>Coloured Forest (NRCF)</td>
<td>The non-redundant CF without balancing.</td>
</tr>
<tr>
<td>Balancing Forest (BF)</td>
<td>Balanced CF with 1-bit colour index.</td>
</tr>
<tr>
<td>Balancing Forest (NRBF)</td>
<td>Non-redundant BF with 1-bit colour index.</td>
</tr>
<tr>
<td>Poset</td>
<td>The filters poset.</td>
</tr>
</tbody>
</table>

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 processor with 512 MB of main memory, Windows XP, and Java JDK 1.4.2. Table 5.1 presents the different data structures used in experimentation. The filters poset algorithm is based on the Siena Java implementation [75] and extended with hashtable-based duplicate filter detection.

5.2 Benchmarks

Basic 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 input filter set. 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 computation of the forwards set is not included in this benchmark, but this is similar for both structures with the exceptions discussed in Section 3.4.

Figure 5.1 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. 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 environ-
5.2. BENCHMARKS

Filters
Workload generator
Notifications

Poset or Forest
1. Add or Add/Remove scenario
2. Matching test
Correctness testing

Figure 5.1: Add and add/remove scenario tests.

ments 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. The algorithm for the forest-based matcher is presented in Figure 3.7 with the difference that for the experimentation we do not use the interface-count specific optimization. The poset-based matcher is similar to the forest-based matcher, but they differ because the poset-based matcher has to keep track of visited nodes. If a filter does not match then all its siblings are added to the visited list. Just ignoring the children of a non-matching node is not enough, because the siblings that cannot be matched by definition may be referenced by some other parts of the poset.

Correctness of operation was tested using assertions on root size invariance using Theorem 3.9, data structure size invariance, and existence of false positives and negatives in matching. We recorded the set of matching interfaces for each notification using the 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 colour property (Property 3.4) were also tested. For the balanced forest also the proper functioning of the colour index was tested. We also test 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 were created once for the replications and the set of notifications to test was created for each replication. The figures present results using the arithmetic mean over the replications.

**Merging Benchmarks** Figure 5.2 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.

![Diagram](image)

**Figure 5.2: Add scenario with merging.**

### 5.3 Redundant Forests

Figure 5.3 presents a comparison between the redundant coloured poset-derived forest (CF), balanced forest (BF), and the filters poset. We measured
both the number of covering operations required by the add and add/remove scenarios and processing time. The filters poset performs considerably slower than the forest structures in both scenarios. In this scenario each filter has a differing interface so they are not pruned. The size of the structures were approximately identical and the size grows linearly with the number of input filters.

Figure 5.3: Comparison of the poset and the forests with a variable number of filters.

Figure 5.4 gives the results for the variable interfaces scenario. 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 also in this scenario, because 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. The coloured forest has overhead in this respect and the balanced forest is able to reduce the filter set size. The filters poset has the smallest set size.

Figure 5.5 presents the matching results. The matching results are similar for all the data structures. The matching algorithm is superior to the naive algorithm, and gives best performance when the number of subscribing interfaces is small.
Figure 5.4: Comparison of the poset and the forests with a variable number of interfaces.

5.4 Non-Redundant Forests

The non-redundant forest benchmarks are similar to the ones described in Section 5.3 and they have the same parameters with the exception of non-redundancy. The forests use the \textit{eliminate-colour-all} extension to remove all redundant filters. Figure 5.6 presents comparison of the non-redundant coloured poset-derived forest (NRCF), balanced forest (NRBF), and the filters poset with a variable number of filters. This scenario represents the worst case for the poset and thus it does not perform well.

Figure 5.7 presents the results for a variable number of interfaces. The poset performs better when the number of interfaces is small. Forests are considerably better than the poset when the number of interfaces grows. This means that the poset should be used only for external interfaces. The non-redundant forests are also useful, because they may be used in hierarchical operation without additional data structures. Colour-based balancing gives the best results and it should be used with non-redundant forests.
5.5 Forwards Sets

The computation of the forwards set for the forest (NRBF) is more complex than for the poset, because the complete successor sets are not 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, the minimum-colour property is maintained, and any unsubscription that is unnecessarily forwarded will have an existing covering subscription.

We have implemented the forwards set computation for the forest and experimented with two cases. First, complete forwards set computation. Second, incomplete forwards set computation for the add operation and full uncovered set computation for del. The computation of the set for both FP and the forests was tested using a naive algorithm for computing the set. The naive algorithm builds the successor and predecessor sets at run-time.

We examined a scenario, in which there are no local clients. The complete case was observed to be heavy when compared with the poset. The incomplete, on the other hand, had similar performance to the poset and, in practice, false positives were not observed. The add is faster for the forest and the del operation is simpler and faster for the poset. If the del procedure is rarely used the forest with the incomplete forwards set computation seems to be the best candidate data structure.

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 colour elimination cannot be performed for filters
from different clients. Second, it may become more likely that 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.

5.6 Merging Benchmarks

Perfect Merging  Figure 5.8 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 scenario is the same as the add scenario in Section 5.3 with the exception that redundant filters are removed before insertion. The merging benchmark compares the insertion and matching performance of interface specific minimal covers with merged minimal covers. 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.
As the number of filters grows the merging algorithm is able to remove redundancy. The root size of the merged forest and poset are the same and root set sizes in merging scenarios are considerably smaller than in normal operation. Based on these results the merging time is reasonable and the merged forest is created quickly. Merging improves also the operation of the poset, but its build time takes considerably longer than the forest.

Figure 5.9 presents merging results for a variable number of interfaces and a static number of filters (500). 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 the number of filters grows.

**Imperfect Merging**  We experimented also 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.

5.7 PosetBrowser

We have developed a graphical tool for experimenting with various content-based routing data structures. The Java-based tool is called PosetBrowser\(^1\) and uses the JGraph toolkit [3] for graph visualization. The PosetBrowser tool supports the following data structures: FP, CF, BF, and NRBF.

Figure 5.10 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.

