# Large-Scale Content-Based Publish/Subscribe Systems

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

**Dissertation**

zur Erlangung des akademischen Grades
eines Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

**Dipl.-Inform. Dipl.-Ing. Gero Mühl**
aus Lüdenscheid

| | |
|---|---|
| Referenten: | Prof. A. P. Buchmann, Ph.D.<br>J. M. Bacon, Ph.D. |
| Datum der Einreichung: | 19. August 2002 |
| Datum der mündlichen Prüfung: | 30. September 2002 |

Darmstadt 2002, Darmstädter Dissertationen D17

# Abstract

Today, the architecture of distributed computer systems is dominated by client/server platforms relying on synchronous request/reply. This architecture is not well suited to implement information-driven applications like news delivery, stock quoting, air traffic control, and dissemination of auction bids due to the inherent mismatch between the demands of these applications and the characteristics of those platforms. In contrast to that, publish/subscribe directly reflects the intrinsic behavior of information-driven applications because communication here is indirect and initiated by producers of information: Producers publish notifications and these are delivered to subscribed consumers by the help of a notification service that decouples the producers and the consumers. Therefore, publish/subscribe should be the first choice for implementing such applications.

The expressiveness of the notification selection mechanism used by the consumers to describe the notifications they are interested in is crucial for the flexibility of a notification service. Content-based notification selection is most expressive because it allows to evaluate filter predicates over the whole content of a notification. The advantage in expressiveness compared to channel- or subject-based selection results in increased flexibility facilitating extensibility and change. On the other hand, scalable implementations of content-based notification services are difficult to realize. Indeed, the expressiveness of notification selection must be carefully chosen in large-scale systems, because expressiveness and scalability are interdependent. Hence, the most fundamental problem in the area of content-based publish/subscribe systems is probably the scalable routing of notifications from their producers to their respective consumers. Unfortunately, existing content-based notification services are not mature enough to be used in large-scale, widely-distributed environments. Most existing notification services are either centralized, use flooding, or use simple routing algorithms that assume that each event broker has global knowledge about all active subscriptions. All these approaches exhibit severe scalability problems in large-scale systems. In contrast to that, this thesis concentrates on mechanisms to improve the scalability of content-based routing algorithms and presents more advanced routing algorithms that do not rely on global knowledge. The algorithms presented here exploit similarities between subscriptions by using identity- and covering-tests, and by merging filters. While identity-based routing is a simplified version of covering-based routing, merging-based routing

is more advanced because it exploits the concept of filter merging. Furthermore, the idea of imperfect routing algorithms is introduced.

The thesis consists of a theoretical and a practical part. The theoretical part presents a formal specification of publish/subscribe systems, a routing framework and a set of routing algorithms, and discusses how the routing optimizations can be broken down to the actual data/filter model. The practical part presents the implementation of the REBECA notification service which supports advertisements and all the routing algorithms mentioned above. A detailed practical evaluation of the implemented algorithms based upon the prototype is also presented.

# Zusammenfassung

Heutzutage wird die Architektur verteilter Systeme von Client/Server-Plattformen dominiert, die auf dem synchronen Request/Reply Paradigma aufbauen. Diese Architektur ist jedoch nicht dafür geeignet, informationsgetriebene Applikationen (z.B. Nachrichtendienste, Aktienkursdienste, Flugüberwachungsdienste oder die Verbreitung von Auktionsgeboten) zu implementieren. Dies liegt daran, dass die Anforderungen dieser Applikationen und die Charakteristika von Client/Server Plattformen sich grundlegend voneinander unterscheiden. Im Gegensatz dazu bilden Publish/Subscribe Systeme die Eigenschaften von informationsgetriebenen Applikationen direkt ab, da in diesem Fall die Kommunikation indirekt ist und vom Produzenten der jeweiligen Informationen angestoßen wird: Produzenten veröffentlichen Benachrichtigungen und diese werden allen subskribierten Konsumenten durch einen zwischengeschalteten Benachrichtigungsdienst zugestellt. Daher bieten sich Publish/Subscribe-basierte Implementationen für die Umsetzung derartiger Applikationen an.

Die Ausdrucksfähigkeit der von den Konsumenten benutzten Nachrichtenselektion ist entscheidend für die Flexibilität eines Benachrichtigungsdienstes. Inhaltsbasierte Nachrichtenselektion ist am ausdrucksstärksten, weil in diesem Fall Prädikate über dem gesamten Inhalt einer Nachricht ausgedrückt werden können. Diese im Vergleich zu kanalbasierter und themenbasierter Selektion erhöhte Ausdrucksfähigkeit vergrößert die Flexibilität und begünstigt somit die Erweiterbarkeit und Änderbarkeit. Andererseits ist ein skalierbarer inhaltsbasierter Benachrichtigungsdienst schwierig zu realisieren. In der Tat muss die Ausdrucksfähigkeit der Nachrichtenselektion in großen Systemen sorgfältig festgelegt werden, weil Ausdrucksfähigkeit und Skalierbarkeit voneinander abhängig sind. Daher ist das wohl grundlegendste Problem im Bereich der inhaltsbasierten Publish/Subscribe Systeme das skalierbare Routen von Nachrichten von den Produzenten zu den entsprechenden Konsumenten. Unglücklicherweise sind die heutigen Benachrichtigungsdienste nicht weit genug entwickelt, um in großen, weit verteilten Umgebungen genutzt werden zu können. Die meisten existierenden Benachrichtigungsdienste sind entweder zentralisiert, benutzen „Flooding“ oder basieren auf einfachen Routing-Algorithmen, die annehmen, dass jeder Broker globales Wissen über alle im System aktiven Subskriptionen hat. Diese Ansätze haben allerdings ausgeprägte Skalierbarkeitsprobleme in großen Systemen. Diese Arbeit konzentriert sich auf Mechanismen, welche die Skalierbarkeit von in-

haltsbasierten Routing-Algorithmen erhöhen und präsentiert verbesserte Algorithmen, die kein globales Wissen voraussetzen. Die Algorithmen basieren auf Tests, die bestimmte „Ähnlichkeiten“ zwischen Subskriptionen erkennen können (Identitäts- und Überdeckungstests) und auf dem Konzept der Verschmelzung von Filtern. Darüber hinaus wird auch die Idee der „nicht perfekten“ Routing-Algorithmen eingeführt.

Im Einzelnen besteht die hier vorgestellte Arbeit aus einem theoretischen und einem praktischen Teil. Der theoretische Teil stellt eine formale Spezifikation von Publish/Subscribe Systemen, ein Routing-Framework und einige Routing-Algorithmen vor. Daneben wird auch diskutiert, wie die vorgeschlagenen Routing-Optimierungen auf das eigentliche Daten- und Filtermodell heruntergebrochen werden können. Der praktische Teil der Arbeit stellt die Implementierung des Benachrichtigungsdienstes REBECA vor, der die diskutierten Routing-Algorithmen und zusätzlich zu Subskriptionen auch Ankündigungen von Produzenten unterstützt. Außerdem wird eine detaillierte praktische Evaluation der Routing-Algorithmen vorgestellt, die auf dem implementierten Prototypen basiert.

# Preface

## Acknowlegements

First of all I would like to thank my advisor Prof. Alejandro P. Buchmann, Ph.D., for his invaluable help and advice, and Jean M. Bacon, Ph.D. for taking over the part of the second referee. Special thanks go to Ludger Fiege and Dr. Felix C. Gärtner. It has been a great pleasure to work with them. I also appreciate my friends and colleagues in the "Databases and Distributed Systems" group as well as in the Ph.D. Program "Enabling Technologies for Electronic Commerce" for their friendship and support. Furthermore, I am grateful to the DFG (Deutsche Forschungsgemeinschaft) that funded my work in form of a scholarship as part of the Ph.D. Program "Enabling Technologies for Electronic Commerce". Last but not least, I would like to thank my family for supporting me.

## Publications

Parts of this Thesis are based on previous publications: Chapter 2 is based on joint work with Ludger Fiege and Felix C. Gärtner [42]. Chapter 4 is partially based on joint work with Ludger Fiege and Alejandro Buchmann [73, 75, 77]. Chapter 6 is based on a publication with Ludger Fiege, Felix C. Gärtner, and Alejandro Buchmann [78]. A number of additional publications has not been incorporated into this thesis [38, 41, 43, 74, 76].

# Contents

# List of Figures

# List of Tables

# List of Symbols

| Symbol | Meaning | Defined |
|---|---|---|
| $n$ | a notification | |
| $\mathcal{N}$ | set of all notifications | p. 13 |
| $F$ | a filter | |
| $\mathcal{F}$ | set of all filters | p. 13 |
| $N(F)$ | set of notifications matched by a filter $F$ | p. 13 |
| $L_B$ | set of local clients of a broker $B$ | p. 25 |
| $N_B$ | set of neighbor brokers of a broker $B$ | p. 25 |
| $\mathcal{C}$ | set of all clients | p. 13 |
| $X, Y, Z$ | clients | |
| $G$ | the graph $G = (V, E)$ representing the broker network | p. 25 |
| $V$ | set of all brokers (node set of $G$) | p. 25 |
| $E$ | set of all connections among brokers (edge set of $G$) | p. 25 |
| $B$ | a broker | |
| $S_X$ | set of all active subscriptions of a client $X$ | p. 13 |
| $P_X$ | set of all notifications published by a client $X$ | p. 14 |
| $A_X$ | set of all active advertisements of a client $X$ | p. 20 |
| $T_B$ | Routing table of a broker $B$ | p. 26 |
| $\nu_B(\mathcal{W})$ | set of notifications that is forwarded by a broker $B$ to a subset of its local clients and neighbors $\mathcal{W}$ | Eq. 3.1 p. 26 |
| $F_B(n)$ | set of all destinations to which a broker $B$ forwards a notification $n$ | Eq. 3.2 p. 26 |
| $F_B^L(n)$ | set of all local clients to which a broker $B$ forwards a notification $n$ (subset of $F_B(n)$) | Eq. 3.3 p. 26 |
| $F_B^N(n)$ | set of all neighbors to which a broker $B$ forwards a notification $n$ (subset of $F_B(n)$) | Eq. 3.4 p. 26 |

| Symbol | Meaning | Defined |
|---|---|---|
| $I_B$ | set of all notifications that are of interest to any local client of a broker $B$ | Eq. 3.5 p. 28 |
| $\eta_{B_i,B_j}$ | The set of notifications that is of interest to the subnet $G_{B_i,B_j}$. | Eq. 3.6 p. 28 |
| $\bar{S}_X$ | set of all active subscriptions of a client $X$ whose update process has terminated | p. 35 |
| $\bar{I}_B$ | subset of $I_B$ considering only subscriptions whose update process has terminated | Eq. 3.7 p. 35 |
| $\bar{\eta}_{B_i,B_j}$ | subset of $\eta_{B_i,B_j}$ considering only subscriptions whose update process has terminated | Eq. 3.8 p. 35 |
| $Q_B^E(D)$ | set of all filters of all routing entries in $T_B$ except those regarding destination $D$ | Eq. 3.9 p. 49 |
| $Q_B^O(D)$ | set of all filters of all routing entries in $T_B$ regarding destination $D$ | Eq. 3.10 p. 49 |
| $C_B^I(F,D)$ | set of all routing entries in $T_B$ regarding destination $D$ whose filter is identical to $F$ | Eq. 3.13 p. 51 |
| $D_B^I(F)$ | set of all destinations $H$ for which there is no filter in $Q_B^E(H)$ that is identical to $F$ | Eq. 3.14 p. 51 |
| $C_B^L(F)$ | set of all routing entries in $T_B$ whose filter is covered by $F$ | Eq. 3.15 p. 57 |
| $C_B^L(F,D)$ | subset of $C_B^L(F)$ regarding destination $D$ | Eq. 3.16, p. 57 |
| $D_B^U(F)$ | set of all destinations $H$ for which there is no filter in $Q_B^E(H)$ that covers $F$ | Eq. 3.17 p. 57 |
| $D_B^{RU}(F)$ | set of all destinations $H$ for which there is no filter in $Q_B^E(H)$ that really covers $F$ | Eq. 3.18 p. 57 |
| $\delta$ | link delay | p. 72 |
| $\pi$ | leasing period | p. 72 |
| $\rho$ | refresh period | p. 72 |
| $d$ | diameter of the network | p. 73 |
| $\Delta T$ | stabilization time | p. 74 |
| $a_i$ | an attribute | p. 81 |
| $A_i$ | an attribute filter | p. 82 |
| $L_A(A_i)$ | set of values matching an attribute filter | p. 82 |
| $S$ | an element selector | p. 98 |
| $C$ | an element filter | p. 98 |
| $P$ | a path filter | p. 98 |

| Symbol | Meaning | Defined |
|---|---|---|
| $L_E(C)$ | set of elements matches by an element filter $C$ | p. 99 |
| $L_S(S)$ | set of elements selected by an element selector $S$ | p. 99 |
| $L_P(P)$ | set of elements matched by a path filter $P$ | p. 99 |
| $m$ | number of stocks | p. 135 |
| $S$ | the source of notifications | p. 136 |
| $x$ | the number of active subscriptions | p. 133 |

# Chapter 1

# Introduction

The architecture of current large-scale and networked computer-systems is dominated by synchronous client/server platforms (e.g., the World Wide Web, CORBA [80], J2EE [104], and COM+ [93]). In client/server systems two roles exist: A component acts as a *client* if it requests data or functionality from another component; it acts as a *server* if it responds to a client's request. Moreover, a client is blocked after it has issued a request, until the corresponding reply arrives. RPC (remote procedure call) [52, 108] and more recently object-oriented successors like RMI (remote method invocation) [106] automated request/reply-based interaction in such a way that calling a remote procedure/method is almost identical to the local case. For example, arguments and return values are automatically marshalled and unmarshalled.

The main advantages that lead to the widespread use of RPC-like platforms are their simplicity and familiarity. Other implementations of the request/reply pattern use explicit message passing. These implementations offer more flexibility but are also more complex. Besides their indisputable advantages, RPC-like platforms also have some inherent disadvantages. One of the main deficiencies is the tight coupling among the involved components, i.e., the clients and the servers: A client needs to explicitly address the server that shall process the request, the server must be ready and able to process the request, and the client is blocked, until it receives the reply. Because of these inherent disadvantages a large range of applications cannot be realized efficiently by using request/reply.

Consider the following example, where an automated stock trading program monitors stocks and bases its decisions to sell or buy certain stocks on current real-time quotes. If this program is realized using request/reply, it has to periodically retrieve the current quote of a specific stock by requesting it from a quoting server. Each time communication and processing cost would be incurred, and in many cases no new information is delivered. This approach is called *polling*. Polling leads to resource waste because it unnecessarily saturates the servers, the network, and the clients [45]. This not only impedes scalability, in a large-scale system, it may even lead to a total breakdown of the network

connections or the server itself.

In the above example, it is important that quote updates are available to the trading program with *minimum latency* and that no quote updates are missed. Both requirements can only be met using a high polling frequency. Hence, data freshness and completeness are inconsistent with a low data fetching overhead. Moreover, synchronous polling blocks the client until the reply arrives. This means that multiple polling requests (e.g., for different stocks) either have to be executed sequentially or error-prone multi-threaded polling must be used. The same would be true if multiple data sources (e.g., different stock exchanges) were involved.

As another example, Bacon, Moody, and Yao [6] proposed to disseminate revocations of security credentials instead of checking them every time they are needed or on a regular basis. Polling is also inappropriate for applications that run on mobile devices (e.g., PDAs). These devices are especially susceptible to resource waste because of their limited processing power and network bandwidth.

These examples show that there are applications that cannot be realized efficiently by using request/reply. Moreover, these examples are not in any way exceptions but rather typical examples of *information-driven* applications, in which information provided by a service depends on information supplied by other services. Realizing such applications using request/reply will always lead to implementations that do not scale and provide data that is inaccurate and probably incomplete.

Of course, the disadvantages of synchronous computing platforms have been recognized before and lead to the development of a number of extensions for these platforms. Some of the disadvantages can be diminished by using asynchronous request/reply. For example, the Synchronous Method Invocation (SMI) of CORBA was supplemented by the CORBA messaging specification [51] which introduced the Asynchronous Method Invocation (AMI) and the Time Independent Invocation (TII). The former achieves that clients are not blocked while they are waiting for a reply, and the latter allows for requests to be buffered until the addressed server becomes available. These extensions circumvent the problems raised by synchronous invocation, but still each client has to request the data from a specific server. These problems are approached by a new communication paradigm called publish/subscribe that recently gained increased publicity in the distributed systems research area. Publish/subscribe is an asynchronous communication paradigm that is also the basis for extensions [71] and supplementary services [81, 82, 92] that have been added to standard middleware recently.

## 1.1 Publish/Subscribe Systems

A *publish/subscribe system* consists of a set of clients that asynchronously exchange notifications, decoupled by a notification service that is interposed between them. *Clients* can be characterized as producers or consumers. *Producers*

publish notifications (such as current stock quotes), and *consumers* subscribe to notifications by issuing *subscriptions*, which are essentially stateless message filters. Consumers can have multiple active subscriptions, and after a client has issued a subscription the notification service delivers all future matching notifications that are published by any producer until the client cancels the respective subscription. In this thesis it is assumed that producers and consumers are implemented in a way such that they publish and subscribe to the intended events as required by the application.

Publish/subscribe systems have a number of interesting characteristics. Firstly, producers do not need to address consumers and vice versa. Instead, consumers simply specify the notifications they are interested in. This loosely coupled approach facilitates flexibility and extensibility because new consumers and producers can be added, moved, or removed easily. Secondly, communication is asynchronous, thereby removing the disadvantages and inflexibility of synchronous communication described above. Thirdly, producers and consumers do not need to be available at the same time. This means that a subscription causes notifications to be delivered even if producers join after the subscription was issued. Finally, publish/subscribe directly reflects the intrinsic behavior of information-driven applications because communication is initiated by producers of information.

The benefits of publish/subscribe make them first choice for implementing information-driven applications. For example, publish/subscribe is well suited for information dissemination applications like news delivery, stock quoting, air traffic control [66], and dissemination of auction bids [12]. The use of publish/subscribe techniques has also been described in the areas of mobile agents [102], work flow systems [29], ubiquitous computing [64], peer-to-peer systems [57], and process control systems [60]. Furthermore, the use of publish/subscribe was proposed for loose coupling of components [10] or several independent distributed applications.

Publish/subscribe systems exhibit a lot of interesting challenges, but a fundamental aspect in any publish/subscribe system is the expressiveness of the notification selection, i.e., how consumers specify subscriptions. Choosing the notification selection mechanism is perhaps the most important (and most difficult) choice to be made when developing a notification service.

## 1.2 Notification Selection Mechanisms

The expressiveness of notification selection is crucial for the flexibility of a notification service. In large-scale systems the expressiveness of notification selection must be carefully chosen because expressiveness and scalability are interdependent. On the one hand, insufficient filter expressiveness can lead to unnecessarily broad subscriptions saturating the network and raising the need for additional consumer-side filtering. In the worst case, when interests cannot be mapped to selection mechanisms, all notifications must be delivered and the selection is

imposed on to the consumer. On the other hand, scalable implementations of more expressive filtering models require more complex delivery strategies [20]. In the following, three common notification selection mechanisms are described: channels, subjects, and content-based selection.

**Channel-based Selection.** The first generation of publish/subscribe systems [55, 82, 109] used *channels*. Here, a notification is published with respect to a specific channel that is specified by the producer, and each notification that is published to a channel is delivered to all consumers that have subscribed to this channel. Channels can be implemented efficiently because they can easily be mapped to multicast groups, but they have some inherent disadvantages. Firstly, the expressiveness, i.e., the filtering capability, of channels is rather limited because notifications can only be classified with respect to a number of channels. If no channel exists that perfectly matches the interests of a consumer, the consumer has to subscribe to multiple channels and/or has to carry out additional filtering on its own. In general, a low number of channels and a high selectivity are contradicting issues. Hence, there will rather be a channel for all stocks of a specific market segment than for every single stock. But what if the user is interested in a single stock only? Secondly, channels are inflexible and inhibit changes. If the assignment of notifications to channels changes, both producer and consumers may have to be changed: Producers may be forced to publish notifications to different channels, while consumers may have to subscribe to various channels and may have to carry out consumer-side filtering. Finally, producers and consumers are not fully decoupled because the producer decides into which channel(s) a notification is published.

**Subject-based Selection.** In *subject-based* publish/subscribe systems [83, 114] producers publish notifications with respect to a certain subject that is usually specified as a dot separated string (e.g., `market.quotes.NASDAQ`). Subjects are arranged in a *subject tree* by using a dot notation, and clients can either subscribe to a single subject (e.g., `market.quotes.NASDAQ.FooInc`) or to a subject and all of its subordinate subjects (e.g., `market.quotes.NASDAQ.*`). In some systems it is also possible to express slightly more complex constraints on the subject of notifications.

The above examples show that subjects provide more powerful notification selection than channels. Nevertheless, subjects have a number of drawbacks. Firstly, they have still a limited expressiveness. With subjects it is possible to have a subject for each single stock, but what if the user is interested in the stock price only if it rises above a certain limit? Secondly, subjects are only suitable to divide the notification space with respect to one dimension. The use of several dimensions leads to an explosion of the tree size because of subtree repetition. In any case, the question arises who defines the subject tree. Thirdly, changes to the subject tree can require major application fixes. For example, consumers may have to subscribe to other subjects and may have to carry out distinct consumer-side filtering. Furthermore, producers and consumers are still

not fully decoupled because the producer determines the subject under which a notification is published.

**Content-based Selection.** *Content-based selection* allows subscriptions to evaluate the whole content of notifications, and so it provides a more powerful and flexible notification selection than channel- or subject-based mechanisms, for which the actual content of a notification is opaque. The increase in expressiveness allows the delivery of uninteresting notifications to be reduced or even to be avoided. In particular, this is important for applications that run on mobile devices having limited processing power and network bandwidth. For example, content-based selection makes it possible to subscribe only to those quotes of a certain stock whose price is above a certain limit. Moreover, only content-based selection provides full decoupling of producers and consumers, facilitating extensibility and continual change. Clearly, content-based selection is the most flexible notification selection mechanism, but on the other hand, scalable implementations are the most complex to realize, too. Indeed, the expressiveness of the selection predicates that can be applied has a large impact on the scalability of any content-based notification service.

In the literature, several systems relying on content-based selection are described. Early work was presented by Carriero and Gelernter [16, 48] in the context of Linda Tuplespaces. Today, content-based selection is used by a set of notification services including Elvin [44, 100, 101], Gryphon [1, 9, 8, 85, 86, 103], Siena [17, 22], Le Subscribe [37, 36, 88, 89], JEDI [13, 14, 28, 29, 30], CEA [4, 5, 7, 10, 56, 71], Hermes [90, 91], and Rebeca [38, 41, 42, 73, 75, 76]. Moreover, the Corba Notification Service Specification [81] and the Java Message Service Specification [111] also rely on content-based filtering.

This thesis concentrates on content-based selection. More precisely, it investigates different possibilities for implementing content-based selection in a scalable way by content-based routing.

## 1.3 Content-based Routing

Clearly, a centralized notification service is easy to implement, but neither scalable nor fault-tolerant. The alternative is to distribute the functionality of the service by using a set of cooperating *event brokers*. In this scenario each broker manages an exclusive subset of the clients, and notifications are propagated from producers to consumers along a path of interconnected brokers. To achieve this, each broker forwards a notification it processes to a (possibly empty) subset of the brokers it is connected to, i.e., its *neighbors*. This must be done in a way that guarantees that a notification is delivered to all interested consumers. In a broker topology with cycles additional care must be taken to avoid delivering duplicate notifications. In order to simplify the discussion, only acyclic topologies are considered in this thesis.

The technique of *flooding* is the simplest approach to implement a notification service that is distributed. Here, every broker forwards a notification that is published by one of its local clients to all of its neighbors, and if a broker receives a notification from a neighbor, it simply forwards it to all other neighbors; of course, a notification is also delivered to all local clients with matching subscriptions. With flooding, routing is trivial, but obviously a lot of unnecessary messages (which have no consumers at the other end) may be exchanged among brokers because each published notification is eventually processed by every broker.

An alternative to flooding is *content-based routing*. Here, each broker has a *routing table* that is used to route notifications based on their content to local clients and neighbors. Compared with flooding, content-based routing reduces the number of notifications that are forwarded, but complicates notification forwarding and introduces the necessity to update routing tables if subscriptions change. In general, the routing tables are maintained by forwarding information regarding new and canceled subscriptions through the broker network. Two categories of content-based routing algorithms can be distinguished. *Perfect* routing ensures that a notification is only forwarded to a neighbor broker iff in the corresponding subnet a consumer with a matching subscription exists. With *imperfect* routing, on the other hand, notifications may be forwarded that have no subscription. Up to now, only perfect routing algorithms were considered.

Two versions of (perfect) content-based routing are known, simple routing and covering-based routing. Moreover, advertisements can be used to further optimize content-based routing.

**Simple Routing.** The most naive filtering-based routing algorithm is *simple routing*. This routing scheme incorporates a routing entry for every active subscription in the routing tables of all brokers by flooding new and canceled subscriptions in the broker network. This ensures that a published notification is delivered to all interested consumers, while minimizing the amount of notification traffic in the system. However, with simple routing the routing tables become rather large, because this routing scheme enforces that *all* brokers have knowledge about *all* active subscriptions. Moreover, every routing table is affected if a subscription changes. Obviously, these are undesirable features in large systems. Simple routing is used, for example, by Gryphon [59].

**Covering-Based Routing.** As an alternative to simple routing Carzaniga [17] has shown that global knowledge about all active subscriptions is not necessary for implementing a perfect routing algorithm. His algorithms use selective subscription forwarding based on covering-tests among subscriptions to avoid flooding *all* subscriptions in the broker network. For example, a new subscription is not forwarded to a neighbor if previously a subscription has been forwarded to that neighbor that covers the former and that has not been canceled yet. Unfortunately, this also implies that if a subscription is canceled, it might

be necessary to forward some other subscriptions to some neighbors. This concerns a subset of the subscriptions that is covered by the canceled subscription. Covering-based routing is used by SIENA [17, 22] and JEDI [29].

**Using Advertisements.** *Advertisements* are filters that are issued by producers to indicate their intention to publish notifications. Advertisements can be used as an additional mechanism to further optimize content-based routing [17]. For this purpose a second routing table is managed by every broker. This *advertisement routing table* can be maintained by the same algorithms as the *subscription routing table*, i.e., by forwarding new and canceled advertisements through the broker network. While the subscription routing tables are used to route notifications from producers to consumers, the advertisement routing tables are used to route subscriptions from consumers to producers: a subscription is only forwarded to a neighbor if it overlaps with an active advertisement that has been received from this neighbor before. Most filtering-based routing algorithms can be combined with advertisements. Advertisements are not supported by previously implemented notification service prototypes.

## 1.4 Shortcomings of Current Approaches

Current approaches to content-based notification services exhibit a number of serious deficiencies which are now explained.

**Lacking Formalization.** A formal specification offers a stable basis for any reasoning about a system and for comparing different systems without misunderstandings. While most work on publish/subscribe imparts an intuitive notion of the correct behavior of publish/subscribe systems, currently no formal treatment of the semantics of publish/subscribe exists. Such a specification would not only be useful to gain a deeper insight into the details of publish/subscribe systems, but would also facilitate reasoning of the correctness.

**Limited Scalability or Expressiveness.** Today most content-based notification services are either centralized (e.g., Le Subscribe [89]) or rely on simple routing (e.g., Gryphon [8, 59, 85]). Both approaches exhibit scalability problems in large-scale systems. SIENA [17, 22] and JEDI [29] exploit covering-based routing. Clearly, covering-based routing can improve the scalability, but in some situations it might be too complex (e.g., if two subscriptions are either disjoint or identical) or even not sophisticated enough. Therefore, alternative routing algorithms should be developed and evaluated by comparing them to each other. In particular, there is currently no work dealing with imperfect routing algorithms. It can be expected that imperfect routing algorithms lead to smaller routing table sizes and maintenance overhead, but also to less efficient filtering. Therefore, they might be a good compromise in (more) dynamic environments. While improving scalability by applying covering-based routing, SIENA and JEDI also

restrict the expressiveness of content-based routing. Both systems only support a number of predefined filtering predicates that work on primitive types (e.g., integers). In contrast to that, filtering should support more complex types, and the set of filtering constraints should be extensible rather than predefined.

**Limited Implementation.** The implementations of current notification services often rely on a single routing algorithm. This reduces the effort needed to realize an implementation, but inhibits comparing routing algorithms to each other in a uniform environment and under the same conditions. This would be facilitated by an implementation that supports a set of distinct routing algorithms and allows to easily add new routing algorithms. Moreover, most current notification services are only validated by trivial example applications like a ticker tape. These applications show that the basic functionality of the notification service is working. Here, only JEDI is an exception which is applied to the development of a more complex application, a work-flow system [29].

**Insufficient Evaluations.** One of the obvious questions that are raised by work on content-based routing is whether and in what environments filtering is superior to flooding. This question is difficult to answer because the efficiency of filtering when compared to flooding depends on many parameters. Because of that, existing evaluations of content-based routing [8, 13, 14, 17, 22, 85] have not answered this question properly. Currently, only a few rules of thumb exist. It is known that in the worst case filtering degrades to flooding. This worst case occurs if for any published notification there is a matching consumer at every broker. It is also clear that the effectiveness of filtering decreases the more often the subscriptions (and advertisements) change. Finally, locality among the interests of clients seems to improve the efficiency of filtering. But besides these reasonable rules, what can be said about other scenarios?

## 1.5 Focus and Contribution of this Thesis

The focus of this thesis is on content-based publish/subscribe systems. This thesis consists of two parts, a theoretical part and a practical part. The theoretical part deals with the formal semantics of publish/subscribe systems and advanced content-based routing algorithms. The practical part describes the implementation of the Rebeca notification service and two example applications and presents a detailed evaluation of the implemented routing algorithms. The contributions of this thesis are:

**Formal Treatment of Publish/Subscribe Semantics.** A formal specification of publish/subscribe systems is introduced. The specification uses sequential traces and is based on the syntax of linear temporal logic. It allows one to reason about the correctness of concrete systems and deficiencies in the description of the semantics of current systems to be pointed out. In this thesis the

specification also serves as a basis for discussing the correctness of the routing algorithms. Moreover, the notion of self-stabilizing publish/subscribe systems is introduced, which is based on a weakened version of the specification of fault-free publish/subscribe systems. Finally, a formalization of routing configurations is presented that allows their correctness to be reasoned about.

**Content-Based Routing Algorithms.** Content-based routing algorithms are proposed including identity-based routing and merging-based routing. While the former is a simplified version of covering-based routing, the latter is more advanced and exploits the concept of filter merging. The correctness of the discussed routing algorithms is proved by using the formalization of publish/subscribe systems and routing configurations. Besides this, how self-stabilizing publish/subscribe systems can be realized through subscription (and advertisement) leasing is presented. This allows a system to recover from arbitrary transient faults within a finite time. To complete this part of the thesis, it is also shown how the proposed routing optimizations can be broken down to the data/filter model that is used. Here, the focus is on name/value pairs, but ideas on supporting semistructured data (e.g., XML) and objects are also described.

**Implementation.** As part of this thesis an implementation was carried out that is part of the REBECA notification service prototype. Moreover, two example applications have been implemented. In contrast to other work, REBECA incorporates not only a single routing algorithm but a set of routing algorithms (all those that were discussed above). Moreover, advertisements are supported and new routing algorithms can easily be added. Hence, the implementation allows to experiment with different routing algorithms and to compare them to each other in a uniform environment. REBECA is built upon a flexible and powerful filtering framework instead of predefined filtering predicates. REBECA also provides service factories and basic support for replaying past events. Besides the notification service, two example applications were implemented, a stock trading platform and an infrastructure for self-actualizing web-pages. These applications show that non-trivial applications can be realized in a scalable way with the implemented notification service.

**Evaluation.** A detailed practical evaluation of the content-based routing algorithms is presented. The evaluation differs from previous ones by a combination of two factors: (1) it is focused on the inherent characteristics of routing algorithms (routing table sizes and filter forwarding overhead) instead of system-specific parameters (CPU load etc.), and (2) it is based on a working prototype (REBECA) instead of simulations. Moreover, several routing algorithms are compared to each other, and the effects of locality regarding the interests of the consumers are investigated. Finally, an evaluation of imperfect merging is carried out. The derived results offer new and detailed insights into the behavior of content-based routing algorithms: 1) Using advanced routing algorithms in

large-scale publish/subscribe systems can be considered *valuable*. 2) The use of advertisements considerably improves the scalability. 3) Advanced routing algorithms operate efficiently in more dynamic environments than was previously thought. 4) The good behavior of the algorithms even improves if the interests of the consumers are not evenly distributed, which can be expected in practice. 5) The evaluation of imperfect merging shows that it is suited to further reduce the routing table sizes.

## 1.6 Structure of the Thesis

Chapter 2 introduces a formal specification of publish/subscribe systems which is based on traces. In Chapter 3 a framework for content-based routing is introduced that builds upon the formal specification of publish/subscribe systems. A set of routing algorithms is described and their correctness is discussed including flooding, simple routing, identity-based routing, covering-based routing, and routing based on filter merging. Chapter 4 discusses how the routing optimization proposed in the preceding chapter can be supported for some data/filter models. The focus of this chapter is on structured records which are based on name/value pairs. In Chapter 5 the implementation that has been carried out as part of this thesis is described. The implementation includes the prototype of the Rebeca notification infrastructure and two example applications, a stock trading platform and an infrastructure for dynamic web pages. Chapter 6 presents a detailed practical evaluation that has been carried out by using the implemented notification service prototype. Chapter 7 summarizes the contributions of this thesis and sketches areas of future work.

# Chapter 2

# Formal Specification of Publish/Subscribe Systems

## Contents



## 2.1 Introduction

In this chapter a formal specification of publish/subscribe systems is described. The specification uses sequential traces and is based on the syntax of linear temporal logic [72, 94]. It permits the correctness of concrete systems to be reasoned about and deficiencies in the description of the semantics of current systems to be pointed out. In this thesis it also serves as the basis for proving the correctness of the routing algorithms. The notion of self-stabilizing publish/subscribe systems is introduced based on a weakened version of the specification of normal publish/subscribe systems. Self-stabilizing systems are able to recover from arbitrary transient faults autonomously.

In the following, first the interface of publish/subscribe systems is described (Sect. 2.2). After that, trace-based specifications are introduced (Sect. 2.3) and a trace-based specification of publish/subscribe systems is presented (Sect. 2.4). Subsequently, self-stabilizing algorithms and a specification for self-stabilizing

publish/subscribe systems is given (Sect. 2.5). Finally, a specification which considers advertisements of producers is presented (Sect. 2.6).

## 2.2 Interface of Publish/Subscribe Systems

There is a considerable amount of work on notification services, and many concrete systems have been designed and implemented (e.g., SIENA [22], JEDI [29], etc.). Unfortunately, understanding and comparing these systems is difficult because of differing and informal semantics. Informal requirements can be demanded for a publish/subscribe system. For example, we could require that

- only notifications should be delivered to a client that match one of its active subscriptions,
- each notification should be delivered to a client at most once,
- notifications should be delivered in some order with respect to their publication (e.g., in causal order etc.), and
- all notifications matching an active subscription should be delivered to the respective client.

But how can these requirements be specified unambiguously and which ones are really mandatory for the basic service level of a useful publish/subscribe system? To answer these questions, first a way to capture the behavior of a publish/subscribe system is needed. After that, it can be defined what it means for a publish/subscribe system to be correct.

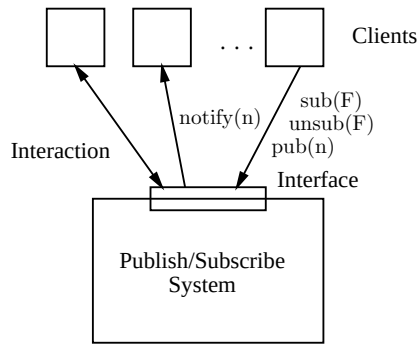



Figure 2.1: Black box view of a publish/subscribe system.

**Interface Operations.** Conceptually, a *system* can be viewed as a black box with an interface. The *interface* of the system offers a number of *operations* each of which may take a number of *parameters*. While *input operations* can be

invoked from the outside, *output operations* are invoked by the system to deliver information to the outside. Note that this does not mean that the *implementation* of the system consists of a single component (cf. Chapter 3) or only has a single interface. Moreover, "invocations" on the interface of a system can be implemented by, for example, message passing. Indeed, the approach is used to abstract from the internal structure and implementation details of a system and to specify, inspect, and verify the behavior of a system by only looking at its interface.

In the case of a *publish/subscribe system* (see Fig. 2.1), a set of *clients* is interacting with the system. Clients publish notifications by invoking the $pub(n)$ operation, giving the notification $n$ as parameter. The published notification can potentially be delivered to all connected clients via an output operation called $notify(n)$. Clients register their interest in specific kinds of notifications by issuing subscriptions via the $sub(F)$ operation which takes a filter $F$ as parameter. Each client can have multiple active subscriptions which must be revoked *separately* by using the $unsub(F)$ operation. All these operations are instantaneous and take parameters from different domains: the set of all clients $\mathcal{C}$ , the set of all notifications $\mathcal{N}$ , and the set of all filters $\mathcal{F}$ . Formally, a filter $F \in \mathcal{F}$ is a mapping from $\mathcal{N}$ to the boolean values *true* and *false*. A notification $n$ *matches a filter* $F$ iff $F(n)$ evaluates to *true*. The set of all notifications that match $F$ is denoted by $N(F)$ . Additionally, two further assumptions are made: Firstly, it is assumed that notifications are unique, i.e., each notification can only be published once. Secondly, every filter is associated with a unique identifier in order to enable the publish/subscribe system to identify a specific subscription.

## 2.3 Trace-Based Specifications

| | |
|---|---|
| $sub(X, F)$ | Client $X$ subscribes to filter $F$ |
| $unsub(X, F)$ | Client $X$ unsubscribes to filter $F$ |
| $notify(X, n)$ | Client $X$ is notified about $n$ |
| $pub(X, n)$ | Client $X$ publishes $n$ |

Table 2.1: Interface operations of a publish/subscribe system

The behavior of the publish/subscribe system is specified by solely looking at its interface. The interface is associated with a set of *variables*. A subset of the variables are *specification variables* which are fictitious devices sometimes necessary to keep track of the internal history of the system within a specification. Two sets of specification variables are assumed at the interface for every client $X \in \mathcal{C}$:

1. a set $S_X$ of *active subscriptions* (i.e., all filters which $X$ has subscribed and not unsubscribed to yet) and

2. a set $P_X$ of *published notifications* (i.e., the subset of $\mathcal{N}$ containing all notifications $X$ has previously published).

It is assumed that these sets are initially empty and that they are updated faithfully according to the operations that occur at the interface of the system. For example, whenever $X$ subscribes to $F$, $F$ is added to $S_X$, and whenever $X$ unsubscribes to $F$, $F$ is removed from $S_X$. Hence, multiple (un)subscriptions to the same filter are idempotent. This also implies that if a client $X$ subscribes to a filter $F$ multiple times and then unsubscribes to this filter once then $F$ is no longer in $S_X$ afterwards.

An assignment of values to the variables is called a *state* of the interface. Invoking operations on the interface results in atomic state changes of the variables. Therefore, individual behaviors of the system can be described as a sequence of *states* $s_n$ interleaved with *operations* $op_n$. Such a sequence

$$\sigma = s_1, op_1, s_2, op_2, s_3, \ldots \tag{2.1}$$

is called a *trace* of the system. In a trace, $s_1$ is called the *initial state* of the system. In order to capture which client is affected by an operation, the operations are extended to include the respective client (see Table 2.1). For example, $sub(X, F)$ means that client $X$ subscribes to filter $F$. Note that a trace in this model reduces time to the relative ordering of operations within a trace. For example, the trace

$$\sigma_1 = s_1, sub(X, F), s_2, pub(Y, n), s_3, notify(X, n), s_4, \ldots \tag{2.2}$$

describes that in the initial state $s_1$ client $X$ subscribes to a filter $F$. After that, in the resulting state $s_2$, client $Y$ publishes a notification $n$, which in turn results in state $s_3$. The next state $s_4$ results from client $X$ receiving the notification $n$, and so on. Note also that the trace $\sigma_1$ does not require that $n$ matches $F$. In fact, many useless traces can be defined. For example, the trace

$$\sigma_2 = s_1, unsub(X, F), s_2, notify(X, n), s_3, \ldots \tag{2.3}$$

describes that $X$ unsubscribes to a filter it has never subscribed to and that it receives a notification although it never subscribed to anything. The task now is to find suitable restrictions on the set of all traces that express exactly what is expected from a publish/subscribe system (e.g., that a delivered notification has to match an active subscription of the respective client).

In order to specify the set of correct traces which a system may exhibit, a way to impose predicates on traces is needed. Let $\sigma = s_1, op_1, s_2, op_2, s_3, \ldots$ be a trace. For every operation $op$ (e.g., $pub(n)$) of the publish/subscribe system, a *predicate* $Op$ (here, $Pub(n)$) is defined on traces in the following way:

$$Op(\sigma) = true \Leftrightarrow op_1 = op. \tag{2.4}$$

This means that the predicate holds if the respective operation is the first one in the trace. For example, the predicate $Sub(X, F)$ holds for trace $\sigma_1$ (Eq. 2.2)

above because $sub(X, F)$ is the first operation in $\sigma_1$. The formal language used to specify sets of traces is built from the above predicates, the quantifiers $\forall$ and $\exists$, the logical operators $\vee$, $\wedge$, $\Rightarrow$, $\neg$ and the "temporal" operators $\Box$ ("always"), $\Diamond$ ("eventually"), and $\circ$ ("next") which are borrowed from temporal logic [94]. For example, the formula $\neg Sub(X, F)$ is true for a trace $\sigma$ iff the first operation in $\sigma$ is *not* $sub(X, F)$. The semantics of the temporal operators is defined as follows: let $\Psi$ be an arbitrary formula. Then

- $\Diamond\Psi$ is true for trace $\sigma$ iff 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 $\sigma$ iff $\Psi$ is true for $s_2, op_2, s_3, op_3, \ldots$.

Note that the temporal operators have higher precedence than the logical operators. Intuitively, $\Diamond\Psi$ means that $\Psi$ will hold eventually, i.e., there exists a point in the trace at which $\Psi$ holds. For example,

$$\Diamond Notify(X, n) \tag{2.5}$$

specifies all traces in which client $X$ eventually is notified about $n$. This predicate holds for $\sigma_2$ (Eq. 2.3). On the other hand, $\Box\Psi$ means that $\Psi$ always holds, i.e., for all "future" points in the trace $\Psi$ holds.

$$\Box\neg Unsub(X, F) \tag{2.6}$$

specifies all traces in which $X$ never unsubscribes to $F$. Finally, $\circ\Psi$ means that $\Psi$ holds after the next step, i.e, for the trace starting with $s_2, op_2$.

$$\Box\big[Notify(Y, n) \;\Rightarrow\; \circ\Box\neg Notify(Y, n)\big] \tag{2.7}$$

specifies all traces in which if $Y$ is notified about $n$ then $Y$ is never notified about $n$ again. Of course, it is also possible to refer to the specification variables. The trace,

$$\Box\big[Notify(Y, n) \;\Rightarrow\; [\exists F \in S_Y.\, n \in N(F)]\big] \tag{2.8}$$

specifies all traces in which the fact that $Y$ is notified about $n$ implies that *at this time*, i.e., in the corresponding state, there exists a subscription $F$ in $S_Y$ that matches $n$. It is important to keep in mind that the temporal operators determine the place in the trace to which the imposed conditions are applied. As a last example,

$$\Box\big[Notify(Y, n) \;\Rightarrow\; [\exists X.\, n \in P_X]\big] \tag{2.9}$$

requires that the fact that $Y$ is notified about $n$ implies that there is a client $X$ for that $n$ is in $P_X$ at this time. Subsequently, this implies that $n$ has been published by $X$ before.

Formally, a *specification* is a set of traces. Since the above formulas represent sets of traces, they will be used as a syntax to express specifications. A system is correct with respect to a specification $\Sigma$ if it exhibits only traces that are in $\Sigma$. Now we are prepared to specify the behavior of a publish/subscribe system.

## 2.4 Publish/Subscribe Systems

In the following, a specification of publish/subscribe systems is presented that relies on the trace-based semantics introduced above. The specification defines which traces a correct publish/subscribe system may exhibit.

**Definition 2.1 (publish/subscribe system)** *A* publish/subscribe system *is a system that exhibits only traces satisfying the following requirements:*

- *(Safety)*

$$\begin{aligned}\Box\Big[\mathit{Notify}(Y,n) \;\Rightarrow\; &\big[\circ\Box\neg\mathit{Notify}(Y,n)\big]\\ &\wedge\big[\exists X.\, n \in P_X\big]\\ &\wedge\big[\exists F \in S_Y.\, n \in N(F)\big]\Big]\end{aligned} \tag{2.10}$$

- *(Liveness)*

$$\begin{aligned}\Box\Big[\mathit{Sub}(Y,F) \;\Rightarrow\; &\big[\Diamond\Box\big(\mathit{Pub}(X,n)\wedge n\in N(F) \;\Rightarrow\; \Diamond\mathit{Notify}(Y,n)\big)\big]\\ &\vee\big[\Diamond\,\mathit{Unsub}(Y,F)\big]\Big]\end{aligned} \tag{2.11}$$

The specification consists of a safety and a liveness condition [62]. A *safety condition* demands that "something irremediably bad" will never happen, while a *liveness condition* requires that "something good" will eventually happen. It is known that many useful system properties can be expressed as the intersection of safety and liveness conditions [2, 47, 46]. Here, the safety condition states that a notification should never be delivered to a consumer more than once, that a delivered notification must have been published by a client in the past, and that a notification should only be delivered to a client if it matches one of the client's active subscriptions. The liveness condition is probably the most complicated to understand. It describes precisely under which conditions a notification must be delivered. The condition can be rephrased as follows: if a client $Y$ subscribes to $F$, then there exists a future time where the publishing of a notification $n$ matching $F$ will lead to a delivery of $n$ to $Y$. This can only be circumvented by $Y$ unsubscribing to $F$.

For example, trace $\sigma_1$ (Eq. 2.2) above satisfies both safety and liveness conditions, while $\sigma_2$ (Eq. 2.3) satisfies the liveness condition but violates the safety condition. As additional examples, consider the following traces where $F$ is a

filter and $n_i$ are notifications matching $F$ while $n'$ is a notification not matching $F$ (the intermediate states are omitted for brevity):

$$\sigma_3 = sub(Y, F), pub(X, n_1), notify(Y, n') \quad (2.12)$$

$$\sigma_4 = pub(X, n), sub(Y, F), unsub(Y, F), notify(Y, n) \quad (2.13)$$

$$\sigma_5 = sub(Y, F), pub(X, n_1), pub(X, n_2), pub(X, n_3), \ldots \quad (2.14)$$

Traces $\sigma_3$ and $\sigma_4$ violate the safety requirement because a notification is delivered to $Y$ that does not match an active subscription. In trace $\sigma_5$ client $Y$ subscribes to $F$ and client $X$ starts to publish a continuous sequence of notifications matching $F$. Since there is no *notify* in $\sigma_5$ it perfectly satisfies safety. However, it violates the liveness requirement (to satisfy liveness, there must be a point in the trace following the subscription where either $Y$ unsubscribes to $F$ or $Y$ begins to receive notifications).

Intuitively, the liveness requirement states that any *finite* processing delay of a subscription is acceptable. By abstracting from physical time, a concise and unambiguous characterization is obtained of what types of actions must be produced by the system under which conditions. According to the specification, delivery of a notification that match an active subscription $F$ of a client is only necessary if the client continuously remains subscribed to $F$. Because the system cannot tell the future, it must nevertheless still make a good effort to prepare delivery even though the client may later unsubscribe to $F$.

A trace is a sequence that is a total ordering of all operations that occur at the system interface. One might argue that assuming a total order of operations is unrealistic in a distributed system because it is not possible or desirable to enforce total ordering of operations. Indeed, specifications can be given that are not implementable, at least not efficiently. However, the specification of Def. 2.1 can be implemented because it imposes ordering relations only on local operations or on operations which intentionally should be causally related in any sensible implementation. For example, consider the safety requirement: (1) Whether or not $Y$ was notified about $n$ previously and whether $Y$ has a matching subscription can be detected locally. (2) Notifying $Y$ about $n$ does not make sense without $n$ having been published previously by some client. On the other hand, SIENA demands that a notification should only be delivered to a client if the client had a matching subscription at the time the notification was published. This requirement is difficult to realize because the publication of a notification and the act of subscribing are generally not causally related.

A system that only satisfies the safety condition is trivial to implement. Any system that never invokes a *notify* operation satisfies the imposed conditions. Similarly, it is easy to implement a system which only satisfies the liveness condition. Any system that delivers every notification that is published to all clients fulfills this condition. Hence, the challenge is to implement a system that satisfies *both* requirements.

## 2.5 Self-Stabilizing Publish/Subscribe Systems

The specification of publish/subscribe systems (see Definition 2.1) requires that the system is correct, i.e., exhibits the desired functionality at its interfaces, in all circumstances. In the context of fault-tolerant systems, satisfying the specification in the presence of faults would mean to *mask* the effects at the interface. In many situations, fault masking is extremely difficult and costly to implement. For example, the specification requires that a notification $n$ is never delivered to a process $X$ that does not have a matching subscription. If we assume that arbitrary transient faults can occur, we cannot guarantee this property because it is easy to construct a fault that corrupts the state such that the system "thinks" that $X$ subscribed to $n$. Of course, arbitrary transient faults are a rather strong fault assumption, but it is not unrealistic. It is quite easy to show that building a correct publish/subscribe system in the presence of arbitrary transient faults is impossible. The best that can be done in this situation is to demand that the system exhibits self-stabilizing behavior.

In the following, self-stabilizing publish/subscribe systems are introduced. Self-stabilizing systems are able to recover from arbitrary transient faults within a finite time. Moreover, they can be realized by a number of techniques known from the literature [98] (e.g., leases, etc.). The specification of self-stabilizing publish/subscribe systems presented here is based on a weakened version of the specification given in Definition 2.1.

### Self-Stabilization Algorithms

The concept of *self-stabilization* was introduced by Dijkstra [31] in 1973. He defined a system being self-stabilizing if "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". In contrast to that, a system which is not self-stabilizing may stay in illegitimate states forever. A comprehensive discussion of self-stabilization is given by Dolev [32]. The property of self-stabilization is generally viewed as a very strong fault-tolerance property which has proved to encompass a formal and unified approach to fault tolerance. The faults which are considered are arbitrary transient faults. A *transient fault* is a fault that may change the state of a system but not its behavior, e.g., memory state perturbations, message losses or alterations, process crashes with subsequent recovery. The property of self-stabilization models the ability of a system to recover from transient faults within a finite time without any intervention from the outside under the assumption that faults do not continue to occur. Hence, if faults do not occur for a long enough period of time, the system will start to work correctly again. Now we can formally define what it means for a system to be self-stabilizing [113].

**Definition 2.2 (self-stabilization)** *A system $S$ is* self-stabilizing for a specification $\Sigma$ *iff there exists a set $L$ of legitimate states such that the following two properties are satisfied:*

1. (Correctness) *Starting from any state in L, the algorithm satisfies* $\Sigma$.
2. (Convergence) *Starting from any state, the algorithm eventually reaches a state in L.*

Roughly speaking, a system is self-stabilizing, if starting from an arbitrary state, the system eventually starts to exhibit its intended behavior.

### Self-Stabilizing Publish/Subscribe Systems

In the previous subsection self-stabilizing systems have been introduced. So the question arises how the definition of publish/subscribe systems can be modified to make sense under the failure model of self-stabilization, i.e, how a temporarily incorrect behavior of the system can be tolerated. In general, under the fault assumption of self-stabilization it is impossible to require *any* property that prohibits certain states. What can be postulated is merely that such a property will be satisfied eventually. Therefore, the safety condition of Definition 2.1 (Eq. 2.10) has to be weakened. In contrast to that, the liveness condition (Eq. 2.11) can be left unchanged.

**Definition 2.3 (self-stabilizing publish/subscribe system)** *A* self-stabilizing publish/subscribe system *is a system which exhibits only traces satisfying the following requirements:*

- (Eventual Safety)

$$\begin{aligned} \Diamond\Box\Big[\mathit{Notify}(Y,n) \;\Rightarrow\; & \big[\circ\Box\neg\mathit{Notify}(Y,n)\big] \\ & \wedge \big[\exists X.\, n \in P_X\big] \\ & \wedge \big[\exists F \in S_Y.\, n \in N(F)\big]\Big] \end{aligned} \tag{2.15}$$

- (Liveness)

$$\begin{aligned} \Box\Big[\mathit{Sub}(Y,F) \;\Rightarrow\; & \big[\Diamond\Box\big(\mathit{Pub}(X,n) \;\wedge\; n \in N(F) \;\Rightarrow\; \Diamond\mathit{Notify}(Y,n)\big)\big] \\ & \vee \big[\Diamond\mathit{Unsub}(Y,F)\big]\Big] \end{aligned} \tag{2.16}$$

The eventual safety condition states that the system starting from any state will eventually begin and continue to satisfy the actual safety property.

## 2.6 Publish/Subscribe Systems with Advertisements

Many implementations of publish/subscribe systems have the notion of advertisements which are issued by producers to indicate their intention to publish

certain kinds of notifications. Today, advertisements are used for two main reasons: First, they are applied to optimize implementations of publish/subscribe systems [17]. Second, consumers may want to inspect the advertisements currently available, for example, in order to issue, change, or cancel subscriptions. Besides this, advertisements should also be used to control the notifications a producer publishes. For example, if a notification is published by a client that does not match any of its active advertisements, it should be discarded and not delivered to any client.

Advertisements can easily be integrated into the formal model of publish/subscribe systems presented here by adding two extra interface operations: Clients indicate their intention to publish certain kinds of notifications by issuing advertisements via the *adv* operation taking a filter $F$ as parameter. Similar to subscriptions, each client can have multiple advertisements which are canceled *separately* via the *unadv* operation which also takes a filter $F$ as parameter. Therefore, two additional interface operations $adv(X, F)$ and $unadv(X, F)$ as well as two new predicates on traces $Adv(X, F)$ and $Unadv(X, F)$ are added. Moreover, an additional specification variable $A_X$ is introduced which is the set of all active advertisements of a client $X$ (i.e., all filters which $X$ has advertised and not yet unadvertised).

**Definition 2.4 (publish/subscribe system with advertisements)** *A* publish/subscribe system *with advertisements is a system which exhibits only traces satisfying the following requirements:*

- (Safety)

$$\begin{aligned}\Box\Big[\Big(&\mathit{Notify}(Y,n) \;\Rightarrow\; \big[\circ\Box\neg\mathit{Notify}(Y,n)\big]\\ &\qquad\wedge\big[\exists X.\, n \in P_X\big]\\ &\qquad\wedge\big[\exists F \in S_Y.\, n \in N(F)\big]\Big)\\ \wedge\;&\Big(\mathit{Pub}(X,n) \;\wedge\; [\forall F \in A_X.\, n \notin N(F)] \;\Rightarrow\; \big[\Box\neg\mathit{Notify}(Y,n)\big]\Big)\Big]\end{aligned} \tag{2.17}$$

- (Liveness)

$$\begin{aligned}\Box\Big[&\big[[\mathit{Sub}(Y,F) \;\wedge\; \Diamond\mathit{Adv}(Z,G)] \;\vee\; [\mathit{Adv}(Z,G) \;\wedge\; \Diamond\mathit{Sub}(Y,F)]\big]\\ &\Rightarrow\; \big[\Diamond\Box\big(\mathit{Pub}(X,n) \;\wedge\; n \in N(F)\cap N(G) \;\Rightarrow\; \Diamond\mathit{Notify}(Y,n)\big)\big]\\ &\qquad\vee\big[\Diamond\,\mathit{Unsub}(Y,F)\big]\\ &\qquad\vee\big[\Diamond\,\mathit{Unadv}(Z,G)\big]\Big]\end{aligned} \tag{2.18}$$

In the above definition, the safety (Eq. 2.10) and the liveness condition (Eq. 2.11) have been changed in order to make sense if advertisements are used.

The safety condition has been sharpened such that if a notification is published that does not match any of the active advertisements of the publishing client, the notification should not be delivered to any client. The liveness condition has been weakened and can be rephrased as follows: if a client $Y$ subscribes to $F$ and a client $Z$ advertises $G$ (in arbitrary order), then there exists a future time where a notification $n$ published by $Z$ that matches $F$ *and* $G$ will lead to a delivery of $n$ to $Y$. This can only be circumvented by $Y$ unsubscribing to $F$ or by $Z$ unadvertising $G$. It is necessary to apply this weaker liveness condition in order to allow using advertisements to optimize the implementations of publish/subscribe systems. Of course it is also possible to combine the above definition with those of self-stabilizing publish/subscribe systems by simply weakening the safety condition to eventual safety.

## 2.7 Related Work

Currently, no other formal specification of publish/subscribe systems is known to the author. For example, in the SIENA system [17, 22], Carzaniga, Rosenblum, and Wolf make an effort in defining the semantics of a notification service but their specification remains rather vague and raises a number of problems:

For example, they demand that a notification is only delivered to a client if the client had a matching subscription at the time the notification was published. In contrast to that, the specification presented here requires only that the client has a matching subscription at the time the notification is delivered. This is much easier to detect (and to implement) in a distributed system. Moreover, clients are required to handle race conditions. For example, notifications may be delivered after cancellation of the respective subscriptions. With the specification presented here the delivery stops immediately. Furthermore, advertisements are not enforced meaning that notifications may be delivered to clients that did not match any of the producers‘ advertisements.

Finally, in SIENA, a client that unsubscribes to a filter implicitly unsubscribes to all filters that are covered by the former filter. This approach burdens the client to keep track of relations among the issued subscriptions. Moreover, the treatment of a subscription that partially overlaps with an unsubscription seems to be inappropriate because in this case the client keeps on receiving notifications that match the unsubscription. Hence, a purely set-oriented approach has not been realized. The interface presented here, is simpler because subscriptions are issued and revoked independently of each other. This avoids undesired side-effects and facilitates a simpler and more precise specification. Nevertheless, a set-oriented approach can easily be realized on top of the presented approach by a simple wrapper if this is desired.

In most other systems, practitioners' approaches dominate and at most the formal semantics of the subscription languages are given, neglecting the semantics of the event service itself.

## 2.8 Discussion

In this chapter a formal specification of publish/subscribe systems has been presented. It offers a number of benefits, but most importantly, it precisely defines what it means for a publish/subscribe system to be correct. The specification is based upon traces of state/operation pairs and uses the syntax of linear temporal logic. It reduces time to the relative ordering of operations within a trace. It comprises safety and liveness conditions, an approach that is well known from literature. In general, a *safety condition* demands that "something irremediably bad" will never happen, while a *liveness condition* requires that "something good" will eventually happen. Here, the safety condition merely states that no wrong notifications are delivered to a client, while the liveness condition requires that after a client has subscribed, eventually all matching notifications are delivered.

The specification of "normal" publish/subscribe systems is based on the assumption that either no faults occur or that all faults can be masked. While the former assumption is a rather unrealistic one, the latter one is nearly impossible to implement. The property of self-stabilization has proved to encompass a formal and unified approach to fault tolerance. It is generally viewed as a very strong fault-tolerance property as it allows systems to recover from arbitrary transient faults within a finite time. Moreover, a number of techniques (e.g., leases, etc.) are known from literature that can be used to realize self-stabilizing systems. Following this promising approach, the notion of self-stabilizing publish/subscribe systems has been introduced. In order to derive a specification of self-stabilizing publish/subscribe systems, the specification of "normal" publish/subscribe systems has been weakened to make sense under the failure model of self-stabilization. More precisely, the safety condition has been weakened to hold only eventually, while the liveness condition has been left unchanged.

# Chapter 3

# Content-Based Routing

## Contents

## 3.1 Introduction

A formal treatment of the foundations of content-based routing is useful to gain full insight into the behavior of publish/subscribe systems using this paradigm. In this chapter a formal framework for content-based routing algorithms is presented building upon the formal specification of publish/subscribe systems previously introduced in Chapter 2. Starting with the fault-free scenario, fault-tolerance in the sense of self-stabilization is added later. The novel contributions include a formalization of routing configurations, their validity, a routing framework, and a universal correctness criterion of legal framework instantiations. A number of routing algorithms are discussed as instances of the framework giving new insights into their detailed operation. The discussion includes new routing algorithms like identity-based routing and merging-based routing as well as known routing algorithms like flooding, simple routing, and covering-based routing. While identity-based routing is a simplified version of covering-based routing, merging-based routing is more sophisticated and exploits filter merging. Finally, it is shown how the framework can be made self-stabilizing and how advertisements can be integrated into the framework.

This chapter is structured as follows: First, the underlying system model is introduced that builds upon asynchronous message passing (see Sect. 3.2). After that, a formalization of routing configurations is presented (see Sect. 3.3), and it is shown what requirements a routing configuration must fulfill to be correct with respect to the specification of publish/subscribe systems presented in Chapter 2. In Section 3.4, a routing framework is introduced that defines how notifications are routed through the broker network by using filter-based routing tables, how control messages are exchanged among the brokers to update the routing tables, and how to implement an instance of a concrete routing algorithm. Sufficient conditions are given which a routing algorithm must satisfy in order to be correct. Subsequently, a number of routing algorithms is discussed as instances of the framework (see Sect. 3.5). This includes flooding, simple routing, identity-based routing, covering-based routing, and merging-based routing. The last two sections of this chapter present how the routing algorithms can be made self-stabilizing through subscription leasing (see Sect. 3.7) and how advertisements can be used for content-based routing (see Sect. 3.6).

## 3.2 System Model

In the following section the system model that underlies the subsequent discussions is presented. The system model assumes a distributed system consisting of a set of processes that communicate with each other by asynchronous message passing using reliable communication channels. It is further assumed that messages are delivered in FIFO order with respect to the sender and that no messages are duplicated, lost, corrupted, or erroneously sent. In Section 3.7 it is presented how the reliable channel assumption can be avoided.

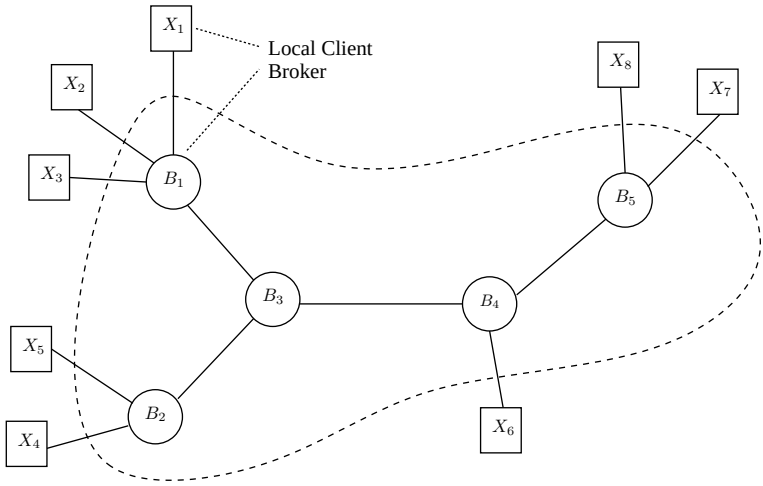



Figure 3.1: A distributed publish/subscribe system.

The publish/subscribe system is realized by a set of cooperating processes called *brokers* that are interconnected in an acyclic topology. Each broker $B$ acts as a local access point to the publish/subscribe system and manages an exclusive set of *local clients* $L_B$ which is a subset of all clients $\mathcal{C}$. Moreover, $B$ communicates with its *neighbor brokers* $N_B$ which are those brokers to which it is directly connected (see Fig. 3.1). Hence, the brokers exchange notifications and control messages to realize the functionality of the distributed notification service. The exact strategy is determined by a distributed content-based routing algorithm that defines how notifications are forwarded through the broker network and how filtering of notifications at intermediary brokers is performed.

More formally, the broker topology is modeled as a connected undirected acyclic graph $G = (V, E)$ with a set of nodes $V = (B_1, \ldots, B_n)$ corresponding to the brokers and a set of edges $E \subseteq \{(B_i, B_j) \mid 1 \leq i < j \leq n\}$ representing bidirectional connections among them. For notational convenience, a function $e(B_i, B_j)$ is defined that returns $(B_i, B_j)$ if $i < j$ and $(B_j, B_i)$ otherwise.

## 3.3 Routing Configurations

In this section a formal model for routing configurations is presented. This model and the derived results offer new insights into how routing configurations can be managed. First of all, it is described how brokers forward notifications by using filter-based routing tables (see Sect. 3.3.1). After that, valid routing configurations are introduced in Section 3.3.2 which ensure that in a static publish/subscribe system all matching notifications are delivered to a consumer. In dynamic publish/subscribe systems it is impossible to ensure that the routing configuration is always valid. Therefore, weakly valid routing configurations are

subsequently introduced (see Sect. 3.3.3). A weakly valid routing configuration demands only the delivery of those notifications which are matched by a subscription that has already been incorporated into the routing configuration.

### 3.3.1 Notification Forwarding based on Routing Tables

Each broker $B$ manages a private *routing table* $T_B$ that comprises a set of routing entries. Each *routing entry* is a pair $(F, U)$ consisting of a filter $F$ and a *destination* $U \in N_B \cup L_B$. The graph $G$ together with all routing tables is called the current *routing configuration* of the publish/subscribe system. The routing configuration of a single broker $B$ consists of two disjoint parts: the *remote routing configuration* that comprises all routing entries whose destination is a neighbor of $B$ and the *local routing configuration* consisting of all routing entries whose destination is a local client of $B$.

The current routing configuration induces the set of notifications that a broker forwards to a destination, i.e., the neighbors and local clients that are connected to it. Formally, for a subset $\mathcal{W}$ of $N_B \cup L_B$,

$$\nu_B(\mathcal{W}) = \{n \mid \exists (F,U) \in T_B.\, n \in N(F) \wedge U \in \mathcal{W}\} \tag{3.1}$$

is the set of notifications that a broker $B$ forwards to any member of $\mathcal{W}$. From this definition follows for two subsets $\mathcal{A}, \mathcal{B}$ of $N_B \cup L_B$ that $\mathcal{A} \subseteq \mathcal{B} \Rightarrow \nu_B(\mathcal{A}) \subseteq \nu_B(\mathcal{B})$ and that $\nu_B(\mathcal{A} \cup \mathcal{B}) = \nu_B(\mathcal{A}) \cup \nu_B(\mathcal{B})$. The destinations, local clients, and neighbors to which a broker $B$ forwards a given notification $n$ are given by $F_B(n)$, $F_B^L(n)$, and $F_B^N(n)$, respectively:

$$F_B(n) = \big\{D \mid D \in N_B \cup L_B \ \wedge\ n \in \nu_B(\{D\})\big\} \tag{3.2}$$

$$F_B^L(n) = F_B(n) \cap L_B \tag{3.3}$$

$$F_B^N(n) = F_B(n) \cap N_B \tag{3.4}$$

Equipped with these definitions it is easy to define an algorithm that forwards notifications based on a routing table (see Fig. 3.2). The algorithm processes incoming messages serially and fairly, i.e., in FIFO order regardless of the sender. The brokers propagate notifications to each other by sending and receiving *forward*$(n)$ messages. A broker receives *pub*$(n)$ messages from and sends *notify*$(n)$ messages to its local clients (cf. Sect. 2.2). The algorithm works in the following way:

- If a broker receives a *pub*$(n)$ message from one of its local clients, it sends a *notify*$(n)$ message to all of its local clients which are in $F_B^L(n)$ and a *forward*$(n)$ message to all of its neighbors in $F_B^N(n)$.

- If a broker receives a *forward*$(n)$ message from one of its neighbors $U$, it sends a *notify*$(n)$ message to all of its local clients which are in $F_B^L(n)$ and a *forward*$(n)$ message to all of its neighbors in $F_B^N(n) \setminus \{U\}$.

```
program StaticNotificationForwarding()
begin
  initialize T_B
  loop
    m ← wait for and return next message
    forwardNotification(m)
  end
end

procedure forwardNotification(Message m)
begin
  if m is "forward(n)" message from neighbor U then
    send "notify(n)" to all local clients in F_B^L(n)
    send "forward(n)" to all neighbors in F_B^N(n) \ {U}
  fi
  if m is "pub(n)" from client X then
    send "notify(n)" to all local clients in F_B^L(n)
    send "forward(n)" to all neighbors in F_B^N(n)
  fi
end
```

Figure 3.2: Static notification forwarding

For example, consider the situation depicted in Fig. 3.3. Here, $B_1$ forwards a notification received from $X_1$ to its local client $X_2$ and its neighbor $B_2$.

Now, that routing configurations and notification forwarding based on routing tables have been introduced, the question arises which conditions a routing configuration has to fulfill in order to ensure that a publish/subscribe system behaves correctly. This question is investigated and answered in the next two subsections.

### 3.3.2 Static Publish/Subscribe Systems: Valid Routing Configurations

In a static publish/subscribe system the set of participating brokers, the connections between them, and the subscriptions of their local consumers are constant over time. In such a system the routing configuration should ensure that a notification that has been published by an arbitrary client is delivered to any client which has a matching subscription. *Valid* routing configurations which are introduced below satisfy this requirement.

Informally, a broker has to distinguish between two sets of notifications:

1. the set of notifications $\nu_{B_Y}(\{Y\})$ it forwards to a *local client* and
2. the set of notifications $\nu_{B_i}(\{B_j\})$ it forwards to a *neighbor broker*.

First, consider $\nu_{B_Y}(\{Y\})$. For the system to be correct with respect to Def. 2.1, a broker must deliver exactly those notifications to one of its local clients which it is interested in. This means that $\nu_{B_Y}(\{Y\})$ always has to be

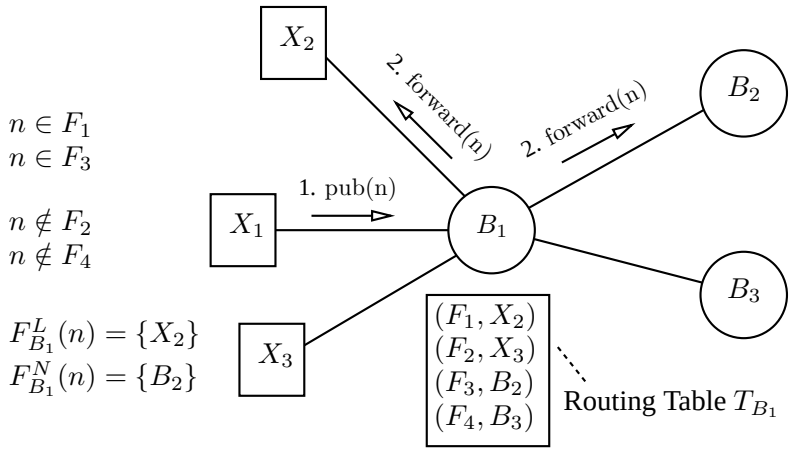



Figure 3.3: Diagram explaining notification forwarding.

equal to $\cup_{F \in S_Y} N(F)$. On the one hand, if $\nu_{B_Y}(\{Y\})$ contains more notifications, the safety requirement would be violated. On the other hand, if $\nu_{B_Y}(\{Y\})$ contains less notifications, these notifications would never be delivered, violating the liveness requirement.

Now, consider $\nu_{B_i}(\{B_j\})$. Intuitively, $\nu_{B_i}(\{B_j\})$ should contain at least all notifications which the neighbor's clients are interested in. But as the topology is acyclic, $\nu_{B_i}(\{B_j\})$ must additionally contain those notifications in which the clients that "lie behind" this neighbor are interested in. If this is not the case, some of these clients would not receive all interesting notifications. In contrast to $\nu_{B_Y}(\{Y\})$, $\nu_{B_i}(\{B_j\})$ is allowed to contain more notifications than necessary. This is because final filtering is achieved through $\nu_{B_Y}(\{Y\})$. Of course, the surplus should be as small as possible in order to limit network bandwidth waste. Later, it is formally proved that these requirements are necessary and sufficient in order to ensure that static notification forwarding leads to a system that is correct with respect to Def. 2.1.

Formally, let $I_B$ be the set of all notifications that are of interest to any of the local clients of $B$, i.e,

$$I_B = \cup_{X \in L_B} \cup_{F \in S_X} N(F). \tag{3.5}$$

Note that $I_B$ changes instantly if a client subscribes or unsubscribes. Now, consider a broker $B_i$ and one of its neighbors $B_j$. If the edge between $B_i$ and $B_j$ is removed, $G$ is partitioned into two not connected subgraphs: one that contains $B_i$ and the other that contains $B_j$. Let $G_{B_i,B_j} = (V_{B_i,B_j}, E_{B_i,B_j})$ be the subgraph whose node set contains $B_i$. In the following, the set of all notifications that are of interest to at least one local consumer of any broker in $V_{B_i,B_j}$ is denoted by

$$\eta_{B_i,B_j} = \cup_{B \in V_{B_i,B_j}} I_B. \tag{3.6}$$

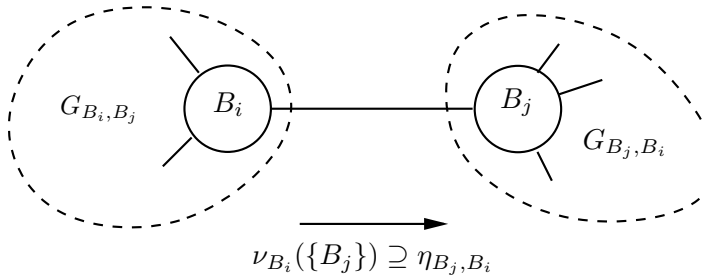



Figure 3.4: Diagram explaining remote validity of valid routing configurations.

Now, valid routing configurations can be defined:

**Definition 3.1 (Valid Routing Configuration)** *A routing configuration is* valid *iff*

1. $e(B_i, B_j) \in E \;\Rightarrow\; \nu_{B_i}(\{B_j\}) \supseteq \eta_{B_j,B_i}$ **(remote validity)**
2. $Y \in L_{B_Y} \;\Rightarrow\; \nu_{B_Y}(\{Y\}) = \cup_{F \in S_Y} N(F)$ **(local validity)**

Remote validity formalizes the requirement placed on $\nu_{B_i}(\{B_j\})$ above. It requires that the remote routing configuration of a broker $B_i$ ensures that *at least* those notifications must be sent over the link from $B_i$ to $B_j$ which are of interest to at least one local client of any broker in $V_{B_j,B_i}$ (see Figure 3.4). Local validity formalizes the requirement placed on $\nu_{B_Y}(\{Y\})$ above. It demands that the local routing configuration ensures that *exactly* those notifications are delivered to a local client $Y$ that are matched by any subscription in $S_Y$ (see Figure 3.5).

A valid routing configuration is called *perfect* if remote validity is also satisfied when replacing the superset relation by an equality. A perfect routing configuration ensures that no notifications are forwarded unnecessarily. Hence, they minimize network bandwidth consumption. Imperfect valid routing configurations, on the other hand, may lead to smaller routing tables.

It can now be proved that if a valid routing configuration is used in conjunction with the algorithm in Fig. 3.2, then the resulting publish/subscribe system satisfies Def. 2.1. Note that the more intricate proofs given in this chapter are

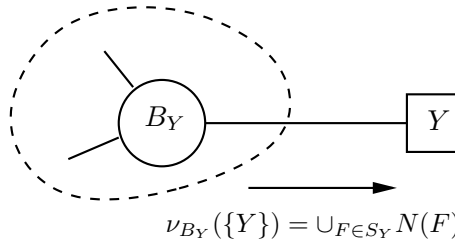



Figure 3.5: Diagram explaining local validity of valid routing configurations.

written in a structured style similar to proof trees of interactive theorem proving environments. This approach is advocated by Lamport who promises that this style "makes it much harder to prove things that are not true" [63]. The proof is a sequence of numbered proof steps at different levels. Every proof step has a proof which may be refined at lower levels by additional proof steps. For example, proof step $\langle 1 \rangle 2$ is the second proof step on level 1. Proofs may also be read in a structured way, for example, by reading only the top level proof steps and going into sublevels only when necessary. In general, it should be sufficient to read solely the proof sketches.

In the following, it is first proved that the safety requirement of Def. 2.1 is satisfied. This is done by proving that the three conjuncts of the safety condition in Def. 2.1 hold (Lemmas 3.1 to 3.3) resulting in Lemma 3.4. After that, the liveness requirement of Def. 2.1 is proved by Lemma 3.5. Together this results in Theorem 3.1 stating the correctness.

**Lemma 3.1** $\Box\big[\mathit{Notify}(Y,n) \Rightarrow \bigcirc\Box\neg\mathit{Notify}(Y,n)\big]$.

PROOF: It has to be proved that if a client is notified about $n$, it is never notified about $n$ again. Each notification $n$ can be published only once because notifications are unique. From the algorithm in Fig. 3.2 (notification forwarding), the fact that $G$ is acyclic, and because of the reliable channel assumption, we know that $n$ can be delivered at most once to a client. □

**Lemma 3.2** *If the routing configuration is valid, then* $\Box\big[\mathit{Notify}(Y,n) \Rightarrow \exists F \in S_Y \,.\, n \in N(F)\big]$ *holds.*

PROOF SKETCH: Here, it is proved that a valid routing configuration implies that only matching notifications are delivered to a client. For a valid routing configuration $Y \in L_{B_Y}$ implies $\nu_{B_Y}(\{Y\}) = \cup_{F \in S_Y} N(F)$ because of the local validity. This together with the reliable channel assumption and the algorithm in Fig. 3.2 ensures that only matching notification are delivered.

ASSUME: 1. The routing configuration is valid.
2. $\mathit{Notify}(Y,n)$

PROVE: $\exists F \in S_Y \,.\, n \in N(F)$

LET: $B_Y$ be the broker that manages $Y$.

PROOF:

$\langle 1 \rangle 1$. $B_Y$ has sent a *notify*$(n)$ message to $Y$.
PROOF: by assumption 2 and algorithm in Fig. 3.2 (lines 13/17). □

$\langle 1 \rangle 2$. $Y \in F^L_{B_Y}(n)$
PROOF: by step $\langle 1 \rangle 1$ and algorithm in Fig. 3.2 (lines 13/17). □

$\langle 1 \rangle 3$. $n \in \nu_{B_Y}(\{Y\})$.
PROOF: by step $\langle 1 \rangle 2$ and definition of $F^L_{B_Y}$. □

$\langle 1 \rangle 4$. $\nu_{B_Y}(\{Y\}) = \cup_{F \in S_Y} N(F)$
PROOF: by definition of valid routing configuration (local validity) (see Def. 3.1) and assumption 1. □

$\langle 1 \rangle 5$. Q.E.D.
PROOF: Step $\langle 1 \rangle 3$ and $\langle 1 \rangle 4$ imply the existence of a filter $F \in S_Y$ that matches $n$. □

**Lemma 3.3** $\Box\big[\mathit{Notify}(Y,n) \Rightarrow \exists X \,.\, n \in P_X\big]$.

PROOF SKETCH: Here, it is proved that if a client is notified about $n$, $n$ has been published by some client before. This is implied by the algorithms and the reliable channel assumption. The detailed proof argues backwards from the delivery of $n$ to the publication of $n$ using an induction on the topology.
ASSUME: $\mathit{Notify}(Y, n)$
PROVE: $\exists X.\, n \in P_X$
PROOF:
$\langle 1 \rangle 1$. $B_Y$ sent *notify(n)* message to $Y$.
PROOF: by assumption and system model. □
$\langle 1 \rangle 2$. $B_Y$ received either $\mathit{pub}(n)$ or a $\mathit{forward}(n)$ message.
PROOF: by step $\langle 1 \rangle 1$ and algorithm in Fig. 3.2 (lines 12/16). □
$\langle 1 \rangle 3$. CASE: $B_Y$ received $\mathit{pub}(n)$ message from client $X \in L_{B_Y}$.
$\langle 2 \rangle 1$. $n \in P_X$.
PROOF: by case assumption and system model. □
$\langle 2 \rangle 2$. Q.E.D.
PROOF: by step $\langle 2 \rangle 1$. □
$\langle 1 \rangle 4$. CASE: $B_Y$ received $\mathit{forward}(n)$ message $m$ from a neighbor.
$\langle 2 \rangle 1$. $m$ must originate from some broker $B$.
PROOF: by reliable channels, algorithm in Fig. 3.2 (lines 14/18), and an induction. □
$\langle 2 \rangle 2$. $B$ received $\mathit{pub}(n)$ message from client $X \in L_B$.
PROOF: by step $\langle 2 \rangle 1$, reliable channels, and algorithm in Fig. 3.2 (line 16). □
$\langle 2 \rangle 3$. Q.E.D.
PROOF: Step $\langle 2 \rangle 2$ with the system model implies that $n \in P_X$. □
$\langle 1 \rangle 5$. Q.E.D.
PROOF: by step $\langle 1 \rangle 2$ in conjunction with $\langle 1 \rangle 3$, and $\langle 1 \rangle 4$ that cover all cases. □

**Lemma 3.4** *If the routing configuration is valid, static notification forwarding satisfies the safety requirement of Def. 2.1.*

PROOF: The Lemmas 3.1, 3.2, and 3.3 proved the three individual conjuncts of the safety requirement of Def. 2.1. Hence the safety requirement holds in its entirety. □

**Lemma 3.5** *If the routing configuration is valid, static notification forwarding satisfies the liveness requirement of Def. 2.1.*

PROOF SKETCH: As the subscriptions are static, it suffices to show that $\big[\mathit{Pub}(X, n) \wedge \exists F \in S_Y.\, n \in N(F)\big]$ implies $\Diamond \mathit{Notify}(Y, n)$. This is proved by induction over the topology. First, the local delivery is proved, i.e., that $B_Y$ sends a $\mathit{notify}(n)$ message to $Y$ if it receives a $\mathit{forward}(n)$ or $\mathit{pub}(n)$ message, and $Y$ has a subscription that matches $n$. After that, the induction step is proved. Informally, the local delivery directly follows from the algorithm and the local validity of the routing configuration. The remote validity, on the other hand, implies that $n$ is forwarded to every broker $B$ managing a local client with a matching subscription. This is simply because remote validity implies that a notification is forwarded over all links representing subnets having a broker with a client with a matching subscription.
ASSUME: 1. $\mathit{Pub}(X, n)$
2. $\exists F \in S_Y.\, n \in N(F)$
3. The routing configuration is valid.

PROVE: $\Diamond Notify(Y, n)$
LET: $B_X$ and $B_Y$ be the broker that manages $X$ and $Y$ respectively.
PROOF:
$\langle 1 \rangle 1$. If $B_Y$ receives a $forward(n)$ or $pub(n)$ message, it sends $notify(n)$ to $Y$.
ASSUME: $B_Y$ receives a $forward(n)$ or $pub(n)$ message.
PROVE: $B_Y$ sends a $notify(n)$ message to $Y$.
$\langle 2 \rangle 1$. $B_Y$ sends a $notify(n)$ message to $Y$ if $Y \in F_B^L(n)$.
PROOF: by assumption and algorithm (lines 13/17). □
$\langle 2 \rangle 2$. $\exists F \in S_Y . n \in N(F)$.
PROOF: by assumption 2. □
$\langle 2 \rangle 3$. $\nu_{B_Y}(\{Y\}) = \cup_{G \in S_Y} N(G)$.
PROOF: by assumption 3. □
$\langle 2 \rangle 4$. $n \in \nu_{B_Y}(\{Y\})$.
PROOF: by step $\langle 2 \rangle 2$ and $\langle 2 \rangle 3$. □
$\langle 2 \rangle 5$. $Y \in F_B^L(n)$.
PROOF: by step $\langle 2 \rangle 4$ and definition of $F_B^L(n)$. □
$\langle 2 \rangle 6$. Q.E.D.
PROOF: by step $\langle 2 \rangle 1$ and $\langle 2 \rangle 5$. □
$\langle 1 \rangle 2$. $n$ is delivered to $Y$ if $B_X = B_Y$.
$\langle 2 \rangle 1$. $B_Y$ receives $pub(n)$ message from $X$
PROOF: by assumption 1, the fact that $B_X = B_Y$, and algorithm. □
$\langle 2 \rangle 2$. Q.E.D.
PROOF: by step $\langle 1 \rangle 1$ and $\langle 2 \rangle 1$. □
$\langle 1 \rangle 3$. LET: $B_k \neq B_Y$ be an arbitrary broker on the path from $B_X$ to $B_Y$ and $B_l$ be next broker on the path from $B_k$ to $B_Y$.
PROOF: The paths are well defined and unique because of the acyclic and connected topology induced by the graph $G$. □
$\langle 1 \rangle 4$. $B_k$ forwards $n$ to $B_l$.
$\langle 2 \rangle 1$. $B_Y \in V_{B_l, B_k}$.
PROOF: by the fact that $B_l$ is nearer to $B_Y$ than $B_k$ and the definition of $V_{B_l, B_k}$. □
$\langle 2 \rangle 2$. $\eta_{B_l, B_k} \supseteq I_{B_Y}$.
PROOF: by definition of $\eta_{B_l, B_k}$ (Eq. 3.6) and step $\langle 2 \rangle 1$. □
$\langle 2 \rangle 3$. $n \in I_{B_Y}$
PROOF: by assumption 2 and definition of $I_{B_Y}$. □
$\langle 2 \rangle 4$. $\nu_{B_k}(\{B_l\}) \supseteq \eta_{B_l, B_k}$.
PROOF: by assumption 3. □
$\langle 2 \rangle 5$. Q.E.D.
PROOF: by step $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, and $\langle 2 \rangle 4$. □
$\langle 1 \rangle 5$. Q.E.D.
PROOF: by step $\langle 1 \rangle 2$ (base case) and $\langle 1 \rangle 4$ (induction step). □

**Theorem 3.1** *If a valid routing configuration is used in conjunction with the algorithm in Fig. 3.2 then the resulting static publish/subscribe system satisfies Def. 2.1.*

PROOF: Follows from Lemmas 3.4 and 3.5. □

Interestingly, it can also be proved that *both* properties which a valid routing configuration must satisfy are also necessary for a static publish/subscribe system satisfying Def. 2.1.

**Lemma 3.6** *Remote validity of a routing configuration is necessary for a static publish/subscribe system using the algorithm in Fig. 3.2 to satisfy Def. 2.1.*

PROOF SKETCH: We assume that property 1 (remote validity) of a valid routing configuration is violated and show that this implies that the liveness condition of Def. 2.1 does not hold. Informally, the violation of the remote validity implies that there are notifications that are not sent over some link although there are some clients in the receiving subset which have a subscription that matches these notifications. Hence, if such a notification is published, it will not be delivered to the interested clients. This implies that the liveness condition is violated.
ASSUME: $\exists B_i, B_j$ with $e(B_i, B_j) \in E$ for which $\nu_{B_i}(\{B_j\}) \supseteq \eta_{B_j,B_i}$ does not hold.
PROVE: Def. 2.1 violated.
PROOF:
⟨1⟩1. $\exists n$ where $n \in \eta_{B_j,B_i} \setminus \nu_{B_i}(\{B_j\})$.
PROOF: by assumption. □
⟨1⟩2. $\exists B \in V_{B_j,B_i}.\, n \in I_B$.
PROOF: by definition of $\eta_{B_j,B_i}$ (Eq. 3.6). □
⟨1⟩3. $\exists Y \in L_B.\, \exists F \in S_Y.\, n \in N(F)$.
PROOF: by definition of $I_B$ (Eq. 3.5). □
⟨1⟩4. ASSUME: $n$ is published by a local client of $B_i$.
PROVE: $n$ is not delivered to $Y$.
⟨2⟩1. $B_i$ does not forward $n$ to $B_j$
PROOF: by step ⟨1⟩1. □
⟨2⟩2. $n$ is not forwarded to $B$.
PROOF: by step ⟨1⟩2, ⟨2⟩1, and the fact that the topology is acyclic, and an induction over the topology. □
⟨2⟩3. Q.E.D.
PROOF: Step ⟨2⟩2 and ⟨1⟩3 imply that $n$ is not delivered to $Y$. □
⟨1⟩5. Q.E.D.
PROOF: step ⟨1⟩3 and ⟨1⟩4 imply that the liveness property of Def. 2.1 is violated. □

**Lemma 3.7** *Local validity of a routing configuration is necessary for a static publish/subscribe system using the algorithm in Fig. 3.2 to satisfy Def. 2.1.*

PROOF SKETCH: We assume that property 2 (local validity) of a valid routing configuration is violated and show that this implies that the safety or the liveness condition of Def. 2.1 does not hold. The violation of the local validity implies that to some clients either not all interesting notification are delivered (violating liveness), or that to some clients uninteresting notifications are delivered (violating safety). This is proved by a case differentiation.
ASSUME: $\exists Y.\, \nu_{B_Y}(\{Y\}) \neq \cup_{F \in S_Y} N(F)$ where $B_Y$ is the broker that manages $Y$.
PROVE: Def. 2.1 violated.
PROOF:
⟨1⟩1. CASE: $\exists n \in \cup_{F \in S_Y} N(F)) \setminus \nu_{B_Y}(\{Y\})$.
ASSUME: A local client of $B_Y$ publishes $n$.
⟨2⟩1. $n$ is not delivered to $Y$.
PROOF: by reliable channel assumption and algorithm in Fig. 3.2. □.
⟨2⟩2. Q.E.D.
PROOF: step ⟨2⟩1 implies that the liveness requirement of Def. 2.1 is violated. □

⟨1⟩2. CASE: $\exists n \in \nu_{B_Y}(\{Y\}) \setminus \cup_{F \in S_Y} N(F)$.
ASSUME: A local client of $B_Y$ publishes $n$.
⟨2⟩1. $n$ is delivered to $Y$.
PROOF: by reliable channel assumption and algorithm in Fig. 3.2. □.
⟨2⟩2. Q.E.D.
PROOF: step ⟨2⟩1 implies that the safety requirement of Def. 2.1 is violated. □
⟨1⟩3. Q.E.D.
PROOF: step ⟨1⟩1 and ⟨1⟩2 cover all cases and therefore, they imply that Def. 2.1 is violated in any case. □

**Theorem 3.2** *A valid routing configuration is necessary for a static publish/subscribe system using the algorithm in Fig. 3.2 to satisfy Def. 2.1.*

PROOF: by Lemmas 3.6 and 3.7. □

**Corollary 3.1** *A valid routing configuration is necessary and sufficient for a static publish/subscribe system using the algorithm in Fig. 3.2 to satisfy Def. 2.1.*

PROOF: Proved by Theorems 3.1 and 3.2. □

Corollary 3.1 shows that valid routing configurations play an important role in static publish/subscribe systems. They are necessary and sufficient to realize a static publish/subscribe system that is correct with respect to Def. 2.1. In the next section, it is shown that validity is too strong a requirement in dynamic publish/subscribe systems. Hence, weakly valid routing configurations are introduced.

### 3.3.3 Dynamic Publish/Subscribe Systems: Weakly Valid Routing Configurations

A dynamic publish/subscribe system must deal with new subscriptions and cancellation of existing subscriptions; for brevity it is assumed that the broker topology is static. It is easy to see that a routing configuration might not ensure the delivery of all notifications that are matched by a new subscription. The formal definition of the correctness of a publish/subscribe system (Def. 2.1) tolerates this as long as eventually all matching notifications are delivered to the subscribing client. To reensure the delivery of all interesting notifications, it might be necessary to update the local routing configuration of the broker of the subscribing client as well as the remote part of other brokers.