---

\(^1\)Available at www.cs.helsinki.fi/u/starkoma/posetbrowser
5.8 Discussion

The coloured and balanced poset-derived forests are more efficient than the filters poset in hierarchical routing. The situation is more complicated for peer-to-peer environments. The two important parameters are: the number of neighbours and the number of clients. The number of neighbours is typically small whereas there may be a large number of clients. Both neighbours and clients are interfaces (colours) from the forest viewpoint. Experimental results indicate that the filters poset performs better with a small number of interfaces, but the forest outperforms it when the number of interfaces and filters grows.

We can derive four useful engineering guidelines for content-based routing systems based on the experimentation:

Figure 5.9: Impact of merging on the forest and poset performance. Results for 500 filters.

The tool contains tests for ensuring the correctness of the algorithms. The tests include addition and deletion of filters. Figure 5.11 shows the filter generation features of the software.
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
Merging We observed significant performance benefits from the merging of interface specific filter sets. The interface specific merging mechanism represents the worst case merging overhead, because in normal distributed environments mergeability is evaluated incrementally. Observed merging overhead was comparable to the insertion time of a filter to a forest. The benefit of merging was constant or near-constant insertion time and matching time of the merged filter set.

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 less attribute filters may cover other filters with larger number of attribute filters. We have also experimented with a static scenario, where the number of attribute filters per filter is fixed. The static scenario gives also good results for covering, but merging does not perform well when the number of attribute filters grows. Imperfect merging, on the other hand, performs well also with a static number of attribute filters, but results in a lower precision.
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 small amount of filters, and more general filters when the subscription space becomes saturated. On the other hand, schema-name based forwarding has a very low precision for equality tests and scenarios where there are many attribute filters per filter and filters are not covered by simpler filters.
Part III

Mobility-aware Routing
Chapter 6

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.

6.1 Introduction

Most research on event systems has focused on event dissemination in the fixed-network, where clients are usually stationary and have reliable, low-latency, and 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 [12]. Mobility protocols are also needed for load balancing subscribers and advertisers between brokers and 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 advertisement 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 com-
pleteness of the subscriptions and advertisements of the mobile client is as-
sumed. We also discuss techniques for supporting incomplete topologies, in
which the subscriptions and advertisements of the mobile client are still be-
ing 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.

6.2 Formal Specification

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 (Table 2.1) in causal order that determine
the execution of the distributed system. The trace \( \sigma \) can be described as
a sequence of states \( s_n \) interleaved with the pub/sub operations denoted by
\( op_n \):

\[
\sigma = s_1, op_1, s_2, op_2, \ldots
\]  

(6.1)

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 \n"eventually”, and \( \circ \ ”next”. The semantics of the temporal operators are
defined as follows for an arbitrary formula \( \Psi \) [65]:

- \( \Diamond \Psi \) is true for trace \( \sigma \) if and only if there exists an \( i \) such that \( \Psi \) is true
  for the trace \( s_i, op_i, s_{i+1}, op_{i+1}, s_{i+2}, \ldots \),

- \( \Box \Psi \) is true for trace \( \sigma \) if and only if for all \( i, \Psi \) is true for the trace \( s_i, op_i, s_{i+1}, op_{i+1}, s_{i+2}, \ldots \), and

- \( \circ \Psi \) is true for trace \( \sigma \) if and only if \( \Psi \) is true for \( s_2, op_2, s_3, op_3, \ldots \).
Property 6.2 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 6.3 gives the safety constraint, which ensures that incorrect events are not processed and delivered. Note that this property does not specify any particular delivery order for notifications, and we need additional constraints to introduce, for example, causal ordering using Lamport’s happened-before relation [58].

The properties are from the definitions in [65] with minor changes in presentation, which also contains the definitions for the safety and liveness properties for advertisement semantics and proofs are given for the correctness of content-based routing. 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 [65]. The safety property may be modified to take self-stabilization into account by requiring eventual safety. In addition, stabilization mechanisms such as leases may be incorporated by assuming that the expiration of a lease leads always to unsubscription. A lease associates an expiration time or date with each subscription. When the lease expires, the subscription is removed from the system unless the lease is renewed.

Property 6.2 Liveness:

\[
□[\text{Sub}(A,F) \Rightarrow [\Diamond\Diamond(\text{Pub}(B,n) \land n \in N(F) \Rightarrow \Diamond\Diamond(\text{Notify}(A,n))) \lor
[\Diamond\Diamond(\text{Unsub}(A,F))]]]
\]

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 6.3 Safety:

\[
□[\text{Notify}(A,n) \Rightarrow [\Diamond\Diamond\neg\text{Notify}(A,n)] \land
[n \in \text{Published}] \land
[\exists F \in \text{Subs}(A): n \in N(F)]],
\]

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.
CHAPTER 6. MOBILITY AND COMPLETENESS

The safety and liveness properties were extended in [98] with the notion of message-completeness and using propositional temporal logic. A message-complete pub/sub system eventually acknowledges subscriptions and guarantees the delivery of notifications matching acknowledged subscriptions.

False Positives and Negatives In order to understand event routing we need to have useful metrics to characterize the system. The two most important metrics are the number of false positives and negatives, and even more so proofs for the existence of false negatives or positives in a particular system. 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.

In addition, we are interested in establishing upper and lower bounds for the cost of notification. The cost of notification may be measured in different units, for example, end-to-end latency and the number of messages sent by routers. Other interesting and useful metrics are end-to-end path length and the size of routing tables.

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 updates process terminates satisfies Properties 6.2 and 6.3 ([65] Theorem 3.3).

We call all update procedures that have ended successfully as complete in the topology, and use completeness to characterize and prove properties of pub/sub mobility. The completeness of subscriptions and advertisements is given by Definition 6.4. Note that Definition 6.4 imposes new requirements for the topology.

Definition 6.4 An advertisement $A$ is complete in a pub/sub system $PS$ iff there does not exist a router $s$ with an overlapping subscription that has not processed $A$. Similarly, a subscription $S$ is complete in $PS$ iff there does not exist a router $s$ such that $s$ has an an advertisement that overlaps with $S$ and $S$ is not active on $s$. 
6.3 Mobility-Safety

In distributed pub/sub systems it is evident that after issuing a subscription it may take several 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. In our subsequent examination we do not count these notifications as false negatives. Mobile clients change the scenario, because the flow end-points are mobile and brokers transfer end-points using a handover process. False negatives that occur during mobile clients are not tolerated. We define a mobility-safe pub/sub system as follows:

**Definition 6.5** A pub/sub system is mobility-safe if starting from an initial configuration $C_0$ at time $T_0$ and ending in a final configuration $C_e$ at time $T_e$ handovers (mobile clients) will not cause any false negatives.