In contrast to a new subscription, an unsubscription does not require the remote part of the routing configuration to be updated. Here, it is sufficient to instantly change the local part in order to prevent the delivery of notifications that are no longer of interest to the unsubscribing client. Nevertheless, further handling of unsubscriptions is desirable for efficiency reasons. Without appropriate processing of unsubscriptions, the set of notifications that a broker forwards to a neighbor is monotonically increasing. In general, this results in network resource waste. Hence, a sensible content-based routing algorithm will continually

update the routing configuration in reaction to subscriptions and unsubscriptions of clients either to ensure the correct semantics of the publish/subscribe system or to optimize.

Updates to the local configuration of a broker can be carried out without any negotiation with other brokers. In contrast to that, updates to the remote configurations of brokers require complex and distributed *update processes*. These may be carried out *periodically* at certain points in time or may be triggered *aperiodically* by messages received from local clients. Also a *hybrid* approach can be used in which, for example, updates caused by subscriptions are propagated instantly and updates caused by unsubscriptions are handled periodically. This thesis concentrates on the aperiodic case. More precisely, in the scenario that is investigated all changes to the routing configuration are triggered by the act of subscribing or unsubscribing. Changes to the local configuration are carried out instantly and updates to the remote part originate from the broker that manages the corresponding client and extend through the broker topology until all necessary changes have been carried out.

In general, many update processes triggered by subscribing or unsubscribing clients may be carried out concurrently. Hence, the routing configuration may never become valid. This is the motivation to introduce the notion of weakly valid routing configurations that impose less restrictive requirements than validity but still ensure that the publish/subscribe system is correct. Informally, a weakly valid routing configuration only guarantees the delivery of notifications matching a subscription whose corresponding update process has terminated.

More formally, let $\bar{S}_X$ be the subset of all active subscriptions $S_X$ of a client $X$ whose update process has terminated. This means that a new subscription is added to $\bar{S}_X$ at the time its update process finishes, while if the subscription is canceled, it is removed immediately. The definition of $\bar{S}_X$ allows "weakened" versions of $I_B$ and $\eta_{B_j,B_i}$ to be defined, respectively:

$$\bar{I}_B = \cup_{X \in L_B} \cup_{F \in \bar{S}_X} N(F), \tag{3.7}$$

$$\bar{\eta}_{B_j,B_i} = \cup_{B \in V_{B_j,B_i}} \bar{I}_B. \tag{3.8}$$

Now, weakly valid routing configurations can be introduced:

**Definition 3.2 (Weakly Valid Routing Configuration)** *A routing configuration is* weakly valid *iff*

1. $e(B_i, B_j) \in E \Rightarrow \nu_{B_i}(\{B_j\}) \supseteq \bar{\eta}_{B_j,B_i}$ **(weak remote validity)**
2. $Y \in L_{B_Y} \Rightarrow \nu_{B_Y}(\{Y\}) = \cup_{F \in S_Y} N(F)$ **(local validity)**

In the definition above, remote validity has been weakened to weak remote validity such that a broker is only required to forward those notifications to a neighbor which are matched by a subscription whose update process has terminated. Local validity remains unchanged because necessary changes to the local part of a routing table can be carried out instantly. From the definition it is trivial to see that every valid routing configuration is also weakly valid because

of $\bar{\eta}_{B_j,B_i} \subseteq \eta_{B_j,B_i}$ and $\bar{S}_Y \subseteq S_Y$. Moreover, a routing configuration is *weakly perfect* if it also satisfies the first property if the superset relationship is replaced by an equality. Now, it can be proved that weakly valid routing configurations are important to ensure a correct publish/subscribe system:

**Theorem 3.3** *If an algorithm ensures that the routing configuration is always weakly valid and that every update process terminates, then the resulting publish/subscribe system satisfies Def. 2.1.*

PROOF: We have to prove the safety and liveness condition of Def. 2.1. These follow from Lemmas 3.4 and 3.5: As the local validity has not been changed, and changes to the local configuration are instantly, Lemma 3.4 can be applied directly giving the safety requirement. The remote validity, on the other hand, has been changed to include only those subscriptions whose update process has terminated. According to the assumption, every update process terminates. Hence, Lemma 3.5 can be applied if $S$ is substituted with $\bar{S}$ and "valid" with "weakly valid", giving the liveness requirement. □

The theorem above shows that weakly valid routing configurations are a useful concept in dynamic publish/subscribe systems: If an algorithm ensures that the routing configuration is always weakly valid, only interesting notifications are delivered to clients and the delivery of notifications is guaranteed after the update process of the respective subscription has terminated. In the next section, a framework for routing algorithms is introduced and it is shown which requirements an instance of the framework must satisfy in order to guarantee that every update process terminates and that the routing configuration is always weakly valid.

## 3.4 Routing Algorithm Framework

In the following, a framework for content-based routing algorithms is presented. It predefines notification forwarding and allows instances of the framework to customize the handling of control messages. After the framework has been introduced, a universal correctness criterion is elaborated in Sect. 3.4.1 that allows to determine legal instantiations of the framework leading to a correct publish/subscribe system.

The framework uses two types of messages for the communication between brokers: $forward(n)$ and $admin(\mathcal{S}, \mathcal{U})$ where $\mathcal{S}$ and $\mathcal{U}$ are both sets of filters whose elements are interpreted as subscriptions and unsubscriptions, respectively. The clients communicate with their respective broker by $sub(F)$, $unsub(F)$, and $pub(n)$ messages while a broker sends its local clients $notify(n)$ messages (cf. Sect. 2.2). Although communication between a client and its broker is conceptually treated as message passing communication, it is assumed that this communication is local and therefore, instantaneous [42]. Moreover, it is assumed that messages can only be sent by the framework algorithm. The framework hardwires the processing of *sub*, *unsub*, *pub*, and *forward* messages and the generation of *notify* messages. The concrete routing algorithm customizes

the handling of *admin* messages by implementing an instance of an `administer` procedure. This allows a wide variety of routing algorithms (see Section 3.5) as instances of the framework to be implemented.

The complete framework algorithm is depicted in Fig. 3.6. The `administer` procedure is called at a broker $B$ if an *admin* message from a neighbor or, if a *sub* or *unsub* message from a local client is received. If its execution was triggered by an *admin* message, it is called with the broker $S$ from which the *admin* message was received and the two filter sets $\mathcal{S}$ and $\mathcal{U}$ that were embedded in the message as parameters (line 19). On the other hand, $sub(F)$ and $unsub(F)$ messages received from a local client $X$ trigger a procedure call `administer`$(X, \{F\}, \emptyset)$ (line 22) and `administer`$(X, \emptyset, \{F\})$ (line 25), respectively. The implementation of the `administer` procedure can identify whether the call was triggered by receiving a message from a local client or from a neighbor by checking if $S$ is in $N_B$ or in $L_B$. As result `administer` returns a set of triples $\mathcal{M}$. To each neighbor except $S$ that is represented in the set of triples, the routing framework sends exactly one message $admin(\mathcal{S}_H, \mathcal{U}_H)$ where $\mathcal{S}_H$ and $\mathcal{U}_H$ are derived from all tuples regarding the respective neighbors (lines 28-32).

### 3.4.1 Legal Instances of the Framework

In principle, the framework algorithm allows an `administer` procedure which exhibits an arbitrary behavior to be defined. In this section a precise specification of legal implementations of an `administer` procedure is given. Subsequently, legal implementations of `administer` together with legal initial routing configurations are shown to be sufficient for an instance of the framework to satisfy Def. 2.1. Hence, the specification can be used as a universally sufficiency criterion that allows legal implementations of the `administer` procedure to be determined by verifying the properties of the specification. This way the criterion is applied to prove the correctness of some routing algorithms in Section 3.5.

Before a definition of legal instances of the `administer` procedure is given, we need some preliminary definitions. This is necessary due to the peculiarities of the dynamic case; we need to refer to the progress of update processes to make correctness statements. It is assumed that each update process has a unique identifier and that the *sub* or *unsub* message that triggered an update as well as all constituting *admin* messages are marked with this identifier. The identifier of the update process of a message $m$ is given by $id(m)$. We use $id(m)$ to refer to the state of variables at the time when a message with this identifier has been processed by the corresponding broker or, sent or received by a specific client. For example, we denote with $\nu_B^{id}$ the state of $\nu_B$ directly after the processing of a message with the identifier $id$. In addition to this, we use a superscript 0 to indicate the initial state of variables. For example, we denote with $\nu_B^0$ the initial state of $\nu_B$. We also need a way to refer to states of variables before the processing of a specific message. Let $lid_{S,B}(m)$ be the identifier of the last *admin*, *sub*, or *unsub* message that $S$ has sent to $B$ before $S$ sent $m$ to $B$ or 0 if $m$ was the first such message that $S$ sent to $B$. Furthermore, let $lid_B(m)$ be

```
program ContentBasedRoutingFramework()
begin
  initialize T_B
  loop
    m ← wait for and return next message
    if m is "forward(n)" or "pub(n)" message then
      forwardNotification(m)
    else
      adminMessages(m)
    fi
  end
end

procedure adminMessages(Message m)
begin
  M ← ∅

  if m is "admin(S, U)" message from broker U then
    M ← administer(U, S, U)
  fi
  if m is "sub (F)" message from client X then
    M ← administer(X, {F}, ∅)
  fi
  if m is "unsub(F)" message from client X then
    M ← administer(X, ∅, {F})
  fi

  forall H ∈ N_B \ {S}
    S_A ← {S | (H, S, U) ∈ M}
    U_A ← {U | (H, S, U) ∈ M}
    send message "admin(S_A, U_A)" to H
  end
end
```

Figure 3.6: Content-Based Routing Framework

the identifier of the *admin* message that $B$ processes before it processes $m$ and 0 if $m$ is the first *admin* message that $B$ processes.

Now, we are ready to define legal initial routing configurations and legal instantiations of `administer`. First, legal initial routing configurations are defined. They are necessary to define a meaningful initial state for a routing algorithm. For brevity, it is assumed that the initial routing configuration does not contain any routing entries for local clients and that $S_Y^0$ is empty for all clients $Y$.

**Definition 3.3 (Legal Initial Routing Configurations)** *An initial routing configuration is* legal *iff the following properties are satisfied:*

1. $e(B_i, B_j) \in E \Rightarrow \nu_{B_i}^0(\{B_j\}) \supseteq \nu_{B_j}^0(N_{B_j} \setminus \{B_i\})$
   **(Legal initial remote routing configuration)**
2. $\nu_{B_i}^0(L_{B_i}) = \emptyset$
   **(Legal initial local routing configuration)**

The first property states that a broker $B_i$ should send at least those notifications over the link to $B_j$ which $B_j$ in turn sends to its other neighbors. This superset relationship among paths in the broker topology ensures that every broker can make routing decisions locally. The second property simply demands that no local routing entries are present in the initial routing configuration.

Now, it is defined what it means for an `administer` procedure to be legal:

**Definition 3.4 (Legal Instantiation of `administer`)** *An instantiation of* `administer` *is* legal *iff there exists at least one initial legal routing configuration such that always if* `administer` *is called at a broker $B$ with a message $m$ received from a local client or neighbor $S$ the following properties are satisfied:*

1. *It returns after a finite time.*
2. *When it returns, the routing configuration satisfies the following properties:*
   (a) *If $S \in L_B$, then $\nu_B^{id(m)}(\{S\}) = \cup_{F \in S_S^{id(m)}} N(F)$.*
   (b) *If $S \in N_B$, then $\nu_B^{id(m)}(\{S\}) \supseteq \nu_S^{id(m)}(L_S \cup N_S \setminus \{B\})$.*
   (c) *$\nu_B^{id(m)}(L_B \cup N_B \setminus \{S\}) = \nu_B^{lid_B(m)}(L_B \cup N_B \setminus \{S\})$.*
3. *The set of returned triples $\mathcal{M}$ contains at least triples $(H, \mathcal{S}_H, \mathcal{U}_H)$ for all brokers $H \in N_B \setminus \{S\}$ for that $\nu_H^{lid_{B,H}(m)}(\{B\}) \not\supseteq \nu_B^{id(m)}(\{S\})$.*

The above definition is a complex construct that is the basis for one of the main theorems (see Theorem 3.4). Because of this, the meaning of the definition is described in detail. It consists of three properties. Property 1 demands `administer` to terminate, property 2 imposes conditions on the routing configuration, and property 3 states requirements that must be met by the set of returned triples.

But what do these properties mean informally?

- Property 1 ensures that a legal `administer` returns after a finite time.
- Property 2 imposes conditions on the routing configuration. The subproperties have the following meaning:
  - Property 2a states that exactly those notifications should be delivered to a client which it has interest in.
  - Property 2b assures that the inclusions established by a legal initial routing configuration are maintained.
  - Property 2c states that a message received from a local client or neighbor can only affect the part of the routing configuration dealing with this destination.
- Property 3 ensures that a broker sends an *admin* message to all neighbors whose remote routing configuration does not ensure that all needed notifications are forwarded to the respective broker.

But what are these properties good for and why are they needed?

- Property 1 together with the framework algorithm, and fact that the topology is acyclic ensures that each update process terminates after a finite time. Without this property the liveness condition of Def. 2.1 could be violated.
- Without property 2a either safety (uninteresting notification may be delivered to a local client) or liveness may be violated (not all interesting notifications may be delivered to a local client).
- The inclusion assured by property 2b guarantees that $B$ forwards at least those notifications to $S$ which $S$ forwards in turn to its other neighbors or delivers to its local clients. This enables brokers to make routing decisions locally.
- Property 2c limits the changes that can be caused by an *admin* message to the part of the routing configuration regarding $S$. This simplifies reasoning.
- Property 3 ensures that an update process "reaches" all neighbors whose routing table must be updated.

In the following, it is proved that if `administer` is legal for a given legal initial routing configuration, a publish/subscribe system satisfying Def. 2.1 is implied. First, it is proved that every update process terminates if `administer` is legal (see Lemma 3.8). After that, it is shown that a routing configuration is always weakly valid if `administer` is legal for a given legal initial routing configuration (Lemmas 3.9, 3.10, and 3.11). Together with Theorem 3.3 this leads finally to Theorem 3.4 giving the desired statement.

**Lemma 3.8** *If the* `administer` *procedure is legal, every update process terminates.*

PROOF SKETCH: The framework algorithm ensures that a broker $B$ does not pass back an *admin* message to the broker from which he got the message. This together with the fact that each update process has a unique identifier, that `administer` returns after a finite time (property 1 of legal `administer`), and that $G$ is acyclic implies that each update process terminates.

Now, we prove an important inclusion among a path of brokers leading to a subscribing client.

**Lemma 3.9** *If* `administer` *is legal for a given legal initial routing configuration, for all clients $Y$ the following property holds:*

$$\Box\big[F \in \bar{S}_Y \wedge \Box\neg Unsub(Y,F) \;\Rightarrow\; \forall B_k \neq B_Y.\, \nu_{B_k}(\{B_m\}) \supseteq N(F)\big]$$

*where $B_Y$ is the broker that manages $Y$ and $B_m$ is the next broker on the path from $B_k$ to $Y$.*

PROOF SKETCH: The property states that if `administer` is legal for a given legal initial routing configuration, the update process of a still active subscription $F$ has terminated, and it is never unsubscribed to $F$, then each broker $B_k \neq B_Y$ forwards at least all notifications which are matched by $F$ to the one neighbor which is the next broker on the unique path leading to the subscribing client. The property is shown by an induction over the path from $B_k$ to $B_Y$. First, the base case is proved. After that, the induction step is shown. The induction is valid because of the acyclic topology.

ASSUME: 1. `administer` is legal for a given legal initial routing configuration.
2. $F \in \bar{S}_Y$.
3. $\Box\neg Unsub(Y, F)$.

PROVE: $\forall B_k \neq B_Y . \nu_{B_k}(\{B_m\}) \supseteq N(F)$ where $B_Y$ is the broker that manages $Y$ and $B_m$ is the next broker on the path from $B_k$ to $Y$.

PROOF:

⟨1⟩1. LET: $B_{p_i}, B_{p_{i-1}}, \ldots, B_{p_1}, B_Y$ be the path from an arbitrary broker $B_{p_i} \neq B_Y$ to $B_Y$ where $B_Y$ is the broker that manages $Y$.
PROOF: The path is well-defined and unique because of the acyclic and connected topology defined by $G$. □

⟨1⟩2. Base Case: $\nu_{B_1}(\{B_Y\}) \supseteq N(F)$
PROOF SKETCH: We first show that the property holds at the time the update process of $F$ terminates. After that, it is proved that the property continues to hold forever.
⟨2⟩1. $Y$ subscribed to $F$ and sent $sub(F)$ message $m$ to $B_Y$.
PROOF: by algorithm. □
⟨2⟩2. `administer` is called at $B_Y$ triggered by $m$ and returns.
PROOF: by step ⟨2⟩1, algorithm, property 1 of legal `administer`, and reliable channels. □
⟨2⟩3. After `administer` returned, $\nu_{B_Y}(\{Y\}) \supseteq N(F)$ holds forever.
PROOF: by step ⟨2⟩2, property 2a, property 2c, and assumption 3. □
⟨2⟩4. Moreover, $B_Y$ either an *admin* message $m'$ with $id(m') = id(m)$ is sent to $B_{p_1}$ or not.
PROOF: by step ⟨2⟩2 and algorithm. □
⟨2⟩5. CASE: $B_Y$ sent an *admin* message $m'$ with $id(m') = id(m)$ to $B_{p_1}$.
PROOF:
⟨3⟩1. $m'$ triggers call of `administer` at $B_{p_1}$ which returns after a finite time.
PROOF: by property 1 of legal `administer`, reliable channels, and algorithm. □
⟨3⟩2. Q.E.D.
PROOF: Property 2b of legal `administer` implies that $\nu_{B_1}(\{B_Y\}) \supseteq N(F)$ holds. □
⟨2⟩6. CASE: $B_Y$ sent no *admin* message $m'$ with $id(m') = id(m)$ to $B_{p_1}$.
PROOF: $\nu_{B_1}(\{B_Y\}) \supseteq N(F)$ holds due to property 3 of legal `administer`. □
⟨2⟩7. After $\nu_{B_1}(\{B_Y\}) \supseteq N(F)$ holds, it continuous to hold forever.
PROOF: by step ⟨2⟩5 and ⟨2⟩6, and property 2b and 2c. □
⟨2⟩8. Q.E.D.
PROOF: by step ⟨2⟩5, ⟨2⟩6 , and ⟨2⟩7. □

⟨1⟩3. Induction Step:
ASSUME: for $B_{p_{i-1}}$ and $B_{p_{i-2}}$ holds $\nu_{B_{p_{i-1}}}(\{B_{p_{i-2}}\}) \supseteq N(F)$
PROVE: for $B_{p_i}$ and $B_{p_{i-1}}$ holds $\nu_{B_{p_i}}(\{B_{p_{i-1}}\}) \supseteq N(F)$
PROOF SKETCH: We first prove that the property holds at the time at which the update process terminates in a case distinction. After that, we argue that the property does not stop to hold after this in the final step.

PROOF:
⟨2⟩1. The induction assumption either holds because it was established by some *admin* message $m'$ sent from $B_{p_{i-2}}$ to $B_{p_{i-1}}$ or it holds because of the initial routing configuration.
PROOF: due to property 2c of legal `administer`, algorithm, and reliable channel assumption $\nu_{B_{p_{i-1}}}(\{B_{p_{i-2}}\})$ can only change if $B_{p_{i-1}}$ receives an *admin* message from $B_{p_{i-2}}$. □
⟨2⟩2. CASE: The induction assumption was established by an *admin* message $m'$.
⟨3⟩1. $m'$ triggers a call of `administer` at $B_{p_{i-1}}$ with $S = B_{p_{i-2}}$.
PROOF: by reliable channels and algorithm. □
⟨3⟩2. `administer` returns.
PROOF: by step ⟨3⟩1 and property 1 of legal `administer`, □
⟨3⟩3. After returning, this either causes $B_{p_{i-1}}$ to send an *admin* message $m''$ with $id(m'') = id(m')$ to $B_{p_i}$ or not.
PROOF: by step ⟨3⟩2 and algorithm in Fig. 3.6. □
⟨3⟩4. CASE: A message $m''$ with $id(m'') = id(m')$ was sent by $B_{p_{i-1}}$ to $B_{p_i}$.
⟨4⟩1. $B_{p_i}$ received $m''$ which established the desired inclusion by triggering a call of `administer`.
PROOF: by property 1 (termination), property 2b of legal `administer`, algorithm, and reliable channels. □
⟨4⟩2. Q.E.D.
PROOF: step ⟨4⟩1 shows that the desired inclusion holds. □
⟨3⟩5. CASE: No message $m''$ with $id(m'') = id(m')$ was sent by $B_{p_{i-1}}$ to $B_{p_i}$.
PROOF: property 3 of legal `administer` implies that the inclusion holds. □
⟨2⟩3. CASE: The induction assumption holds from the legal initial routing configuration.
PROOF: The definition of legal initial routing configuration implies that the desired inclusion holds. □
⟨2⟩4. After $\nu_{B_{p_i}}(\{B_{p_{i-1}}\}) \supseteq N(F)$ holds, it continuous to hold forever.
PROOF: Property 2b and 2c of legal `administer`, assumption 4, and reliable channels. □
⟨2⟩5. Q.E.D.
PROOF: by step ⟨2⟩1, ⟨2⟩2, ⟨2⟩2, ⟨2⟩3, and ⟨2⟩4. □
⟨1⟩4. Q.E.D.
PROOF: Step ⟨1⟩2 proves the base case and step ⟨1⟩3 the induction step. Lemma follows from induction over acyclic topology. □

**Lemma 3.10** *If* `administer` *is legal for a given initial routing configuration,* $\Box\big[Y \in L_{B_Y} \Rightarrow \nu_{B_Y}(\{Y\}) = \cup_{F \in S_Y} N(F)\big]$.

PROOF SKETCH: Here, it is shown that the local validity always holds. This is proved by induction. It is shown that the property is satisfied for the initial routing configuration and then for all subsequent configurations.
ASSUME: 1. `administer` is legal for a given initial routing configuration.
2. $S_Y^0 = \emptyset$
PROVE: $\Box\big[Y \in L_{B_Y} \Rightarrow \nu_{B_Y}(\{Y\}) = \cup_{F \in S_Y} N(F)\big]$
PROOF:
⟨1⟩1. $\nu^0_{B_Y}(\{Y\}) = \cup_{F \in S_Y^0} N(F)$

⟨2⟩1. $\nu^0_{B_Y}(\{Y\}) = \emptyset$
PROOF: by assumption 1 and definition of legal initial routing configuration. □
⟨2⟩2. $\cup_{F \in S^0_Y} N(F) = \emptyset$
PROOF: assumption 2. □
⟨2⟩3. Q.E.D.
PROOF: by step ⟨2⟩1 and ⟨2⟩2.
⟨1⟩2. ASSUME: $\nu^{lid_{Y,B_Y}(m)}_{B_Y}(\{Y\}) = \cup_{F \in S^{lid_{Y,B_Y}(m)}_Y} N(F)$
PROVE: $\nu^{id(m)}_{B_Y}(\{Y\}) = \cup_{F \in S^{id(m)}_Y} N(F)$
PROOF:
⟨2⟩1. If $m$ is received, `administer` is called and returns.
PROOF: by property 1 of legal `administer` and algorithm. □
⟨2⟩2. Q.E.D.
PROOF: by step ⟨2⟩1 and property 2a (can be applied because of induction assumption) and 2c of legal `administer`. □
⟨1⟩3. Q.E.D.
PROOF: by induction; step ⟨1⟩1 proved the base step and ⟨1⟩2 proved the induction step. □

**Lemma 3.11** *If* `administer` *is legal for a given legal initial routing configuration, the routing configuration is always weakly valid.*

ASSUME: 1. initial routing configuration is legal.
2. `administer` is legal.
PROVE: The routing configuration is always weakly valid.
PROOF SKETCH: It is shown that the routing configuration is always weakly valid by proving that both local validity and weakly remote validity always hold.
PROOF:
⟨1⟩1. $\Box\big[e(B_i, B_j) \in E \Rightarrow \nu_{B_i}(\{B_j\}) \supseteq \bar{\eta}_{B_j,B_i}\big]$.
ASSUME: $e(B_i, B_j) \in E$
PROVE: $\Box\big[\nu_{B_i}(\{B_j\}) \supseteq \bar{\eta}_{B_j,B_i}\big]$
⟨2⟩1. $\Box\big[F \in \bar{S_Y} \wedge \Box\neg Unsub(Y, F) \Rightarrow \forall B_k \neq B_Y . \nu_{B_k}(\{B_m\}) \supseteq N(F)\big]$ where $B_Y$ is the broker that manages $Y$ and $B_m$ is the next broker on the path from $B_k$ to $Y$.
PROOF: by assumption 1 and 2, and Lemma 3.9. □
⟨2⟩2. $\nu_{B_i}(\{B_j\}) \supseteq \cup_{B \in V_{B_j,B_i}} \cup_{X \in L_B} \cup_{F \in \bar{S}_Y} N(F)$.
PROOF: Step ⟨2⟩1 holds for all subscriptions $F \in \bar{S_Y}$ of all clients $Y$ of any broker $B_Y$ which is on a path starting with $B_i, B_j$. □
⟨2⟩3. $\bar{\eta}_{B_j,B_i} = \cup_{B \in V_{B_j,B_i}} \cup_{X \in L_B} \cup_{F \in \bar{S}_Y} N(F)$
PROOF: by definition of $\bar{\eta}_{B_j,B_i}$ (Eq. 3.8). □
⟨2⟩4. Q.E.D.
PROOF: by Step ⟨2⟩2 and ⟨2⟩3. □
⟨1⟩2. $\Box\big[Y \in L_{B_Y} \Rightarrow \nu_{B_Y}(\{Y\}) = \cup_{F \in S_Y} N(F)\big]$.
PROOF: by Lemma 3.10. □
⟨1⟩3. Q.E.D.
PROOF: by step ⟨1⟩1 and ⟨1⟩2. □

**Theorem 3.4 (Correctness of legal Framework Instantiations)**

*If* `administer` *is valid for a given legal initial routing configuration, the algorithm in Fig. 3.6 satisfies Def. 2.1.*

PROOF: follows from Lemmas 3.8 and 3.11 and Theorem 3.3. □

## 3.5 Routing Algorithms

In this section it is described how some routing algorithms can be realized with the content-based routing framework. The algorithms and their behavior are discussed in detail and it is argued for their correctness by giving proofs. Each algorithm is defined by an instantiation of the `administer` procedure. So only the correctness criterion given in Def. 3.4 needs to be checked. All presented algorithms except flooding ensure that the routing configuration is always weakly perfect. Such routing algorithms are called *perfect* because they minimize unnecessary forwarding of notifications.

### 3.5.1 Flooding

```
Set procedure administer(Sender S, Set 𝔖, Set 𝔘)
begin
   T_B ← T_B ∪ {(F, S) | F ∈ 𝔖}
   T_B ← T_B \ {(G, S) | G ∈ 𝔘}
5  return ∅
end
```

Figure 3.7: Flooding-Based Routing

In the following, flooding is described as an instance of the routing framework and the correctness of flooding is proved. With flooding, the routing table of each broker $B$ is initialized to the constant set $\{(F_T, U) \mid U \in N_B\}$ with $F_T(n) = true$ at system startup. As $N(F_T) = \mathcal{N}$ this routing configuration implies that $F_B^N(n) = N_B$. Hence, a broker will forward a notification received from a local client to all neighbors and a notification received from a neighbor to all other neighbors. This routing strategy is called *flooding* (see Fig. 3.7) because it implies that any published notification is processed by every broker. As the topology is acyclic and as no messages are duplicated, flooding also ensures that every notification is processed at most once by every broker. Flooding is the only routing strategy that does not require the remote routing configuration to be updated. Therefore, no *admin* messages are exchanged. After the initialization, each broker solely adds and deletes routing entries regarding its local clients as they subscribe and unsubscribe:

- If a client $X$ subscribes to a filter $F$, the corresponding broker adds $(F, X)$ to its routing table.

- If a client $X$ unsubscribes to a filter $F$, the corresponding broker deletes $(F, X)$ from its routing table.

**Correctness Proof.** Now, the correctness of flooding is proved. First, it is shown that the initial routing configuration is legal (see Lemma 3.12). After that, it is proved that the `administer` procedure depicted in Fig. 3.7 is legal (see Lemma 3.13). Together with the correctness of the framework this implies the correctness of flooding (see Theorem 3.5).

**Lemma 3.12** *If the routing table of every broker $B$ is initialized to $\{(F_T, U) \mid U \in N_B\}$, the initial routing configuration is legal.*

PROOF SKETCH: It has to be shown that both properties, a legal initial routing configuration has to fulfill, are satisfied. First, we show that the initial remote configuration is legal. After that, we show that the initial local configuration is legal. The proof follows almost directly from the assumptions.
ASSUME: For all $B \in V$, $T_B$ is initialized to $\{(F_T, U) \mid U \in N_B\}$.
PROVE: 1. $e(B_i, B_j) \in E \Rightarrow \nu^0_{B_i}(\{B_j\}) \supseteq \nu^0_{B_j}(N_{B_j} \setminus \{B_i\})$
2. $\nu^0_{B_i}(L_{B_i}) = \emptyset$
PROOF:
⟨1⟩1. $e(B_i, B_j) \in E \Rightarrow \nu^0_{B_i}(\{B_j\}) \supseteq \nu^0_{B_j}(N_{B_j} \setminus \{B_i\})$
ASSUME: $e(B_i, B_j) \in E$
PROVE: $\nu^0_{B_i}(\{B_j\}) \supseteq \nu^0_{B_j}(N_{B_j} \setminus \{B_i\})$
⟨2⟩1. $\nu^0_{B_i}(B_j) = \mathcal{N}$
PROOF: by Lemma assumption and assumption of step ⟨1⟩1. □
⟨2⟩2. $\nu^0_{B_j}(N_{B_j} \setminus \{B_i\}) = \mathcal{N}$
PROOF: by Lemma assumption and assumption of step ⟨1⟩1. □
⟨2⟩3. Q.E.D.
PROOF: by step ⟨2⟩1, ⟨2⟩2, and lemma assumption. □
⟨1⟩2. $\nu^0_{B_i}(L_{B_i}) = \emptyset$
PROOF: follows directly from Lemma assumption. □
⟨1⟩3. Q.E.D.
PROOF: by Step ⟨1⟩1 and ⟨1⟩2. □

After we have shown that the initial routing configuration of flooding is legal, it is now proved that the `administer` procedure shown in Figure 3.7 is legal for this initial configuration.

**Lemma 3.13** *The* `administer` *procedure shown in Figure 3.7 (flooding) is legal for the legal initial routing configuration of Lemma 3.12.*

PROOF SKETCH: We prove that the `administer` procedure shown in Figure 3.7 is legal by showing that the properties 1 to 3 of Def. 3.4 are satisfied.
PROOF:
⟨1⟩1. Property 1 is satisfied.
PROOF: The procedure returns immediately. □
⟨1⟩2. Property 2 is satisfied.
⟨2⟩1. Property 2a is satisfied.

PROOF SKETCH: Shown by a simple induction. After initialization, no local routing entries are present in a routing table. Hence, the equality holds. If a client subscribes a routing entry is added. If a client unsubscribes, the corresponding routing entry is removed. In both cases, the equality holds. Moreover, the local routing entries of a client can only be affected by this client subscribing or unsubscribing. This implies that always exactly the matching notifications are delivered.

PROOF: Implied by the algorithms in Fig. 3.6, Fig. 3.7, and a simple induction. □

⟨2⟩2. Property 2b is satisfied.

PROOF: As `administer` always returns ∅, no broker can ever receive an *admin* message. Therefore, `administer` cannot be called with $S \in N_{B_i}$ and property 2b is satisfied. □

⟨2⟩3. Property 2c is satisfied.

PROOF: The algorithms in Fig. 3.6 (framework) and Fig. 3.7 (flooding) imply that the routing entries regarding a destination can only be affected by a message received from this destination (see Fig. 3.7, lines 3/4). □

⟨2⟩4. Q.E.D.

PROOF: by step ⟨2⟩1, ⟨2⟩2, and ⟨2⟩3.

⟨1⟩3. Property 3 is satisfied.

PROOF SKETCH: As `administer` never returns any triple (see Fig. 3.7, line 5), we have to prove that $\nu_H^{lid_{B,H}(m)}(\{B\}) \supseteq \nu_B^{id(m)}(\{S\})$ holds.

LET: $H \in N_B \setminus \{S\}$

PROVE: $\nu_H^{lid_{B,H}(m)}(\{B\}) \supseteq \nu_B^{id(m)}(\{S\})$

⟨2⟩1. $\nu_H^{lid_{B,H}(m)}(\{B\}) = \mathcal{N}$.

PROOF: follows from property 2c proved in step ⟨1⟩2, that `administer` cannot be called with $S \in N_{B_i}$, and assumption of step ⟨1⟩3. □

⟨2⟩2. $\nu_B^{id(m)}(\{S\}) \subseteq \mathcal{N}$

PROOF: by definition of $\nu$ (Eq. 3.1). □

⟨2⟩3. Q.E.D.

PROOF: by step ⟨2⟩1 and ⟨2⟩2. □

⟨1⟩4. Q.E.D.

PROOF: by step ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3. □

**Theorem 3.5 (Correctness of Flooding)** *The routing framework (see Figure 3.6) with the implementation of* `administer` *shown in Fig. 3.7 (flooding) and the legal initial routing configuration of Lemma 3.12 satisfies Def. 2.1.*

PROOF: follows from Lemmas 3.12, 3.13, and Theorem 3.4. □

Flooding is probably the simplest way to implement a distributed publish/subscribe system. It is also well suited to systems in which subscriptions are changing at a very high rate because routing tables do not need to be updated. The main drawback of flooding is that a lot of notifications may be forwarded unnecessarily because a notification is sent to every broker regardless of whether or not it has a local client with a matching subscription. This leads to the idea of filter-based routing algorithms.

### 3.5.2 Simple Filter-Based Routing

In this section simple filter-based routing is described. This routing strategy uses filter forwarding to update the routing configuration in reaction to subscribing and unsubscribing clients: new and canceled subscriptions are simply flooded into the broker network such that they reach every broker. This allows the brokers to update their routing tables accordingly. The only assumption that underlies simple routing is that each filter can be uniquely identified. The algorithm works as follows:

- Initially, the routing table $T_B$ of each broker $B$ is initialized to $\emptyset$.
- If a client $X$ subscribes to a filter $F$, the corresponding broker adds $(F, X)$ to its routing table and sends an $admin(\{F\}, \emptyset)$ message to all neighbors (see Fig. 3.9).
- If a client $X$ unsubscribes to a filter $F$, the broker removes $(F, X)$ from its routing table and sends an $admin(\emptyset, \{F\})$ message to all neighbors.
- If a broker receives an $admin(\{F\}, \emptyset)$ message from a neighbor $U$, it adds an entry $(F, U)$ to its routing table (see Fig. 3.9). Moreover, the broker forwards the received *admin* message to all of its neighbors except $U$.
- If a broker receives an $admin(\emptyset, \{F\})$ message from a neighbor $U$, it extracts the entry $(F, U)$ from its routing table. Moreover, the broker forwards the received *admin* message to all of its neighbors except $U$.

This strategy is simple but it implies that every broker has to handle a new or canceled subscription. Moreover, the algorithm requires that every broker has global knowledge about all active subscriptions because any routing table contains a routing entry for every active subscription. The corresponding instantiation of the `administer` procedure is shown in Figure 3.8.

```
Set procedure administer(Sender S, Set 𝔖, Set 𝔘)
begin
  T_B ← T_B ∪ {(F, S) | F ∈ 𝔖}
  T_B ← T_B \ {(G, S) | G ∈ 𝔘}
  𝔐 ← {(H, 𝔖, 𝔘) | H ∈ N_B \ {S}}
  return 𝔐
end
```

Figure 3.8: Simple Filter-Based Routing

**Correctness Proof.** Following, the correctness of simple routing is proved (Theorem 3.6) by showing that the initial routing configuration is valid (Lemma 3.14) and that the instantiation of `administer` (Lemma 3.18) is legal for that initial routing configuration. This proves the correctness of simple routing (see Theorem 3.6).

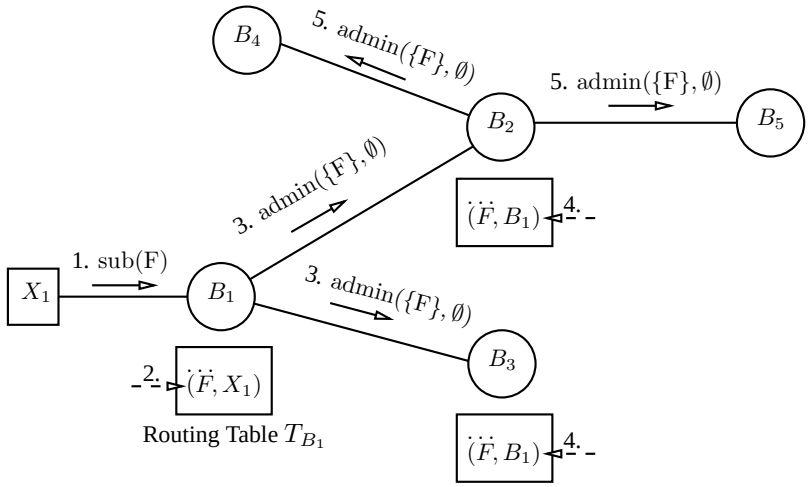



Figure 3.9: Diagram explaining simple routing (new subscription).

**Lemma 3.14 (Legal Empty Routing Configuration)** *If for each broker $B_i$ the routing table $T_{B_i}$ is initialized to $\emptyset$, the initial routing configuration is legal.*

PROOF SKETCH: Here, it is simply shown that both requirements of legal initial routing configurations are met. First, it is shown that the remote initial routing configuration is legal. After that, it is proved that the local initial routing configuration is legal.
ASSUME: $T_{B_i}$ is initialized to $\emptyset$
PROVE: 1. $e(B_i, B_j) \in E \Rightarrow \nu^0_{B_i}(\{B_j\}) \supseteq \nu^0_{B_j}(N_{B_j} \setminus \{B_i\})$
2. $\nu^0_{B_i}(L_{B_i}) = \emptyset$
PROOF:
⟨1⟩1. $e(B_i, B_j) \in E \Rightarrow \nu^0_{B_i}(\{B_j\}) \supseteq \nu^0_{B_j}(N_{B_j} \setminus \{B_i\})$
ASSUME: $e(B_i, B_j) \in E$
PROVE: $\nu^0_{B_i}(\{B_j\}) \supseteq \nu^0_{B_j}(N_{B_j} \setminus \{B_i\})$
⟨2⟩1. $\nu^0_{B_i}(\{B_j\}) = \emptyset$
PROOF: Implied by Lemma assumption and assumption of step ⟨1⟩1. □
⟨2⟩2. $\nu^0_{B_j}(N_{B_j} \setminus \{B_i\}) = \emptyset$
PROOF: Implied by Lemma assumption and assumption of step ⟨1⟩1. □
⟨2⟩3. Q.E.D.
PROOF: by step ⟨2⟩1,⟨2⟩2, and assumption of step ⟨1⟩1. □
⟨1⟩2. $\nu^0_{B_i}(L_{B_i}) = \emptyset$
PROOF: follows directly from Lemma assumption. □
⟨1⟩3. Q.E.D.
PROOF: by step ⟨1⟩1 and ⟨1⟩2. □

After we have proved that the empty initial routing configuration is legal, it is shown that simple routing is legal for this initial configuration. Before we prove this property we need some preparation. Let $Q^E_B(D)$ ($E$ stands for "except") be the set of all filters of all routing entries in the routing table of a broker $B$ except those regarding destination $D$. Moreover, let $Q^O_B(D)$ ($O$ stands

for "only") be the set of all filters of all routing entries in the routing table of a broker $B$ regarding destination $D$:

$$Q_B^E(D) = \{F \mid \exists (F, D') \in T_B \wedge D' \neq D\} \quad (3.9)$$

$$Q_B^O(D) = \{F \mid \exists (F, D) \in T_B\} \quad (3.10)$$

Now consider for a broker $B$ and one of its neighbors $H$ the following sets:

$$\alpha = Q_B^E(H) \quad (3.11)$$

$$\beta = Q_H^O(B) \quad (3.12)$$

The sets $\alpha$ and $\beta$ are those routing entries that determine $\nu_B(L_B \cup N_B \setminus \{H\})$ and $\nu_H(\{B\})$, respectively. More precisely, the definition of $\nu$ (Eq. 3.1) implies that $\nu_B(L_B \cup N_B \setminus \{H\}) = \cup_{F \in \alpha} N(F)$ and $\nu_H(\{B\}) = \cup_{F \in \beta} N(F)$.

**Lemma 3.15** *If $\alpha = \beta$, then $\nu_H(\{B\}) = \nu_B(L_B \cup N_B \setminus \{H\})$.*

PROOF: The property follows directly from the Definitions of $\alpha$ (Eq. 3.11), $\beta$ (Eq. 3.12), and $\nu$ (Eq. 3.1). □

The above Lemma states that if $\alpha$ and $\beta$ contain the same filters, then $H$ forwards exactly those notifications to $B$ that $B$ forwards in turn to all other destinations. This situation is depicted in Fig. 3.10. Simple routing ensures that this property holds among corresponding versions of $\alpha$ and $\beta$.

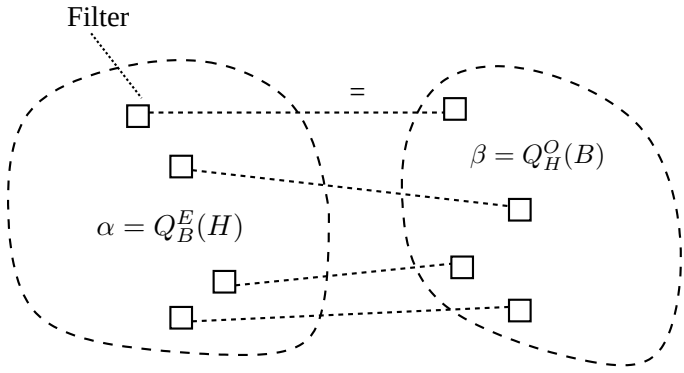



Figure 3.10: Relation among $\alpha$ and $\beta$ for simple routing.

The following Lemma proves that $B$ sends an *admin* message to $H$ if $\alpha$ changes.

**Lemma 3.16** *If simple routing is used and $\alpha$ changes, then $B$ sends an admin message to $H$.*

PROOF: From the framework algorithm (see Fig. 3.6) and the algorithm in Fig. 3.8 (simple routing) we know that $\alpha$ can only change, if $B$ receives a *sub* or *unsub* message from a local client or an *admin* message from a neighbor except $H$. In all these cases, `administer` is called and a triple is returned for $H$ (see Fig. 3.8, line 5). According to the framework this implies that an *admin* message is sent to $H$. □

**Lemma 3.17** *If simple routing with the empty legal initial routing configuration is used, the following property holds: If $H$ receives an admin message $m$ from $B$, then $\alpha^{id(m)} = \beta^{id(m)}$.*

PROOF SKETCH: The Lemma is proved by an induction. First, it is shown that the property holds for the first *admin* message that $H$ receives from $B$. Here, the fact that the property holds initially due to the legal initial routing configuration is used. After that, the induction step is shown that uses Lemma 3.16.

Now, we are prepared to show that simple routing is legal for the empty initial routing configuration.

**Lemma 3.18** *The* `administer` *procedure shown in Fig. 3.8 (simple routing ) is legal for the legal empty initial routing configuration.*

PROOF SKETCH: We prove that the `administer` procedure shown in Fig. 3.8 (simple routing) is legal by showing that the properties 1 to 3 of Def. 3.4 are satisfied.
PROOF:
⟨1⟩1. Property 1 is satisfied.
PROOF: It is easy to see that the procedure returns. □
⟨1⟩2. Property 2 is satisfied.
⟨2⟩1. Property 2a is satisfied.
PROOF: As *unsub* and *sub* messages are handled in the same way, the same proof as in Lemma 3.13 holds. □
⟨2⟩2. Property 2b is satisfied.
ASSUME: $S \in N_B$
PROVE: $\nu_B^{id(m)}(\{S\}) \supseteq \nu_S^{id(m)}(L_S \cup N_S \setminus \{B\})$
PROOF: by Lemmas 3.15 and 3.17. □
⟨2⟩3. Property 2c is satisfied.
PROOF: The algorithms in Fig. 3.6 (framework) and Fig. 3.8 (simple routing) imply that the routing entries regarding a destination can only be affected by a message received from this destination (see Fig. 3.8, lines 3/4). □
⟨2⟩4. Q.E.D.
PROOF: by ⟨2⟩1, ⟨2⟩2, and ⟨2⟩3. □
⟨1⟩3. Property 3 is satisfied.
⟨2⟩1. Q.E.D.
PROOF: Properties 3 is satisfied because `administer` shown in Fig. 3.8 (simple routing) returns exactly one triple for each neighbor except $S$ (see Fig. 3.8, line 5). □
⟨1⟩4. Q.E.D.
PROOF: by ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3. □

**Theorem 3.6 (Correctness of Simple Filter-Based Routing)** *The routing framework with* `administer` *shown in Fig. 3.8 (simple routing) and the empty legal initial routing configuration satisfy Def. 2.1.*

PROOF: by Lemmas 3.14, 3.18, and Theorem 3.4. □

### 3.5.3 Routing based on Filter Identity

The simple filter-based routing algorithm described in the last section enforces that every broker has knowledge about all active subscriptions. Therefore, the size of each routing table grows linearly with the number of active subscriptions. In the next subsections, routing algorithms are presented that avoid global knowledge by taking into account *similarities* among the subscriptions. These algorithms are based on the following idea: The set of notifications that a broker $B_i$ forwards to a broker $B_j$, i.e., $\nu_{B_i}\{B_j\}$, is the set of all notifications that are matched by any routing entry $(F, B_j)$ in $T_{B_i}$. In general, a subset of these routing entries might be sufficient. For example, there can be two routing entries $(F, B_j)$ and $(G, B_j)$ with $N(F) = N(G)$. Clearly, one of these entries is sufficient as both have identical sets of matching notifications. This fact is used by the identity-based routing algorithm to systematically avoid redundant routing entries and unnecessary forwarding of subscriptions and unsubscriptions.