We also relax Definition 6.5 by not considering server failures or assuming that faults may be masked. False positives may be removed by using event identifiers so they are not considered.

Now we may formulate the goals of a mobility-aware distributed event-based system. Mobility happens when a subscribing or publishing end-point relocates to some other router. It is also possible that a subscriber and a publisher of the same event both relocate at the same time. Mobile end-points have two immediate implications for the system. First, in order to maintain mobility-safety the system must buffer notifications for a sink, when the sink relocates to another router. Typically, this relocation requires a virtual sink to be established at the old location, which buffers events for later playback. The second implication is a requirement for an update procedure that first sets up a routing path to the new router, and then tears down the old routing path if it is no longer needed. Both buffering and the update procedure require signalling.

Intuitionally, 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 — satisfying the requirement for mobility-safety. This is our basic hypothesis for the state-transfer 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. Even though the flooding model guarantees correct routing it does not solve the buffering problem posed by disconnected clients.
A good mobility protocol is mobility-safe, minimizes the number of false positives, and minimizes the signalling cost.

### 6.4 Background

Mobility support [53,83] 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 [56,78]. Higher level mobility protocols are also needed in order to provide efficient middleware solutions, for example SIP (Session Initiation Protocol) mobility [91] and Wireless CORBA [74]. Event-based systems require their own mobility protocols in order to update the event routing topology and optimize event flow.

Siena, Rebeca, and Hermes [80] 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 [37]. 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 [97].

Siena is a scalable architecture based on event routing that has been extended to support mobility [17–19]. 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 [116]. 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 [68].

JECho is a mobility-aware event system that uses opportunistic event channels in order to support mobile clients [30]. 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.

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 $A$ to $B$.
2. $B$ subscribes 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 [12]. 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).

6.5 Generic Mobility Support

The Siena event system was extended with generic mobility support, which uses existing pub/sub primitives: publish and subscribe [18, 19]. The mobility-safety of the protocol was formally verified. The benefit of a generic protocol is 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 APIs. 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 use of advertisement semantics was not discussed for the Siena mobility support service so we extend the generic model to cover also advertisements.

The Siena generic mobility support service is implemented by proxy objects that reside on access routers. Figure 6.1 presents an overview of the process: 1. the client arrives to access point \( B \) and sends the move-in request to the proxy. 2. a ping request is sent and a response will be eventually received (3). The response can also be called a pong. 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.

The mobility protocol proceeds in four distinct phases: first, the target subscribes all 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 the events are relocated. In addition, there may be further costs triggered by changes in the subscription tables of the intermediate routers. Let \( N \) denote the number of brokers in the network and \( n \) the number of brokers between the source and target of mobility. The cost structure of the procedure is given by Table 6.1 as the number of brokers that will be updated during each phase. For advertisement semantics we assume that the publication of a ping message includes also the advertisement. The unsubscription cost depends also on other active subscriptions on the servers and is a worst-case estimate.

For subscription semantics the basic problem with the ping-based synchronization protocol is that the ping message is guaranteed not to be sub-
6.5. GENERIC MOBILITY SUPPORT

Figure 6.1: Move-in function in mobility support service [19].

scribed by any other broker on the network and hence the ping subscription message will be introduced at every broker on the network. If uniqueness is not guaranteed, simultaneous mobility by different clients becomes impossible.

With subscription semantics 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 subscription status of the broker. The Siena support service does not require API support, because the ping messages are continuously resent [19], 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.

Advertisement semantics presents several problems for mobility, because the ping and pong messages need to be advertised before they may be subscribed. The source advertises pong and subscribes ping, whereas the target advertises ping and subscribes pong. This has a high cost, because the advertisement from source needs first to reach the target before target’s subscription starts to propagate on the reverse path. We envisage that some of
CHAPTER 6. MOBILITY AND COMPLETENESS

Table 6.1: Cost structure for generic mobility.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Sub semantics</th>
<th>Adv semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source: Subsc. Ping(id)</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>Target: Subsc. Filter</td>
<td>(0 \leq x \leq N)</td>
<td>(0 \leq x \leq N)</td>
</tr>
<tr>
<td>Target: Subsc. Pong(id)</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>Target: Pub. Ping(id)</td>
<td>(n + \text{periodic})</td>
<td>N</td>
</tr>
<tr>
<td>Source: Pub. Pong(id)</td>
<td>(n)</td>
<td>N</td>
</tr>
<tr>
<td>Source: Unsub. Ping(id)</td>
<td>N</td>
<td>(n)</td>
</tr>
<tr>
<td>Target: Unsub. Pong(id)</td>
<td>N</td>
<td>(n)</td>
</tr>
<tr>
<td>Target: Unadv. Ping(id)</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>Source: Unadv. Pong(id)</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>

this cost may be avoided by using virtual advertisements for the ping/pong messages so that they are propagated using subscription semantics. Another solution is to bundle the advertisement and notification into one, which we have proposed previously [104]. The pub/sub API needs to support these optimizations.

Publisher mobility for subscription semantics does not require any changes, but it is supported directly by the pub/sub system. Publisher mobility for advertisement semantics follows the ping/pong 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.

Unsubscription and unadvertisement cost may be reduced if the underlying system supports leases. The ping/pong advertisement does not change the behaviour of the system even when unadvertishments are not used for ping/pong messages, because the identifiers are assumed to be unique and advertisements simply denote the intent to publish an event in the future. However, multiple subscriptions for the same ping/pong identifiers may result in the failure of the mobility protocol unless the synchronization request carries information about the client.

6.6 Acyclic Graphs with Advertisements

6.6.1 Overview

An outline of the protocol we describe was presented in [68, 116] with mobility restricted between border brokers. A mobility protocol for a hierarchical
routing topology with more assumptions was examined in [12]. Formally, the network of application-level event brokers is an undirected acyclic graph $G = (E, V)$, where $E$ is the set of edges and $V$ is the set of vertices 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. At the start of the handover, we assume that the routing topology is complete — that subscriptions have terminated successfully. We also assume that clients know the subscriptions and the address of the origin server (for direct communication). Some handover protocols use pub/sub routing to find the origin server; however, this approach may require the use of flooding to find the origin server, for example in scenarios where the relocated subscriptions are not advertised. In this case, the protocol breaks down. 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. Moreover, text and XML-based notifications typically compress very well, for example using gzip, and we believe that by transferring the notifications in one compressed message the overhead may be reduced considerably. 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 [12] in mobility scenarios. We consider the number of exchanged messages, because update processing and propagation causes the most stress for the pub/sub system.

### 6.6.2 Mobile Subscribers

Figure 6.2 presents an overview of the subscription handover both for complete and incomplete topologies. Prior to the handover, a set of brokers have advertised the events that are being relocated. Handover is performed between two brokers in the network: $a, b \in V$. The handover starts when $b$ receives a message, typically from the client, that a set of subscriptions $S$ should be relocated from $a$ to $b$. Now, $b$ may optimize the operation if $S$ has been already subscribed 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 6.2: Subscription handover with a complete and incomplete topology. Coloured nodes represent the inactive path.

In the figure the complete subscription topology update proceeds as follows: 1. \( b \) sends the update message, 2. update message is propagated towards \( a \) and will meet \( a \)'s subscriptions at node M (which can also be \( a \) itself), 3. \( M \) sends the update message towards \( a \), 4. session is transferred and \( a \) may unsubscribe to \( M \) if necessary. For the incomplete case the procedure is similar, but the subscription update message may be sent to several directions as depicted in the 1. phase of the figure. In 2. subscriptions from \( a \) and \( b \) meet and the meeting point, \( M \), notifies \( a \) in 3. It is clear that all subscriptions contained in \( S \) must be covered at \( M \). The covering optimization 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 view a path 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 \( S \), if \( S \) has been sent on that edge. A path is active if all of its edges are active. Later we will also say this edge is active towards \( b \) if \( S \) was sent from \( a \) to \( b \), and that a path \( a \sim b \) is active towards \( b \) if all of its edges are active in the direction of the directed path \( a \sim b \). An inactive path is one that is not active, i.e. contains at least one inactive edge.
We know that $a$ and $b$ are connected since $G$ is connected. Also the path between $a$ and $b$ is unique, since $G$ is acyclic. If the covering optimization is not applicable we know that $S$ is not active on all edges on the path from $a$ to $b$. Let $S_a$ be the set of active edges on the path, and $S_i$ the set of inactive edges on the path. The topology update cost is the cost of updating $S_i$.

**Proposition 6.6** There exists a node $M$ on the path $a \leadsto b$ such that the path $a \leadsto M$ consists of exactly the edges in $S_a$ (and then $S_i$ is the path $b \leadsto M$).

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

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

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 3, and in the latter case the path $b \leadsto P$ is already active, so no update is necessary.

Given that there are simultaneous unadvertisements, Proposition 6.6 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. The protocol may cope with this scenario by using timeouts or periodic update messages. Simultaneous advertisements do not pose such a problem.

When \( S \) reaches \( M \) the subscription has taken effect. After this \( M \) contacts \( a \), and \( a \) may then unsubscribe \( S \) if necessary. The problem with covering and merging is that \( M \) does not know the exact interface from which \( a \) is reachable. Therefore \( M \) needs to forward the completion message to all interfaces that have covering subscriptions. In essence, this is content-based flooding. If there cannot be concurrent subscriptions and advertisements on the subgraph, \( M \) may also directly contact \( a \). If there is concurrent activity, the path from \( M \) to \( a \) needs to be tested.

If the whole subscription configuration is complete, it follows that if an update is required it is sent by \( b \) to exactly one output interface. If this were not the case there would be a publisher on an interface not connected with the subscriptions from \( a \), which would violate the assumption that the topology is complete. If the whole topology is complete, \( M \) may contact \( a \) directly using out-of-band signalling.

**Communication Cost**

Assuming completeness, the total cost for the mobility in terms of exchanged messages for the subgraph defined by the path from \( a \) to \( b \) is \( |S_i| \) messages for the path, one message from \( M \) to \( a \), the cost of transferring the buffered events, and then the cost of unsubscription \( C_{unsub} \). The worst case size of \( S_i \) is \( n - 1 \), where \( n \) is the number of brokers in the routing topology. Therefore for the complete topology the update cost for subscriptions is at most \( n + C_{unsub} \). If \( S \) is already covered at \( b \) then the cost is \( 1 + C_{unsub} \).

**Mobility-Safety**

**Lemma 6.7** 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 following Proposition 6.6 indicates that the
For simultaneous advertisement activity it is sufficient to consider only the cases 1 and 2. In the 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 the 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.

**Handovers on Incomplete Topology**

It may also be the case that the set of subscriptions, $S$, is still being propagated and the subscription path is not totally active at the time of the handover. This happens when the handover happens while the system propagates and processes subscription messages or update messages. This kind of behaviour may occur because the handover is triggered by out-of-band mechanisms, for example terminal mobility. In this case Proposition 6.6 is not necessarily valid. If mobility is activated by sending a message using the pub/sub system this problem may be avoided.

Clearly, $a$ does not know when the topology is complete. Node $b$ may detect the local incompleteness of the subgraph defined by the path from $a$ to $b$; however, $b$ cannot detect the completeness of the subgraph based on local information. For example, consider the scenario where there is one broker between $a$ and $b$ that has not yet processed the subscription from $a$ and $S$ is covered at $b$. Now, since $S$ is covered at $b$, $b$ will ask $a$ to send the session even though the subscriptions may not be complete on the path at the time. This incompleteness of the path means that the handover is not necessarily mobility-safe. This may be solved by synchronizing $a$ and $b$ using a message passed through the path. The update message is needed even when $S$ is covered at $b$ and requires at least $|S_a \cup S_i|$ messages. The problem is that $b$ does not know where $a$ is located so the update is flooded on the reverse-path of covering subscriptions or overlapping advertisements. Incompleteness may also cause flooding when $S$ is not covered at $b$ and the protocol needs to cope with the existence of multiple meeting points $M$.