**Basic Idea of Identity-Based Routing.** Roughly speaking, the basic idea of identity-routing is the following:

- A subscription is not forwarded to a neighbor if an identical subscription that has not been canceled was forwarded to that neighbor.
- An unsubscription is not forwarded to a neighbor if there is another subscription of a local client or another neighbor that is identical to the former.

**Basic Definitions.** Before identity-based routing is described in more detail, some basic definitions are needed. Formally, two filters $F$ and $G$ are *identical*, denoted by $F \equiv G$, if $N(F) = N(G)$. We denote the set of all routing entries in a routing table of a broker $B$ whose filter is identical to a given filter $F$ and whose destination equals a given destination $D$ with $C_B^I(F, D)$ (the $I$ stands for "identity"). Moreover, we denote with $D_B^I(F)$ the set of all neighbors $H$ for which no routing entry $(G, D)$ in the routing table of $B$ exists where $G$ is identical to $F$ and $D$ is distinct from $H$:

$$C_B^I(F, D) = \{(G, D) \mid (G, D) \in T_B \ \wedge \ F \equiv G\}, \tag{3.13}$$

$$D_B^I(F) = \{H \in N_B \mid \nexists G \in Q_B^E(H).\, F \equiv G\}. \tag{3.14}$$

Now, the processing of subscriptions and unsubscriptions can be described in more detail.

**Processing of Subscriptions and Unsubscriptions.** If a broker $B$ receives a subscription or unsubscription $F$ from a neighbor or a local client $S$, it does the following:

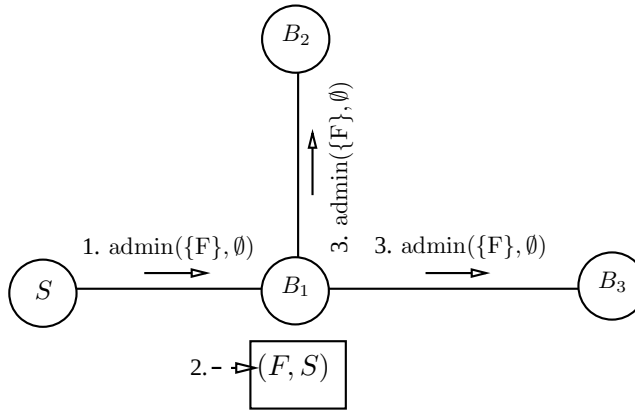
- First, $B$ updates its routing table:

  - If $S$ is a neighbor, $B$ removes all routing entries whose filter is identical to $F$ and that refer to the destination $S$, i.e., $C_B^I(F, S)$.
  - If $S$ is a local client, $B$ removes solely $(F, S)$. This is to ensure the disjunctive interpretation of multiple active subscriptions of a client (cf. Sect. 2.2) and idempotent resubscriptions (cf. Sect. 3.7).

- After that, $B$ forwards $F$ to all neighbors which are in $D_B^I(F)$ except $S$.
- Finally, if $F$ is a subscription, $B$ inserts a routing entry $(F, S)$ into its routing table.

**Examples.** In Figure 3.11, broker $B_1$ receives a new subscription $F$ from a neighbor $S$. $B_1$ inserts $(F, S)$ into its routing table and forwards $F$ to its neighbors $B_2$ and $B_3$ because they are both in $D_B^I(F) \setminus \{S\}$.

In the second example (see Fig. 3.12), broker $B_1$ receives a new subscription $F$ from a local client $S$. Here, $B_1$ also inserts $(F, S)$ into its routing table but forwards $F$ only to its neighbor $B_3$ which is the only neighbor in $D_B^I(F) \setminus \{S\}$. $B_2$ is not in that set due to the routing entry $(F', B_3)$ where $F' \equiv F$.




$$D_B^I(F) \setminus \{S\} = \{B_2, B_3\}$$

Figure 3.11: Processing a new subscription from a neighbor.

**Messages regarding Subscriptions and Unsubscriptions.** With identity-based routing, a broker has to handle four types of messages regarding subscriptions and unsubscriptions:

- *sub*$(F)$: a subscription received from a local client.
- *unsub*$(F)$: an unsubscription received from a local client.
- *admin*$(\{F\}, \emptyset)$: a new subscription received from a neighbor.

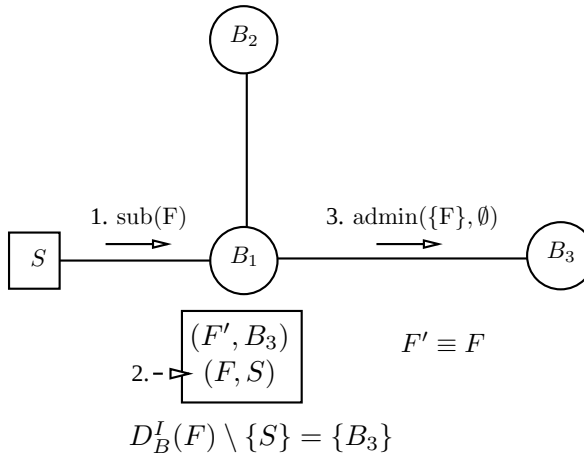



Figure 3.12: Processing a new subscription from a local client.

- *admin*$(\emptyset, \{F\})$: an unsubscription received from a neighbor.

These messages are converted into appropriate calls of `administer` according to the framework.

**Instantiation of `administer`.** The complete code of the identity-based version of `administer` is shown in Fig. 3.13. The procedure uses a helper procedure `generate` (see Fig. 3.14) which takes as arguments two sets of tuples $\mathcal{S}_S$ and $\mathcal{S}_U$ which are routing entries. Each entry $(F, U)$ in $\mathcal{S}_S$ is interpreted as a subscription $F$ that should be sent to $U$. Similar, each entry $(F, U)$ in $\mathcal{S}_U$ is interpreted as an unsubscription $F$ that should be sent to $U$. From these two sets the procedure generates a set of triples which contains exactly one triple for each broker to which a filter should be forwarded. In particular, each triple $(H, \mathcal{S}_H, \mathcal{U}_H)$ contains all the subscriptions $\mathcal{S}_H$ and unsubscriptions $\mathcal{U}_H$ that should be sent to $H$.

**Correctness Proof.** In the following, the correctness of identity-based routing is proved. It is shown that the `administer` procedure shown in Fig. 3.13 is legal for the empty legal initial routing configuration. This directly gives Theorem 3.7.

Again, consider $\alpha = Q_B^E(H)$ (see Eq. 3.9) and $\beta = Q_H^O(B)$ (see Eq. 3.10).

**Lemma 3.19** *If for each filter $F$ in $\alpha$ there is a filter $G$ in $\beta$ such that $G \equiv F$, then $\nu_H(\{B\}) \supseteq \nu_B(L_B \cup N_B \setminus \{H\})$.*

PROOF: by definitions of $\alpha$ (Eq. 3.11), $\beta$ (Eq. 3.12), and $\nu$ (Eq. 3.1). □

The above Lemma states that if for each filter in $\alpha$ there is a filter $\beta$ that is identical, then $H$ forwards *at least* those notifications to $B$ that $B$ forwards in turn to all other destinations.

```
Set procedure administer(Sender S, Set 𝔖, Set 𝔘)
begin
  F_S ← ∅
  F_U ← ∅
5
  // handle subscriptions and unsubscriptions
  forall F ∈ 𝔖 ∪ 𝔘 do
    if S ∈ N_B then
      T_B ← T_B \ C_B^I(F, S)
10  else
      T_B ← T_B \ {(F, S)}
    fi

    A ← {(F, H) | H ∈ D_B^I(F) \ {S}}
15  if F ∈ 𝔘 then
      F_U ← F_U ∪ A
    else
      F_S ← F_S ∪ A
      T_B ← T_B ∪ {(F, S)}
20  fi
  end

  // generate triples
  M ← generate(F_S, F_U)
25 return 𝔐
end
```

Figure 3.13: Routing based on Filter Identity

**Lemma 3.20** *If for each filter $G$ in $\beta$ there is a filter $F$ in $\alpha$ such that $G \equiv F$, then $\nu_H(\{B\}) \subseteq \nu_B(L_B \cup N_B \setminus \{H\})$.*

PROOF: by definitions of $\alpha$ (Eq. 3.11), $\beta$ (Eq. 3.12), and $\nu$ (Eq. 3.1). □

The above Lemma states that if for each filter in $\beta$ there is a filter in $\alpha$ that is identical, then $H$ forwards *at most* those notifications to $B$ that $B$ forwards in turn to all other destinations. This implies that if the conditions of Lemmas 3.19 and 3.20 are satisfied, then $\nu_H(\{B\}) = \nu_B(L_B \cup N_B \setminus \{H\})$ holds. This situation is depicted in Fig. 3.15. $\alpha$ and $\beta$ are denoted *identity-equivalent* if (a) for each filter in $\alpha$ there is a filter in $\beta$ that is identical and if (b) for each filter in $\beta$ there is a filter in $\alpha$ that is identical. Identity-based routing ensures that this property holds among corresponding versions of $\alpha$ and $\beta$.

**Lemma 3.21** *If identity-based routing is used with the empty legal initial routing configuration the following properties hold:*

1. *If* `administer` *is called at a broker $B$ triggered by a message $m$ received from a neighbor or local client $S$ and $\alpha^{id(m)}$ is not identity-equivalent to $\beta^{lid_{B,H}(m)}$ for a neighbor $H \neq S$, then a triple for $H$ is returned.*

2. *If a triple was returned for a neighbor $H$, $H$ receives an admin message*

```
Set procedure generate(Set 𝔖_S, Set 𝔖_U)
begin
  𝔐 ← ∅
  forall H ∈ N_B do
    𝔐_S ← {F | (F, H) ∈ 𝔖_S}
    𝔐_U ← {F | (F, H) ∈ 𝔖_U}
    if (𝔐_S ≠ ∅ ∨ 𝔐_U ≠ ∅) then
      𝔐 ← 𝔐 ∪ {(H, 𝔐_S, 𝔐_U)}
    fi
  end
  return 𝔐
end
```

Figure 3.14: Procedure `generate`

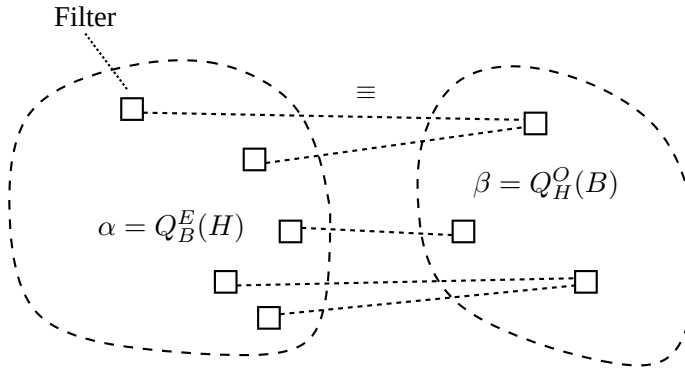



Figure 3.15: Relation among $\alpha$ and $\beta$ for identity-based routing.

*$m'$ with $id(m') = id(m)$ from $B$,* `administer` *is called at $H$ triggered by $m'$, and $\alpha^{id(m)}$ is identity-equivalent to $\beta^{id(m)}$.*

PROOF SKETCH: The Lemma is proved by an induction. First, the base case is proved. Here, it is shown that the properties hold with respect to the first time `administer` is called at $B$. This is done in a case distinction that uses the fact that $\alpha^0$ and $\beta^0$ are both initially empty. After that, the induction step is shown. The induction assumption is that $\alpha^{lid_B(m)}$ is identity-equivalent to $\beta^{lid_{B,H}(m)}$. The properties are proved in a tedious case distinction.

Now, we are prepared to show that identity-based routing is legal for the empty initial routing configuration.

**Lemma 3.22** *The* `administer` *procedure shown in Fig. 3.13 (identity-based routing) is legal for the legal empty initial routing configuration.*

PROOF SKETCH: We prove that the `administer` procedure shown in Fig. 3.13 is legal by showing that the properties 1 to 3 of Def. 3.4 are satisfied.
PROOF:

⟨1⟩1. Property 1 is satisfied.
PROVE: `administer` returns.
PROOF: It is easy to see that the procedure returns. □
⟨1⟩2. Property 2 is satisfied.
⟨2⟩1. Property 2a is satisfied.
PROOF: As *unsub* and *sub* messages are handled in the same way, the same proof as in Lemma 3.13 holds. □
⟨2⟩2. Property 2b is satisfied.
ASSUME: $S \in N_B$
PROVE: $\nu_B^{id(m)}(\{S\}) \supseteq \nu_S^{id(m)}(L_S \cup N_S \setminus \{B\})$
PROOF: by Lemmas 3.19 and 3.21. □
⟨2⟩3. Property 2c is satisfied.
PROVE: $\nu_B^{id(m)}(L_B \cup N_B \setminus \{S\}) = \nu_B^{lid_B(m)}(L_B \cup N_B \setminus \{S\})$
PROOF: follows from algorithm in Fig. 3.13. □
⟨2⟩4. Q.E.D.
PROOF: follows from step ⟨2⟩1, ⟨2⟩2, and ⟨2⟩3. □
⟨1⟩3. Property 3 is satisfied.
ASSUME: $\nu_H^{lid_{B,H}(m)}(\{B\}) \not\supseteq \nu_B^{id(m)}(\{S\})$
PROVE: $\exists(H, \mathcal{S}_H, \mathcal{U}_H) \in \mathcal{M}$
PROOF: by Lemmas 3.19 and 3.21. □
⟨1⟩4. Q.E.D.
PROOF: follows from step ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3. □

**Theorem 3.7 (Correctness of Routing based on Filter Identity)** *The framework with the* `administer` *implementation shown in Figure 3.13 (identity-based routing) and the empty initial routing configuration satisfy Def. 2.1.*

PROOF: by Lemmas 3.14, 3.22, and Theorem 3.4. □

### 3.5.4 Routing based on Filter Covering

After discussing identity-based routing, an obvious idea is to exploit more complex similarities among subscriptions. The next step is to take advantage of covering among filters, a concept that was first mentioned in the area of notification services by Carzaniga [17]. A filter *covers another filter* if the former matches all notifications the latter matches. Therefore, a routing entry $(F, U)$ is obsolete if there exists a routing entry $(G, U)$ where $G$ covers $F$. This fact is used by the covering-based routing algorithm to systematically avoid redundant routing entries and unnecessary forwarding of subscriptions and unsubscriptions.

**Basic Idea of Covering-Based Routing.** Roughly speaking, the basic idea of covering-based routing is the following:

- A subscription is not forwarded to a neighbor if a subscription that covers the former was forwarded to that neighbor that has not been canceled.
- If a subscription is forwarded, the receiving broker deletes all routing entries whose filters are covered by the new subscription and that refer to the same destination in order to get rid of the obsolete routing entries.

- An unsubscription is not forwarded to a neighbor if there is a subscription of a local client or another neighbor that covers the former.
- If an unsubscription is forwarded to a neighbor, the sending broker also forwards a possibly empty subset of *uncovered* subscriptions. This is done to ensure the delivery of notifications that match these existing subscriptions.

The algorithm presented in the following either processes a single subscription or a single unsubscription that comes along with a set of uncovered subscriptions. Before these steps are described in detail, some basic definitions are needed.

**Basic Definitions.** Formally, a filter $F$ *covers a filter* $G$, denoted by $F \sqsupseteq G$ iff $N(F) \supseteq N(G)$. The pair $(\mathcal{F}, \sqsupseteq)$ defines a reflexive partial order over the set of all filters $\mathcal{F}$. If $F \sqsupseteq G$ then $n \in N(G)$ implies $n \in N(F)$. $F$ is a *real cover* of $G$, denoted by $F \sqsupset G$, iff $N(F) \supset N(G)$.

We define the set $C_B^L(F)$ (the $L$ stands for "lower") that comprise the set of all routing entries in the routing table of a broker $B$ that are covered by a given filter $F$. We also define $C_B^L(F, D)$ as the restriction of $C_B^L(F)$ to a given destination $D$. Additionally, we denote with $D_B^U(F)$ (the $U$ stands for "upper") as the set of all neighbors $H$ for which no routing entry $(G, D)$ in the routing table of $B$ exists where $G$ covers $F$ and $D$ is distinct from $H$. With $D_B^{RU}(F)$ (the $RU$ stands for "real upper") the set of all neighbors $H$ for which no routing entry $(G, D)$ in the routing table of $B$ exists where $G$ is a real cover of $F$ and $D$ is distinct from $H$:

$$C_B^L(F) = \{(G,U) \mid (G,U) \in T_B \ \wedge \ F \sqsupseteq G\} \tag{3.15}$$

$$C_B^L(F,D) = \{(G,D) \mid (G,D) \in C_B^L(F)\} \tag{3.16}$$

$$D_B^U(F) = \{H \in N_B \mid \nexists G \in Q_B^E(H).\, G \sqsupseteq F\} \tag{3.17}$$

$$D_B^{RU}(F) = \{H \in N_B \mid \nexists G \in Q_B^E(H).\, G \sqsupset F\} \tag{3.18}$$

**Processing of a Subscription.** If a broker $B$ receives a new subscription $F$ from a neighbor or a local client $S$, it does the following:

- First, $B$ updates its routing table:
  - If $S$ is a neighbor, $B$ removes all entries whose filters are covered by $F$ that refer to $S$, i.e., $C_B^L(F, S)$, to get rid of the obsolete routing entries. This is also done to ensure idempotent resubscriptions (cf. Sect. 3.7).
  - If $S$ is a local client, $B$ removes solely $(F, S)$. This is to ensure the disjunctive interpretation of multiple active subscriptions of a client (cf. Sect. 2.2) and idempotent resubscriptions (cf. Sect. 3.7).
- Next, $B$ forwards $F$ to all neighbors which are in $D_B^U(F)$ except $S$.
- Finally, $B$ inserts $(F, S)$ into its routing table.

**Processing of an Unsubscription.** An unsubscription $F$ is first received by a broker from a local client as *unsub*($F$) message. The fact that complicates covering-based routing is that simply to forward an unsubscription to some neighbors is not sufficient. Instead, to each neighbor to which $F$ is forwarded also a possibly empty subset of filters which are really covered by $F$ has to be forwarded. This is done to ensure the delivery of notifications that match these existing subscriptions which are called *uncovered* in the following. Hence, an unsubscription $F$ that is received from a neighbor comes along with a possibly empty set of uncovered subscriptions $\{F_1, \ldots F_n\}$ (where $\forall i.F \sqsupset F_i \wedge (\forall i, j.i \neq j \Rightarrow F_i \not\sqsupseteq F_j)$) and may newly uncover subscriptions at the receiving broker. It is important that the unsubscription and the corresponding uncovered subscriptions are forwarded in a single message in order to guarantee that the change to the routing table of the receiving broker is atomic. Otherwise, in the intermediate time between the cancellation of the unsubscription and the time at which the uncovered subscriptions become effective, notifications may be lost.

Now, the processing of the unsubscription is described in detail. If a broker $B$ receives an unsubscription $F$ from a neighbor or a local client $S$ it does the following:

- First, $B$ updates its routing table:
  - If $S$ is a neighbor, $B$ removes all routing entries whose filter is covered by $F$ and that refer to the destination $S$, i.e, $C_B^L(F, S)$.
  - If $S$ is a local client, $B$ removes solely $(F, S)$. This is to ensure the disjunctive interpretation of multiple active subscriptions of a client (cf. Sect. 2.2) and idempotent resubscriptions (cf. Sect. 3.7).
- Next, $B$ forwards $F$ to all neighbors which are in $D_B^U(F)$ except $S$.
- Finally, the newly uncovered subscriptions, i.e., all filters in $C_B^L(F) \backslash C_B^I(F)$, are added into the set $P$ which serves as a temporary storage.

**Processing of newly uncovered Subscriptions.** The uncovered subscriptions are processed in the following way:

- First, the uncovered subscription that came along with the subscription are added to $P$ and inserted into the routing table.
- Second, for each entry $(F, U) \in P$ representing an uncovered subscription $F$ regarding destination $U$, $B$ does the following:
  - $B$ determines the number of destinations $k$ in $P$ with subscriptions identical to $F$, i.e., $k = \#\{H \mid (G, H) \in P \ \wedge \ G \equiv F\}$.
  - $(F, U)$ and all entries with identical filters are removed from $P$.
  - Next, $B$ forwards $F$ to all neighbors which are in $D_B^{RU}(F)$ except $S$, and except $U$ if $k = 1$. Note that the value of $k$ has no effect if $S = U$.

**Messages regarding Subscriptions and Unsubscriptions.** With covering-based routing a broker has to handle four types of messages regarding subscriptions and unsubscriptions:

- $sub(F)$: a subscription received from a local client.
- $unsub(F)$: an unsubscription received from a local client.
- $admin(\{F\}, \emptyset)$: a new subscription received from a neighbor.
- $admin(\{F_1, \ldots, F_n\}, \{F\})$ where $\forall i.\, F \sqsupset F_i$ and $(\forall i, j.\, (i \neq j \Rightarrow F_i \not\sqsupseteq F_j))$: an unsubscription with uncovered subscriptions received from a neighbor.

These messages are converted into appropriate calls of `administer` according to the framework.

**Examples.** In Figure 3.16, $B_1$ receives a new subscription $F$ from a local client $S$. Therefore, $B_1$ adds $(F, S)$ to its routing table. Moreover, $B_1$ forwards $F$ only to its neighbor $B_3$ because $B_3$ is the only neighbor in $D_B^U(F) \setminus \{S\}$. $B_2$ is not in this set because of the routing entry $(G, B_3)$ where $G \sqsupseteq F$.

In the second example (see Fig. 3.17), $B_1$ receives a subscription $F$ from a neighbor $S$. $B_1$ removes the entry $(G, S)$ from its routing table because the entry is in $C_B^L(F, S)$. Moreover, $B_1$ inserts $(F, S)$ into its routing table. Finally, $B_1$ forwards $F$ to its neighbors $B_2$ and $B_3$ because they are both in $D_B^U(F) \setminus \{S\}$.

In Figure 3.18, broker $B_1$ receives an unsubscription $F$ from a neighbor $S$. Hence, $B_1$ removes $(F, S)$. Furthermore, $B_1$ forwards the unsubscription to its neighbors $B_2$ and $B_3$ as both are in $D_B^U(F) \setminus \{S\}$.

In the next example, (see Fig. 3.19), $B_1$ receives an unsubscription $F$ from a local client $S$. Hence, $B_1$ removes $(F, S)$. In this case, $B_1$ forwards the unsubscription only to $B_3$ because it is the only broker in $D_B^U(F) \setminus \{S\}$. $B_2$ is not in this set because of the routing entry $(G, B_3)$ where $G \sqsupseteq F$.

In Figure 3.20, broker $B_1$ receives an unsubscription $F$ from a local client $S$. Hence, it removes $(F, S)$ from its routing table. In this example the unsubscription $F$ uncovers a subscription $G$. While the subscription $F$ is forwarded to $B_2$ and $B_3$, the uncovered subscription $G$ is solely forwarded to $B_3$. $G$ is not forwarded to $B_2$ although it is in $D_B^{RU}(G) \setminus \{S\}$ because $k = 1$.

In the last example (see Fig. 3.21), broker $B_1$ receives an unsubscription $F$ that comes along with an uncovered subscription $G'$. Moreover, in the routing table of $B_1$ there is an entry $(G, S)$ where $G \equiv G'$. Here, $B_1$ removes $(F, S)$ from and inserts $(G', S)$ into its routing table. The unsubscription $F$ and the uncovered subscription $G$ are sent to $B_2$ *and* $B_3$. $G$ is forwarded to $B_2$ and $B_3$ because they are both in $D_B^{RU}(G) \setminus \{S\}$ and additionally $k = 2$ holds.

**Instantiation of `administer`.** The complete algorithm is shown in Fig. 3.22. The algorithm consists of two main parts (1) the processing of a subscription (lines 9-19) and (2) the processing of an unsubscription that comes along with

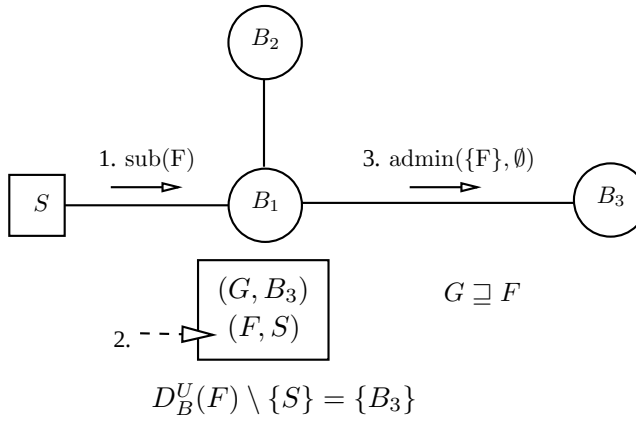



Figure 3.16: Processing of a new subscription from a local client.

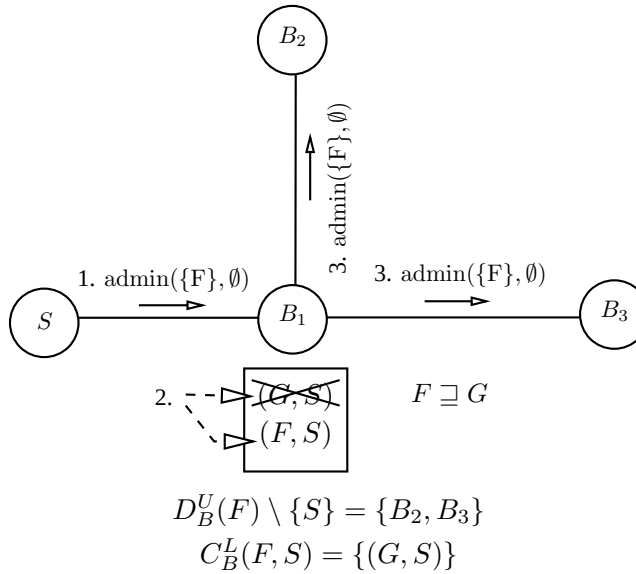



Figure 3.17: Processing of a new subscription from a neighbor.

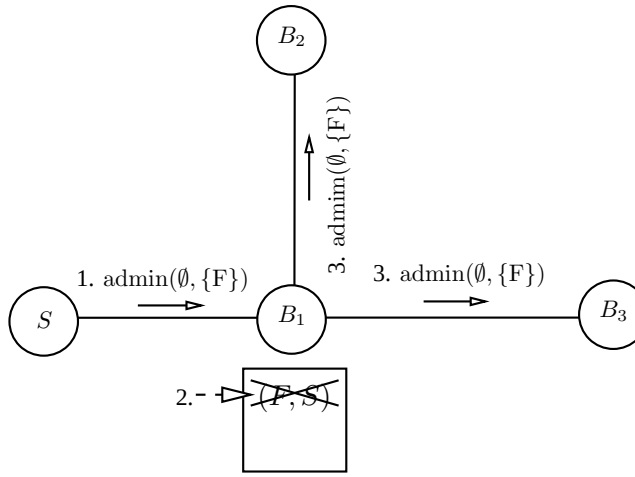



Figure 3.18: Processing of an unsubscription from a neighbor.

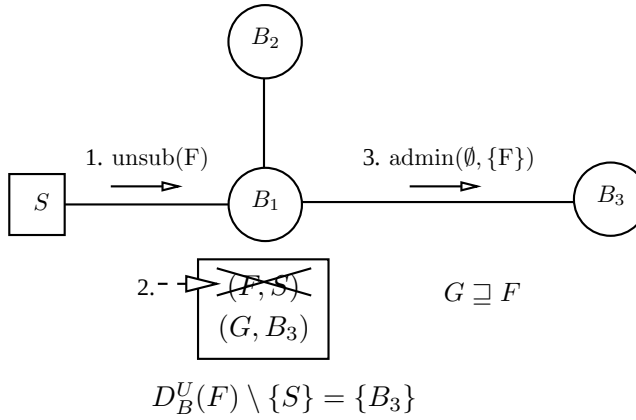



Figure 3.19: Processing of an unsubscription from a local client.

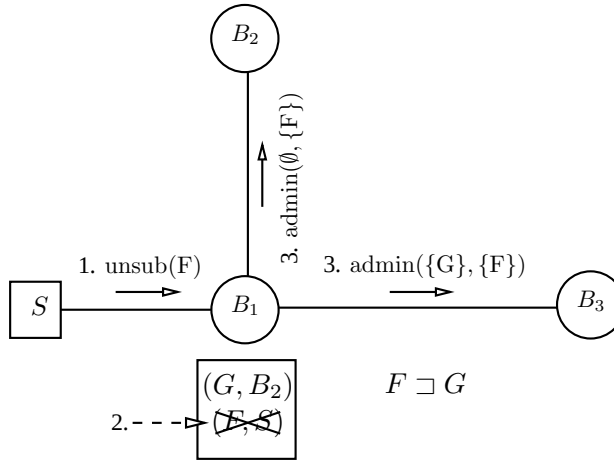



Figure 3.20: Processing of an unsubscription from a local client.

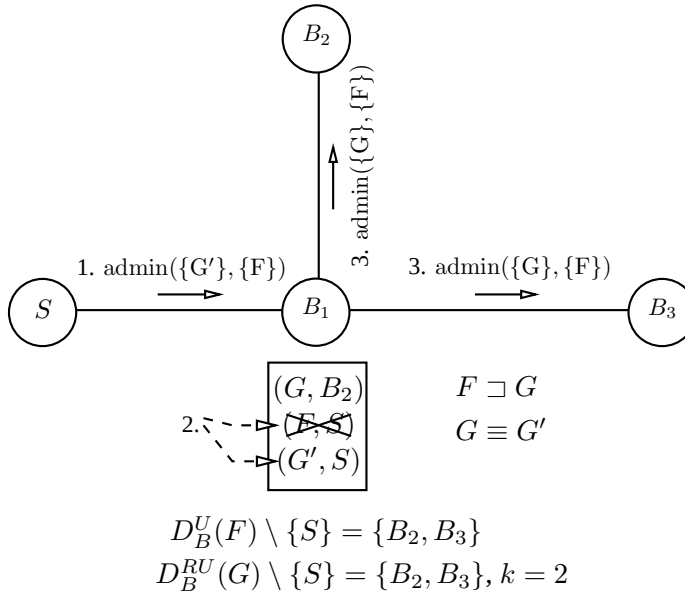



Figure 3.21: Processing of an unsubscription from a neighbor.

a set of uncovered subscriptions and may newly uncover subscriptions (lines 20-42). Finally, the set of triples is determined by calling the `generate` procedure and returned (lines 46-47).

**Correctness Proof.** In the following, the correctness of covering-based routing is proved. Again, it is assumed that the initial routing configuration is empty. First, it is shown that the `administer` procedure shown in Fig. 3.22 is legal. This directly gives Theorem 3.8.

Again, consider $\alpha = Q_B^E(H)$ (see Eq. 3.9) and $\beta = Q_H^O(B)$ (see Eq. 3.10). If $B$ receives a *sub*, *unsub*, or *admin* message, $\alpha$ changes.

**Lemma 3.23** *If for each filter $F$ in $\alpha$ there is a filter $G$ in $\beta$ such that $G \sqsupseteq F$, then $\nu_H(\{B\}) \supseteq \nu_B(L_B \cup N_B \setminus \{H\})$.*

PROOF: by definitions of $\alpha$ (Eq. 3.11), $\beta$ (Eq. 3.12), and $\nu$ (Eq. 3.1). □

Lemma 3.23 states that if for each filter in $\alpha$ there is a filter $\beta$ that covers the former, then $H$ forwards *at least* those notifications to $B$ that $B$ forwards in turn to all other destinations. This implies that if the conditions of Lemmas 3.20 and 3.23 are satisfied, then $\nu_H(\{B\}) = \nu_B(L_B \cup N_B \setminus \{H\})$ holds. This situation is depicted in Fig. 3.23. $\alpha$ and $\beta$ are denoted *covering-equivalent* if (a) for each filter in $\alpha$ there is a filter in $\beta$ that covers the former and if (b) for each filter in $\beta$ there is a filter in $\alpha$ that is identical. Covering-based routing ensures that this property holds among corresponding versions of $\alpha$ and $\beta$. Additionally, it guarantees that for each filter in $\beta$ there is no other filter in $\beta$ that covers the former.

**Lemma 3.24** *If covering-based routing is used the following properties hold:*

1. *If* `administer` *is called at a broker $B$ triggered by a message $m$ received from a neighbor or local client $S$ and $\alpha^{id(m)}$ is not covering-equivalent to $\beta^{lid_{B,H}(m)}$ for a neighbor $H \neq S$, then a triple for $H$ is returned.*
2. *If a triple was returned for a neighbor $H$, $H$ receives an admin message $m'$ with $id(m') = id(m)$ from $B$,* `administer` *is called at $H$ triggered by $m'$, and $\alpha^{id(m)}$ is covering-equivalent to $\beta^{id(m)}$.*

PROOF SKETCH: The Lemma is proved by an induction. First, the base case is proved. Here, it is shown that the properties hold with respect to the first time `administer` is called at $B$. This is done in a case distinction that uses the fact that $\alpha^0$ and $\beta^0$ are both initially empty. After that, the induction step is shown. The induction assumption is that $\alpha^{lid_B(m)}$ is covering-equivalent to $\beta^{lid_{B,H}(m)}$. The properties are proved in a tedious case distinction.

Now, we are prepared to show that covering-based routing is legal for the empty initial routing configuration.

**Lemma 3.25** *The* `administer` *procedure shown in Fig. 3.22 (covering-based routing) is legal for the empty legal initial routing configuration.*

```
Set procedure administer(Sender S, Set 𝔖, Set 𝔘)
begin
  ℱ_S ← ∅
  ℱ_U ← ∅
  P ← ∅

  if 𝔘 = ∅
    // handle subscription
    for the filter F ∈ 𝔖 do
      if S ∈ N_B then
        T_B ← T_B \ C_B^L(F, S)
      else
        T_B ← T_B \ {(F, S)}
      fi
      ℱ_S ← ℱ_S ∪ {(F, H) | H ∈ D_B^U(F) \ {S}}
      T_B ← T_B ∪ {(F, S)}
    end
  else
    // handle unsubscription
    for the filter F ∈ 𝔘 do
      if S ∈ N_B then
        T_B ← T_B \ C_B^L(F, S)
      else
        T_B ← T_B \ {(F, S)}
      fi
      ℱ_U ← ℱ_U ∪ {(F, H) | H ∈ D_B^U(F) \ {S}}
      P ← P ∪ (C_B^L(F) \ C_B^I(F))
    end

    // handle uncovered subscriptions
    P ← P ∪ {(F, S) | F ∈ 𝔖}
    T_B ← T_B ∪ {(F, S) | F ∈ 𝔖}
    forall (F, U) ∈ P do
      k ← #{H | (G, H) ∈ P ∧ G ≡ F}
      P ← P \ {(G, H) | (G, H) ∈ P ∧ G ≡ F}

      𝒜 ← D_B^RU(F) \ {S}
      if k = 1
        𝒜 ← 𝒜 \ {U}
      fi
      F_S ← F_S ∪ {(F, H) | H ∈ 𝒜}
    end
  fi

  // generate triples
  M ← generate(F_S, F_U)
  return ℳ

end
```

Figure 3.22: Routing based on Filter Covering

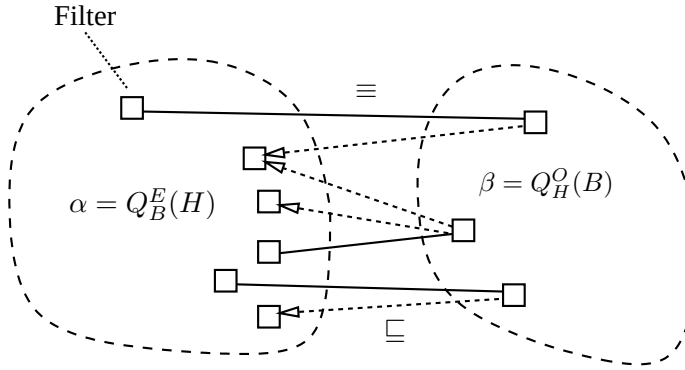



Figure 3.23: Relation among $\alpha$ and $\beta$ for covering-based routing.

PROOF SKETCH: We prove that the `administer` procedure shown in Fig. 3.22 is legal by showing that the properties 1 to 3 of Def. 3.4 are satisfied.
PROOF:
⟨1⟩1. Property 1 is satisfied.
PROVE: `administer` returns.
PROOF: It is easy to see that the procedure returns. □
⟨1⟩2. Property 2 is satisfied.
⟨2⟩1. Property 2a is satisfied.
PROOF: As *unsub* and *sub* messages are handled in the same way, the same proof as in Lemma 3.13 holds. □
⟨2⟩2. Property 2b is satisfied.
ASSUME: $S \in N_B$
PROVE: $\nu_B^{id(m)}(\{S\}) \supseteq \nu_S^{id(m)}(L_S \cup N_S \setminus \{B\})$
PROOF: by Lemmas 3.23 and 3.24. □
⟨1⟩3. Property 3 is satisfied.
ASSUME: $\nu_H^{lid_{B,H}(m)}(\{B\}) \not\supseteq \nu_B^{id(m)}(\{S\})$
PROVE: $\exists(B_H, \mathcal{S}_H, \mathcal{U}_H) \in \mathcal{M}$
PROOF: by Lemmas 3.23 and 3.24. □
⟨1⟩4. Q.E.D.
PROOF: follows from step ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3. □

**Theorem 3.8 (Correctness of Routing based on Filter Covering)** *The routing framework with the* `administer` *implementation shown in Figure 3.22 (covering-based routing) and the empty legal initial routing configuration satisfy Def. 2.1.*

PROOF: by Lemmas 3.14 and 3.25, and Theorem 3.4. □

### 3.5.5 Routing based on Filter Merging

This section describes *merging-based* routing, a routing algorithm that is based on filter merging [73] and that can be implemented on top of covering-based

routing. First, the basic ideas underlying merging-based routing are presented which can be used to implement many variants of merging-based routing. After that, the main questions underlying merging-based routing are described. Finally, a concrete merging-based routing algorithm is presented that is also the basis for the implementation described in Chapter 5.

**Basic Idea of Merging-Based Routing.** The basic idea of merging-based routing is rather simple. In contrast to covering-based routing, merging-based routing does not merely rely on the filters which have been issued by the clients. Instead, if a merging-based routing algorithm is applied, a broker can merge the filters of existing routing entries and forward this merger to a subset of its neighbors.

Formally, a filter $F$ is a *merger of (or covers) a set of filters* $\{F_1, \ldots, F_N\}$, denoted by $F \sqsupseteq \{F_1, \ldots, F_N\}$, iff $N(F) \supseteq \big( \cup_i N(F_i) \big)$. $F$ is a *perfect* merger if the equality holds and an *imperfect* merger, otherwise. A merging-based routing algorithm which only generates perfect mergers and additionally ensures that the generated mergers are forwarded in a way such that only interesting notifications are delivered to a broker is called *perfect*, otherwise the algorithm is *imperfect*.

**Main Questions underlying Merging-Based Routing.** The idea of routing based on filter merging presented above puts up a framework for possible algorithms. In order to implement a concrete merging-based routing algorithm, a number of questions must be answered. The main questions underlying merging-based routing are the following:

- Which routing entries are merged and how is the merger derived from the constituting filters?
- To which neighbors is a new merger forwarded?
- In which cases is a merger canceled?
- To which neighbors is a canceled merger forwarded?
- How are the mergers administered?

To which neighbors a merger should be forwarded either as subscription or as unsubscription depends on the routing entries from which the merger was generated. For example, assume that we have generated a perfect merger $F$ from two routing entries $(F_1, H_1)$ and $(F_2, H_2)$ where $H_1 \neq H_2$. If we are interested in a perfect merging-based routing algorithm only, the merger should be prefect and should at most be forwarded to all neighbors except $H_1$ and $H_2$ because otherwise non interesting events might be forwarded from either $H_1$ that match $F_1$ or from $H_2$ that match $F_2$. More generally, a perfect merger $F$ generated from a set of routing entries $\{(F_1, H_1), \ldots, (F_n, H_n)\}$ can be forwarded to all neighbors while preserving perfect merging-based routing if any notification $n$

that is matched by the perfect merger $F$, is matched by at least one filter $F_i$ of a local client or by two distinct filters $F_i$ and $F_j$ with $H_i \neq H_j$.

For the other questions depicted above similar considerations can be carried out. Instead of discussing them in full detail, a concrete merging algorithm is presented in the following that also serves as the basis for the implementation described in Chapter 5.

### Concrete Merging-Based Routing Algorithm

Subsequently, a concrete merging-based routing algorithm is presented that is perfect and which is implemented on top of covering-based routing. The algorithm allows every broker solely to merge routing entries that refer to the *same* destination. This keeps the algorithm simple enough to be applied in a dynamic publish/subscribe system.

**Generation and Forwarding of a New Merger.** In order to enable filter merging, a broker $B$ can replace a set of routing entries $\{(F_1, D), \ldots, (F_n, D)\}$ with the *same* destination $D$ by a single merged entry $(F, D)$ if $F$ is a perfect merger of $\{F_1, \ldots, F_n\}$. The merged routing entries are removed from the routing table and $(F, D)$ is added to the routing table instead. As $F$ is a perfect merger this does not affect the set of notifications that $B$ is forwarding to $D$, i.e., $\nu_B(\{D\})$. The algorithm also stores what filters are associated with a merger for the case the merger is canceled. As several routing entries are replaced by a single one the size of the routing table of the broker that introduced the new merger is reduced. This results in faster matching. The merger is then forwarded exactly in the same way as a normal subscription that would have been received from $D$. Every time a new subscription is received, the broker tries to generate a new merger or to add the new subscription to an existing merger.

**Cancellation of a Merger.** There are three cases in which a merger has to be canceled:

1. If a subscription arrives from $D$ that covers the whole merger, the merger is simply removed from the routing table.

2. If an unsubscription for a part of a merger is received from $D$, the merger is removed and the merger is forwarded as unsubscription. The other entries that constitute the merger are added to the routing table again and are forwarded as subscriptions. This is done in a single message in order to guarantee that the change to the routing table of the receiving broker is atomic.

3. If a subscription arrives from $D$ that covers one or more of the filters that constitute a merger, the merger is removed and forwarded as unsubscription. The routing entries whose filters are not covered by the new

subscription are added to the routing table again and are forwarded as subscriptions. Again, this is done in a single message.

In the last two cases, the algorithm tries to generate a new merger from the remaining parts of the canceled merger.

**Examples.** In Figure 3.24, $B_1$ receives a new subscription $G$ from a local client $S$. $B_1$ merges $G$ with the filter $F$ of an existing routing entry $(F, S)$ to a broader filter $H$ that covers $F$ and $G$. Hence, $B_1$ removes $(F, S)$ from and inserts $(H, S)$ into its routing table. Moreover, it records the information that $H$ consists of $F$ and $G$ in an additional data structure. Finally, $B_1$ forwards the new merger $H$ to its neighbors $B_2$ and $B_3$.

In the second example (see Fig. 3.25), $B_1$ receives an unsubscription $F$ from a local client $S$ where $F$ is part of a merger $H$. Here, $B_1$ removes $(H, S)$ and adds $(G, S)$ instead. $B_1$ forwards an unsubscription for $H$ along with a subscription for $G$ to both its neighbors $B_2$ and $B_3$. The fact that $k = 1$ has no effect here, since $S = U$.

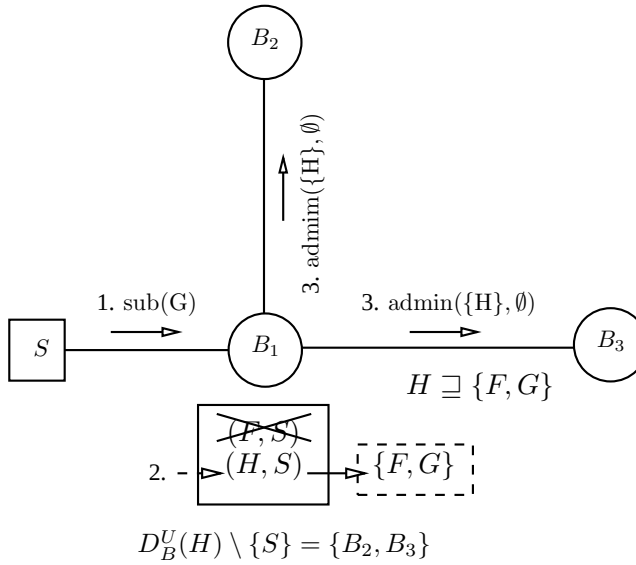



Figure 3.24: Forwarding a new merger.

## 3.6 Routing with Advertisements

In this section it is shown how advertisements can be integrated into the routing framework. *Advertisements* are filters that are issued by clients to indicate their intention to publish certain kinds of notifications and only notifications matching an active advertisement of the respective producer should be delivered to

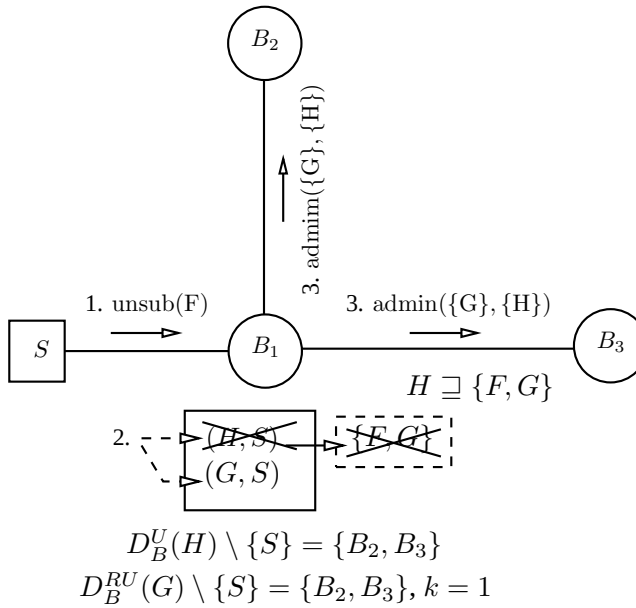



Figure 3.25: Forwarding a canceled merger.

interested consumers (cf. Sect. 2.6). In the context of content-based routing, advertisements can be used as an additional mechanism for further optimization because it is sufficient to forward a subscription only into those subnets where matching events can be produced, i.e., where a client has issued an advertisement that overlaps with the given subscription [17]. The only assumption that underlies the use of advertisements is that it can be detected whether or not a subscription and an advertisement overlap. Formally, two filters $F_1$ and $F_2$ are *overlapping* iff $N(F_1) \cap N(F_2) \neq \emptyset$.

**Basic Idea of Routing with Advertisements.** If advertisements are utilized for optimized routing, each broker manages two routing tables, the known *subscription-based* routing table $T_B^S$ (formerly $T_B$) and an additional *advertisement-based* routing table $T_B^A$. While the former is used (as described before) to route notifications from producers to interested consumers, the latter is used to route (un)subscriptions from interested consumers to producers. This means that forwarding of notifications and (un)subscriptions is restricted by using the routing tables:

- A *notification* $n$ is only forwarded to a destination $D$ if there is a routing entry $(D, F) \in T_B^S$ with $n \in N(F)$.
- An *(un)subscription* $F$ is only forwarded to a neighbor $H$ if there is a routing entry $(H, G) \in T_B^A$ with $N(F) \cap N(G) \neq \emptyset$.

In a dynamic publish/subscribe system both routing tables need to be updated as clients issue or revoke subscriptions and advertisements. For this purpose all routing algorithms described in Sect. 3.5 except flooding can be used but now they must be applied on two levels that are partially dependent. The first level is responsible for updating the subscription-based routing table, while the second level keeps the advertisement-based routing table up to date. For each of the levels one of the presented routing algorithms can be chosen independently of the other. For example, simple routing can be used for the former, while at the same time covering-based routing is applied to the latter.

**Processing of Subscriptions and Unsubscriptions.** To be more precise regarding the dependency among the two levels, the notification forwarding remains unchanged, while the (un)subscription forwarding of the routing algorithm is changed: (un)subscriptions are only forwarded to those neighbors for which there is an overlapping advertisement. For example, suppose that the routing algorithm that updates the subscription routing table decides that a subscription $F$ should be forwarded to the neighbors $H_1$ and $H_2$. If there is only an advertisement from $H_1$ in $T_B^A$ that overlaps with $F$ and none for $H_2$, $F$ is only forwarded to $H_1$.

**Processing of Advertisements and Unadvertisement.** If a broker receives an (un)advertisement it is simply forwarded according to the used routing algorithms. Additionally, subscriptions are forwarded and existing routing entries are dropped in reaction to new and canceled advertisements, respectively:

- If a broker receives a new advertisement from a neighbor $H$, it (potentially) forwards all subscriptions to $H$ which

  1. come from a destination $D \neq H$,
  2. overlap with $F$, and
  3. for which $F$ is the only advertisements from $H$ that overlaps.

- If a broker receives an unadvertisement $F$ from a neighbor $H$, it drops all routing entries of all neighbors $U \neq H$ for whose filter there is no other advertisement from any other destination $D \neq U$ that overlaps.

Now, the main disadvantage of routing with advertisements becomes clear: notifications which match only an advertisement that has been recently issued by a producer, may not be delivered to all interested consumers. This was also the main reason to apply a weakened liveness condition (see Sect. 2.6) if advertisements are used. Indeed, delivery is only guaranteed after the new advertisement has been propagated and the subscriptions that are forwarded in turn have also been propagated. Both processes are guaranteed to terminate after a finite time. Hence, the proposed solution satisfies Def. 2.4.

**Integration with the Routing Framework.** From the explanations above it is easy to imagine how advertisements can be integrated into the routing framework:

- Two new messages $adv(\mathcal{S}, \mathcal{U})$ and $unadv(\mathcal{S}, \mathcal{U})$ are introduced that correspond to new and canceled advertisements issued by clients.
- There are now two categories of *admin* messages, $admin_S$ and $admin_A$ to distinguish among subscription and advertisement-related *admin* messages.
- There are now two `administer` procedures `administerA` and `administerS` which are called according to the received *sub*,*unsub* and *adv*, *unadv* messages, respectively.
- From each of the triples from the set of triples returned by `administerS` those filters are deleted for which there is no overlapping advertisement issued by the intended receiver.
- In reaction to new and canceled advertisements, "activated" subscriptions are forwarded and "unserviceable" routing entries in the subscription-based routing table are dropped accordingly.

In the way depicted above, advertisements were integrated in the implementation of the REBECA notification service (see Chapter 5).

## 3.7 Ensuring Self-Stabilization

Up to this section, it was assumed that no faults can occur. Here, it is shown how fault-tolerance in the sense of self-stabilization can be achieved. In a system where arbitrary transient faults (like message losses, memory perturbations) can occur, the framework presented in the previous sections will obviously not be a correct implementation of a publish/subscribe system with respect to Def. 2.1. However, it is relatively easy to augment the framework so that it satisfies the specification of self-stabilizing publish/subscribe systems (Definition 2.3) in the presence of arbitrary transient faults. This is done by the concept of *subscription leasing*. Interestingly, this approach does not only allow the publish/subscribe system to recover from internal faults but also from certain external faults related to the clients. For example, if a client crashes, its subscriptions are automatically removed after their leases have expired.

### 3.7.1 Fault Assumption

In the following, arbitrary faults are assumed to possibly happen within the publish/subscribe system as long as they are transient (faults do not affect the clients). This includes for example arbitrary message corruption, deletion or insertion, arbitrary sequences of process crashes and subsequent recoveries, and

arbitrary perturbations of the data structures of processes. Obviously, it must also be assumed that the number of occurrences of these faults is finite. This behavior is modeled by assuming that the system begins its execution in an arbitrary state. Then the concept of self-stabilization is applied to achieve stabilizing publish/subscribe semantics.

### 3.7.2 Routing Table Entry Leasing

Self-stabilization is achieved by applying the well-known concept of leasing: subscriptions of clients are only *leased*, i.e., to maintain a subscription, client $X$ must regularly renew the lease for its subscription to a filter $F$ by "resubscribing" to $F$. This is done in the same way as subscribing, i.e., $X$ sends a $sub(F)$ message to its broker. A prerequisite for making this approach work is that the routing algorithms ensure that subsequent resubscriptions are idempotent. This means that they should cause no change to the routing table but should cause *admin* messages to be sent out properly. The algorithms presented in Sect. 3.5 exhibit this behavior or need only small changes to do so. Therefore, brokers within the publish/subscribe system continue to run the routing framework as usual and the routing table is updated by the `administer` routine. Brokers also control the validity of routing table entries in the following way: within the routing table, each broker implicitly stores the *lease expiration time* for each individual entry. Lease renewals, i.e., idempotent insertions of routing entries update this timestamp. They can be caused by *sub* or *admin* message. If filter merging is performed the resulting additional routing entries must be renewed on a regular basis, too. Note that lease times can be adaptive but have some known *maximal lease time*.

### 3.7.3 Lease Expiry and Timing Conditions

Periodically, brokers validate their routing table and discard entries whose lease has expired. This must be done atomically and can be achieved by running a concurrent task which operates in mutual exclusion to the routing framework. This process requires some notion of global time, i.e., some synchrony assumptions about processes and communication. This is because clients and brokers must be able to set and compare lease timestamps approximately within the same time frame and communication channels must be timely enough to renew the lease for a routing table entry before it expires. For the following analysis we will assume a global clock and refer timing information to this clock. However, this clock is only a fictitious device facilitating analysis.

The two main parameters in an analysis of the timing conditions for stabilization are the *leasing period* $\pi$ , i.e., the amount of time for which leases are granted, and the *refresh period* $\rho$ , i.e, the amount of time after which clients initiate a renewal of each individual lease. Other important parameters are the *link delay* $\delta$ , i.e., the amount of time taken for a link (process and communication channel) to process and forward a message. In the following, it is assumed

that $\delta$ is between $\delta_{min}$ and $\delta_{max}$. Moreover, let $d$ be the *network diameter*, i.e., the length of the longest path in the broker network.

A critical issue here is that the timing assumptions must be conservative enough so that clients are able to renew their leases everywhere in the network before they expire. So how large must $\pi$ be with respect to $\rho$ in this case? To answer this question, consider two brokers $B$ and $B'$ connected by the longest path in the network. Assume a local client $X$ of $B$ leases a routing table entry of $B$ at time $t_0$ and renews this lease at time $t_1 = t_0 + \rho$. In the worst case, $X$'s lease causes other leases to be granted all along the path to broker $B'$ (if for example simple filter-based routing is used). Considering the best and worst cases of the link delay, the first lease reaches $B'$ at time

$$a_0 = t_0 + d \cdot \delta_{min} \tag{3.19}$$

in the best case and the lease renewal reaches $B'$ at time

$$a_1 = t_1 + d \cdot \delta_{max} \tag{3.20}$$

in the worst case. In the worst case, i.e., if $X$ refreshes its leases after $\rho$ time and if network delays are unfavorable, two lease renewals will arrive at $B'$ within at most $a_1 - a_0$ time. Hence, the leasing period $\pi$ must be greater than $a_1 - a_0$:

$$\pi > \rho + d(\delta_{max} - \delta_{min}). \tag{3.21}$$

If clocks are not perfectly synchronized, an additional value $\Delta$ must be added to $\pi$, where $\Delta$ is a bound on the differences of clock readings throughout the network.

### 3.7.4 Stabilization Proof

The timing analysis in the previous section is the basis for the proof that the augmented framework is self-stabilizing. First, it is argued that without faults the augmented framework satisfies Def. 2.1 (see Lemma 3.26). After, that it is shown that starting from an arbitrary state, the system will eventually reach a correct state within a finite time (see Lemma 3.27). Together this gives Theorem 3.9 stating the self-stabilizing property.

**Lemma 3.26 (Correctness)** *If $\rho + d(\delta_{max} - \delta_{min}) < \pi$, the system is in a weakly valid routing configuration, and all messages in the channels have been initiated by some client, then the system satisfies Def. 2.1.*

PROOF SKETCH: The timing condition and the condition on the channels guarantee that leases get renewed in time everywhere. This means that we can "ignore" the periodic leasing activity because it is transparent from a functional point of view. The proof then follows from the original "time-free" correctness proof in Theorem 3.4.

**Lemma 3.27 (Convergence)** *Starting from an arbitrary state, eventually the system will reach a weakly valid routing configuration where all messages in the channels have been initiated by some client.*

PROOF SKETCH: The idea of the proof is rather simple and independent of real-time timing assumptions: spurious messages will be eventually delivered to their destination processes and may cause spurious routing table entries. These (and other) spurious entries will eventually be removed from the routing table by the lease expiry mechanism. Note that spurious entries in the local configuration temporarily affect the safety property while spurious entries in the in the remote configuration solely degrade efficiency. In any case, eventually, routing tables contain only entries which can be traced back to clients requests, and the channels will only contain messages initiated by some client. Of course, the `administer` should not introduce any additional state.

**Theorem 3.9** *If the timing conditions satisfy* $\rho + d(\delta_{max} - \delta_{min}) < \pi$*, the routing framework augmented with the lease mechanism satisfies Def. 2.3 in the presence of arbitrary transient faults.*

PROOF SKETCH: The convergence property of Lemma 3.27 implies that the system recovers to a weakly valid routing configuration once the transient faults have stopped. The correctness property of Lemma 3.26 implies that from this point onwards the safety and the liveness properties of the original Def. 2.1 are satisfied forever. So overall the system satisfies Def. 2.3 in the presence of arbitrary transient faults.

It is worth noting that a (temporary) violation of the timing assumptions can also be considered as a transient fault. So overall stabilization can be guaranteed even if the timing bounds only hold *eventually*.

### 3.7.5 Stabilization Time

The *stabilization time* $\Delta T$ , i.e., the time it takes for the system to reach a legal state starting from an arbitrary state, depends on the value of $\pi$. In the worst case, it must be waited until all garbage messages reach their destination nodes, i.e. $\delta_{max}$. Since garbage messages can contaminate the network too, a delay of $d(\delta_{max} - \delta_{min})$ time for the final garbage messages to take effect must be assumed. Finally, after $\pi$ time, the effects will be removed everywhere. Overall, the stabilization time sums up to

$$\Delta T = \delta_{max} + d(\delta_{max} - \delta_{min}) + \pi. \tag{3.22}$$

There is an obvious tradeoff between the values of $\pi$ and $\rho$. To have low message overhead, $\rho$ (the refresh period) should be as large as possible. However, this implies a large value of $\pi$, but $\pi$ should be as small as possible to facilitate fast recovery. Choosing the parameters in practice is difficult and out of the scope of this thesis.

## 3.8 Related Work

In this section the work presented in this chapter is related to other approaches dealing with content-based routing.

### 3.8.1 Siena

Siena [17, 22] is a notification service that uses covering-based routing. Two variants of covering-based routing are presented by the authors, a hierarchical and a peer-based variant. The peer-based variant is similar to the one presented here. The hierarchical variant propagates subscriptions only from each broker to its parent broker. A notification is always forwarded to the parent broker and selectively forwarded to subordinate brokers. Other routing algorithms than covering-based routing are not considered and fault tolerance is not discussed.

### 3.8.2 Gryphon

Gryphon [8, 85] uses simple routing but proposes to exploit multicast for inter-broker communication in order to reduce the used network bandwidth. Several multicast strategies are discussed and compared to a fictitious ideal multicast strategy where for each permutation of consumers a multicast group exists. Neither other routing algorithms than simple routing nor fault tolerance aspects are discussed. Nevertheless, the ideas developed in Gryphon are well suited to be integrated with the algorithms presented here.

### 3.8.3 Hermes

Recently, Pietzuch and Bacon presented Hermes [90, 91], a notification service that implements content-based notification delivery on top of a peer-to-peer overlay network. Before this, only two subject-based notification services, Scribe [97] and Bayeux [121], have been built in this way. In all approaches rendezvous nodes are used. In Hermes, each rendezvous node is responsible for a certain event-type, while in the subject-based approaches they refer to specific topics. Similar to the approach presented here, in Hermes subscriptions are only leased and need to be periodically refreshed. This results in the state of the broker being "soft" and allows certain faults to be tolerated.

## 3.9 Discussion

In this chapter the foundations of content-based routing have been discussed in detail. Starting with the fault-free scenario, fault-tolerance in the sense of self-stabilization is added later on. First, a formalization of routing configurations has been proposed that builds upon filter-based routing tables. The routing table of a broker determines to what neighbors it forwards a notification that it processes. Furthermore, it has been shown that valid routing configurations are sufficient and necessary to ensure a correct static publish/subscribe system. The sufficiency has also been proved for weakly valid routing configurations and dynamic public subscribe systems.

Based upon the formalization of routing configurations, a framework for content-based routing algorithms has been described. The framework hard-

wires the notification forwarding based on routing tables and allows an abstract function to be customized to yield a concrete routing algorithm. This approach is flexible enough to realize a variety of routing algorithms. A universal sufficiency criterion has been elaborated allowing to identify legal instances of this abstract function that lead to a correct publish/subscribe system. The criterion consists of a set of properties that must be satisfied. The meaning of the individual properties has been described giving new insights into the requirements a content-based routing algorithm has to fulfill.

As examples of framework instances, flooding, simple routing, identity-based routing, covering-based routing, and merging-based routing have been discussed. Treating these algorithms as instances of the framework offered new insights into their behavior. Routing algorithms are explicitly given and it is argued for their correctness in a more structured way than previous work did. It has also been discussed how advertisements can be integrated into the routing framework.

In the last section of this chapter, it has been shown how the routing framework can be made self-stabilizing by using subscription leasing: Subscriptions of clients are not permanent but need to be renewed on a periodical basis. This allows a publish/subscribe system to recover from arbitrary transient faults within a finite time.

# Chapter 4

# Models and Routing Optimizations

## Contents

## 4.1 Introduction

The content-based routing algorithms presented in Chapter 3 do not depend on the actual underlying data/filter model. Instead they are based on abstract tests for overlapping, identity, and covering, and on filter merging. Hence, the routing algorithms only assume that these tests among filters can be computed and that merging can be carried out. In this chapter it is described how these tests and filter merging can be efficiently supported. The emphasis is on structured records that are based on name/value pairs since this is the most prominent model for content-based publish/subscribe systems. In addition, routing optimizations for semistructured records and objects are discussed, too.

This Chapter is structured as follows: First, some content-based data/filter models are described in Section 4.2 including tuples, records (which are further divided into structured and semistructured records), and objects. After that introductory discussion all models except tuples are discussed in more detail. In Section 4.3 a model based on structured records is presented which relies on conjunctive filters consisting of attribute filters. Subsequently, a powerful and flexible filtering framework is discussed that allows new data types and filtering predicates to be introduced easily instead of relying on a fixed set of predefined types and predicates (e.g., used by SIENA and JEDI, etc.). As examples and to show the feasibility of the approach, covering implications and merging rules for attribute filters are described for a set of data types with typical operators. After that, covering, identity, overlapping, and merging of conjunctive filters are discussed and new algorithms for identity tests and covering tests as well as for detecting merging candidates are presented. These algorithms are necessary to realize the routing algorithms of Chapter 3. Section 4.4 presents a model for semistructured records which is a generalization of structured record model and essentially builds upon XML documents and XPath expressions. It is shown how routing optimizations can be applied to semistructured records. Section 4.5 presents how matching can be carried out, and how routing optimizations can be applied if notifications and filters are objects.

## 4.2 Content-Based Data/Filter Models

This section discusses some prominent content-based data models in conjunction with corresponding filter models. Informally, a *data model* defines how the content of notifications is structured while a *filter model* defines how subscriptions can be specified, i.e., how notifications can be selected by applying filters that evaluate predicates over the content of notifications. In consequence, the filter model always depends on the underlying data model and there can be more than one filter model for a given data model. The data/filter model has to be chosen carefully because it has a large impact on the expressiveness and the scalability of a content-based notification service.

Several prominent data models are discussed including tuples, records, and

objects where records are further divided into structured and semistructured records.

### 4.2.1 Tuples

In tuple-oriented models a notification is a *tuple*, i.e., an *ordered* set of *attributes*. All approaches using tuples deploy some sort of templates as subscription mechanisms. Similar to a query-by-example mask, a *template* specifies matching notifications by a partial tuple which can contain wildcards. The attributes in the notification are matched to the attributes in the template according to their position. For example, the notification (*StockQuote*, *"Foo Inc."*, 45.7) is matched by the subscription template (*StockQuote*, *"Foo Inc."*, $*$). "Matching by position" is inflexible because attributes cannot be optional. Tuples in conjunction with templates were first proposed by Gelernter in work on Linda Tuplespaces [48] which use typed attributes. The original version of Linda, however, did not support a subscription mechanism but newer approaches based on Tuplespaces, e.g., JavaSpaces [110], do. Also some notification services are built upon tuples: JEDI models a notification as a tuple of strings [28] in which the first string corresponds to the notification name, while the others are normal attributes. They support the equality and the prefix operator for matching. Bates et al. [10] define notifications as instances of classes. An instance consists of a tuple of typed attributes derived from a class definition. Here, a template either specifies the exact value of an attribute or it does not care about the value. Concluding, tuples with templates provide a simple model that is not flexible enough because attributes of notifications and templates are matched to each other according to their position. This disadvantage is diminished by record-oriented models which use "matching by attribute names". However, matching by position is more efficient.

### 4.2.2 Records

In a record-oriented model a notification consists of a *named* set of attributes. Record-oriented models can be divided in two categories which are structured records and semistructured records, respectively. Roughly speaking, the models can be distinguished by the fact that in structured records attribute names are unique, while in the semistructured models several attributes with the same name can exist. In the following both categories of record-oriented models are discussed.

#### Structured Records

Many systems model notifications similar to structured records consisting of a set of name/value pairs called attributes. Examples are Siena [17], Gryphon [1, 8], Rebeca [40], JMS [111], and the Corba Notification Service [81]. In this model filters address attributes by their unique name and impose constraints on

the values of the respective attributes. In most models a constraint is assumed to evaluate to *false* if the addressed attribute is not contained in the notification. Therefore, each constraint implicitly defines an existential quantifier over the notification. Besides *flat* records in which values are atomic types, *hierarchical* records in which attributes may be nested can also be supported easily by using a dotted naming scheme (e.g., *Position*.x).

Some systems (e.g., SIENA) restrict constraints to depend on a single attribute (e.g., $\{x = 1\}$). This class of constraints is called *attribute filters*. Other systems (e.g., ELVIN) allow constraints to evaluate multiple attributes which are combined by operators (e.g., $\{x + y = 5\}$). In general, multiple constraints can be combined to form filters by boolean operators (e.g., $\{y < 3 \wedge x = 4\}$). SIENA and REBECA restrict filters to be conjunctions of attribute filters. On one hand, this restriction reduces the expressiveness of the filter model, but on the other hand it enables routing optimizations like covering to be applied efficiently. The limitation is also not as serious as it seems at first. For example, a filter that is defined by an arbitrary boolean expression can always be converted to and treated as a collection of conjunctive filters.

Although records and tuples seem to be quite similar at first glance, records are clearly more powerful because they allow for optional attributes in the notifications, avoid unnecessary "don't care" constraints in the templates, and enable the easy addition of new attributes without affecting existing filters. How to support routing optimizations for structured records is discussed in Section 4.3.

**Semistructured Records**

According to Bunemann [15] semistructured data can be characterized as some kind of graph-like or tree-like structure that is often called *self-describing* because the schema of the data is contained in the data itself. At the moment, the most prominent semistructured data model is XML [118]. Similar to structured records, a semistructured record is a set of nested attributes, but in contrast to structured records, in semistructured records sibling attributes can have the same name. In consequence, a single attribute can no longer be uniquely addressed by its name alone. Instead names (e.g., *car.price*), which are usually called *paths* in this context, select sets of attributes. Therefore, filtering strategies assuming that a single attribute is addressed by a given name cannot directly be used in this scenario. One way to approach this problem is to use path expressions (e.g., XPath [119]) which select a set of attributes and impose constraints on the selected attributes.

Clearly, the semistructured model is more powerful than structured records, but work in this area related to content-based routing is still in its early stages. Lately, using XML and path expressions has gained increased attention. Nguyen et al. [79] and Chen et al. [24] described approaches for XML continuous queries. Altinel and Franklin [3] presented an efficient method for filtering XML documents using XPath expressions. All this work concentrates on efficient local matching and does not deal with distributed content-based routing. First ideas

on how to support routing optimizations like covering and merging for semistructured records was presented by Mühl and Fiege [75]. These ideas are discussed in Section 4.4.

### 4.2.3 Objects

Using objects as notifications is widely used in GUIs (e.g., JAVA AWT [105]) and visual components (e.g, JavaBeans [107]). The JAVA Distributed Event Specification [109] which is built upon JAVA RMI also uses Objects. The difference of this approach to a notification service is that consumers must directly register with the source of an event. Eugster and Guerraoui [35] present an approach to use structural reflection for content-based filtering of notifications. The object-oriented model is most flexible and powerful, but routing optimizations like covering and merging are difficult to achieve if filters can contain arbitrary code. Mühl and Fiege [75] have presented first ideas on how to support routing optimizations like covering and merging for objects. These ideas are discussed in Section 4.5.

## 4.3 Structured Records

In this section structured records are discussed in detail. A flexible and extensible filtering framework is described that builds upon the model and which is also the basis for the implementation (see Chapter 5). Finally, it is depicted how the routing optimizations can be supported efficiently. This includes the presentation of theoretical results and new algorithms.

### 4.3.1 Data Model

#### Notifications

A *notification* is a message that contains information about an event that has occurred. Formally, a notification $n$ is a nonempty set of *attributes* $\{a_1, \ldots, a_n\}$ where each $a_i$ is a *name value pair* $(n_i, V_i)$ with name $n_i$ and value $v_i$. It is assumed that names are unique, i.e., $i \neq j \Rightarrow n_i \neq n_j$, and that there exists a function that uniquely maps each $n_i$ to a type $T_j$ that is the type of the corresponding value $v_i$.

In the following it is distinguished between *simple values* that are a single element of the domain of $T_j$, i.e., $v_i \in dom(T_j)$, and *multi values* that are a finite subset of the domain, i.e., $v_i \subseteq dom(T_j)$. An example of a simple notification is $\{(type, StockQuote), (name, \text{“}Infineon\text{”}), (price, 45.0)\}$.

### 4.3.2 Filter Model

#### Filters

A *filter* $F$ is a stateless boolean function that is applied to a notification, i.e., $F(n) \rightarrow \{true, false\}$. A notification *matches* $F$ if $F(n)$ evaluates to *true*. Consequently, the set of matching notifications $N(F)$ is defined as $\{n \mid F(n) = true\}$. Two filters $F_1$ and $F_2$ are *identical*, written $F_1 \equiv F_2$, if and only if $N(F_1) = N(F_2)$. Moreover, they are *overlapping*, denoted by $F_1 \sqcap F_2$, iff $N(F_1) \cap N(F_2) \neq \emptyset$. Otherwise they are *disjoint*, denoted by $F_1 \not\sqcap F_2$.

A filter is usually given as a boolean expression that consists of predicates that are combined by boolean operators (e.g., *and*, *or*, *not*). A filter consisting of a single atomic predicate is a *simple filter* or *constraint*. Filters that are derived from simple filters by combining them with boolean operators are *compound filters*. A compound filter that is a conjunction of simple filters is called a *conjunctive filter*. In the model proposed filters are restricted to be conjunctive filters. It is sufficient to consider conjunctive filters because a compound filter can always be broken up into a set of conjunctive filters that are interpreted disjunctively and can be handled independently.

#### Attribute Filters

An *attribute filter* is a simple filter that imposes a constraint on the value of a single attribute (e.g., $\{name = \text{“Foo Inc.”}\}$). It is defined as a triple $A_i = (n_i, Op_i, C_i)$ where $n_i$ is an attribute name, $Op_i$ is a test operator and $C_i$ is a set of constants that may be empty. The name $n_i$ determines to which attribute the constraint applies. If the notification does not contain an attribute with name $n_i$ then $A_i$ evaluates to *false*. Therefore, each constraint implicitly defines an existential quantifier over the notification. Otherwise, the operator $Op_i$ is evaluated using the value of the addressed attribute and the specified set of constants $C_i$. It is assumed that the types of operands are compatible with the used operator. The outcome of $A_i$ is defined as the result of $Op_i$ that evaluates either to *true* or *false*. Furthermore, an attribute filter is provided that simply checks whether a given attribute is contained in $n$. For the sake of simplicity the more readable notation $\{price > 10\}$ is used instead of $\{(price, >, \{10\})\}$. In contrast to most other work (e.g., SIENA) constraints that depend on more than one constant are considered in this chapter. This enables more operators and enhances the expressiveness of the filtering model and can be done without affecting scalability.

By $L_A(A_i) \subseteq dom(T_k)$ the set of all values is denoted that cause an attribute filter to match an attribute, i.e., $\{v_i \mid Op_i(v_i, C_i) = true\}$. It is assumed that $L_A(A_i) \neq \emptyset$. An attribute filter $A_1$ *covers* an attribute filter $A_2$, written $A_1 \sqsupseteq A_2$, iff $n_1 = n_2 \wedge L_A(A_1) \supseteq L_A(A_2)$. For example, $\{price > 10\}$ covers $\{price \in [20, 30]\}$. $A_1$ and $A_2$ are *identical*, denoted by $A_1 \equiv A_2$, iff $n_1 = n_2 \wedge L_A(A_1) = L_A(A_2)$. $A_1$ and $A_2$ are *overlapping* iff $n_1 = n_2 \wedge L_A(A_1) \cap L_A(A_2) \neq \emptyset$, denoted by $A_1 \sqcap A_2$. Otherwise they are *disjoint*, denoted by $A_1 \not\sqcap A_2$. For

example, $\{price > 10\}$ and $\{price < 20\}$ are overlapping, while $\{price < 10\}$ and $\{price > 20\}$ are disjoint.

In the described model a *filter* is defined as a conjunction of attribute filters, i.e., $F = A_1 \wedge \ldots \wedge A_n$. To enable efficient evaluation of routing optimizations like covering and merging at most one attribute filter for each attribute is allowed (see Sects. 4.3.5 to 4.3.7). A notification *n matches a filter* $F$ iff it satisfies all attribute filters of $F$. Moreover, a filter with an empty set of attribute filters matches any notification. An example for a conjunctive filter consisting of attribute filters is $\{(type = StockQuote), (name = \text{“Foo Inc.”}), (price \notin [30, 40])\}$.

The limitation to at most one attribute filter for each attribute is not as serious as it seems at first glance because the proposed model provides complex data types as attribute values and an extensible set of constraints that can be imposed (see Sect. 4.3.3). Moreover, it is often possible to merge several conjunctive constraints imposed on a single attribute into a single constraint on the same attribute. Especially suited for this kind of merging are constraints which are either contradicting (if they are conjuncted) or can be replaced by a single constraint of the same type. Such types of constraints and their corresponding attribute filters are called *conjunction-complete*. For example, interval constraints and constraints testing whether a point is in a given rectangle in a two-dimensional plane are conjunction-complete. As an example, $\{x \in [3, 7] \ \wedge \ x \in [5, 8]\}$ can be substituted by $\{x \in [5, 7]\}$. If a constraint type is not conjunction-complete it is often possible to substitute a set of such constraints by a single constraint of a more general type. For example, a set of ordering constraints defined on a totally ordered set (e.g., integer numbers) are either contradictory or can be replaced by a single interval constraint. As an example, $\{x \geq 3 \ \wedge \ x \leq 5\}$ can be merged to $\{x \in [3, 5]\}$.

#### Subscriptions and Advertisements

*Subscriptions* and *advertisements* are simply filters that are issued by consumers and producers of notifications, respectively. There is no difference in their model, and hence, subscriptions and advertisements are the exact dual of each other. This is in contrast to Siena where subscriptions and advertisements are not exactly complementary raising a number of problems (see related work section).

### 4.3.3 Generic Constraints and Types

Former work dealing with content-based notification selection mechanisms often tightly integrated the constraints that can be put on values and the types of values supported with the matching and routing algorithms [1, 8]. An exception is Siena where matching and routing algorithms are separated from constraints. However, they only support a fixed set of constraints on some predefined primitive types.

This thesis proposes to use a collection of abstract attribute filter classes. Each of these classes offers a generic implementation of the methods needed

by the matching and routing algorithms (e.g., the covering and matching test) and imposes a certain type of constraint on an attribute that can be used with values of all types that implement the operators needed. The appropriate implementation of the operators are called by the constraint class at runtime using polymorphism. This enables new constraints and types to be defined and supported without requiring changes to the routing and matching algorithms. Note that although an object-oriented approach is suggested it is not mandatory to use it.

For example, a constraint class has been implemented that realizes comparison constraints on totally ordered sets (see Sect. 3.4). This class can be used to impose comparison constraints on all kinds of ordered values (e.g., integer numbers). Consider a type "person" that consists of first and second name, the date of birth, and the place of birth. This type is easily supported by providing implementations for the comparison operators which are called by the constraint class to provide the covering and matching methods using polymorphism.

In the following subsections, some generic attribute constraints are presented that cover a wide range of practically relevant constraints, but more importantly, they illustrate the feasibility of the approach. Of course this collection is not exhaustive but other constraints can be integrated easily. For example, intervals could be used as values. In this case the same operators as for set constraints can be used because intervals are essentially sets. The investigation of a subset of regular expressions seems to be promising, too. Most paragraphs also present a table that gives an overview of covering implication dealing with the discussed type of constraint. The meaning of a single row in the Tables 1 through 6 is: Given $A_1$ and $A_2$ as specified in column 1 and 2, $A_1 \sqsupseteq A_2$ if and only if the condition in column 3 is satisfied. In order to test whether a filter covers another, covering must hold for all attributes as will be shown later.

### General Constraints

Two general constraints are considered that can be imposed on all attributes regardless of the type of their value: *exists*$(n)$ tests whether an attribute with name $n$ is contained in a given notification, i.e., whether $\exists A_i . n_i = n$. The *exists* constraint covers all other constraints that can be imposed on an attribute.

### Constraints on the Type of Notifications

Most work on notification services has a notion of types or classes of notifications. Usually, the type of a notification is specified by a textual string that can be tested for equality and prefix. If a dot notation is used, a type hierarchy with single inheritance can be supported allowing for the automatic propagation of interest in sub-classes [10]. Unfortunately, multiple inheritance cannot be supported by a dotted naming scheme. In contrast to that, a direct support of notification types has a number of advantages. Such an approach can enable multiple inheritance and achieve a better programming language integra-

tion [34]. Moreover, type inclusion tests can be evaluated more efficiently than the corresponding string operation (i.e., whether the string starts with a given prefix) [116].

Consequently, a separate constraint that evaluates to *true* if $n$ is an instance of type $T$ and *false* otherwise, written $n$ *instanceof* $T$, is defined. A constraint $n$ *instanceof* $T_1$ covers a constraint $n$ *instanceof* $T_2$ if and only if $T_1$ is either the same type or a supertype of $T_2$ (see Table 4.1). Naturally, it is assumed that the set of attributes that can be contained in a notification of type $T$ is a superset of the union of all attribute names of all supertypes of $T$.

| $A_1$ | $A_2$ | $A_1 \sqsupseteq A_2$ iff |
|---|---|---|
| $n$ *instanceof* $T_1$ | $n$ *instanceof* $T_2$ | $T_1 = T_2 \ \vee \ T_1$ *supertype of* $T_2$ |

Table 4.1: Covering among notification types

### Equality and Inequality Constraints on Simple Values

The simplest constraints that can be imposed on a value are tests for equality and inequality. Covering implications among these tests can always be reduced to a simple comparison of their respective constants (see Table 4.2).

| $A_1$ | $A_2$ | $A_1 \sqsupseteq A_2$ iff |
|---|---|---|
| $x = c_1$ | $x = c_2$ | $c_1 = c_2$ |
| $x \neq c_1$ | $x = c_2$ | $c_1 \neq c_2$ |
| | $x \neq c_2$ | $c_1 = c_2$ |

Table 4.2: Covering among (in)equality constraints on simple values

### Comparison Constraints on Simple Values

Another common class of constraints are comparisons on values for which the domain and the comparison operators define a totally ordered set (e.g., integers with the usual comparison operators). Again, covering among these tests can be reduced to a simple comparison of their respective constants. Table 4.3 depicts covering implications of inequality and greater than, for brevity the other comparison operators are left out.

### Interval Constraints on Simple Values

Interval constraints make it possible to test whether a value $x$ is within a given interval $I$ or not, i.e., $x \in I$ and $x \notin I$ respectively, where I is a closed interval $[c_1, c_2]$ with $c_1 \leq c_2$. Here, computing coverages involves two comparisons (see Table 4.4).

| $A_1$ | $A_2$ | $A_1 \sqsupseteq A_2$ iff |
|---|---|---|
| $x \neq c_1$ | $x < c_2$ | $c_1 \geq c_2$ |
| | $x \leq c_2$ | $c_1 > c_2$ |
| | $x = c_2$ | $c_1 \neq c_2$ |
| | $x \geq c_2$ | $c_1 < c_2$ |
| | $x > c_2$ | $c_1 \leq c_2$ |
| $x > c_1$ | $x = c_2$ | $c_1 < c_2$ |
| | $x > c_2$ | $c_1 \leq c_2$ |
| | $x \geq c_2$ | $c_1 < c_2$ |

Table 4.3: Covering among comparison constraint on simple values

| $A_1$ | $A_2$ | $A_1 \sqsupseteq A_2$ iff |
|---|---|---|
| $x \in I_1$ | $x \in I_2$ | $I_1 \supseteq I_2$ |
| $x \notin I_1$ | $x \notin I_2$ | $I_1 \subseteq I_2$ |

Table 4.4: Covering among interval constraints on simple values

### Constraints on Strings

In addition to the comparison operators based on the lexical order, a prefix, a substring, and a postfix operator is defined. Constraints on strings can be used to realize subjects. Computing coverages among them requires a single test (see Table 4.5).

| $A_1$ | $A_2$ | $A_1 \sqsupseteq A_2$ iff |
|---|---|---|
| $s$ *prefix* $S_1$ | $s$ *prefix* $S_2$ | $S_1$ *prefix* $S_2$ |
| $s$ *postfix* $S_1$ | $s$ *postfix* $S_2$ | $S_2$ *postfix* $S_1$ |
| $s$ *substring* $S_1$ | $s$ *substring* $S_2$ | $S_1$ *substring* $S_2$ |

Table 4.5: Covering among constraints on strings

### Set Constraints on Simple Values

Set constraints on simple values test whether a value is or is not a member of a given set. In order to compute a coverage among two of these constraints a single set inclusion test is sufficient (see Table 4.6). Its complexity depends on the characteristics of the underlying set. Set constraints can also be combined with comparison constraints if the domain of the value is a totally ordered set.

| $A_1$ | $A_2$ | $A_1 \sqsupseteq A_2$ iff |
|---|---|---|
| $x \in M_1$ | $x \in M_2$ | $M_1 \supseteq M_2$ |
| $x \notin M_1$ | $x \notin M_2$ | $M_1 \subseteq M_2$ |

Table 4.6: Covering among set constraints on simple values

**Set Constraints on Multi Values**

The idea of multi values is to allow a value to be a set of elements. This enables set-oriented operators which are defined on a multi value $X = \{v_1, \ldots, v_n\}$. For example, the following common operators can be defined:

$$\begin{aligned} X \ \mathit{subset}\ M &\Leftrightarrow X \subseteq M \\ X \ \mathit{superset}\ M &\Leftrightarrow X \supseteq M \\ X \ \mathit{contains}\ a_1 &\Leftrightarrow a_1 \in X \\ X \ \mathit{notcontains}\ a_1 &\Leftrightarrow a_1 \notin X \\ X \ \mathit{disjunct}\ M &\Leftrightarrow X \cap M = \emptyset \\ X \ \mathit{overlaps}\ M &\Leftrightarrow X \cap M \neq \emptyset \end{aligned}$$

To determine covering with respect to these constraints either the evaluation of a set inclusion test or of a set membership test is needed (see Table 4.7).

| $A_1$ | $A_2$ | $A_1 \sqsupseteq A_2$ iff |
|---|---|---|
| $X$ *subset* $M_1$ | $X$ *subset* $M_2$ | $M_1$ *superset* $M_2$ |
| $X$ *contains* $a_1$ | $X$ *superset* $M_2$ | $a_1 \in M_2$ |
| $X$ *superset* $M_1$ | $X$ *superset* $M_2$ | $M_1$ *subset* $M_2$ |
| $X$ *notContains* $a_1$ | $X$ *disjunct* $M_2$ | $a_1 \in M_2$ |
| $X$ *disjunct* $M_1$ | $X$ *disjunct* $M_2$ | $M_1$ *subset* $M_2$ |
| $X$ *overlaps* $M_1$ | $X$ *overlaps* $M_2$ | $M_1$ *superset* $M_2$ |

Table 4.7: Covering among set constraints on multi values

### 4.3.4 Identity of Conjunctive Filters

In the following it is shown how identity of conjunctive filters can be reduced to the respective attribute filters. An identity test among filters is necessary to implement identity-based routing (see Sect. 3.5.3) and covering-based routing (see Sect. 3.5.4).

**Lemma 4.1** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters, the following holds: the fact that $F_1$ and $F_2$ contain the same number of attribute filters and that $\forall A_i^1 \exists A_j^2 . A_i^1 \equiv A_j^2$ implies that $F_1$ and $F_2$ are identical.*

PROOF: The proof is rather trivial. A notification that matches $F_1$ satisfies all attribute filters $A_i^1$. For each of these $A_i^1$ there is an identical $A_j^2$. Hence, $A_j^2$ is matched, too. As $F_1$ and $F_2$ contain the same number of attribute filter this implies that all attribute filters of $F_2$ are matched, too. Therefore, $F_2$ is also matched. As the same argumentation can be applied to notifications that match $F_2$ this implies that $F_1$ and $F_2$ match identical sets of notifications, i.e., they are identical. □

It is necessary to restrict filters to contain at most one attribute filter for each attribute in order to strengthen Lemma 4.1 to an equivalence. As a simple example, $\{x > 5 \ \wedge \ x < 5\}$ is identical to $\{x \neq \ 5\}$ although neither $\{x > 5\} \equiv \{x \neq 5\}$ nor $\{x < 5\} \equiv \{x \neq 5\}$.

**Lemma 4.2** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter for each attribute, the following holds: $F_1 \equiv F_2$ implies $\forall A_i^1 \exists A_j^2.\ A_i^1 \equiv A_j^2$.*

PROOF SKETCH: The proof is by contradiction.
ASSUME: 1. $F_1 \equiv F_2$.
2. $\forall A_i^1 \exists A_j^2.\ A_i^1 \equiv A_j^2$ does not hold.
PROVE: false
PROOF: The second assumption implies that there is an $A_i^1$ for that no identical $A_j^2$ exists. This means that either no attribute filter with the same name is contained in $F_2$ or that $L(A_i^1) \neq L(A_j^2)$. In the first case, a notification can be constructed that does not contain the respective attribute and which matches $F_2$ but does not match $F_1$. Hence, $F_1$ and $F_2$ cannot be identical and the first assumption is violated. In the second case, a notification can be constructed where the value of the respective attribute is in $L(A_i^1)$ but not in $L(A_j^2)$ if $L(A_i^1) \supset L(A_j^2)$. This notification matches $F_1$ but not $F_2$. The other way around, a notification can be constructed where the value of the respective attribute is in $L(A_j^2)$ but not in $L(A_2^j)$ if $L(A_i^1) \subset L(A_j^2)$. This notification matches $F_2$ but not $F_1$. At least one of these two cases need to occur because $L(A_i^1) \neq L(A_j^2)$. Hence, $F_1$ and $F_2$ cannot be identical and the first assumption is violated. The above cases cover all possible cases. □

**Lemma 4.3** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter for each attribute, the following holds: $F_1 \equiv F_2$ implies that $F_1$ and $F_2$ contain the same number of attribute filters.*

PROOF: by Lemma 4.2 and the fact the identity relation among filters is symmetrical. □

**Corollary 4.1** *Two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter for each attribute are identical iff they contain the same number of attribute filters and $\forall A_i^1 \exists A_j^2.\ A_i^1 \equiv A_j^2$.*

PROOF: by Lemmas 4.1, 4.2, and 4.3. □

The above corollary essentially states that two filters are identical iff they constrain the same attributes and iff the attribute filters of each constrained attribute are pairwise identical (see Fig. 4.1).

$$\begin{array}{ccccc} F_1 & = & \{x \geq 2\} & \wedge & \{y > 5\} \\ | & & | & & | \\ \equiv & & \equiv & & \equiv \\ | & & | & & | \\ F_2 & = & \{x \geq 2\} & \wedge & \{y > 5\} \end{array}$$

Figure 4.1: Identity of filters consisting of attribute filters.

### 4.3.5 Covering of Conjunctive Filters

In the following it is shown how covering of conjunctive filters can be reduced to the respective attribute filters. A covering test among filters is necessary to implement covering-based routing (see Sect. 3.5.4).