An interesting scenario is where $b$ has received advertisements from several interfaces, but the subscription has not yet been connected with any of them and thus $S$ is not covered at $b$. Now, when $b$ forwards the update message to the interfaces and there are simultaneous subscriptions beyond $b$, it may happen that the update message will meet covering subscriptions on the route creating several meeting points. This problem is difficult to avoid, because
CHAPTER 6. MOBILITY AND COMPLETENESS

we do not know in which direction $a$ is located. Therefore, $M$ may not notify $a$ directly when the topology is incomplete, but has to use content-based flooding to find $a$. If the update was forwarded to many interfaces only one will have $M$, but potentially content-based flooding is used in different parts of the network and update messages are propagated to the end of each branch to locate $a$.

Incompleteness of advertisements creates problems for the protocol, because the source may not be locatable at all if the advertisements have not reached the target. This problem is solved by using periodic updates, which is not practical.

**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.

**6.6.3 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, a publisher mobility support is not needed for 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 6.3 presents an overview of the process. 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 $M$ that the topology has been updated. 3. $a$ sends an update message to $b$ to ensure completeness. Finally, $a$ unadvertises if necessary.

If we assume that the advertisement topology is complete we find that the set $A$ of advertisements to be relocated from $a$ to $b$ is already advertised to $b$. The advertisement may be more general due to covering and merging. The problem is that the advertisements point to $a$, which needs to be changed.
Figure 6.3: Advertisement handover with complete topology. Black arrows indicate subscriptions, grey arrow indicate advertisements.

We have the two paths, the active path \( A_a \), and the inactive path \( A_i \) and \( M \) at their intersection. If there are no publishers that advertise covering advertisements on the path \( A_a \cup A_i \) it follows that \( A_a = \emptyset \).

If all interfaces advertise \( A \) at \( b \) then the handover is complete. If all interfaces do not advertise \( A \) at \( b \), they need to be updated. If there are more than one interface that does not advertise a set of advertisements that covers \( A \), then the topology is incomplete. (The one is the interface that received the advertisement). By definition, each router advertises a received advertisement to all other interfaces.

Now, \( b \) needs to locate \( a \) and sends the update message to interfaces that have received covering advertisements and advertises \( A \) from \( b \). Given that \( a \)'s advertisements are initially complete, \( b \)'s update message will reach \( M \) and \( a \). This means that subscriptions on the path are hop-by-hop connected with \( b \). Here we have the inactive path \( S_i \) towards \( a \), which is the path from \( b \) to subscribers that are not yet connected. In order to avoid false negatives \( a \) and \( b \) must wait that the subscriptions are connected. In order to avoid false negatives, \( a \) sends a ping message to \( b \) to indicate when the subscriptions on the path have been updated.

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 \( A \) 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 we assume initial completeness of the relocated advertisement, periodic update messages are not needed.

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.

**Communication Cost**

The total cost for the mobility in terms of exchanged messages for the subgraph defined by the path from \( a \) to \( b \) is at least \( |A_i| \) messages for the advertisement update. The subscription update requires at least \( |S_i| \) for the ping message. Subscriptions are connected hop-by-hop towards \( b \) as \( A \) is sent towards \( a \) and then the ping message is sent from \( a \) to \( b \). This happens after \( b \) receives the ping message. Finally, we have the cost of unadvertisement \( C_{unadv} \). Assuming completeness this gives the total cost of \( |A_i| + |S_i| + C_{unadv} \).

The worst case size for both \( A_i \) and \( S_i \) is \( n - 1 \), where \( n \) is the number of brokers in the routing topology. Therefore the handover cost for relocating advertisements from \( a \) to \( b \) is at most \( 2(n - 1) + C_{unadv} \). If \( A \) is already covered at \( b \), then the cost is \( 1 + C_{unadv} \).

**Mobility-Safety**

**Lemma 6.8** 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 6.6 applies here too (but we will name the nodes \( a, b, \) and \( S \) here for clarity). Similarly we may collapse \( S \) to \( M \), since the active path \( S \sim M \) is not affected by the publisher moving. Also the case 3 is trivial, since the subscription is then already existing at \( b \), and so the path \( b \sim S \) is already active.

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

The final case to consider is 1. Here if \( a \) couples the subscription with its ping, the path \( b \sim a \) 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 6.7.

**Theorem 6.9** A pub/sub system with both subscriber and publisher mobility and initially complete subscription and advertisement 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 6.4, where only relevant brokers are shown and the edges between brokers are actual paths. In case I, since the path $M_2 \sim M_1$ is active and not affected by mobility, the situation is equivalent to the subscriber and publisher moving serially, which is mobility-safe by Lemmas 6.7 and 6.8.

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 has been propagated 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 6.7 and 6.8.

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 $M_1 \sim P_O$ 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.

### 6.7 Rendezvous-Point Models

Hermes is a peer-to-peer event system based on an overlay called Pan that supports a variant of the advertisement semantics. Hermes [80, 81] 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.

A special node in the network called the *Rendezvous Point* (RP) is used to coordinate advertisement and subscription propagation. The RP manages
the event type and Hermes supports chaining RPs into type hierarchies. In general, the RP of an event type is obtained by hashing the event type to the flat addressing space of the overlay [81]. Most current overlays are based on flat addressing spaces, for example Pastry [87], Tapestry [117], and Chord [94]. This means that rendezvous points are uniformly distributed over the addressing space. The placement of event types (RPs) using uniform distribution is motivated by the fact that the types are disjoint from the viewpoint of matching. On the other hand, event traffic distribution may well be non-uniform, which should also be taken into account. The problem with non-uniform traffic distribution is that an RP may be located on the other side of the network. The RP may then become a performance and scalability bottleneck.

Hermes rendezvous-points are established using a special message that establishes an event type to a rendezvous point, which owns the address of the hashed type identifier. The event type conforms to a schema that the client software may request using an API call. This is required for type-safe subscriptions. Hermes supports two routing algorithms: type-based routing and type/attribute-based routing.