**Lemma 4.4** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters, the following holds: $\forall i \exists j.\ A_i^1 \sqsupseteq A_j^2$ implies $F_1 \sqsupseteq F_2$.*

ASSUME: $\forall i \exists j.\ A_i^1 \sqsupseteq A_j^2$
PROVE: $F_1 \sqsupseteq F_2$
PROOF: If an arbitrary notification $n$ is matched by $F_2$ then $n$ satisfies all $A_2^j$. This fact together with the assumption implies that $n$ also satisfies all $A_1^i$. Therefore, $n$ is matched by $F_1$, too. Hence, $F_1 \sqsupseteq F_2$. □

If several attribute filters can be imposed on the same attribute then $\forall i \exists j.\ A_i^1 \sqsupseteq A_j^2$ is not a necessary condition for $F_1 \sqsupseteq F_2$ (see also Fig. 4.2). For example, $\{x \in [5,8]\}$ covers $\{x \in [4,7] \ \wedge \ x \in [6,9]\}$ although $\{x \in [5,8]\}$ covers neither $\{x \in [4,7]$ nor $\{x \in [6,9]\}$. If conjunctive filters are restricted to have at most one attribute filter for each attribute then Lemma 4.4 can be strengthened to an equivalence:

**Lemma 4.5** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter for each attribute, the following holds: $F_1 \sqsupseteq F_2$ implies $\forall i \exists j.\ A_i^1 \sqsupseteq A_j^2$.*

ASSUME: $\neg(\forall i \exists j.\ A_i^1 \sqsupseteq A_j^2)$
PROVE: $\neg(F_1 \sqsupseteq F_2)$
PROOF: A notification $n$ is constructed that matches $F_2$ but not $F_1$ to prove that $F_1$ does not cover $F_2$. The assumption implies that there is at least one $A_k^1$ that

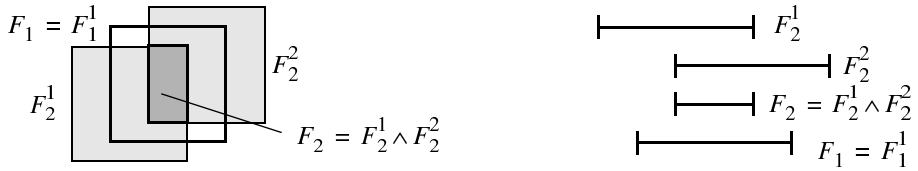



Figure 4.2: $F_1 \sqsupseteq F_2$ although neither $F_1^1 \sqsupseteq F_2^1$ nor $F_1^1 \sqsupseteq F_2^2$ (two examples).

does not cover any $A_j^2$. If there exists an $A_l^2$ that constrains the same attribute as such an $A_k^1$ then choose for this attribute a value that matches $A_l^2$ but not $A_k^1$. Such a value exists because $L_A(A_k^1) \neq \emptyset$ and $A_k^1 \not\sqsupseteq A_l^2$. Add name/value pairs for all other attributes that are constrained in $F_2$ such that they are matched by the appropriate attribute filters of $F_2$. The constructed notification matches $F_2$ but not $F_1$. Therefore, $F_1$ does not cover $F_2$. □

**Corollary 4.2** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter per attribute, the following holds: $F_1 \sqsupseteq F_2$ is equivalent to $\forall i \exists j.\ A_i^1 \sqsupseteq A_j^2$.*

Proof: by Lemmas 4.4 and 4.5. □

The above corollary essentially states that a filter $F_1$ covers a filter $F_2$ iff for each attribute filter in $F_1$ there is an attribute filter in $F_2$ that is covered by the former (see Fig. 4.3).

$$
\begin{array}{ccccccccc}
F_1 & = & \{x \geq 2\} & \wedge & \{y > 5\} & & \\
| & & | & & | & & \\
\sqsupseteq & & \sqsupseteq & & \sqsupseteq & & \\
| & & | & & | & & \\
F_2 & = & \{x = 4\} & \wedge & \{y = 7\} & \wedge & \{z \in [3,5]\}
\end{array}
$$

Figure 4.3: Covering of filters consisting of attribute filters.

### 4.3.6 Overlapping of Conjunctive Filters

In the following it is shown how overlapping of conjunctive filters can be reduced to the respective attribute filters. An overlapping test among filters is necessary to implement advertisements.

**Lemma 4.6** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters, $\exists A_i^1, A_j^2.\ \left(n_i^1 = n_j^2 \ \wedge\ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\right)$ implies that $F_1$ and $F_2$ are disjoint.*

PROOF: Suppose that $F_1$ and $F_2$ contain attribute filters $A_i^1$ and $A_j^2$ such that $\big(n_i^1 = n_j^2 \ \wedge \ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\big)$. This means that both filters require the existence of an attribute with name $n_i^1$ and that the value of this attribute must match $L_A(A_i^1)$ in order to make a notification match $F_1$ and $L_A(A_j^2)$ in order to match $F_2$. As $L_A(A_i^1)$ and are $L_A(A_j^2)$ disjoint this implies that a given notification can be matched either by $F_1$ or by $F_2$. Hence, $F_1$ and $F_2$ are disjoint. □

It is necessary to restrict filters to contain at most one attribute filter for each attribute in order to strengthen Lemma 4.6 to an equivalence. As a simple example, $\{x \in \{3,5\} \ \wedge \ x \in \{4,5\}\}$ is disjoint with $\{x \in \{3,5\} \ \wedge \ x \in \{3,4\}\}$ although there are no disjoint attribute filters.

**Lemma 4.7** *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter for each attribute, the fact that $F_1$ and $F_2$ are disjoint implies that $\exists A_i^1, A_j^2.\ \big(n_i^1 = n_j^2 \ \wedge \ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\big)$.*

PROOF: The proof is by contradiction. Suppose that $F_1$ and $F_2$ are disjoint and that there are no $A_i^1, A_j^2$ such that $n_i^1 = n_j^2 \ \wedge \ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset$. We construct a notification that matches $F_1$ and $F_2$ to imply a contradiction in following way: For each attribute that is constrained in $F_1$ or $F_2$ add an attribute whose value satisfies the attribute filters contained in $F_1$ and $F_2$ regarding this attribute. This value must exist because there are no $A_i^1, A_j^2$ such that $n_i^1 = n_j^2 \ \wedge \ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset$. Hence, the constructed notification matches $F_1$ and $F_2$, and therefore, $F_1$ and $F_2$ are not disjoint. □

**Corollary 4.3** *Two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter for each attribute are disjoint, i.e., not overlapping, iff $\exists A_i^1, A_j^2.\big(n_i^1 = n_j^2 \wedge L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\big)$.*

PROOF: by Lemmas 4.6 and 4.7. □

$$
\begin{array}{ccccc}
F_1 & = & \{x \geq 2\} & \wedge & \{y > 5\} \\
| & & | & & | \\
\not\sqcap & & \not\sqcap & & \sqcap \\
| & & | & & | \\
F_2 & = & \{x < 1\} & \wedge & \{y < 7\}
\end{array}
$$

Figure 4.4: Disjoint filters consisting of attribute filters.

The above corollary essentially states that two filters are disjoint iff for an attribute that is constrained in both filters the corresponding attribute filters are disjoint (see Fig. 4.4). Hence, two filters are overlapping iff no such attribute filters exist (see Fig. 4.5).

$$\begin{array}{ccccc} F_1 & = & \{x \geq 2\} & \wedge & \{y > 5\} \\ | & & | & & | \\ \sqcap & & \sqcap & & \sqcap \\ | & & | & & | \\ F_2 & = & \{x < 5\} & \wedge & \{y < 7\} \end{array}$$

Figure 4.5: Overlapping filters consisting of attribute filters.

### 4.3.7 Merging of Conjunctive Filters

In Chapter 3 merging-based routing algorithms were presented which used abstract merging operations. In this section merging of conjunctive filters is discussed for the current model of structured records. The aim of filter merging is to determine a filter that is a merger of a set of filters. Merging of filters can be used to drastically reduce the amount of subscriptions and advertisements that have to be stored by the routing algorithm.

#### Perfect Merging

A set of conjunctive filters with at most one attribute filter for each attribute can be perfectly merged into a single conjunctive filter if for all except a single attribute their corresponding attribute filters are identical and if the attribute filters of the distinguishing attribute can be merged into a single attribute filter. For example, the two filters $F_1 = \{x = 5 \wedge y \in \{2, 3\}\}$ and $F_2 = \{x = 5 \wedge y \in \{4, 5\}\}$ can be merged to $F = \{x = 5 \wedge y \in \{2, 3, 4, 5\}\}$. Moreover, a set of attribute filters imposed on the same attribute with name $n$ can be merged to an *exists*$(n)$ test if at least one of them is satisfied by any value. Note that an existence test is equivalent to no constraint if the attribute is mandatory for the corresponding type of notification.

An algorithm that determines the possibly empty set of filters which are candidates to be merged with a given filter is depicted later in Section 4.3.8. From the set of merging candidates the set of attribute filters to be merged can easily be extracted. This set is used as input of a merging algorithm which has a specialized implementation for each type of constraint. In the general case purely algebraic merging techniques have exponential time complexity. Alternatively, a predicate proximity graph can be used to implement a greedy algorithm [61]. For many practical cases (e.g., set operators) efficient algorithms exist. Only in rare cases it is necessary to use an exhaustive combinatorial or a suboptimal greedy algorithm.

The characteristics of the constraints that are used to define attribute filters are important for merging. Constraints which only exist in a normal and a negated form can be directly merged by using some basic laws of boolean algebra. For example, the filters $F_1 = (y = 3 \wedge x = 5)$ and $F_1 = (y = 3 \wedge x \neq 5)$ can be

merged to $F = (y = 3 \wedge \exists x)$. In general, constraints are not restricted to be the negated form of each other and hence, better merging can be achieved by taking the specific characteristics of the imposed constraints into account.

A class of constraints that is *complete under disjunction* allows to merge a set of constraints of this class into a single constraint of the same class. Examples for disjunction-complete constraints are *set inclusions* (e.g., $x \in \{2, 3, 7\}$) and *set exclusions* (e.g., $x \notin \{2, 3, 7\}$) while *comparison constraints* (e.g., $x < 4$) are not disjunction-complete. If a constraint class is not disjunction-complete it may still be possible to carry out merging if a specific *merging condition* is met. For example, a set of *interval tests* (e.g., $x \in [2, 4]$ and $x \in [3, 5]$) can be merged into a single interval test (here, $x \in [2, 5]$) if the intervals form a connected set. Otherwise, merging may be possible if a more general constraint is considered as merging result. For example, two comparison constraints (e.g., $x < 4$ and $x > 7$) can be merged to an interval test (here, $x \notin [4, 7]$).

Merging on the level of attribute filters is implemented by each generic attribute filter class. Table 4.8 below presents some versatile perfect merging rules. The meaning of a single row is that $A_1$ and $A_2$ can be perfectly merged to the indicated merger (column 4) if the given merging condition (column 3) holds.

| $A_1$ | $A_2$ | Condition | $A_1 \cup A_2$ |
|---|---|---|---|
| $x \in M_1$ | $x \in M_2$ | - | $x \in M_1 \cup M_2$ |
| $x \notin M_1$ | $x \notin M_2$ | $M_1 \cap M_2 = \emptyset$ | $\exists x$ |
| | | $M_1 \cap M_2 \neq \emptyset$ | $x \notin M_1 \cap M_2$ |
| $X$ *overlaps* $M_1$ | $X$ *overlaps* $M_2$ | - | $X$ *overlaps* $M_1 \cup M_2$ |
| $X$ *disjunct* $M_1$ | $X$ *disjunct* $M_2$ | $M_1 \cap M_2 = \emptyset$ | $\exists X$ |
| | | $M_1 \cap M_2 \neq \emptyset$ | $X$ *disjunct* $M_1 \cap M_2$ |
| $x = a_1$ | $x \neq a_1$ | $a_1 = a_2$ | $\exists x$ |
| $x < a_1$ | $x > a_2$ | $a_1 > a_2$ | $\exists x$ |
| | $x \geq a_2$ | $a_1 \geq a_2$ | |
| $x \leq a_1$ | $x > a_2$ | $a_1 \geq a_2$ | $\exists x$ |
| | $x \geq a_2$ | | |

Table 4.8: Some versatile perfect merging rules for attribute filters

The first two rules can also be applied to equality and inequality tests because $x = a_1 \Leftrightarrow x \in \{a_1\}$ and $x \neq a_1 \Leftrightarrow x \notin \{a_1\}$.

### Imperfect Merging

At a first glance, imperfect merging seems to be less promising but in situations in which perfect merging is either too complex or not computable it is a good compromise. Clearly, there exists a trade-off between filtering overhead and network resource consumption. Imperfect merging may result in notifications

being forwarded that do not match any of the original subscriptions, but on the other hand, it reduces the amount of subscriptions and advertisements that must be dealt with.

In order to use imperfect merging heuristics are necessary that define in what situations and to what degree imperfect merging should be carried out. For example, filters that differ in few attribute filters could be merged imperfectly by imposing on each attribute a constraint that covers all original constraints. In order to decide whether two given filters should be merged a heuristic that allows the amount of introduced imperfection to be estimated is needed. This could also be accomplished by explicitly replacing an attribute filter with another that only tests for the existence of the given attribute or by simply dropping the attribute filter. Statistical online evaluation of filter selectivity would be also a good basis for merging decisions that enables adaptive filtering strategies. Imperfect merging requires further investigation.

### 4.3.8 Algorithms

In this section algorithms are presented that are superior to the naive algorithms. They use the generic approach presented in Section 4.3.3: Each generic constraint class (e.g., constraints on ordered values) offers specialized indexing data structures to efficiently manage constraints on attributes. For example, hashing is used for equality tests. In the following algorithms for matching, covering, and for detecting merging candidates are described that are all based on the predicate counting algorithm. Algorithms for detecting identity and overlapping among filters can be derived similarly.

#### Matching Algorithm

The naive algorithm separately matches a given notification against all filters to determine the set of matching filters. This implies that the same attribute filter may be evaluated many times. More advanced algorithms (see related work section for a detailed overview) avoid this. Some of these more advanced matching algorithms require a costly compilation step (e.g., [50]) that makes them less suitable for publish/subscribe systems in which subscriptions change dynamically. In contrast to that, the algorithm presented here allows filters to be added or removed at any time. The algorithm is based on the idea of *predicate counting* [89, 120] and makes use of the generic approach. The algorithm is depicted in Fig. 4.6. It determines all filters that match a given notification.

#### Covering Algorithm

Covering-based routing is built upon two tests: a first test that determines all filters that cover a given filter and a second one determines all filters that are covered by a given filter. The naive implementation simply tests each filter against all others sequentially. The algorithms presented here are more efficient.

```
Matching Algorithm
Input:  notification n, set of filters F
Output: the set M of all filters in F that match n.
{
   <For each filter in F a counter is initialized to zero.>
   for <each A_i contained in n> {
      for <each filter S in F that has a constraint on A_i that
             is satisfied by the value of the corresponding
             attribute of n> {
         <Increment the counter of S>
      }
   }
   M:=<all filters in F whose counter is equal to their number
          of attribute filters>
}
```

Figure 4.6: Matching algorithm based on counting satisfied attribute filters

They are derived from the matching algorithm presented above (see Fig. 4.7 and 4.8).

```
Covering Algorithm I
Input:  filter F_1, set of filters F
Output: the set C of all filters in F that cover F_1.
{
   <For each filter in F a counter is initialized to zero.>
   for <each A_i contained in F_1> {
      for <each filter S in F that has a constraint A_j that
             covers A_i> {
        <Increment the counter of S>
      }
   }
   C:=<all filters in F whose counter is equal to their number
          of attribute filters>
}
```

Figure 4.7: Covering algorithm that determines all covering filters.

### Merging Algorithm

Here, an algorithm is presented that determines all possible merging candidates which are those filters that are identical to a given filter in all but a single attribute. The algorithm avoids testing all filters against all others. It counts the number of identical attribute filters to find merging candidates (see Fig. 4.9).

The further handling of the set of merging candidates depends on the constraints involved. For all constraints discussed in section 3 (e.g., set constraints on simple values) there exists an efficient algorithm which outputs a single merged filter and a set of filters not included in the merger. For other constraints, an optimal algorithm requires exponential time complexity [27]. In this

```
Covering Algorithm II
Input: filter F1, set of filters F
Output: the set C of all filters in F that are covered by F1.
{
   <For each filter in F a counter is initialized to zero.>
   for <each Ai contained in F1> {
      for <each filter S in F that has a constraint Aj that
      is covered by Ai> {
        <Increment the counter of S>
      }
   }
   C:=<all filters in F whose counter is equal to the number
           of attribute filters of F1>
}
```

Figure 4.8: Covering algorithm that determines all covered filters.

case the use of greedy algorithms or heuristics (e.g., using a predicate proximity graph) seem to be promising.

```
Merging Algorithm
Input: filter F1, set of filters F
Output: set M of all merging candidates
{
   <For each filter in F a counter is initialized to zero.>
   for <each Ai contained in F1> {
      for <each filter S in F that has a constraint Aj that is
      identical to Ai> {
         <Increment the counter of S>
      }
   }
   M:=<all filters in F whose counter is one smaller than or
           equal to their number of attribute filters>
}
```

Figure 4.9: Merging algorithm based on counting identical attribute filters.

## 4.4 Semistructured Records

In the last section structured records have been discussed in detail. In this section a model for semistructured records is presented. The structured and the semistructured model are mainly distinguished by the following fact: In the structured model attribute names are unique, and hence, an attribute name uniquely addresses a single attribute. On the contrary, in the semistructured model sibling attributes can have the same name, and therefore, names address sets of attributes.

In the following, a model for semistructured records is presented in which notifications are essentially XML [118] documents. The filtering mechanisms

are similar to but less powerful than XPath [119]. After the model has been introduced, how routing optimizations can be achieved is discussed.

### 4.4.1 Data Model

In the semistructured data model a *notification* is a well-formed XML document [118] and consists of a set of elements that are arranged in a hierarchy with a single root element uniquely named "notification". Each *element* consists of a set of attributes whose names must be distinct and a set of subordinate *child elements* which are named but whose names must not necessarily be distinct. An *attribute* $A$ is a pair $(n_i, v_i)$ with name $n_i$ and value $v_i$. Names of attributes must be unique with respect to elements. A simple notification that describes an auction is shown in Fig 4.10. In this example, the element *auction* has two subelements which are named *item*. Furthermore, the element *cpu* contains an attribute *clock* whose value is 800. Note that XML documents can contain free text between the opening and the closing tag of an element. Here, this text is simply ignored.

```
<notification>
   <auction
      endtime="05/18/02 22:17:42"
      minprice="$50">
      <seller
         name="Smith"
         id="1234"/>
      <item>
         <board
           manufacturer="Elitegroup"
           type="K7S5"
           socket="Socket A"/>
      </item>
      <item>
         <cpu
            manufacturer="AMD"
            type="Athlon"
            socket="Socket A"
            clock="800"/>
      </item>
   </auction>
</notification>
```

Figure 4.10: A simple notification

### 4.4.2 Filter Model

In the semistructured filter model a filter is a conjunction of path filters. Each of the path filters selects a subset of the elements in a notification by an element selector and places constraints on the attributes of the selected elements by an

element filter which consists of a set of attribute filters. In the following, this model is described in full detail.

**Element Selectors.** An *element selector* selects a subset of the elements of a notification and is specified by an attribute path. It is distinguished between absolute and abbreviated paths. An *absolute path* is a slash separated string that starts with a single slash (e.g., */notification/auction*). An *abbreviated path* is a slash separated string that starts with two slashes (e.g., *//cpu*). An absolute/abbreviated path selects all elements whose path is equal to/ends with the given path. For example, *//item* selects both *item* elements of the notification in Fig. 4.10.

**Attribute Filters.** An *attribute filter* is a pair $A = (n, Q)$ consisting of a name $n$ (e.g., *manufacturer*) and a constraint $Q$ (e.g., $=$ *"AMD"*). An element *matches an attribute filter* if the element contains an attribute, e.g., (*manufacturer*, *"AMD"*), with name $n$ whose value $v$ satisfies $Q$. This means that an attribute filter evaluates to false if the element does not contain an attribute with name $n$. Therefore, an attribute filter implicitly defines an existential quantifier over an element.

**Element Filters.** An *element filter* $C$ is a conjunction of a nonempty set $A$ of attribute filters $\{A_1, \ldots, A_i\}$, i.e., $C = \wedge_i A_i$. Hence, an element *matches an element filter* iff all attribute filters are satisfied. An example of an element filter based on the syntax of XPath is [*@manufacturer* $=$ *"AMD"* $\wedge$ *@clock* $\geq 700$]. Note that in this notation attribute names are prefixed by an "@".

**Path Filters.** A *path filter* $P = (S, C)$ consists of an element selector $S$ and an element filter $C$. A notification $n$ *matches a path filter* $P$ if at least one element of $n$ is selected by $S$ that matches $C$. It is possible to extend this model in such a way that an interval constraint can be imposed on both the number of elements that match an element filter and the number of elements that must not match. These extensions are excluded for brevity. An example of a complete path filter based on an absolute path is: */notification/auction/item/cpu*[*@manufacturer* $=$ *"AMD"* $\wedge$ *@clock* $\geq 700$].

**Filters.** A *filter* $F$ is a conjunction of path filter $\{P_1, \ldots, P_n\}$. Hence, a notification *matches a filter* if all path filters are satisfied. The set of all notifications that match a given filter $F$ is $N(F)$.

### 4.4.3 Covering

This subsection shows how covering among filters in the semistructured model can be detected. Similar results can easily be obtained for identity and overlapping, too. These are left out for brevity.

**Covering of Attribute Filters.** Let $L_A(A)$ be the set of all values that cause an attribute filter $A$ to match an attribute. An attribute filter $A_1 = (n_1, Q_1)$ *covers* an attribute filter $A_2 = (n_2, Q_2)$, denoted by $A_1 \sqsupseteq A_2$, iff $n_1 = n_2 \wedge L_A(A_1) \supseteq L_A(A_2)$. For example, [@*clock* $\geq$ 600] covers [@*clock* $\geq$ 700].

**Covering of Element Filters.** Let $L_E(C)$ be the set of all elements that match an element filter $C$. An element filter $C_1$ *covers* an element filter $C_2$, denoted by $C_1 \sqsupseteq C_2$, iff $L_E(C_1)$ is a superset of $L_E(C_2)$. For example, [*@clock* $\geq$ 600] covers [*@manufacturer* = *"AMD"* $\wedge$ *@clock* $\geq$ 700].

Furthermore, $C_1$ is *disjoint* with $C_2$ with respect to the constrained attributes if there exists no attribute that is constrained in both element filters. For example, [@*minprice* < \$100] is disjoint with [*@name* = *"Pu"*] with respect to their constrained attributes.

**Corollary 4.4** *Given two element filters $C_1$ and $C_2$, which do not contain two attribute filters with the same name, the following holds: $C_1 \sqsupseteq C_2$ is equivalent to $\forall j \exists i.\ A_i^1 \sqsupseteq A_j^2$.*

**Covering of Element Selectors.** Let $L_S(S)$ be the set of all elements that are selected by an element selector $S$. An element selector $S_1$ *covers* an element selector $S_2$, denoted by $S_1 \sqsupseteq S_2$, iff $L_S(S_1) \supseteq L_S(S_2)$. Furthermore, $S_1$ is *disjoint* with $S_2$, iff $L_S(S_1) \cap L_S(S_2) = \emptyset$.

In the model presented here, an absolute path covers another absolute path iff both are identical. An absolute path only covers an abbreviated path iff the former is */notification* and the latter is *//notification* as the root element has a unique name. An abbreviated path covers another (abbreviated or absolute) path iff the former is a suffix of the latter (without the leading // or /). For example, *//cpu* covers *//item/cpu* because the former path selects all elements named *cpu*, while the latter only selects those elements named *cpu* which are a subelement of an element with name *item*.

**Covering of Path Filters.** Let $L_P(P)$ be the set of all elements that match a path filter $P$. A path filter $P_1 = (S_1, C_1)$ *covers* another path filter $P_2 = (S_2, C_2)$, written $P_1 \sqsupseteq P_2$, if and only if $L_P(P_1) \supseteq L_P(P_2)$. For example, the path filter *//cpu*[*@manufacturer* = *"AMD"*] covers *//cpu*[*@manufacturer* = *"AMD"* $\wedge$ *@clock* $\geq$ 700]. $P_1$ is *disjoint* with $P_2$, iff either $S_1$ is disjoint with $S_2$ or if $C_1$ is disjoint with $C_2$ with respect to their constrained attributes.

**Corollary 4.5** *Given two path filters $P_1 = (S_1, C_1)$ and $P_2 = (S_2, C_2)$ the following holds: $P_1 \sqsupseteq P_2$ is equivalent to $S_1 \sqsupseteq S_2 \wedge C_1 \sqsupseteq C_2$.*

**Covering of Filters.** A filter $F_1$ *covers* a filter $F_2$, denoted by $F_1 \sqsupseteq F_2$, iff $N(F_1) \supseteq N(F_2)$.

**Corollary 4.6** *Given two filters* $F_1 = P_1^1 \wedge \ldots \wedge P_n^1$ *and* $F_2 = P_1^2 \wedge \ldots \wedge P_m^2$ *which are conjunctions of disjoint path filters the following holds:* $F_1 \sqsupseteq F_2$ *is equivalent to* $\forall i \exists j.\ P_i^1 \sqsupseteq P_j^2$.

For example, the filter $\{//cpu[@type = \text{“}Athlon\text{”}]\}$ covers $\{//seller[@name = \text{“}Pu\text{”}] \wedge //cpu[@type = \text{“}Athlon\text{”} \wedge @clock \geq 600]\}$ .

## 4.5 Objects

A purely object-oriented approach models notifications and filters as objects. A clear advantage of such a model is that it can easily be integrated with object-oriented programming languages. In contrast to that, models that are based on e.g., name/value pairs, can only operate on serialized instances of objects violating object encapsulation. Unfortunately, routing optimizations and in particular covering and merging are difficult to achieve if filters can contain arbitrary code. In this section three scenarios for that covering and merging can be supported are described.

### 4.5.1 Calling Methods on Attribute Objects

Regardless whether the data models depends structured or on semistructured records it is possible to embed objects in notifications. In this case public members can be accessed and public inspector methods can be invoked on the embedded object after it has been instantiated. The returned member or the return value of the inspector method can either be a boolean value that is directly interpreted as result of the attribute filter or a value that is used in order to evaluate the actual constraint.

For example, suppose that an instance of a class `StockQuote` has been embedded in a notification as an attribute with name *quote*. Then an attribute filter that evaluates this attribute could be specified like this $\{quote.id() = \text{“}IBM\text{”}\}$. For example, this filter covers $\{quote.isRealTime() \wedge quote.id() = \text{“}IBM\text{”} \wedge quote.Price() > 45.0\}$. Moreover, it could be merged with a filter $\{quote.id() = \text{“}MSFT\text{”}\}$ to a filter $\{quote.id() \in \{\text{“}IBM\text{”}, \text{“}MSFT\text{”}\}\}$. As stated in [33, 35] structural reflection (e.g., supported by JAVA) can be used to invoke the specified methods. Unfortunately, the model does not allow to detect all covering relations among filters. For example, a filter $\{quote.Volume() > \$10,000\}$ covers a filter $\{quote.Price() > \$100 \wedge quote.Quantity() > 100\}$ because the volume is defined as the product price multiplied by the quantity.

### 4.5.2 Filtering on Notification Classes

Here, notifications are objects and consequently they are an instance of some class. Hence, class filters can be used that evaluate the class of a notification: a notification matches a filter if it is assignable to the specified class. It is also possible to support covering and merging. A class filter covers another class filter

if an instance of the latter class can be assigned to an instance of the former one. A set of class filters can be merged perfectly if they either contain a class which covers all other classes or if they represent all direct subclasses of their common superclass. Figure 4.11 shows the implementation of a `ClassFilter` in JAVA.

```
class ClassFilter {
   protected Class _class;

   public boolean covers(ClassFilter filter) {
      return _class.isAssigneableFrom(filter._class);
   }

   public static ClassFilter merge(ClassFilterSet filters) {
      Class superClass=filters.getCommonSuperClass();
      if (superclass!=null) {
         if (filters.contain(superClass))
            return new ClassFilter(superClass);
         if (filters.containAllSubclasses(superClass))
            return new AllSubclassesFilter(superClass);
      }
      return null;
   }

   public boolean match(Notification n){
      return _class.isInstance(n);
   }
}
```

Figure 4.11: Implementation of a `ClassFilter` in JAVA

The integration with content-based filtering can be achieved by supporting filters that are conjunctions of a class filter and a specialized filter object whose match method is invoked if the class filter returned true.

### 4.5.3 Specialized Filter Objects

Another possibility is to use specialized filter objects, an approach that can also be combined with class filters. Such a filter implements a `match` method that evaluates whether a notification matches this filter instance or not. Moreover, it can also implement methods for covering and merging. Figure 4.12 shows the implementation of a `QuoteFilter` in JAVA. Note that the filters can also be built upon a more generic filter library which offers, for example, set-oriented filters.

## 4.6 Related Work

**Support of Routing Optimizations.** ELVIN [101] supports quenching in which notifications are first evaluated against a broader subscription that covers the disjunction of all subscriptions but no algorithms for quenching are described.

```
public class QuoteFilter {

   public boolean covers(QuoteFilter qf){
      return getSymbolSet().isSuperSet(qf.getSymbolSet());
   }

   public static QuoteFilter merge(QuoteFilter[] qf){
      return new QuoteFilter(QuoteFilter.unionOfSymbolSets(qf));
   }

   public boolean match(Event e) {
      if (!(e instanceof QuoteEvent))
         return false;
      return (qf.getSymbolSet().contains(
                ((QuoteEvent)e).getSymbol()));
   }
}
```

Figure 4.12: Implementation of a `QuoteFilter` in JAVA

SIENA [19, 17] exploits covering among filters, and uses overlapping of filters to support advertisements. The data/filter model of SIENA is similar to structured records but allows for multiple attribute filters on the same attribute. If multiple attribute filters are imposed on an attribute, they are interpreted differently with respect to subscriptions and advertisements. In the case of a subscription they are interpreted conjunctively, while for advertisements they are interpreted disjunctively. Hence, their model is not symmetrical. This choice causes no problems with the simple types and operators supported in SIENA, but it inhibits supporting routing optimizations for more complex types and operators. Algorithms that determine coverage or overlapping among filters are not presented.

**Matching Algorithms.** Yan and Garcia-Molina [120] describe several matching algorithms including the predicate counting, the key, and the tree algorithm in the context text documents which are matched against keyword-based profiles. In this paper they also present performance results obtained from simulation.

Fabre et al. [37] and Pereira et al. [89] present matching algorithms which exploit similarities among predicates. In a first step the satisfied predicates are computed and after that the number of predicates satisfied by a subscription are counted using an association table. Two variants of this algorithm are described which incorporate special treatment of equality tests and of constraints having only inequality tests.

A predicate matching algorithm for database rule systems is presented by Hanson et al. [54] that indexes the most selective predicate that is determined by the query optimizer. They use a special indexing data structure called interval binary search tree to support the efficient evaluation of interval tests.

Gough and Smith [50] present a matching algorithm that is based on au-

tomata theory. They show how a set of conjunctions of predicates, each dependent on exactly one attribute, can be transformed to a deterministic finite state automaton. In the paper different types of test predicates are considered and complexity results are obtained. Their algorithm is very efficient, but its worse case space complexity is exponential. The proposed solution is also not suited for dynamic environments as the automaton has to be newly constructed from scratch if subscriptions change.

Pu et al. [70, 112] present indexing strategies for continual queries based on trigger patterns. In particular, a strategy which uses an index on the most selective predicate is described. More complex indexing strategies exploit similarities among trigger patterns to reduce the processing costs. They restrict optimizations to constraints which place a constraint on a single attribute involving at most one constant.

Gryphon uses the content-based matching algorithm presented by Aguilera et al. [1]. This algorithm traverses a parallel search tree where non-leaf nodes correspond to simple tests and edges from non-leaf nodes represent results. Leaf-nodes are associated with matched subscriptions. Banavar et al. [8] present a multicast routing algorithm that executes the matching algorithm at each broker. The algorithm presented is limited to equality tests.

**Answering Queries using Views.** Covering relations are known from database theory and in particular from the area of answering queries using views [53, 115]. There, the question is whether the result set of a given query $Q$ can be solely obtained from a set of predefined views $V$ whose elements can be combined by the usual relational operators, i.e., whether $Q$ is covered by some combination of the views in $V$. Answering this question for relational expressions is $NP$-hard even without comparison operators. If only the union operator is allowed, this is still a more general scenario than the one presented here. Although special cases have been investigated, an approach that is closely related does not seem to exist.

**Semantic Caching.** Lee and Chu [65] describe a semantic caching algorithm for conjunctive point queries that exploits covering between conjunctive predicates to find cache entries which cover a given query. However, this work is restricted to point queries involving the equivalence and the like operator. Godfrey and Gryz [49] depict an architecture for predicate-based caching that is similar to answering queries using views. Therefore, it is not surprising that their algorithms are $NP$-complete, too. Keller and Basu [61] propose a predicate-based caching scheme for client/server database architectures. They perfectly merge predicates in the cache to obtain a more compact cache description and to speed up query processing. Their algorithm has exponential time complexity.

**Query Merging.** Crespo et al. [27] propose merging of queries that are evaluated periodically against a database. As example, they use geographical queries

represented by a rectangle. Before the queries are processed a merging algorithm is run that combines similar queries and outputs a set of merged queries whose answers contain all tuples of the original query. Their aim is to find a set of mergers which is cost optimal. They show that in the general case query merging is *NP*-complete and discuss optimal and heuristic algorithms.

**Geometrical Algorithms.** In the context of geometrical algorithms [95], for example, polygon inclusion, intersection, and containment of convex polygons are investigated. These algorithms can be integrated with the work presented here to support efficient matching, covering, and merging of notifications containing geometric objects. Such objects are, for example, prevalent in geographical information systems.

## 4.7 Discussion

In this chapter several data/filter models have been discussed and it was shown how the routing optimizations that are used by the content-based routing algorithm described in Chapter 3 can be mapped to the actual data and filter model. The emphasis was placed on structured records consisting of name/value pairs because this is currently the most prominent model. Additionally, routing optimizations for semistructured records and objects have been discussed. This discussion should be seen as a first step towards a more complete treatment of these models.

For structured records, first detailed data and filter models have been presented that addressed some shortcomings of previous approaches. The filtering model builds upon conjunctive filters that consist of attribute filters. To enable efficient evaluation of the tests used by the routing algorithms and to facilitate filter merging, only at most one attribute filter for each attribute is allowed. Subsequently, a generic filtering framework has been described that is separated from the routing and matching algorithms, supports the proposed routing optimizations, and that is not restricted to specific data types or operators. Instead, new data types and operators can be added easily. As examples, covering implications and merging rules for a set of relevant data types with typical operators have been described. Based on the model, matching, covering, and merging algorithms that support the generic approach have been presented.

For semistructured records, a data model in which notifications are essentially XML documents has been proposed. The filter model builds upon path expressions being a subset of XPath. Roughly speaking, the model for semistructured records is a generalization from the one of structured records because sets of attributes instead of single attribute are selected by a path, i.e., a partial attribute name. After the model has been described, some initial ideas on how to support covering and merging have been presented.

For the object-oriented model, three scenarios in which covering and merging can be supported have been described. The first one relies on extending the

record-oriented models by allowing to call methods on attribute objects. The second and the third dealt with filters on notification classes and specialized filter objects, respectively.

# Chapter 5

# Implementation

## Contents

## 5.1 Introduction

This chapter describes the implementation that has been carried out as part of this thesis. The implementation has been realized in JAVA and consists of two major parts which are a content-based notification infrastructure, called REBECA, and two example applications which serve as proof of concept.

REBECA is a recursive acronym that stands for **R**ebeca **E**vent-**B**ased **E**lectronic **C**ommerce **A**rchitecture. The system can be distinguished among other notification service prototypes by a number of characteristics:

- Instead of relying on a single routing scheme, a set of routing algorithms has been implemented. Moreover, new routing algorithms can be added easily. This allows to test and to compare various routing algorithms to each other in a uniform environment.

- While other notification services only support a fixed set of primitive filtering constraints, REBECA relies on a powerful, flexible, and extensible filtering framework that supports routing optimizations.

- REBECA supports replay of past notifications and service factories. In order to enable these features, subscription and unsubscription events, as well as corresponding subscriptions for these notifications have been introduced. Subscription and unsubscription events are automatically published by the infrastructure if a client subscribes or unsubscribes.

- REBECA has also served as the basis for the evaluation that is described in Chapter 6.

Two example applications (see Sect. 5.4) have been implemented, an infrastructure for self-actualizing web pages (see Sect. 5.4.1) and a stock trading platform (see Sect. 5.4.2). The discussion of these applications also shows how event-based applications can be engineered. Note that a demo of the self-actualizing web pages can be accessed on-line [39].

In the following, first the design and the concept that underly the REBECA notification infrastructure are described. This includes a description of the most important classes. After that, it is depicted how the REBECA infrastructure can be used to build a simple event-based application. Subsequently, the functionality and the design of the implemented example applications are described.

## 5.2 The REBECA Notification Infrastructure

In this section the design and the concepts of the REBECA Notification Infrastructure are described.

### 5.2.1 General Architecture

The REBECA infrastructure consists of a set of interconnected event brokers. Brokers are divided into two categories: *Local brokers* are the access points of the publish/subscribe system and therefore, they manage the clients. Each of them is connected to exactly one router. The *routers*, on the other hand, do not manage clients. They solely concentrate on forwarding incoming notifications to their respective neighbors. Currently, brokers are connected to routers and routers are interconnected by communicating serialized JAVA Objects over TCP sockets or by local queues.

### 5.2.2 Available Routing Algorithms

The infrastructure can be configured to use one of the following routing algorithms that have been described in Section 3.5:

- Flooding (cf. Sect. 3.5.1),
- Simple Routing (cf. Sect. 3.5.2),
- Identity-Based Routing (cf. Sect. 3.5.3),
- Covering-Based Routing (cf. Sect. 3.5.4), and
- Merging-Based Routing (cf. Sect. 3.5.5).

Additionally, it can be chosen whether or not advertisements are used for routing (cf. Sect. 3.6). The routing algorithms are built upon a filter framework that uses the name/value pair data model (cf. Sect. 4.3) and offers an extensible set of attribute filters. For more details please refer to Chapter 4.

### 5.2.3 Replay Mechanism

The REBECA infrastructure includes a replay mechanism which allows consumers to subscribe to and to receive past notifications that have been recorded on their behalf. This is often necessary to initialize new components. For example, if a watch list that displays current stock quotes is initialized, the user normally does not want to wait until for each monitored stock a new quote arrives. Instead the watch list should display the current prices immediately.

In order to support replay of events, histories can be connected to the REBECA infrastructure. A *history* is responsible for recording and replaying certain kinds of notifications. Therefore, a history issues corresponding subscriptions and saves the notifications it receives in an internal storage. On a regular basis or triggered by some event, the history investigates the recorded notifications and removes those notifications which are no longer needed. A consumer indicates its wish to receive past notifications by attaching a *replay description* to the subscription it issues. In order to notify histories about new subscriptions, for each new subscription a corresponding *subscription event* is automatically

published by the infrastructure that contains the subscription and the replay description. Hence, a history subscribes to those subscription events whose embedded subscription and replay description concern the notifications it records. If a history receives such a subscription event, it determines those of the stored notifications that match both the subscription and the replay description. After that, each of the corresponding notifications is embedded in a replay event and published by the history. The infrastructure makes sure that a replay event is delivered to the specific consumer only.

### 5.2.4 Factory Concept

In large-scale publish/subscribe systems it might be impossible to produce all notifications to which consumers may subscribe. Therefore, a mechanism is needed that on one hand ensures the production of all notifications that are currently needed and that on the other hand suppresses the production of notifications for which there are currently no subscribers. This is especially important when considering multi-step information flows. If a producer stops to publish notifications, it may also unsubscribe to the notifications it consumes. Hence, the producer of these events might also be deactivated if there are no further consumer of them. As a first step towards automatic instantiation and de-instantiation of producers, REBECA introduces the concept of factories.

The factory concept relies on subscription and unsubscription events that are automatically published by the REBECA infrastructure if a consumer subscribes or unsubscribes. A *factory* manages a set of service instances and subscribes to those subscription and unsubscription events whose embedded subscription overlaps with the notifications its service instances are responsible to publish. If a service factory receives a subscription event whose subscription is not completely served by the active services, it either activates an existing service or creates a new service that produces the desired notifications. If a service factory receives an unsubscription event, it checks whether there are any service instances that could be deactivated or de-instantiated and does so. Of course, these policies must be handled in a flexible way. Often, factories need the help of histories at the time a service is created or an existing service is reactivated.

### 5.2.5 Reference of Essential Classes

The implementation of the REBECA notification infrastructure consists of classes and interfaces. This section shortly describes the most important classes in form of an enumerative reference. The relation among the classes is visualized by a set of UML (unified modeling language) [84] diagrams. The usage of the infrastructure is discussed in Section 5.3.

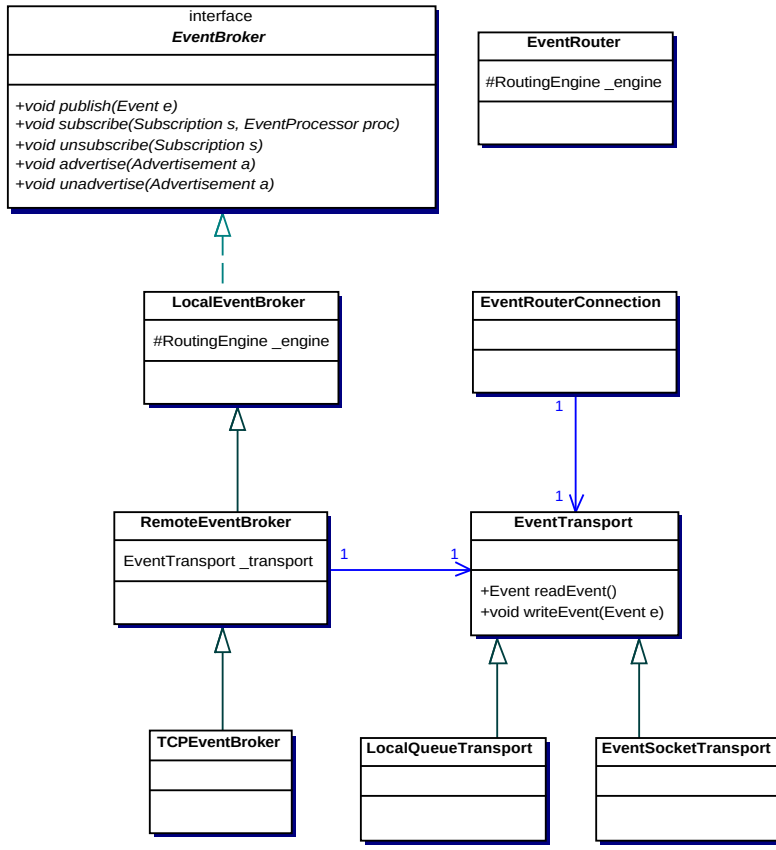



Figure 5.1: Infrastructure Classes

### `Event`

The `Event` class (see Fig. 5.2) is the base class of all notifications. `AdminEvent` is an important subclass of the `Event` class that is used to communicate control messages inside of the infrastructure.

### `Filter`

The `Filter` class (see Fig. 5.3) is the abstract base class of all filters. Each filter has a unique id which is used by the routing algorithms. All sub-classes of `Filter` must implement the `boolean match(Event e)` method.

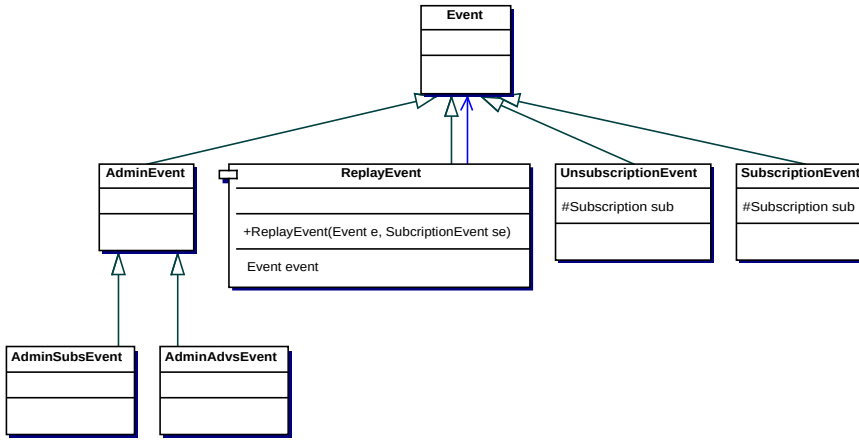



Figure 5.2: Important Event Classes

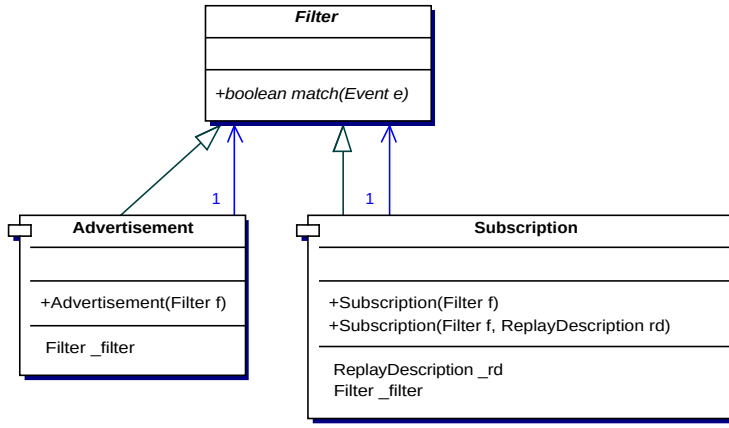



Figure 5.3: Filtering Classes

#### Subscription

The Subscription class extends Filter and is the base class of all subscriptions. A new Subscription is created from a given Filter and an optional ReplayDescription (see Fig. 5.3).

#### Advertisement

Advertisement is a subclass of Filter that serves as the base class of all advertisements (see Fig. 5.3). A new Advertisement is constructed from a given Filter.

#### EventRouter

The functionality of an event router is implemented by the EventRouter class (see Fig. 5.1). A RoutingEngine is responsible for notification forwarding and routing table actualization. An EventRouter has a RoutingEngine and a ServerSocket. The ServerSocket is used to accept connections from EventBrokers and EventRouters. If a connection is established, an EventRouterConnection is created that handles all communication regarding this connection. At creation time, an EventRouter can connect to another EventRouter which is specified by its IP address and port number. Currently, no other methods to establish connections are supported. Although, it is possible to have multiple EventRouter instances in a single JAVA VM this is in general not useful.

#### RoutingEngine

RoutingEngine is the base class of all routing algorithms (see Fig. 5.4). Each RoutingEngine has two RoutingTables which are the subscription routing table and the advertisement routing table, respectively. While the former is used to route notifications from producers to consumers, the latter is used to route subscriptions from consumers to producers.

A RoutingEngine processes incoming notifications serially and in FIFO-order. If the processed notification is an AdminEvent, the RoutingEngine updates its routing tables accordingly and hands out proper AdminEvents to some of the connected EventRouterConnections according to the applied routing algorithm. If the notification is not an AdminEvent, the RoutingEngine hands it out to the EventRouterConnections with matching subscriptions.

The RoutingEngine class is extended by the following classes that implement the corresponding routing algorithms: Flooding, SimpleRouting, IdentityRouting, CoveringRouting, and MergingRouting (see Fig. 5.4).

#### EventRouterConnection

An EventRouterConnection (see Fig. 5.1) encapsulates the communication between an EventRouter and an EventBroker or another EventRouter. An

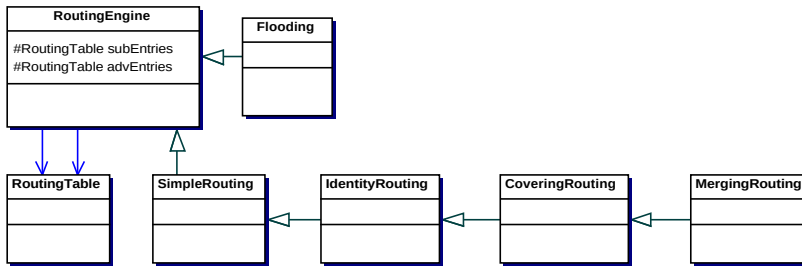



Figure 5.4: Routing Algorithm Classes

`EventRouterConnection` has a pointer to the `RoutingEngine` of its associated `EventRouter`. To this `RoutingEngine` an `EventRouterConnection` hands out incoming notifications for further processing. Similar, the `RoutingEngine` calls the `void process(Event e)` method of an `EventRouterConnection` to send out notifications. An `EventRouterConnection` has also an `EventTransport` which is responsible for carrying out the actual low-level communication.

#### `EventBroker`

`EventBroker`s (see Fig. 5.1) are the access points of the publish/subscribe system. Hence, the `EventBroker` interface offers an API with the usual semantics:

```
public interface EventBroker {
    void publish(Event e);
    void subscribe(Subscription s, EventProcessor p);
    void unsubscribe(Subscription s);
    void advertise(Advertisement a);
    void unadvertise(Advertisement a);
}
```

#### `LocalEventBroker`

A `LocalEventBroker` is an `EventBroker` that is not connected to any `EventRouter` (see Fig. 5.1). Therefore, this class alone is only useful to realize a local publish/subscribe system. A `LocalEventBroker` has a `RoutingEngine` that implements the applied routing algorithm.

#### `RemoteEventBroker`

A `RemoteEventBroker` (see Fig. 5.1) is a `LocalEventBroker` that additionally establishes a connection to an `EventRouter`. Over this connection `Events` are exchanged according to the used routing algorithm. The connection itself is implemented by an `EventTransport`.

#### TCPEventBroker

A TCPEventBroker (see Fig. 5.1) is a RemoteEventBroker that uses an EventSocketTransport to connect to an EventRouter specified by its IP address and port number.

#### DefaultEventBroker

The DefaultEventBroker is an EventBroker that encapsulates an instance of an EventBroker. It is usually initialized by DefaultEventBroker.init(args); in the main method.

#### EventTransport

An EventTransport (see Fig. 5.1) realizes a bidirectional connection over which Event instances are exchanged:

```
public abstract class EventTransport {
   public Event readEvent()}
   public void  writeEvent(Event e)}
}
```

Currently, there are two implementations available which are described in the following.

#### EventSocketTransport

An EventSocketTransport (see Fig. 5.1) is an EventTransport that uses a Socket to communicate Events as serialized JAVA objects by an ObjectInputStream and an ObjectOutputStream.

#### EventQueueTransport

An EventQueueTransport (see Fig. 5.1) is an EventTransport that uses a local queue to communicate events. Therefore, it can only be used to connect partners that reside in the same JAVA VM.

#### RoutingTable

A RoutingTable (see Fig. 5.4) implements the functionality of a routing table that is needed by the filtering-based routing algorithms. Essentially, a RoutingTable consists of a set of RoutingEntry instances that can be added, removed, and queried. For example, the set of destinations that match a given notification can be determined by the Set getDestinations(Event e) method.

### EventProcessor

The `EventProcessor` interface (see Fig. 5.5) is essentially a callback that is implemented by a consumer to define an endpoint for notification delivery. If a matching notification arrives, the method `process(Event e)` is called with the notification as parameter.

```
public interface EventProcessor {
    public void process(Event e);
}
```

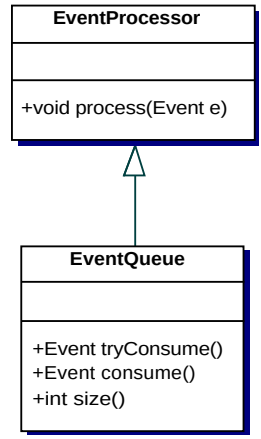



Figure 5.5: Event Processor Classes

### EventQueue

An `EventQueue` (see Fig. 5.5) is an `EventProcessor` that queues incoming notifications. The topmost notification can be extracted either by the blocking `Event consume()` or by the non-blocking `Event tryConsume()` method.

### SubscriptionEvent

If a consumer issues a new subscription, automatically a `SubscriptionEvent` (see Fig. 5.2) is generated that contains the corresponding `Subscription`. Currently, this is done by the corresponding `LocalEventBroker`.

### UnsubscriptionEvent

If a consumer cancels an active subscription, automatically an `UnsubscriptionEvent` (see Fig. 5.2) is generated that contains the corresponding `Subscription`.

Currently, this is done by the corresponding `LocalEventBroker`.

#### `SubscriptionSubscription`

A `SubscriptionSubscription` is a `Subscription` that matches `SubscriptionEvent`s whose embedded `Subscription` overlaps with a given `Filter`. Additionally, it can be tested whether the optional `ReplayDescription` of the subscription overlaps with a given `ReplayDescription`, too. `UnsubscriptionSubscription`s are used by histories and factories.

#### `UnsubscriptionSubscription`

An `UnsubscriptionSubscription` is a `Subscription` that matches `UnsubscriptionEvent`s whose embedded `Subscription` overlaps with a given `Filter`. `SubscriptionSubscription`s are used by factories.

#### `ReplayEvent`

A `ReplayEvent` (see Fig. 5.2) is published by histories in reaction to the receipt of a `SubscriptionEvent`. It is created from a given `Event` and the received `SubscriptionEvent`. The infrastructure ensures that a `ReplayEvent` is only delivered to the consumer that has subscribed to the subscription that is embedded in the `SubscriptionEvent`.

## 5.3 Using the Infrastructure

In this section it is shortly described how one can use the REBECA notification infrastructure to build a simple distributed application that is based on publish/subscribe.

Suppose we have potential consumers of `ExampleEvent`s that are published by a respective producer. Moreover, the producer should only be active if there is at least one consumer subscribed, and the last ten occurrences of published `ExampleEvent`s should be delivered to a consumer that has newly subscribed. This application consists of a factory that manages the producer, a history, and at least a single consumer. In the following, the classes that are necessary to realize this example are described and it is depicted how the example can be run. All mentioned classes are part of the REBECA distribution and can be found in the `Events.example` package.

### 5.3.1 Implementing a Sub-Class of `Event`

The `ExampleEvent` class extends the `Event` class and provides a method to print some information (see Fig. 5.6).

```
public class ExampleEvent extends Event {
    ObjectId _id;

    public ExampleEvent() {
        _id=new ObjectId();
    }

    public String toString() {
        return "Events.example.ExampleEvent: {" + _id + "}";
    }
}
```

Figure 5.6: A simple user-defined event

### 5.3.2 Implementing a Consumer

The `ExampleConsumer` class (see Fig. 5.7) realizes a consumer that is interested in `ExampleEvent`s. It implements the `EventProcessor` interface by providing an implementation of the `public void process(Event e)` method. This endpoint is then bound to a subscription that matches `ExampleEvent`s. An `ExampleConsumer` can be started by:

```
java Events.example.ExampleConsumer
```

```
public class ExampleConsumer implements EventProcessor {

    private EventBroker _broker;
    private Subscription _sub;

    public ExampleConsumer() {
        _broker = DefaultEventBroker.getEventBroker();
        _sub = new Subscription(
                  new EventClassFilter(ExampleEvent.class));
        _broker.subscribe(_sub, this);
    }

    public void process(Event e) {
        System.out.println(e.toString());
    }

    public static void main(String args[]) {
        DefaultEventBroker.init(args);
        ExampleConsumer consumer = new ExampleConsumer();
    }
}
```

Figure 5.7: A simple event consumer

### 5.3.3 Implementing a Producer

The `ExampleProducer` class (see Fig. 5.8) extends the `DefaultEventProducer` class. First, it issues an advertisement that indicates that it will publish `ExampleEvents`, and after that it publishes an `ExampleEvent` every 3 seconds. This is done in a separate thread to avoid blocking the calling thread. An `ExampleProducer` can be started by invoking

```
java Events.example.ExampleProducer
```

in a shell.

```
public class ExampleProducer extends DefaultEventProducer {
    boolean _b=true;
    Advertisement _adv;

    public ExampleProducer() {
        super.setEventBroker(
           DefaultEventBroker.getEventBroker());
        _adv = new Advertisement(
            new EventClassFilter(ExampleEvent.class));
        advertise(_adv);

        Thread th = new Thread() {
            public void run() {
                while (_b) {
                    publish(new ExampleEvent());
                    try {
                        Thread.sleep(3000);
                    }
                    catch (InterruptedException e) {}
                }
            }
        };
        th.start();
    }

    public void shutdown() {
      _b=false;
      unadvertise(_adv);
    }

    public static void main(String args[]) {
        DefaultEventBroker.init(args);
        ExampleProducer producer = new ExampleProducer();
    }
}
```

Figure 5.8: A simple event producer

### 5.3.4 Implementing a History

An `ExampleHistory` (see Fig. 5.9) is a simple history that records and replays `ExampleEvent`s. More precisely, it subscribes to `ExampleEvent`s and continually keeps track of the last ten occurrences it has received. Furthermore, it subscribes to `SubscriptionEvent`s whose embedded subscription matches `ExampleEvent`s. If it receives such a `SubscriptionEvent`, it publishes each of the recorded events as `ReplayEvent`. The code of the example also shows how to use an anonymous subclass of `EventProcessor` to dispatch matching event to a specific method. An `ExampleHistory` can be started by:

```
java Events.example.ExampleHistory
```

### 5.3.5 Implementing a Factory

The `ExampleFactory` (see Fig. 5.10) is a simple factory managing a single `ExampleProducer`. The factory subscribes to `SubscriptionEvent`s and `UnsubscriptionEvent`s that deal with `ExampleEvent`s. If the factory receives a `SubscriptionEvent`, it activates a producer that publishes `ExampleEvent`s if it is not already active, and adds the id of the specific subscription to its set of active subscriptions. If it receives an `UnsubscriptionEvent`, it deletes the specific subscription id from its set of active subscriptions and deactivates the producer if there are no subscribers left. The code of this example also shows how to use the `instanceof` operator for event demultiplexing. An `ExampleFactory` can be started by:

```
java Events.example.ExampleFactory
```

### 5.3.6 Starting an `EventRouter`

An `EventRouter` is created with the following command:

```
java Events.EventRouter [-lport <portnumber>]
     [-rport <portnumer>] [-rhost <hostname>]
```

The `lport` option specifies on which port the router listens for incoming connections. If omitted the router listens on port 8020. If the `rport` or the `rhost` option is present, the `EventRouter` tries to connect itself to a parent `EventRouter` which resides on the specified host (or the local host if the `rhost` option is omitted) and listens for incoming connections on the given port (or on port 8020 if the `rport` option is omitted).

### 5.3.7 Running the Example

In the following, it is described how to run our simple example. First, we have to select the routing algorithm that is used. Currently, this is done by editing

```
public class ExampleHistory {

    EventBroker _broker;
    Subscription _sub;
    SubscriptionSubscription _subSub;
    int _n=0;
    Event[] history=new Event[10];

    public ExampleHistory() {
        _broker = DefaultEventBroker.getEventBroker();

        _sub = new Subscription(
                  new EventClassFilter(ExampleEvent.class));
        _broker.subscribe(_sub, new EventProcessor() {
            public void process(Event e) {
                processExampleEvent((ExampleEvent)e);
            }
        });

        SubscriptionSubscription _subSub =
            new SubscriptionSubscription(
               new EventClassFilter(ExampleEvent.class));
        _broker.subscribe(_subSub, new EventProcessor() {
            public void process(Event e) {
                processSubEvent((SubscriptionEvent)e);
            }
        });
    }

    protected void processExampleEvent(ExampleEvent ee) {
        history[_n%10]=ee;
        _n++;
    }

    protected void processSubEvent(SubscriptionEvent se) {
        ReplayEvent re;
        ExampleEvent ee;
        for (int i=0;i<10;i++) {
            ee=(ExampleEvent)history[i];
            if (ee!=null) {
                re=new ReplayEvent(ee,se);
                _broker.publish(re);
            }
        }
    }

    public static void main(String args[]) {
        DefaultEventBroker.init(args);
        ExampleHistory history = new ExampleHistory();
    }
}
```

Figure 5.9: A simple event history

```
public class ExampleFactory implements EventProcessor {
    EventBroker _broker;
    ExampleProducer _producer;
    SubscriptionSubscription _subSub;
    UnsubscriptionSubscription _unsubSub;
    HashSet _subs=new HashSet();

    public ExampleFactory() {
        _broker = DefaultEventBroker.getEventBroker();
        _subSub = new SubscriptionSubscription(
           new EventClassFilter(ExampleEvent.class));
        _broker.subscribe(_subSub, this);
        _unsubSub = new UnsubscriptionSubscription(
           new EventClassFilter(ExampleEvent.class));
        _broker.subscribe(_unsubSub, this);
    }

    protected void addSubscription(SubscriptionEvent se) {
        if (_producer==null) {
             System.out.println("ExampleFactory: Starting
                                 Service");
            _producer=new ExampleProducer();
        }
        _subs.add(se.getSubscription().getId());
    }

    protected void removeSubscription(UnsubscriptionEvent ue) {
        _subs.remove(ue.getSubscription().getId());
        if (_subs.size()==0 && _producer!=null) {
            System.out.println("ExampleFactory: Stopping
                                Service");
            _producer.shutdown();
            _producer=null;
        }
    }

    public void process(Event e) {
        if (e instanceof SubscriptionEvent) {
            addSubscription((SubscriptionEvent)e);
            return;
        }
        if (e instanceof UnsubscriptionEvent) {
            removeSubscription((UnsubscriptionEvent)e);
            return;
        }
    }

    public static void main(String args[]) {
        DefaultEventBroker.init(args);
        ExampleFactory ef=new ExampleFactory();
    }

}
```

Figure 5.10: A simple service factory

and compiling the `Config` class. It can be chosen among flooding and the following filter-based routing algorithms: simple, identity-based, covering-based, and merging-based routing, each with or without advertisements.

After selecting the routing algorithm, we have to start an `EventRouter` which is done by the command:

```
java Events.EventRouter
```

Subsequently, we start the factory

```
java Events.example.ExampleFactory
```

and the history

```
java Events.example.ExampleHistory
```

Finally, we start the consumer by:

```
java Events.example.ExampleConsumer
```

The consumer starts to output `ExampleEvent`s after a short delay. If we start a second consumer after some time, it receives the last (at most) 10 instances of `ExampleEvent`s that have been recorded by the history, too. Of course, it is also possible to directly start the producer instead of using the factory:

```
java Events.example.ExampleProducer
```

## 5.4 Example Applications

In this section two example applications are described, a stock trading application and an infrastructure for self-actualizing web pages. The implemented applications are typical examples of information-driven applications that benefit from an event-based approach. Each of the applications has been implemented in JAVA by exclusively using the REBECA event notification infrastructure. In the following, the example applications are described in detail.

### 5.4.1 Self-Actualizing Web Pages

In this subsection an infrastructure for self-actualizing web pages is described. A demo of the infrastructure can be accessed on-line on the Internet [39].

Self-actualizing web pages include two important aspects: Firstly, the content of static web pages can be actualized in reaction to the occurrence of specific notifications, and secondly, notifications can be used to trigger clients to update the pages that they display. These two parts are called the *server-side* and the *client-side*, respectively. They are described in detail below.

### Server-Side

On the server-side, static web pages are updated in reaction to the occurrence of specific events. For example, a page whose content depends on the current price of a stock should be updated if the price changes. This can easily be realized by using the REBECA infrastructure: The component that is responsible for updating certain web pages subscribes to those notifications on which the pages depend and updates the affected pages accordingly if a notification arrives.

### Client-Side

In the preceding subsection, the server-side of self-actualizing web pages has been discussed. To update the pages on the web server does not cause the browser of a web client to update the pages they display. A simple but inefficient approach is to use the HTML `REFRESH` directive to periodically reretrieve the displayed page. To update the displayed page efficiently, a mechanism to notify web clients about the change is needed. There are three basic approaches to achieve that a client always displays the current version of a web page:

- **Pull.** In this scenario the client periodically issues page requests in order to get the current web page.
- **Push.** Here, the client is asynchronously notified about page updates. The complete page or equivalent information to construct it, e.g., from the previous version, is embedded in the notification.
- **Push-Triggered Pull.** The client is asynchronously notified about a page update, but the page is retrieved by a conventional HTTP page request, i.e., a pull.

To realize the push and the push-triggered pull approach, REBECA has been integrated with standard Internet browsers: A lean and invisible JAVA applet is embedded into the HTML code of the web page. If a page is loaded into a browser window, the applet is activated. In this case it connects to a remote event broker and subscribes to interesting events. To which host and port the applet should connect and which events are of interest is configured by a set of `<PARA>` tags. After issuing the subscription(s), all matching events are subsequently delivered to the client. Triggered by the receipt of such a notification either the displayed page is directly updated (push) or a new version of the page is retrieved (push-triggered pull). The next two paragraphs describe how this is achieved.

**Push.** When using the push-based approach, only a single page request occurs to retrieve the initial version of the page to be displayed. Subsequent updates of the page or new pages to be displayed are extracted from the received notifications. For example, either the whole HTML code of the page can be embedded in the notification or the embedded information can be used by the client to construct the page. In the latter case, the client must know how to derive the page.

The pure push approach is realized by calling Javascript from JAVA by using the `netscape.javascript` package that is a part of LiveConnect that is shipped with newer version of the SUN JRE or JDK. As underlying JAVA platform the JAVA Plug-in from SUN is used because this plug-in is supported by current versions of both Netscape Communicator and Microsoft Internet Explorer.

**Push-Triggered Pull.** Here, the notification that has arrived triggers the client to pull a new version or a new page from a web server by a standard HTTP request. For example, the URL of the page to be loaded can be extracted from the received notification. The actual request is achieved by calling the `showDocument()` method on the class `Applet` with the appropriate URL. One problem with push-triggered pull is that many clients may by notified at nearly the same time. Therefore, these clients will almost simultaneously request the same page. This may overload the web server and cause longer response times. A simple solution to this that better shapes the traffic is to use client specific delays that are applied before retrieving the new page. A more advanced approach is to use snoopy caches that act as web proxies and offer the current versions of the web pages to be retrieved. Which proxy should be used is encoded in the URL that is delivered to the client.

### 5.4.2 Stock Trading Platform

As second and more complex example, a stock trading platform has been implemented which allows users to monitor stocks in real-time without requiring continuous attention. The fact that distinguishes its realization among others is that it is engineered in a completely event-driven way.

The stock trading applications consists of three main parts which are a trigger list (see Fig. 5.11), a portfolio (see Fig. 5.12), and charts (see Fig. 5.13). All information that is displayed inside of their corresponding windows is real-time and self-actualizing. Commands to buy and sell stocks can be input on a command line prompt. Moreover, the user can be notified by a SMS (short message service) that is sent to its mobile phone if a certain situation occurs.

In the following, the functionality of the main parts of the application is explained. After that, the overall architecture of the application is described.

#### Trigger List

The most important part of the stock trading application is a self-actualizing real-time trigger list, which is depicted in Fig. 5.11. This comfortable tool allows a user to monitor a large set of stocks without requiring continuous attention by defining two kinds of triggers which are called absolute and relative limits, respectively. An *absolute limit* monitors whether the price of a certain stock is within a given range. Therefore, it is *triggered* if the price of the respective stock is either below or above this range. A *relative limit* monitors whether the price of a given stock has increased or decreased a given number of percents within a

given time interval. Here, the user can choose among a set of predefined relative limits. If at least one absolute limit is triggered, the program beeps twice. Otherwise, the program beeps once if at least one relative limit is triggered.

In Fig. 5.11 a screen-shot of a trigger list is shown. The triggered absolute limits are shown in the upper part of the window, the non-triggered absolute limits are depicted in the middle, and the triggered relative limits are shown in the lower part of the window. For both types of limits, the first two columns refer to the name of the respective stock, its id, and its current price, respectively. For an absolute limit, the fourth and the fifth column refer to the lower and the upper limit of the range, respectively. For a relative limit, on the other hand, the fourth and the following columns refer to the relative change in percent, the minimum price, the maximum price, and the applied rule.

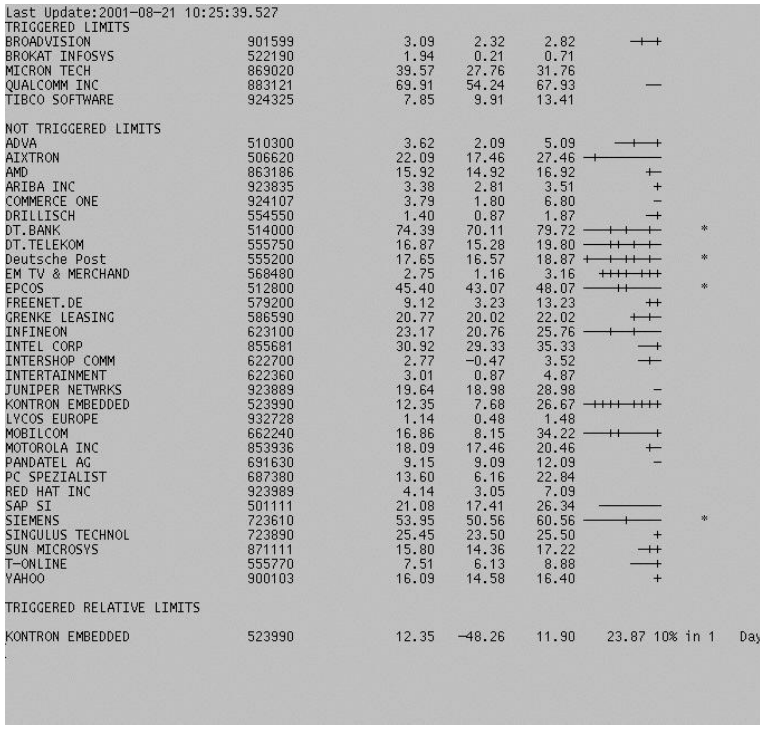



Figure 5.11: A self-actualizing real-time trigger list

### Portfolio

The second important feature of the stock trading application is a self-actualizing real-time portfolio which depicts detailed information about the stocks that a user currently holds (see Fig. 5.12). For each stock the quantity the user holds,

the purchase price of the stock, and its current price is shown. Besides this, the purchase price of each position and its current value are also given. Finally, the overall profit (or loss) is shown at the bottom of the portfolio.

Last Update: 2001-08-21 10:23:41.929

| warrantname | wkn | | | | | |
|---|---|---|---|---|---|---|
| ADVA | 510300 | 100 | 15.00 | 3.62 | 1500.00 | 362.00 −1138.00 |
| AMD | 863186 | 60 | 33.50 | 15.92 | 2010.00 | 955.20 −1054.80 |
| ARIBA INC | 923835 | 50 | 16.70 | 3.38 | 835.00 | 169.00 −666.00 |
| BROADVISION | 901599 | 100 | 7.46 | 3.09 | 746.00 | 309.00 −437.00 |

Profit: −3295.80 Euro
−6446.12 DM

Figure 5.12: A self-actualizing real-time portfolio

### Charts

The stock trading application also provides self-actualizing real-time charts that visualize the price of a certain stock over time (see Fig. 5.13). If a new real-time chart is created, the evolution of the stock price of, for example, the last two hours is shown, and if a new quote arrives, the diagram is updated instantly.

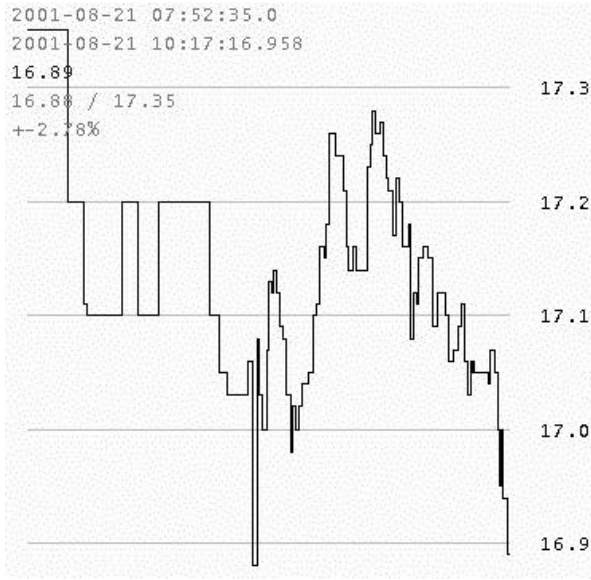



Figure 5.13: A self-actualizing real-time chart

**Overall Architecture**

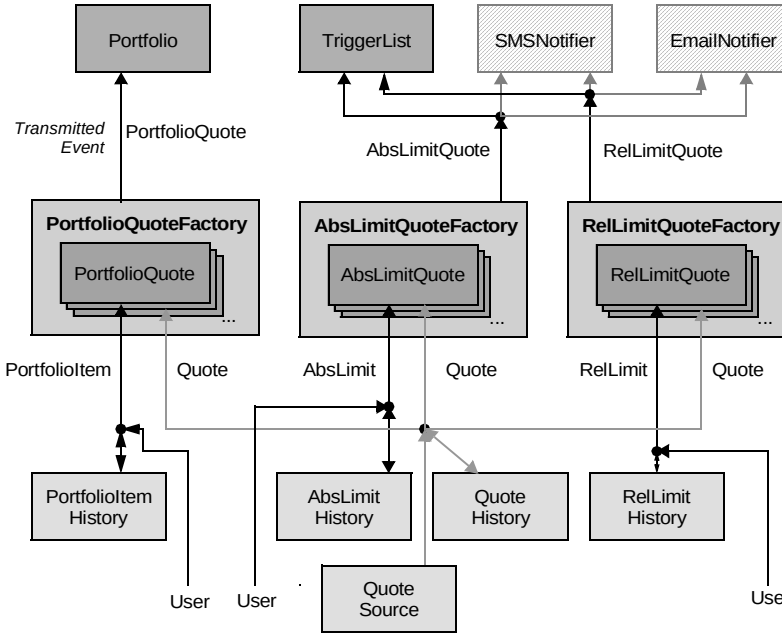



Figure 5.14: The architecture of the stock trading application

In order to get the current stock quotes, every minute a set of web pages is retrieved from which the quotes are extracted. If the price of a stock has changed, a corresponding `QuoteEvent` is published. To avoid a burst of quotes occurring every minute, all quotes of a certain period are uniformly distributed over this period.

Fig. 5.14 depicts the overall architecture of the stock trading application. To realize the functionality in a completely event-driven way, a set of histories and factories is used. For example, the `QuoteHistory` records `QuoteEvent`s that are needed by the real-time charts, as well as by the absolute and the relative limits.

## 5.5 Related Work

In this section implementations of other notification services are related to the REBECA notification infrastructure.

### 5.5.1 SIENA

Currently, SIENA [17, 18, 20, 21, 22, 23, 96, 117] offers two prototypes, one written in C++ that implements the peer-to-peer variant and a second one implemented in JAVA that realizes the hierarchical version of the covering-based routing algorithm. Currently, the use of advertisements is not supported. Moreover, the C++ version is no longer supported and incompatible with the JAVA version. This means that, in contrast to REBECA, SIENA currently does not allow different routing algorithms to be compared. On the other hand, SIENA includes a mechanism to efficiently recognize sequences of events that is currently missing in REBECA. At the moment, SIENA is limited to name/value pairs, because this data/filter model is hard-coded in the base classes of, for example, the notifications. Moreover, SIENA uses its own restricted data format for serialization/deserialization. In contrast to this, it is possible by using REBECA to experiment with various data/filter models, serialization/deserialization mechanisms, and routing algorithms. The SIENA prototypes are publicly available, but currently no implemented example applications exist.

### 5.5.2 JEDI

The current implementation of JEDI [13, 14, 28, 29, 30] uses the hierarchical version of covering-based routing described by Carzaniga [17]. Other routing algorithms are not looked at. Moreover, the expressiveness of their data/filter model is rather limited because notifications and filters are ordered sets of strings and matching is solely based on equality and prefix tests. Cugola, Di Nitto, and Fuggetta also depicted how JEDI can be used to engineer an application called OPSS (ORCHESTRA Process Support System) which is essentially a WFMS (Work Flow Management System) [29].

### 5.5.3 Gryphon

The Gryphon prototype [1, 9, 8, 85, 59, 86, 103] developed at the IBM T. J. Watson Research Center offers an implementation of the JAVA Message Service (JMS) API [111] through. Gryphon uses simple routing without advertisements and its matching algorithm (which is based on a parallel search tree) is fast but restricted to simple tests. Alternative routing algorithms are not considered, and currently, no example applications are available. Moreover, the prototype is currently not publicly available.

### 5.5.4 ELVIN

ELVIN is a content-based publish/subscribe middleware developed at the Distributed Systems Technology Center (DSTC) [44, 100, 101]. It uses a subscription language that is very expressive but probably inhibits routing optimizations. ELVIN exploits a federation of event brokers, but unfortunately, the routing algorithms are not described. Interestingly, a concept called *quenching* is offered

by Elvin which allows producers to detect whether there are no consumers currently subscribed to the notifications they produce. Affected producers can cease production of those events in this case. This approach is somewhat related to the factory concept of Rebeca. Unfortunately, it is not described how quenching in Elvin is achieved. The Elvin project is rather advanced and a couple of example applications exist. Both, the prototype and the example applications are publicly available.

### 5.5.5 Hermes

Recently, Bacon and Pietzuch presented Hermes [90, 91], a notification service that implements content-based notification delivery on top of a peer-to-peer overlay network. The prototype uses rendezvous nodes for event types and supports advertisements. Hermes is implemented in Java and communication between the components takes place by passing XML-defined messages. XML Schema is used to define message formats and event types allowing for a rich type system including user-defined types. In order to hide the complexity of XML messages from the client, a mapping between XML Schema and Java is performed.

## 5.6 Discussion

The implemented content-based notification infrastructure Rebeca is a contribution to the area of publish/subscribe systems for several reasons. Firstly, and in contrast to most other work, Rebeca offers a whole set of routing algorithms. This allows routing algorithms to be compared to each other in a uniform environment. The implemented routing algorithms include flooding and diverse filter-based routing algorithms with and without advertisements. Moreover, new routing algorithms can be added easily.

Secondly, instead of carrying out simulations the prototype also served as the basis for the experiments that are described in the next chapter. This increases the validity of the results compared to simulation based results (see Chapter 6). Moreover, Rebeca provides basic support for replaying past events and service factories. The necessity of these concepts became evident while implementing the stock trading application which uses histories and factories.

Finally, the implemented example applications, a stock trading platform and an infrastructure for self-actualizing web pages show that reasonable applications can be realized by using the implemented notification infrastructure.

# Chapter 6

# Experimental Results

## Contents

## 6.1 Introduction

This chapter presents an evaluation of the implemented filter-based routing algorithms, giving a detailed insight into their behavior. The evaluation focuses on the inherent characteristics of routing algorithms (routing table sizes and filter forwarding overhead) instead of system-specific parameters (CPU load etc.). Moreover, it is based on an implemented prototype (cf. Chapter 5) instead of simulations, increasing the validity of the results. Besides the comparison of the implemented routing algorithms, the effects of locality among the interests of consumers and imperfection when carrying out filter merging are also investigated. At appropriate places analytical results are also derived and compared to the outcome of the experiments to corroborate their validity.

To the author's knowledge, the evaluation described here is the first that presents a detailed investigation of the routing table sizes and the filter forwarding overhead. These fundamental characteristics of content-based routing algorithms have been neglected by previous evaluations (e.g., in the context of SIENA [17, 22], JEDI [13, 14], and Gryphon [8, 85]); instead they mainly concentrated on system-specific parameters (e.g., the load induced on brokers) which depend on further assumptions, like the available network bandwidth and processing power (and their costs), as well as the used matching algorithm. These magnitudes should be reinvestigated after the inherent tradeoffs and parameter relations of content-based routing algorithms are well understood.

The derived results offer new and detailed insights into the behavior of content-based routing algorithms: First, using advanced routing algorithms in large-scale publish/subscribe systems can be considered *valuable*. They significantly reduce both the routing table sizes and the filter forwarding overhead. Second, the use of advertisements can considerably improve the scalability. Third, advanced routing algorithms operate efficiently in more dynamic environments than was previously thought. Fourth, the good behavior of the algorithms even improves if the interests of the consumers are not evenly distributed, which can be expected in practice. Finally, the evaluation of imperfect merging shows that it further reduces the routing table sizes. Altogether, the evaluation shows that advanced content-based routing algorithms extend the application domains in which publish/subscribe can be applied significantly.

This chapter is structured as follows: Sect. 6.2 describes the general setup of the experiments. After that, the experiments themselves and their results are explained in detail. Sect. 6.3 and 6.4 investigate the routing table sizes and the

filter forwarding overhead, respectively. The effects of locality and the evaluation of imperfect merging is presented in Sect. 6.5. Finally, the results of this chapter are related to previous work (see Sect. 6.6).

## 6.2 General Setup

This section describes the general setup of the experiments which were performed in the context of a stock trading information system (see Section 5.4.2) where clients can subscribe and unsubscribe to certain stock information and a central publisher of this information (e.g., a stock exchange) exists. Besides the used routing algorithm, the results are influenced by the characteristics of the broker topology, the consumers, and the producers. Investigating the relations among all parameters that are involved is beyond the scope of this work. Therefore, this evaluation varies some main parameters (e.g., the number of active subscriptions $x$) and assumes a simple but meaningful scenario in which the other parameters remain constant (see Table 6.1). In the following subsections, the setup with respect to the mentioned parameters is described in more detail.

| | |
|---|---|
| Number of consumers per local broker | $1 - 200$ |
| Number of subscriptions per consumer | 10 |
| Number of stocks | 1000 |
| Number of notification sources | 1 |
| Number of event routers | 40 |
| Number of local event brokers | 67 |
| Number of neighbor broker | fix |
| Number of hierarchy levels | 5 |
| Distribution of clients to brokers | random |
| Distribution of stocks to clients | random |
| Degree of locality | none |

Table 6.1: Fixed and varied parameters of the setup

### 6.2.1 Broker Topology

The broker topology that is used has a major impact on any experiment. The main parameters that characterize a broker topology are:

- the number of brokers,
- the number of neighbor brokers which may be constant or vary,
- the existence or absence of connectivity cycles, and
- the diameter of the network which is the longest path connecting two arbitrary brokers.

In line with most previous work, this work concentrates on a hierarchical, symmetrical, and acyclic, i.e. tree-like, topology. To use a symmetrical topology facilitates the interpretation of the findings, and the hierarchical structure is justified by the hierarchical structure of real networks, like the Internet. Future work will include topologies with cycles to diminish single points of failure. Due to implementation reasons, brokers which have local clients and those that have not are distinguished. The former are called *local brokers* while the latter are called *routers*. A local broker is connected to exactly one router.

The investigated topology has 5 levels of brokers. Starting from a single router, called the *root router*, all routers except the leaves are connected to 3 subordinate routers. To each non-leaf router a single local broker is connected, while to each leaf router two local brokers are connected. Therefore, the used topology consists of 40 routers and 67 local brokers. In Fig. 6.1 a topology of the same type with only 4 levels is shown. The circles refer to routers while the squares refer to local brokers.

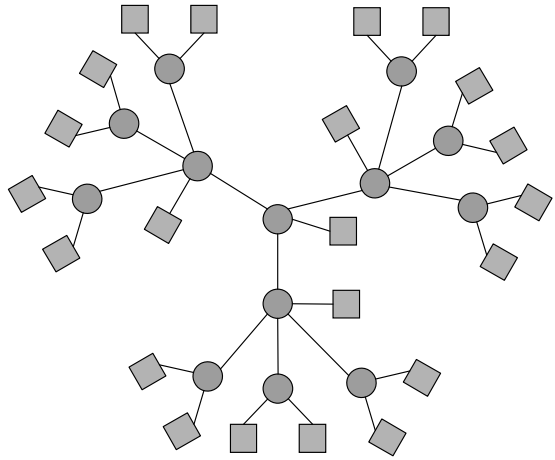


Figure 6.1: Broker topology with 4 levels.

### 6.2.2 Characteristics of the consumers

The characteristics of its consumers have a large impact on the performance of a publish/subscribe system. Here, the main parameters are

- the number of consumers,
- the number, type, and distribution of the subscriptions,

- the assignment of the subscriptions to the consumers (e.g. locality of interests),
- the assignment of the consumers to the local brokers, and
- the rate of subscribing and unsubscribing.

In the experiments the consumers are equally distributed among the local brokers. It is distinguished between *active* clients that are subscribed, i.e., that have issued their subscriptions, and *inactive* clients that have no active subscriptions. If the size of the routing tables is investigated, all clients are active, while, when investigating the filter forwarding overhead, on average only half of them are active; this has to be kept in mind when interpreting the figures. In the latter case, a large number of iterations is carried out: Each time an arbitrary client is picked and its state is toggled from active to inactive or vice versa.

Each consumer has 10 random but distinct subscriptions. As the basis for the investigations quote subscriptions, which are borrowed from the implemented stock trading application (see Sect. 5.4.2), are used. A *quote subscription* matches all quotes of a specific stock that is specified by its unique symbol. For simplicity it is assumed that the number of different stocks $m$ equals 1000. This choice may seem to be arbitrary, but indeed, the used subscriptions cover a wider range of subscription types than it seems at a first glance. For example, if $m$ is different, findings can be derived by scaling. Moreover, if subscriptions have more than one attribute, results can be obtained by assuming that $m$ equals the product of the number of distinct values each attribute can have. For example, subscriptions with three attributes where each attribute has 10 possible values would lead to the same results. In order to compare covering-based with merging based-routing additionally *quote interval subscriptions* are applied which are slightly more complex. This type of subscription matches all quotes of a certain stock whose price is within a certain range, e.g. $\$10 - \$20$. The rational for this distinction will become clear later.

### 6.2.3 Characteristics of the producers

The characteristics of the producers also influence the behavior of a publish/subscribe systems. The main parameters of a set of producers are

- the absolute number of producers,
- the number, type, and distribution of the advertisements,
- the assignment of the advertisements to the producers,
- the assignment of the producers to the local brokers, and
- the rate of advertising and unadvertising.

In the experiments the root daemon serves as a single source of notifications, denoted by $S$ . At system start-up this 'producer' issues an advertisement that is never revoked and overlaps with all possible subscriptions. This simple scenario is sufficient to deduce the effects of static advertisements. Facts about the advertisement routing tables that would arise if more than one producer is present can also be inferred from the experiments because the advertisement routing tables are managed by the same algorithms as subscription routing tables. Of course, the factor by which the use of advertisements reduce the size of the subscription routing tables in such a scenario depends on the characteristics of the advertisements, e.g., their number and the degree of overlap. In the worst case, if at each local broker a producer issues an advertisement that overlaps with all subscriptions, advertisements would be completely useless. Fortunately, the probability for this case is very low. Instead, it can be expected that advertisements are only partly overlapping.

Scenarios with dynamic advertisements are out of the scope of this thesis and are left for future work. Indeed, the filter forwarding overhead induced by dynamic advertisements is difficult to predict and depends, for example, on the rate of advertising and unadvertising, but it seems reasonable to assume that compared to subscriptions this rate will be rather low.

## 6.3 Routing Tables Sizes

The first key characteristic of any routing algorithm is the evolvement of the size of the routing tables with respect to the number of active subscriptions. The results of these experiments are described in the subsequent subsections, separately for each individual routing algorithm. The routing table size is an indication of the *space complexity* of the algorithms in the "size" of the system. Note that only remote routing entries are counted here to concentrate on the behavior of filtering-based routing algorithms. The number of local routing entries in the system equals the number of active subscriptions; they also exist if flooding is performed.

### 6.3.1 Simple Routing

If simple content-based routing without advertisements is used, the size of a routing table (local and remote entries) is equal to the number of active subscriptions $x$. In particular, the characteristics of the subscriptions have no impact on the size of the routing tables. The sum of all remote routing table entries $\Sigma|R|$ is given by $(|V| - 1) \cdot x$. Therefore, $\Sigma|R|$ grows linearly with both the number of active subscriptions and the number of brokers (see Fig. 6.2). This is due to the fact that each subscription is stored on every broker.

### 6.3.2 Simple Routing with Advertisements

The use of advertisements significantly reduces the size of the routing tables because in this case a subscription is only stored on brokers that are on a path from the respective consumer to the sources of the notifications of interest. This means that a local broker can only have remote routing entries if at least one of its clients has issued an advertisement. Therefore, local brokers have no remote routing entries at all while the size of the routing table of a router depends on the level to which it belongs (see Tab. 6.2). The root router might become overloaded first, because it has the largest routing table whose size equals the number of active subscriptions. Hence, the use of advertisements has not reduced the routing table size of the root router. The routing tables of the brokers on the lower levels are much smaller. Their size corresponds to the number of subscriptions that are active in the respective subnet.

| Level | Number of Routers | Size of Routing Table |
|---|---|---|
| 1 | 1 | $1.00 \cdot x$ |
| 2 | 3 | $0.33 \cdot x$ |
| 3 | 9 | $0.10 \cdot x$ |
| 4 | 27 | $0.03 \cdot x$ |

Table 6.2: Routing Table Size for Simple Routing with Advertisements

The sum of all remote routing entries $\Sigma|R|$ is given by $P_{avg} \cdot x$ where $P_{avg}$ is the average path length from a consumer to the notification source, i.e., the root router. This means that the routing tables still grow linearly with the number of active subscriptions (see Fig. 6.2) but logarithmically instead of linearly in the number of brokers. The average path length depends on the topology and the positioning of the notification source. For the given topology $P_{avg}$ with the source at the root router $P_{avg}$ is minimal and equals $(54 \cdot 4 + 9 \cdot 3 + 3 \cdot 2 + 1 \cdot 1)/67 = 3.73$. Therefore, the use of advertisements reduces $\Sigma|R|$ by a factor of $(|V|-1)/P_{avg} = 28.4$ which is independent of the number of active subscriptions (see Fig. 6.4). If the source is positioned at a local broker of a leaf router (which is the worst case), $P_{avg}$ would be equal to $(1 \cdot 2 + 1 \cdot 3 + 5 \cdot 4 + 3 \cdot 5 + 14 \cdot 6 + 6 \cdot 7 + 36 \cdot 8)/67 = 6.78$. Importantly, if the hierarchy has more levels, the factor by which advertisements reduce the routing table sizes would drastically increase because the number of brokers grows exponentially in the number of levels. Therefore, advertisements increase the scalability of large publish/subscribe systems.

### 6.3.3 Routing based on Identity

Due to the identity-based routing algorithm, routing table entries that have identical filters and destinations are diminished. In consequence, the size of the routing table of a broker $B$ is limited by the number of stocks multiplied by the number of its neighbors, i.e., $m \cdot |N_B|$. This means that, in contrast to simple

routing, the size of a routing table does not grow unboundedly with respect to the number of active subscriptions. Since no advertisements are used, this limit is approached for all brokers at equal speed.

For relatively small numbers of active subscriptions the sum of all remote routing tables entries grows as in the case of simple routing, i.e., nearly linearly, while for only slightly larger numbers the gradient monotonically decreases (Fig. 6.2). This is because for larger numbers of subscriptions the probability increases that subscriptions are identical. In the investigated scenario, $\Sigma|R|$ converges to $212,000$ for large numbers of subscriptions. More generally, if $G$ is an arbitrary acyclic graph $\Sigma|R|$ converges to

$$\begin{aligned}\Sigma|R| &= \sum_{B \in V} (|N_B| \cdot m) \\ &= 2 \cdot m \cdot |E| \\ &= 2 \cdot m \cdot (|V| - 1)\end{aligned}$$

### 6.3.4 Routing based on Identity with Advertisements

The use of advertisements offers similar benefits for identity-based routing as in simple routing, e.g., only the routers have remote routing entries. For all routers $B$ except the source, the size of a routing table is limited by $m \cdot (|N_B| - 1)$, while for the source it is limited by $m \cdot |N_B|$. These limits are reached for a relatively small number of subscriptions for the topmost router, while for the routers on the lower levels a larger number is necessary to reach the saturation point.

For relatively small numbers of subscriptions the sum of all remote routing entries approximates that of simple routing with advertisements, but the gradient decreases rapidly and the sum finally converges to $106,000$. Indeed, if $G$ is an arbitrary acyclic graph, the sum of all remote routing entries is limited by

$$\begin{aligned}\Sigma|R| &= m \cdot |N_S| + \sum_{B \in V \setminus \{S\}} \big((|N_B| - 1) \cdot m\big) \\ &= m \cdot \Big(\sum_{B \in V} (|N_B| - 1)\Big) + m \\ &= m \cdot \Big(\sum_{B \in V} |N_B|\Big) + m(1 - |V|) \\ &= 2 \cdot m \cdot |E| + m(1 - |V|) \\ &= 2 \cdot m \cdot (|V| - 1) + m(1 - |V|) \\ &= m \cdot (|V| - 1)\end{aligned}$$

Hence, the use of advertisements halves the limit without advertisements. Interestingly, this limit does not depend on the positioning of the source.

The factor by which the use of advertisements reduces $\Sigma|R|$ depends on the number of issued subscriptions (see Fig. 6.4). The factor is near to $(|V| - 1)/P_{avg} = 28.4$ for very small numbers of subscriptions but it quickly decreases

for larger numbers of subscriptions. Finally, it converges to 2 meaning a minimum improvement of 50%.