In type-based routing all messages are propagated towards the RP: subscriptions, advertisements, and notifications. Type-based routing does not support filtering, but compares event based on their type. Subscriptions and advertisements are local to a branch of the multicast tree rooted at the RP and they are not forwarded by the RP. This means that notifications always
have to be sent to the RP.

Type/attribute-based routing is similar to type-based routing, but supports filtering with covering relations and, instead of sending all notifications to the RP, notifications are sent on the reverse path of subscriptions. In this case, advertisements are only sent to the RP. Subscriptions are always sent on the reverse path of advertisements. The RP forwards subscription messages to overlapping advertisements. The type/attribute-based routing is more suitable to scenarios, where event traffic is not uniformly distributed, because notifications are not always sent to the RP.

One key feature of Hermes is connecting RPs into type hierarchies. In subscription inheritance routing, advertisements are only sent to the RP that maintains the event type. Subscriptions are forwarded by the RP to all RPs with descendant types. In advertisement inheritance routing, the RP forwards the advertisement recursively to all RPs of all ancestor event types. Also notifications are forwarded to all ancestor event types, because they are sent on the same forward path as advertisements.

Hermes uses heartbeat messages to detect server and RP failures. The underlying overlay allows to locate a new server that takes over the responsibilities of a failed node. Control messages pertaining to routing are simply sent towards the RP and the overlay will provide a new route with a new server. Hermes supports RP replication by synchronizing advertisement and subscription status between different replicas. The replicas are placed in the same multicast tree to avoid overhead due to message propagation. Load balancing of traffic between RPs is not discussed.

Figure 6.5 illustrates rendezvous-point based notification using 6 phases:

1. Publisher advertises an event type (and a filter in type/attribute-based routing).
2. Advertisement is forwarded to the rendezvous-point.
3. Subscriber subscribes 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 forwarded to the rendezvous-point.
5. Another subscriber subscribes.
6. Subscription message is propagated towards the RP.

After the last phase, the two routing models 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 RP.

- Subscriptions are introduced on the path from the subscriber to 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 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 requires more complex investigation.

![Figure 6.5: Example of the Hermes model.](image-url)
6.7. RENDEZVOUS-POINT MODELS

6.7.1 Mobility-safety

Lemma 6.10 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 we implement subscription semantics using advertisement semantics and thus the behaviour of $PS_S$ is not changed.

Theorem 6.11 An initially complete rendezvous type/attribute-based routing model is mobility-safe.

Proof sketch: Follows from Theorem 6.9 and Lemma 6.10 by establishing virtual advertisements at each RP for the corresponding event type.

6.7.2 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 origin 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 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 that are complete to RP. The two central problems that are solved are: false negatives due to incompleteness of the path and a ping between source and destination is not needed, and by using the overlay address to locate the source, content-based flooding does not need to be used.

6.7.3 Communication Cost

The RP completeness simplifies subscriber and publisher mobility protocols and the communication costs. Within one branch the update message from
b to a eventually meets the subscription from a to RP. Between branches
the cost is the distance from b to the RP. Hence, the worst-case number of
nodes that need to be updated if RP completeness is assumed is given by
$RP_{\text{max}}$, where $RP_{\text{max}}$ is the maximum number of nodes between any node
and a RP. One message is needed to notify a and one is needed to transfer
buffered events. The case when a relocated subscription is covered is handled
as before. Publisher mobility is also simpler than before and for a new branch
it is sufficient to synchronize b with the RP or RPs in question.

Although RPs are good for mobility, the cost of a handover grows with the
number of RP updates for cyclic overlay routing. 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.

6.8 Discussion

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

- The generic protocol is mobility-safe and applicable to various under-
lying pub/subs systems, but it is very inefficient and does not allow
pub/sub system or topology specific optimizations. Incompleteness is
not an issue, because the protocol ensures that the source and target
access routers are synchronized.

- The general acyclic graph routing topology is more efficient than the
generic protocol, but suffers from the problem that the source router
needs to be located using event routing. Since covering and merging
do not preserve information pertaining to the original router that is-
issued a subscription or advertisement, the use of content-based flooding
may be required. It is not realistic to assume that all routers know
the whole router topology. Furthermore, network-level addressing may
not provide enough detail to find the correct application-layer output
interface, because there may be any number of network-level routers be-
tween two application-level routers. This routing topology suffers also
from the incompleteness of subscriptions and thus the covering opti-
mization that uses out-of-band communication cannot be performed if
mobility-safety is required. Incompleteness may also cause a router to
flood subscriptions to several exiting interfaces. Incompleteness of the
subpath from the origin router to the target router may be corrected, but it has a high cost due to potential content-based flooding.

- Rendezvous points 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 points 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 techniques for improving mobility-aware pub/sub systems:

- Overlay-based routing. Overlay addresses prevent the content-based flooding problem. This abstracts the point-to-point 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 points. Rendezvous 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. Completeness checking ensures that the subscriptions and advertisements are fully established in the topology. This is needed to perform the covering optimization.

6.9 Summary

In this chapter, 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.
The generic mechanism has a high cost for mobility. The other two mobility mechanisms have considerably smaller cost.

If an acyclic graph-based routing topology is incomplete, content-based flooding may or must be used to complete the handover successfully. This means that the optimizations discussed for complete topologies may not be applied 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 practice. Mobility-safety cannot be guaranteed, if protocols engineered with the completeness assumption are used for incomplete topologies.

The rendezvous-based semantics used in the Hermes overlay pub/sub system was identified to be the best topology for pub/sub mobility support, because it may be extended to guarantee the completeness of subscriptions and advertisements. In addition, the model does not require the flooding of the whole network with advertisement messages. Rendezvous-points with acyclic topology cannot have greater cost than generic acyclic graphs.

We presented three useful techniques for optimizing mobile pub/sub systems, namely using overlay routing, rendezvous-points, and completeness-checking. Completeness checking enables the covering optimization, which prevents unnecessary topology updates.
Part IV

Conclusions
Chapter 7

Conclusions

In this thesis we have presented and investigated efficient content-based routing for static and mobile environments. 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.

The main contributions of the second part is a formal definition of the poset-derived forest data structure and its variants, and results with filter merging. This work addresses the requirement for frequent updates to routing tables, and so far efficient data structures have not been proposed in literature. Typically, event systems represent 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 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.

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 external filters for peer-to-peer routing tables is more difficult. We observed significant performance benefits from the merging of interface specific filter sets.

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 identified to be good for mobility.

If an acyclic graph-based routing topology is incomplete, content-based flooding may or must be used to complete the handover successfully. This means that the optimizations discussed for complete topologies may not be applied 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 identified to be a suitable 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 network/transport 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.

Future work includes experimentation with different peer-to-peer routing techniques combined with filter merging. We also plan to continue research on pub/sub mobility protocols and especially on how to combine load balancing with mobility support.
References


REFERENCES

of the 1st Workshop on Network and System Support for Games, pages 3–9, Braunschweig, Germany, 2002. ACM Press.


REFERENCES


[34] Simon Courtenage and Steven Williams. Automatic hyperlink creation using P2P and publish/subscribe. In Workshop on Peer-to-Peer and Agent Infrastructures for Knowledge Management (PAIKM), Kaiserslautern, Germany, April 2005.


REFERENCES

[48] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


[57] Markus Keidl, Alexander Kreutz, Alfons Kemper, and Donald Kossmann. A publish & subscribe architecture for distributed metadata
management. In *Proceedings of the 18th International Conference on Data Engineering (ICDE’02)*, 2002.


REFERENCES


REFERENCES


Appendix A

Filter Merging Mechanism

This appendix presents the filter model and merging mechanism used in the experimentation.

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, 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. Disjunctions would give more power over the structure of the notification. On the other hand, disjunctions complicate the algorithms and 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 one of those disjunctive terms is satisfiable. A term is satisfiable if there are no contradictions [4].

Table A.1 presents the predicates for integer and string types. All predicates are unary except the range predicate, which is binary. The predicates
Table A.1: Predicate list for the filter clause.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
<th>String</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equ</td>
<td>Exact matching of any type</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Neq</td>
<td>Not equal to given value</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>In</td>
<td>Substring within string value</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Nin</td>
<td>Subtring not within string value</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Stw</td>
<td>Starts with the given string</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>Ends with the given string</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Exists</td>
<td>Tuple exists in notification</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Gr/Gre</td>
<td>Value $\geq$ to given value</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Lt/Lte</td>
<td>Value $\leq$ to given value</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Rng</td>
<td>Range test $[a, b]$</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>$$name</td>
<td>Placeholder for the value of the tuple name</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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.

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 $A_i$ [65]:

$$\forall i \exists j A_i \sqsupseteq B_j \iff A \sqsupseteq B.$$  \hspace{1cm} (A.1)

Figure A.1 presents the covering relations for integer predicates. Each cell gives the condition for $F_1 \sqsubseteq F_2$. 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: $F_1$ and $F_2$ denote the values of unary input predicates. If a predicate is binary, the values are given by $F_1.x$ and $F_1.y$ and similarly for $F_2$. The binary
range predicate is also written using \( rng(x, y) \). Sentence "\( F_1 \) is in \( rng(x, y) \)" denotes that the single value of the unary predicate \( F_1 \) is contained in the range \([x, y]\) of the binary predicate.

<table>
<thead>
<tr>
<th>( F_2 )</th>
<th>( F_1 )</th>
<th>Equ</th>
<th>Neq</th>
<th>Gr</th>
<th>Lt</th>
<th>Gre</th>
<th>Lge</th>
<th>Rng</th>
<th>Exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equ</td>
<td>( F_2 = F_1 )</td>
<td>( F_2 \neq F_1 )</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &gt; F_1 )</td>
<td>( F_1 &lt; F_2 )</td>
<td>( F_1 &gt; F_2 )</td>
<td>( F_1 \subseteq F_2 )</td>
<td>( F_2 \subseteq F_1 )</td>
<td></td>
</tr>
<tr>
<td>Neq</td>
<td>( F_2 = F_1 )</td>
<td>( F_2 \neq F_1 )</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &gt; F_1 )</td>
<td>( F_1 &lt; F_2 )</td>
<td>( F_1 &gt; F_2 )</td>
<td>( F_1 \subseteq F_2 )</td>
<td>( F_2 \subseteq F_1 )</td>
<td></td>
</tr>
<tr>
<td>Gr</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &gt; F_1 )</td>
<td>( F_1 &lt; F_2 )</td>
<td>( F_1 &gt; F_2 )</td>
<td>( F_1 \subseteq F_2 )</td>
<td>( F_2 \subseteq F_1 )</td>
<td></td>
</tr>
<tr>
<td>Lt</td>
<td>( F_2 &gt; F_1 )</td>
<td>( F_2 &gt; F_1 )</td>
<td>( F_2 &gt; F_1 )</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_1 &lt; F_2 )</td>
<td>( F_1 &gt; F_2 )</td>
<td>( F_1 \subseteq F_2 )</td>
<td>( F_2 \subseteq F_1 )</td>
<td></td>
</tr>
<tr>
<td>Gre</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &lt; F_1 )</td>
<td>( F_2 &gt; F_1 )</td>
<td>( F_1 &lt; F_2 )</td>
<td>( F_1 &gt; F_2 )</td>
<td>( F_1 \subseteq F_2 )</td>
<td>( F_2 \subseteq F_1 )</td>
<td></td>
</tr>
<tr>
<td>Lge</td>
<td>( F_2 \leq F_1 )</td>
<td>( F_2 \leq F_1 )</td>
<td>( F_2 \leq F_1 )</td>
<td>( F_2 \geq F_1 )</td>
<td>( F_1 \leq F_2 )</td>
<td>( F_1 \geq F_2 )</td>
<td>( F_1 \subseteq F_2 )</td>
<td>( F_2 \subseteq F_1 )</td>
<td></td>
</tr>
<tr>
<td>Rng</td>
<td>( F_2 \in rng(x, y) )</td>
<td>( F_2 \in rng(x, y) )</td>
<td>( F_2 \in rng(x, y) )</td>
<td>( F_2 \in rng(x, y) )</td>
<td>( F_2 \in rng(x, y) )</td>
<td>( F_2 \in rng(x, y) )</td>
<td>( F_2 \in rng(x, y) )</td>
<td>( F_2 \in rng(x, y) )</td>
<td></td>
</tr>
<tr>
<td>Exists</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td></td>
</tr>
</tbody>
</table>

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 [65]. Theorem A.2 presents covering for disjunctive attribute filters. A pre-requisite is that filter \( A \) has the same name and type than \( B \). 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.2 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.2** \( \forall i \exists j A_j \supseteq B_i \Rightarrow A \supseteq B \).

**Proof:** Assume the left side. 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 \( A \supseteq B \). \( \square \)

**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, \( \sim \), 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 [21, 65] and overlap detection for filters in the disjunctive normal form is discussed in [4].

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 \texttt{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.

A.5 Perfect Merging

The perfect merging algorithm merges two structurally equivalent filters by applying the attribute merging algorithm to the distinctive attribute filter, if it exists. Perfect merging is based on Definition A.3. For disjunctive attribute filters the distinctive attribute filter is always mergeable. Conjunctive attribute filters are not necessarily mergeable. For any two mergeable filters
A.5. PERFECT MERGING

Table A.2: The rules for number existence test.

<table>
<thead>
<tr>
<th>( F_1 )</th>
<th>( F_2 )</th>
<th>Condition</th>
<th>( F_1 )</th>
<th>( F_2 )</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = a_1 )</td>
<td>( x \neq a_2 )</td>
<td>( a_1 = a_2 )</td>
<td>( x \neq a_1 )</td>
<td>( x = a_2 )</td>
<td>( a_1 = a_2 )</td>
</tr>
<tr>
<td>( x &lt; a_1 )</td>
<td>( x &gt; a_2 )</td>
<td>( a_1 &gt; a_2 )</td>
<td>( x &lt; a_1 )</td>
<td>( x \geq a_2 )</td>
<td>( a_1 \geq a_2 )</td>
</tr>
<tr>
<td>( x \leq a_1 )</td>
<td>( x &gt; a_2 )</td>
<td>( a_1 \geq a_2 )</td>
<td>( x \leq a_1 )</td>
<td>( x \geq a_2 )</td>
<td>( a_1 \geq a_2 )</td>
</tr>
<tr>
<td>( x &gt; a_1 )</td>
<td>( x &lt; a_2 )</td>
<td>( a_1 &lt; a_2 )</td>
<td>( x &gt; a_1 )</td>
<td>( x \leq a_2 )</td>
<td>( a_1 \leq a_2 )</td>
</tr>
<tr>
<td>( x \geq a_1 )</td>
<td>( x &lt; a_2 )</td>
<td>( a_1 \leq a_2 )</td>
<td>( x \geq a_1 )</td>
<td>( x \leq a_2 )</td>
<td>( a_1 \leq a_2 )</td>
</tr>
<tr>
<td>( x \neq a_1 )</td>
<td>( x \neq a_2 )</td>
<td>( a_1 \neq a_2 )</td>
<td>( x \neq a_1 )</td>
<td>( x &gt; a_2 )</td>
<td>( a_1 &gt; a_2 )</td>
</tr>
<tr>
<td>( x \neq a_1 )</td>
<td>( x \geq a_2 )</td>
<td>( a_1 \geq a_2 )</td>
<td>( x \neq a_1 )</td>
<td>( x \leq a_2 )</td>
<td>( a_1 \leq a_2 )</td>
</tr>
<tr>
<td>( x \neq a_1 )</td>
<td>( x &lt; a_2 )</td>
<td>( a_1 &lt; a_2 )</td>
<td>( x &gt; a_1 )</td>
<td>( x \neq a_2 )</td>
<td>( a_1 &lt; a_2 )</td>
</tr>
<tr>
<td>( x \neq a_1 )</td>
<td>( x \neq a_2 )</td>
<td>( a_1 \leq a_2 )</td>
<td>( x &lt; a_1 )</td>
<td>( x \neq a_2 )</td>
<td>( a_1 &gt; a_2 )</td>
</tr>
<tr>
<td>( x \neq a_1 )</td>
<td>( x \neq a_2 )</td>
<td>( a_1 \geq a_2 )</td>
<td>( x &lt; a_1 )</td>
<td>( x \neq a_2 )</td>
<td>( a_1 &gt; a_2 )</td>
</tr>
<tr>
<td>( x &lt; a )</td>
<td>( x \geq a )</td>
<td>( a &lt; a )</td>
<td>( x &lt; a )</td>
<td>( x \neq a )</td>
<td>( a \neq a )</td>
</tr>
<tr>
<td>( x = b )</td>
<td>( x \neq b )</td>
<td>( a = b )</td>
<td>( x = b )</td>
<td>( x \neq b )</td>
<td>( a \neq b )</td>
</tr>
</tbody>
</table>

\( F_1 \) and \( F_2 \) the operation \( \text{merge}(F_1, F_2) \) either merges the distinctive attribute filter or the filters are identical.

**Definition A.3** Perfect merging may be performed if and only if at least \( n - 1 \) conjuncts are identical. Perfect merging is based on the tautology: 
\[
(C_1 \land \ldots \land C_n \land B) \lor (C_1 \land \ldots \land C_n \land D) \iff C_1 \land \ldots \land C_n \land (B \lor D),
\]
where all \( C_i \), \( B \), and \( D \) are conjuncts. If \( B \) and \( D \) are mergeable: 
\[
C_1 \land \ldots \land C_n \land (B \lor D) \iff C_1 \land \ldots \land C_n \land \text{merge}(B, D).
\]

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 [4, 65].

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 perfect merging approach presented in [64, 65] for conjunctive attribute filters requires that all but one-attribute filter of an attribute are identical and the distinctive attribute filters can be merged. 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 attribute filters. The presented approach is more expressive, because it supports disjunctions.
Appendix B

Example Scenario: Smart Office

We have developed a smart office scenario that highlights the features of the Fuego Core middleware service set 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 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 B.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 smart-phone and a laptop.

Key activities in the scenario and demonstration 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.
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, event session is transferred back to the smartphone. 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 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. Context-sensitive messaging using content-based routing is implemented by subscribing 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 presence service builds on these services and adds the capability for users to monitor the presence of other entities and also control the visibility of their own presence variables. The synchronizing file-system 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.