### 6.3.5 Routing based on Covering

Covering-based routing behaves exactly like identity-based routing if quote subscriptions are used because a quote subscription covers another quote subscription iff they are identical. Therefore, quote interval subscriptions have been used to investigate the behavior of covering-based routing. The following type of quote interval subscriptions has been used: $symbol = stock \wedge price \in [50 - t_1, 50 + t_2]$ with $t_1, t_2 \in [0, 50]$ randomly selected. This choice seems to be justified because in practice it can be expected that most quote interval subscriptions that refer to the same stock share a common price which is near to the current price of the stock. For two given quote interval subscriptions the probability that one of them covers the other is 50%. Of course, the benefits of covering-based routing would be less evident if this probability was lower.

Interestingly, the size of routing tables and the sum thereof are larger and converge more slowly for covering-based routing with quote interval subscriptions than for identity-based routing with quote subscriptions (see Fig. 6.3). This is due to the fact that, opposed to quote subscriptions, several quote interval subscriptions can exist that refer to the same stock and that do not cover each other. On the other hand, identity-based routing would nearly degrade to simple routing if it was applied to quote interval subscriptions. Therefore, the use of covering inherently improves the scalability if these more complex subscriptions are applied.

### 6.3.6 Routing based on Covering with Advertisements

Compared to identity-based routing with advertisements, the size of the routing tables and the sum thereof are larger and converge more slowly (see Fig. 6.3). The rational for this has already been presented in the preceding subsection. Interestingly, the difference is not as great as if advertisements are not used.

The curve of the factor by which advertisements reduce the sum of all remote routing entries is depicted in Fig. 6.4. It looks similar to that of identity-based routing but it declines more slowly and therefore, the use of advertisements offers even more advantages if covering-based routing is used.

### 6.3.7 Routing based on Merging

The implemented merging-based routing algorithm leads to a routing table that has at most one remote entry for each neighbor if quote subscriptions are used. Hence, the sum of all remote routing entries $\Sigma|R|$ is limited by $\sum_{B \in V} |N_B| = 2 \cdot |E| = 2 \cdot (|V| - 1)$. Because of that, and in order to compare it with covering-based routing, quote interval subscriptions have been used to evaluate merging-based routing. The resulting curve is shown in Fig. 6.3. It is identical to that

of identity-based routing with quote subscriptions. This is because two given quote interval subscriptions of the used form can always be merged if they refer to the same stock. Of course, the effect of filter merging would be smaller if the merging probability was lower than 100%, but imperfect merging can moderate this (see Sect. 6.5.2). In any case, the use of merging improves the scalability if quote interval subscriptions are used.

### 6.3.8 Routing based on Merging with Advertisements

If merging with advertisements is used in conjunction with quote subscriptions, the routing table of the source contains at most one remote routing entry for each neighbor while those of the other brokers contain at most the number of neighbors minus one. The limit of the sum of all remote routing entries can be derived in the following way:

$$\begin{aligned}
\Sigma|R| &= |N_S| + \sum_{B \in V \setminus \{S\}} (|N_B| - 1) \\
&= \Big( \sum_{B \in V} (|N_B| - 1) \Big) + 1 \\
&= \Big( \sum_{B \in V} |N_B| \Big) - |V| + 1 \\
&= 2 \cdot |E| - |V| + 1 \\
&= 2 \cdot (|V| - 1) - |V| + 1 \\
&= |V| - 1
\end{aligned}$$

Due to this fact and to achieve comparability, quote interval subscriptions have been used. Fig. 6.3 shows the resulting curve. Again, it is identical to that of identity-based routing for quote subscriptions because of the same reason that has been stated in the preceding subsection.

## 6.4 Filter Forwarding Overhead

In this section the second key characteristic of a routing algorithm, namely the filter forwarding overhead, is investigated with respect to the number of active subscriptions. As a realistic measure of this overhead the number of control messages that are processed by the brokers has been chosen so the measurements offer an insight into the *message complexity* of the protocols. The following subsections discuss the behavior of the implemented routing algorithms with respect to the filter forwarding overhead in detail.

### 6.4.1 Simple Routing

If simple routing is used every new or canceled subscription is forwarded to any broker and hence, all routing tables are affected by an update. Therefore, the

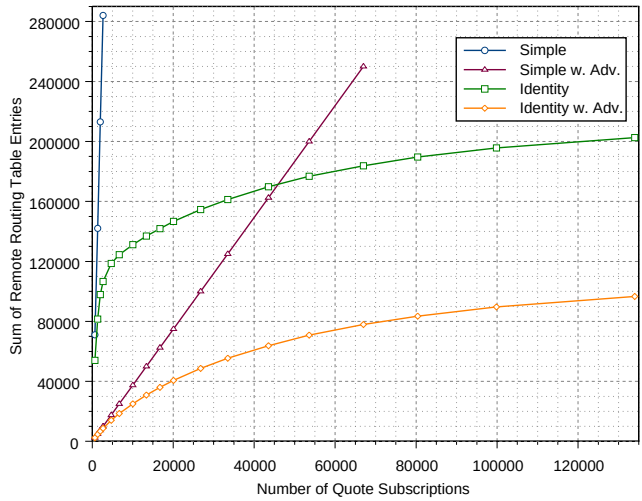



Figure 6.2: Simple vs. identity-based routing (routing table size).

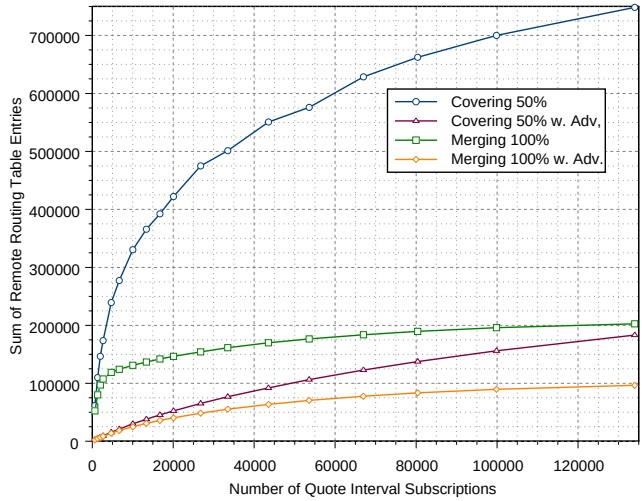



Figure 6.3: Covering- vs. merging-based routing (routing table size).

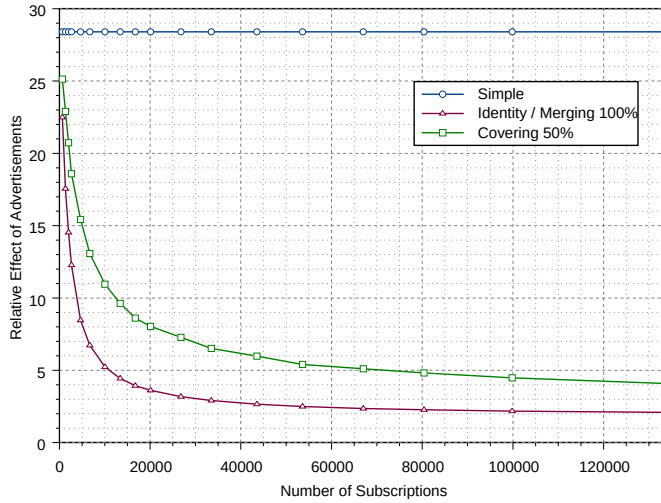



Figure 6.4: Relative effect of the use of advertisements (routing table size).

number of control messages that is necessary to update the routing tables is equal to the number of brokers minus one, i.e., $|V| - 1$, and does not dependent on the number of active subscriptions (see Fig. 6.5).

### 6.4.2 Simple Routing with Advertisements

Interestingly, the use of advertisements reduces not only the size of routing tables but also the filter forwarding overhead. This is because a subscription is only stored in the routing tables of those brokers that lie on the path from the respective consumer to the notification source. In consequence, $P_{avg} = 3.73$ control messages are necessary on average (see Fig. 6.5). This means that compared to simple routing without advertisements, the number of necessary control messages is reduced by a constant factor of $(|V| - 1)/P_{avg} = 28.4$ (see Fig. 6.7). In consequence, advertisements reduce the filter forwarding overhead significantly, especially for large systems because for the investigated type of topology $P_{avg}$ grows only logarithmically in the number of brokers $V$.

### 6.4.3 Routing based on Identity

If identity-based routing is used the filter forwarding overhead depends on the number of active subscriptions. From Fig. 6.5 it can be inferred that, for small numbers of active subscriptions, the majority of routing tables are affected by

a new or canceled subscription. This number decreases very quickly at first and still nearly exponentially afterwards. This is due to the fact that a new or canceled subscription is only forwarded to a neighbor if there is no other active subscription that has been received from a distinct neighbor which is identical to the subscription considered. The probability for this to occur decreases with an increasing number of active subscriptions.

### 6.4.4 Routing based on Identity with Advertisements

Here, the average number of control messages that is necessary to update the routing tables starts at the level of simple routing with advertisements (see Fig. 6.5). Interestingly, the resulting curve quickly approximates that of identity-based routing without advertisements. This effect is caused by the uniform distribution of interests. Hence, the factor by which the number of control messages is reduced by using advertisements is quite large for a small number of subscriptions but this advantage quickly disappears (see Fig. 6.7). In fact, for large numbers of subscriptions, the use of advertisements does not reduce the filter forwarding overhead at all.

### 6.4.5 Routing based on Covering

If quote subscriptions are used covering-based routing behaves exactly like identity-based routing. If quote interval subscriptions are used the number of necessary control messages decreases much more slowly (see Fig. 6.6). This is because two quote interval subscriptions that refer to the same stock do not need to cover each another. This also implies that sometimes a subscription is forwarded that does not cause any additional notifications to be received. This disadvantage is diminished by merging-based routing.

### 6.4.6 Routing based on Covering with Advertisements

Here, the filter forwarding overhead starts at the level of simple routing with advertisements but decreases afterwards (see Fig. 6.6). If quote subscriptions are used, covering-based routing behaves exactly like identity-based routing. If quote interval subscriptions are used the curve drops much more slowly. Interestingly, the curves with and without advertisements approach one another only slowly and therefore, advertisements remain useful for larger numbers of subscriptions, too (see Fig. 6.7). This fact becomes even more evident for larger systems.

### 6.4.7 Routing based on Merging

The filter forwarding overhead that is induced by merging-based routing is similar to that of covering-based routing but it decreases more slowly (see Fig. 6.6). In fact, the gradient of the curve is smaller in all areas of the graph. The reasons for this behavior still need to be investigated, but it can be expected that it is caused by the perfectness of the merging-based routing algorithm.

### 6.4.8 Routing based on Merging with Advertisements

Here, the filter forwarding overhead is very similar to that of covering-based routing with advertisements. In fact, the overhead induced by merging-based routing is only slightly larger. This is interesting because the difference is much bigger if advertisements are not used.

The filter forwarding overhead without advertisements is clearly larger than that with advertisements, even for larger numbers of subscriptions. Therefore, the use of advertisements offers an advantage in this case, too (see Fig. 6.7).

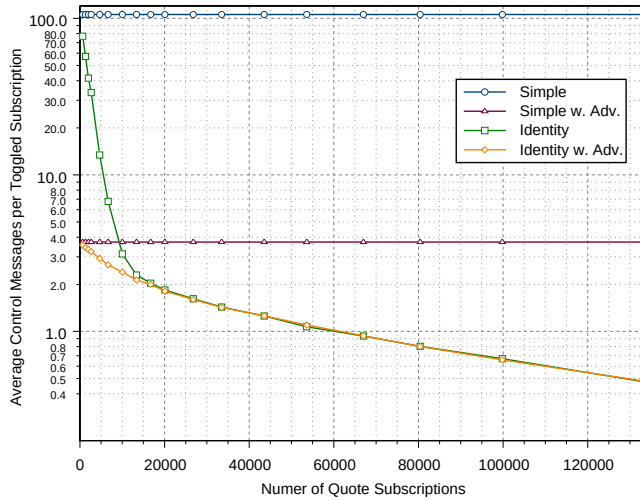



Figure 6.5: Simple vs. identity-based routing (filter forwarding overhead).

## 6.5 Supplementary experiments

This section describes two supplementary experiments dealing with aspects that are not covered by the experiments that have been described in the preceding sections: The first experiment investigates the effect of varying locality, while the second presents an investigation of imperfect merging. This additional material allows to extend the validity of the results presented.

### 6.5.1 Effects of Locality

In the experiments of Sections 6.3 and 6.4, it has been assumed that the interests of the clients are uniformly distributed over the entire system. In this experiment

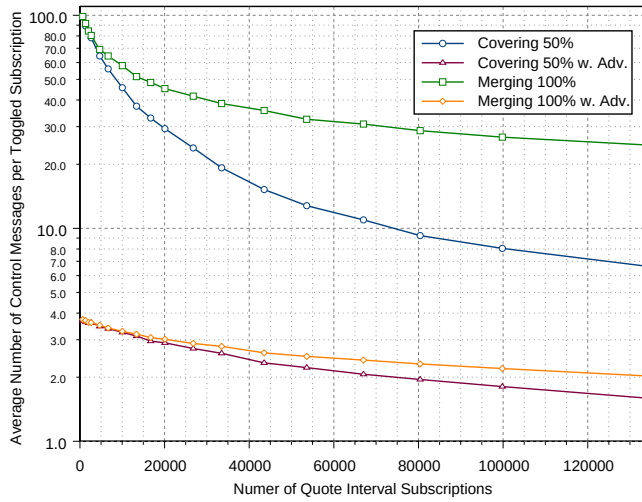



Figure 6.6: Covering- vs. merging-based routing (filter forwarding overhead).

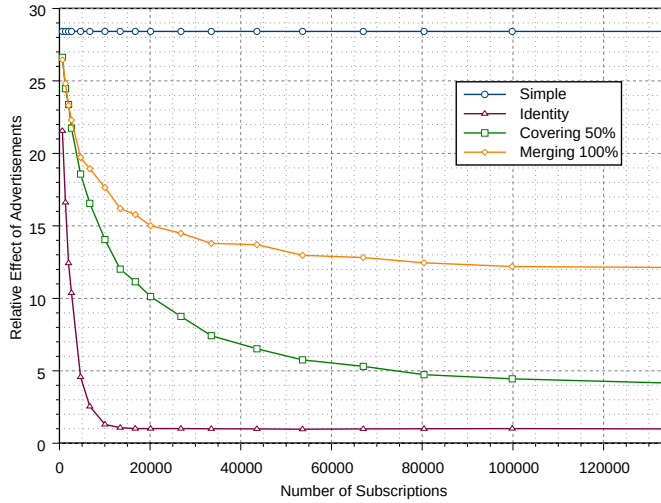



Figure 6.7: Relative effect using advertisements (filter forwarding overhead).

the effects that are caused by varying the degree of locality among the interests of the consumers have been investigated. In particular, it has been studied how locality influences the sum of all remote routing entries, the saved proportion of the payload traffic, the relative routing table size, and the filter forwarding overhead. The identity-based routing algorithm has served as the basis for these investigations; the findings for the other routing algorithms would be similar. In order to capture the behavior, the degrees of locality of the three subnets that are induced by the subordinate routers of the root router have been varied. The local broker that is connected to the root router has been left without any subscriptions.

The parameter $d$ is a measure of the amount of locality among the sets of clients of the respective subnets. For $d = 1/3$ the interests are disjoint and for $d = 1.0$ they are identical (see Table 6.3). The resulting curves vary $d$ stepwise

| d | Subnet 1 | Subnet 2 | Subnet 3 |
|---|---|---|---|
| 1/3 | 0-333 | 333-666 | 666-999 |
| 0.4 | 0-400 | 300-700 | 600-1000 |
| 0.5 | 0-500 | 250-750 | 500-1000 |
| 0.6 | 0-600 | 200-800 | 400-1000 |
| 0.7 | 0-700 | 150-850 | 300-1000 |
| 0.8 | 0-800 | 100-900 | 200-1000 |
| 0.9 | 0-900 | 50-950 | 100-1000 |
| 1.0 | 0-1000 | 0-1000 | 0-1000 |

Table 6.3: Relation between $d$ and the corresponding subnet interests

from 1/3 to 1. It is important to note that the depicted effects would be even more evident if the probability that a subscription refers to a specific stock was not uniformly distributed.

From the Figures 6.8, 6.9, 6.10, and 6.11 it can be inferred that the degree of locality has a great impact on the size of the routing tables and the filter forwarding overhead. An interest distribution without locality results in the largest routing tables and the highest filter forwarding overhead. For smaller values of $d$ the routing tables and the filter forwarding overhead monotonically decrease and reach their minimum for $d = 1/3$. Interestingly, the sum of all remote routing entries is limited by $d \cdot m \cdot (|V| - 2)$ if advertisements are used, and by $2 \cdot \frac{2+d}{3} \cdot m \cdot (|V| - 2)$ if advertisements are not used. Here again, the use of advertisements shows its advantages.

The degree of locality also influences the amount of traffic that is saved when filtering is compared to flooding (see Fig. 6.12). For perfect routing algorithms the amount of saved payload traffic does not dependent on the underlying routing algorithm; it only depends on the type and number of issued subscriptions. For uniform interests ($d = 1.0$) and large numbers of subscriptions the saved amount of traffic slowly converges to 0. Opposed to that, for $1/3 \leq d < 1.0$ a constant

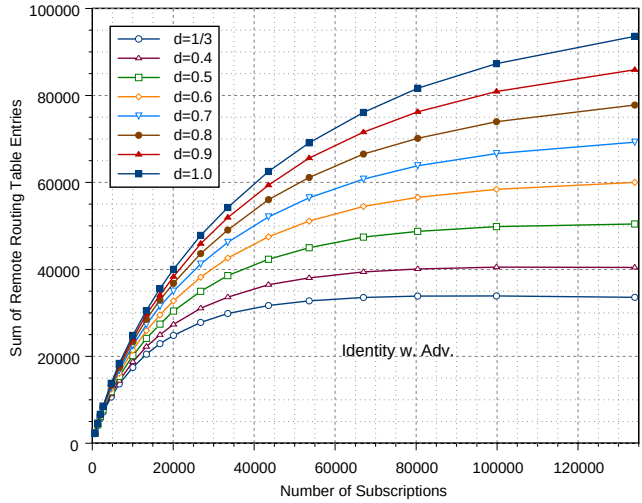



Figure 6.8: Routing table size.

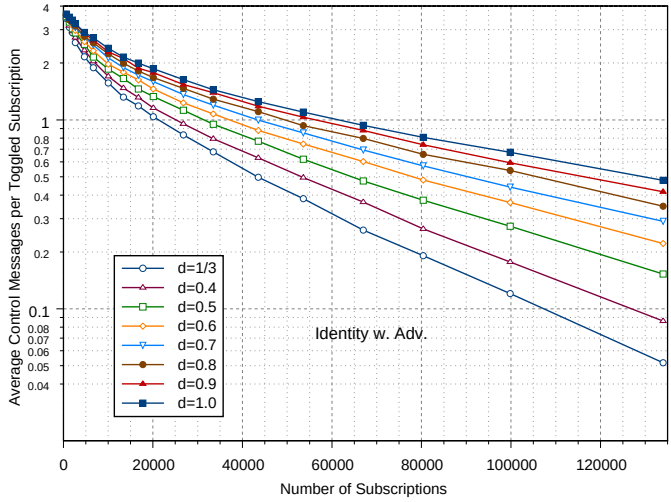



Figure 6.9: Filter forwarding overhead.

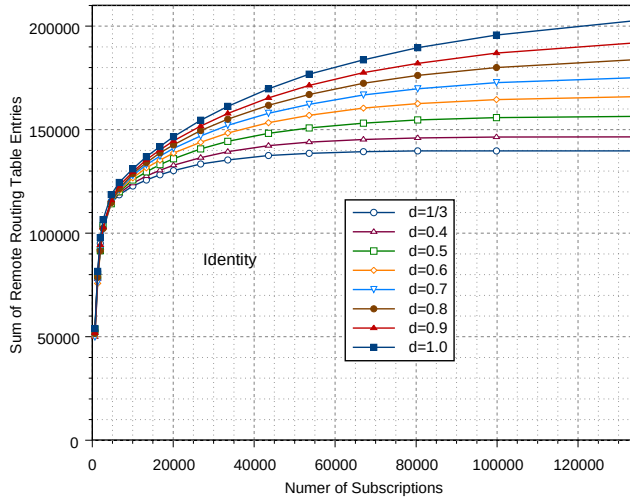



Figure 6.10: Routing table size.

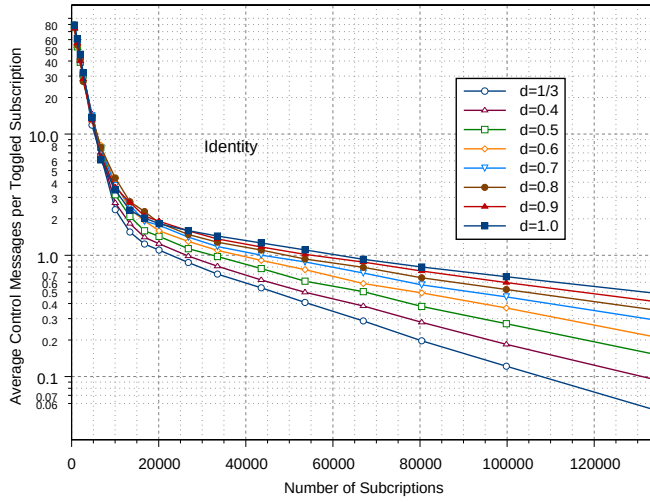



Figure 6.11: Filter forwarding overhead.

amount of traffic which equals $1 - d$ is saved for large numbers of subscriptions.

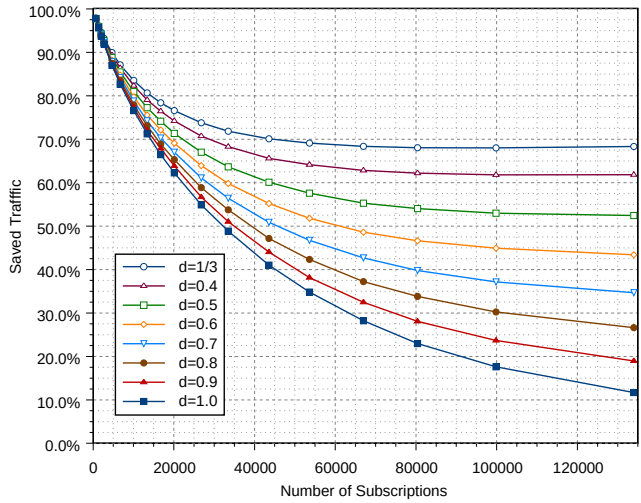



Figure 6.12: Amount of saved payload traffic compared to flooding.

A simple measure of the usefulness of filtering is the *relative routing table size* which is given by the sum of all remote routing entries divided by the amount of saved traffic. Fig. 6.13 depicts the relative routing table size if advertisements are used and Fig. 6.14 if they are not used. From these figures it can be inferred that the relative routing table size is largely dependent on the degree of locality. For $d = 1.0$ the relative routing overhead converges to infinity because the saved traffic converges to 0. If $1/3 \leq d < 1.0$ it converges to $d \cdot m \cdot (|V| - 2)/(1 - d)$ if advertisements are used (see Fig. 6.15).

The results about the saved traffic and the filter forwarding overhead can be combined in a way such that the scenarios in which filtering is advantageous can be estimated. In fact, it is possible to determine the ratio of the event production rate to the subscription change rate for which the number of payload messages per time unit that is saved (by applying filtering) equals the number of control messages (that are necessary to update the routing tables). This gives an indication on the "break even point", i.e., up to how much dynamic activity the outing schema still saves messages.

Surprisingly, the minimum ratio for which filtering is advantageous decreases for increasing numbers of subscriptions and degrees of locality if identity-based routing is used. This becomes evident by comparing, for example, Figures 6.16 and 6.18. In simple routing (Fig. 6.18), the system must be increasingly static in order for the routing scheme to still save messages. In identity-based routing

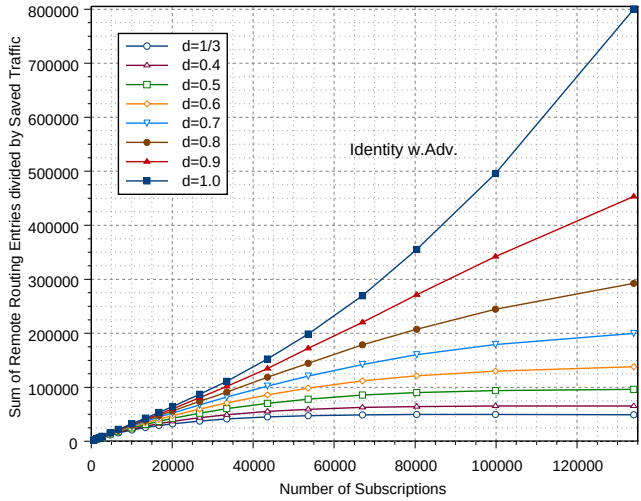



Figure 6.13: Relative routing table size (with adv.).

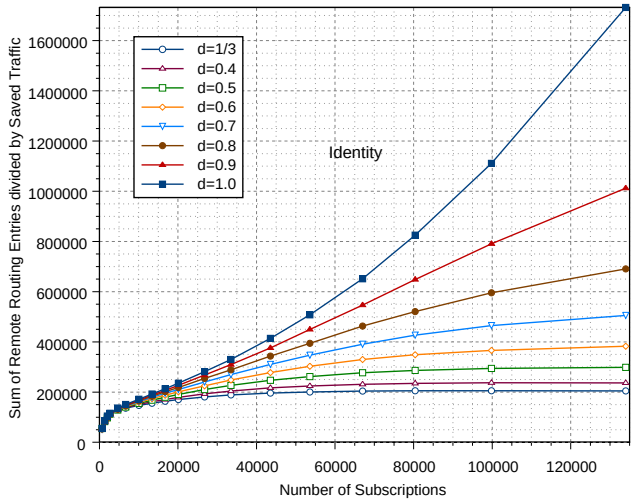



Figure 6.14: Relative routing size (without adv.).

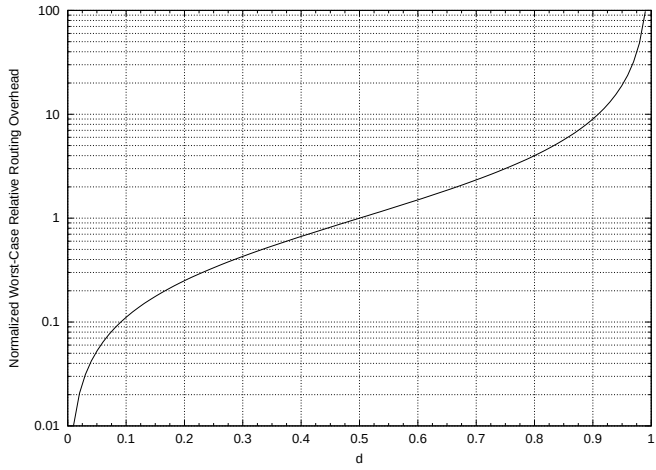



Figure 6.15: Normalized worst case relative routing table size (with adv.).

(Fig. 6.16), the gradient is negative meaning that the dynamics of the system can even "get worse" (i.e., increase) while still saving messages through filter-based routing. In particular, the degree of locality determines the gradient with which the minimum ratio decreases (see Fig. 6.16 and 6.17). Apparently, the use of filtering is advantageous in scenarios which are more dynamic than was previously thought if advanced routing algorithms are applied.

### 6.5.2 Evaluation of Imperfect Merging

In the experiments of Sections 6.3 and 6.4 a merging algorithm has been evaluated that generates perfect mergers, i.e., mergers that do not match any additional notifications than the filters it was generated from. This experiment investigates an imperfect merging algorithm that can be configured to use different degrees of imperfectness. As a basis for the evaluation, quote interval subscriptions of the form $symbol = stock \wedge price \in [p_1, p_2]$ with $p_1, p_2 \in [0, 100] \wedge p_2 - p_1 = 20$ are used. In consequence, there can be at most 4 subscriptions that refer to the same stock that cannot be perfectly merged. Let $[p_1, p_2]$ and $[p_3, p_4]$ be the intervals of two subscriptions that refer to the same stock. The implemented imperfect merging algorithm merges these subscriptions iff

$$\frac{max\{p_2, p_4\} - min\{p_1, p_3\}}{p_2 - p_1 + p_4 - p_3} \leq f$$

is satisfied. For $f = 1$ the merging algorithm is perfect while for larger values of $f$ the grade of imperfectness increases. If $f = 2.5$ the algorithm merges any two subscriptions that refer to the same stock.

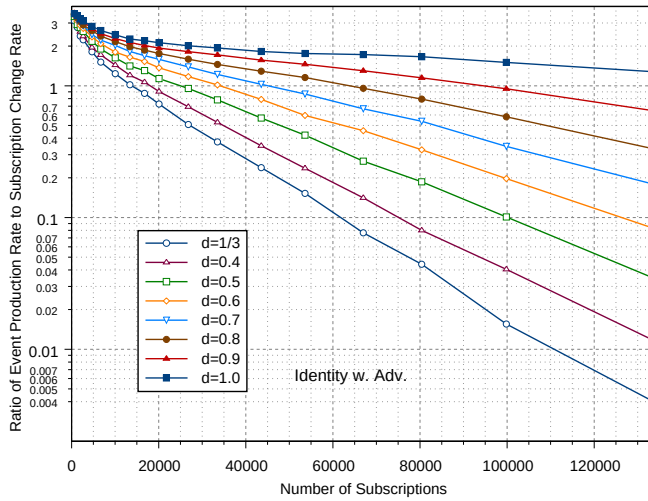



Figure 6.16: Maximum degree of dynamics (identity with adv.).

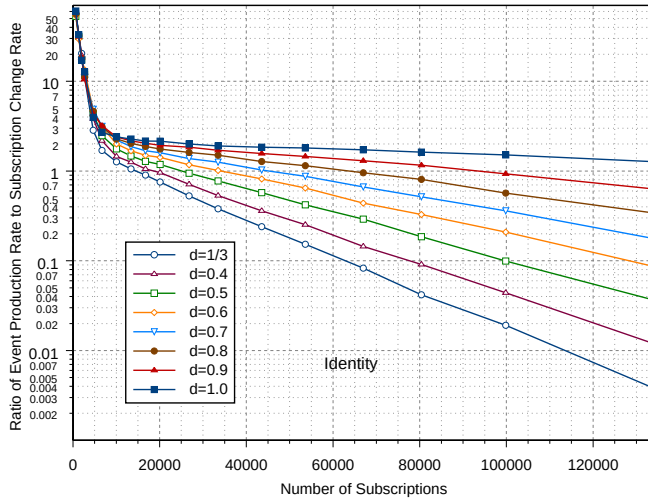



Figure 6.17: Maximum degree of dynamics (identity without adv.).

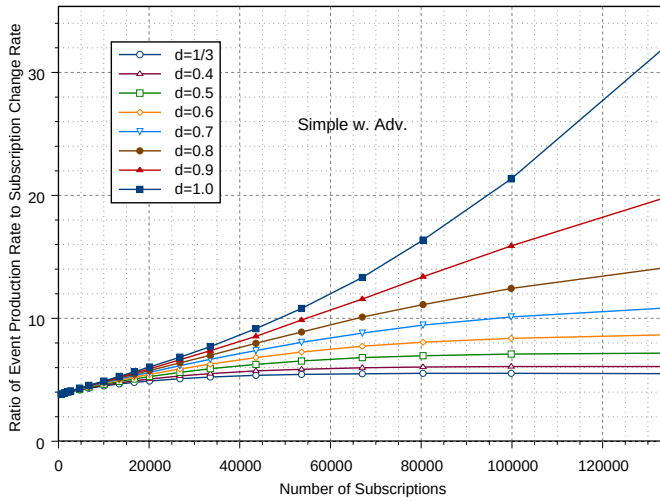



Figure 6.18: Maximum degree of dynamics (simple with adv.).

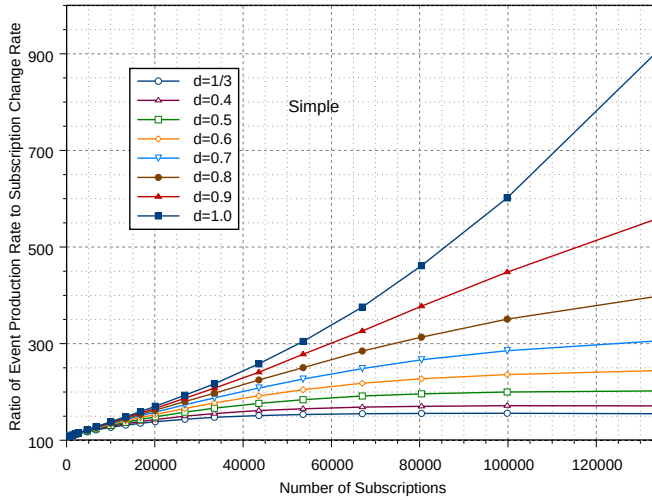



Figure 6.19: Maximum degree of dynamics (simple without adv.).

The evolution of the sum of all remote routing table entries and the filter forwarding overhead is shown in Fig. 6.20 and 6.21 where $f$ is varied from 1 to 2.5. It can be inferred that the routing table size decreases with increasing degrees of imperfectness while the filter forwarding overhead decreases only very slowly. This is because the implemented merging algorithm has not been optimized for imperfect merging. It can be expected that future imperfect merging algorithms will diminish this disadvantage. Imperfect merging appear to be very promising and requires further investigation.

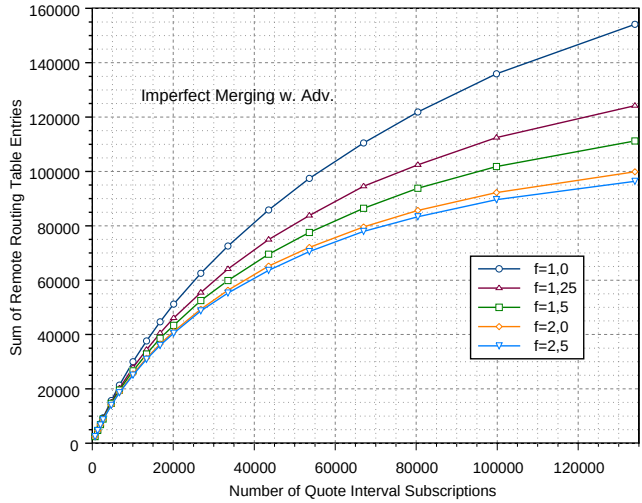



Figure 6.20: Effect of varying imperfectness (routing table size).

## 6.6 Related Work

This section discusses previous evaluations of notification services and relates them to the results of this chapter. Interestingly, none of them has investigated the routing tables sizes or the filter forwarding overhead.

### 6.6.1 Siena

Carzaniga, Rosenblum, and Wolf [17, 22] presented performance results which are based upon a simulation framework. Their work investigated two variants of covering-based routing, a peer-based and a hierarchical version. The simulated

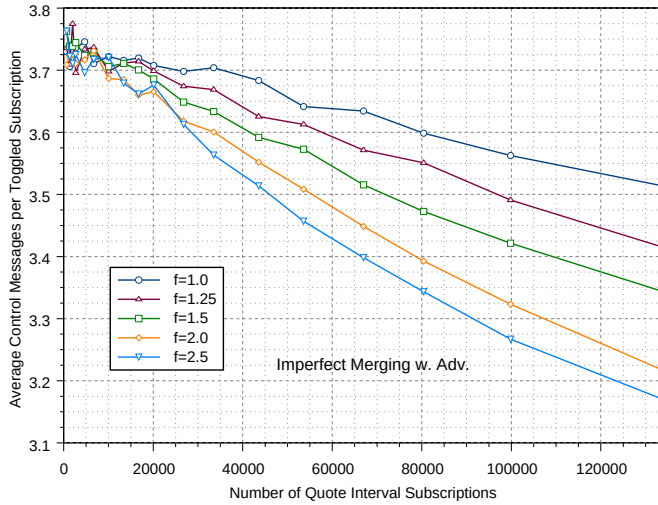



Figure 6.21: Effect of varying imperfectness (filter forwarding overhead).

algorithms are also incorporated into their publish/subscribe prototype called SIENA. Other routing algorithms are not considered.

The simulations investigated the total cost induced by the notification service, the cost induced on individual brokers (and its variance), the average cost per subscription (and its worst case), and the per notification cost. Unfortunately, it is not easy to interpret their results because the setup of main parameters influencing the results are not described. This includes the metric underlying their cost analysis, the structure of the notifications, subscriptions, and advertisements, the rate of subscribing/unsubscribing and advertising/unadvertising.

### 6.6.2 JEDI

The current implementation of JEDI exploits a hierarchy of event brokers in conjunction with the hierarchical version of covering-based routing [17]. The algorithm implies that a notification is always propagated to the root broker regardless of the interests of the consumers. Moreover, an improved version is suggested that extends the hierarchical algorithm by using advertisements, and simulations have been carried out to compare the original with the improved version [13, 14]. Bricconi, Di Nitto, and Tracanella [14] also presented the analytical model that underlies their simulations and which allows the average number of notifications that is processed by an event broker to be estimated.

### 6.6.3 Gryphon

Performance results related to the prototype of the Gryphon notification service are presented by Banavar et al. [8] and Opyrchal et al. [85]. The routing algorithm exploited by Gryphon is similar to simple routing without advertisements. Their work concentrates on the use of multicast and efficient matching of events to subscriptions [1]. The matching algorithm clearly outperforms the simple sequential algorithm, but it depends on and supports only a few types of attribute filters, limiting its usability. Moreover, updating the matching data structure if clients subscribe and unsubscribe is costly.

The load caused at the individual brokers was investigated in the first article mentioned above [8]. The results presented show that flooding overloads at the same publishing rate regardless of the percentage of matches or the number of active subscriptions. Filtering-based routing on the other hand can handle much higher publication rates if subscriptions are highly selective or highly local, which can be expected in large scale publish/subscribe systems.

The second article [85] concentrates on bandwidth utilization. It compares flooding to four multicast-enabled routing algorithms and ideal multicast which assumes that for each event a perfect multicast group exists. The authors state that filtering-based routing is superior to flooding under conditions of high selectivity and high locality of subscriptions. This opinion supports the findings of this Thesis. Nevertheless, it can be expected that their results are still too pessimistic because their work depends on simple routing, i.e., the routing algorithm does not exploit covering and merging. They also assume that event brokers are not placed nearby to the multicast routers and therefore, the use of multicast may even introduce a bandwidth penalty. Moreover, they did not investigate the use of advertisements.

## 6.7 Discussion

The evaluation has shown that for the investigated scenarios the advanced routing algorithms are clearly superior to the simple routing algorithm. Both the size of the routing tables and the filter forwarding overhead are significantly reduced by applying the proposed routing optimizations.

It also became evident that the use of advertisements further reduces the routing table sizes and the filter forwarding overhead by a large factor that depends on the average path length in the given topology. This is because if advertisements are used, a subscription is only forwarded to and stored in the routing table of those neighbors that are on a path from the respective consumer to a producer that has issued an overlapping advertisement. In general, the average path length increases logarithmically in the number of brokers and therefore, advertisements are especially advantageous in content-based publish/subscribe systems with many brokers.

The above findings hold for the worst case that subscriptions are uniformly distributed. If this is not the case (i.e., if *locality* among the interests of the

consumers exists and increases), the performance improves drastically. In this case the routing table sizes and the filter forwarding overhead are reduced substantially, and a constant amount of payload traffic is saved even for very large numbers of subscriptions. Furthermore, the results about the filter forwarding overhead and the saved traffic were combined. This allowed to derive the ratio of the event production rate to the subscription change rate for that the number payload messages that is saved (by applying filtering) equals the number of control messages (that is necessary to update the routing tables). This showed that, by using advanced routing algorithms, filtering is advantageous in even more dynamic environments than was previously thought.

In addition to the experiments whose results have been described above, an evaluation of a simple imperfect merging algorithm has been carried out. The evaluation showed that the use of imperfect merging can keep the routing tables small compared to perfect merging if subscriptions are more complex. Currently, imperfect merging is still in its early stages, and hence, it can be expected that future algorithms can improve these results.

In conclusion, advanced routing algorithms improve the scalability of content-based publish/subscribe systems. Future work can build upon these results, e.g., by considering other and also more general scenarios. For example, the effects of several producers and of dynamic advertisements should be investigated. Moreover, algorithms that can adapt to changing environments are especially interesting.

# Chapter 7

# Conclusions and Future Work

## Conclusions

Today, the architecture of distributed computer systems is dominated by client/server platforms relying on synchronous request/reply. This architecture is not well suited to implement information-driven applications like news delivery, stock quoting, air traffic control, and dissemination of auction bids due to the inherent mismatch between the demands of these applications and the characteristics of those platforms. In contrast to that, publish/subscribe directly reflects the intrinsic behavior of information-driven application because communication is indirect and initiated by producers of information. Therefore, publish/subscribe should be the first choice for implementing such applications.

The expressiveness of the notification selection mechanism is crucial for the flexibility of a notification service. Content-based notification selection is most expressive because it allows filtering predicates to be evaluated over the whole content of a notification. The advantage in expressiveness compared to channel- or subject-based selection results in increased flexibility facilitating extensibility and change. On the other hand, scalable implementations of content-based notification services are most difficult to realize. Indeed, the expressiveness of notification selection must be carefully chosen in large-scale systems because expressiveness and scalability are interdependent. Hence, the most fundamental problem in the area of content-based publish/subscribe systems is probably the scalable routing of notifications. This thesis concentrated on mechanisms to improve the scalability of publish/subscribe systems which are based on content-based routing algorithms.

This thesis consists of a theoretical and a practical part. The theoretical part presented a formal specification of publish/subscribe systems (Chapter 2), a routing framework and novel routing algorithms (Chapter 3), and discussed how

the routing optimizations (that rely on identity- and covering-tests as well as on filter merging) can be broken down to the actual data/filter model (Chapter 4). Altogether, the theoretical part gave a number of new insights into the theory of content-based publish/subscribe systems. The practical part presented the implementation of the REBECA notification service prototype which includes all discussed routing algorithms (Chapter 5), and a detailed practical evaluation of the implemented routing algorithms based upon the prototype (Chapter 6). Hence, this thesis captures the theory, the implementation, and the evaluation of content-based publish/subscribe systems. In the following, the content of the individual chapters is summarized.

**Chapter 2: Formal Specification of Publish/Subscribe Systems.** Chapter 2 presented a formal specification of publish/subscribe systems. It is based on sequential traces of state/operation pairs and uses the syntax of linear temporal logic. The specification offers a stable basis for any further reasoning about the behavior (e.g., the correctness) of publish/subscribe systems and allows deficiencies in the description of the semantics of current systems to be pointed out. The specification consists of safety and liveness conditions which are known to be suited for specifying any useful system behavior. Here, the safety condition states that only matching notifications should be delivered to consumers and that a notification should never be delivered to a consumer more than once. The liveness condition demands that if a consumer has subscribed to a filter, eventually, i.e., after a finite time, all matching notifications which are published afterwards must be delivered to this consumer. The same specification applies to systems which require to mask internal faults. Since this is not always possible, the notion of stabilizing publish/subscribe systems was introduced that is based on a weakened version of the specification of fault-free publish/subscribe systems. Stabilizing systems can be realized by a number of techniques known from the literature. They are able to recover from arbitrary transient faults within a finite time.

**Chapter 3: Content-Based Routing.** A formal framework for content-based routing algorithms was described in Chapter 3. A formalization of routing configurations was introduced that is based on the notion of valid routing configurations. A valid routing configuration ensures that all matching notifications are delivered to a consumer. In dynamic publish/subscribe systems it is impossible to ensure that the routing configuration is always valid. Therefore, weakly valid routing configurations were introduced. A weakly valid routing configuration demands only the delivery of those notifications which are matched by a subscription which has already been incorporated into the routing configuration. After the formalization of routing configurations, a routing framework was described in which an abstract function must be instantiated to yield concrete routing algorithms. Sufficient conditions were given which the abstract function must satisfy in order for a routing algorithm to be correct. This routing framework was subsequently used to describe a set of routing algorithms and to

discuss their correctness by applying the above correctness criterion. The routing algorithms discussed include flooding, identity-based routing, covering-based routing, and routing based on filter merging. While identity-based routing is a simplified version of covering-based routing, merging-based routing is even more sophisticated. It is based upon filter merging reducing the number of filters that must be dealt with. The last two sections of this chapter presented how advertisements can be integrated into the routing framework and how the given routing algorithms can be made stabilizing by using subscription leasing.

**Chapter 4: Models and Routing Optimizations.** In Chapter 4, it was shown how the proposed routing optimizations can be broken down to the underlying data/filter model. The focus of this investigation was on structured records that consist of name/value pairs. In the name/value pair model, a notification is essentially a record consisting of attributes being name/value pairs. Subsequently, a filtering model was introduced that builds upon conjunctive filters consisting of attribute filters, each of which imposing a constraint on a single attribute. To enable the efficient evaluation of identity- and covering tests as well as filter merging, at most one attribute filter per attribute is allowed. After defining the data/filter model, a generic filtering framework was proposed that is not restricted to specific data types or operators. As examples, covering implications and merging rules for a set of data types with typical operators were described. Furthermore, new algorithms for identity tests and covering tests as well as for detecting merging candidates were presented. These algorithms are necessary to realize the routing algorithms presented in Chapter 3. Moreover, some preliminary ideas how to support semistructured data (e.g., XML) and objects were described.

**Chapter 5: Implementation.** The implementation that had been carried out as part of this thesis was described in Chapter 5. It consists of the Rebeca notification service and two example applications. Instead of relying on a single routing scheme, a set of routing algorithms was implemented, and new routing algorithms can be added easily. This allows various routing algorithms to be tested and compared in a uniform environment. Rebeca relies on a filtering framework instead of a fixed set of primitive filtering constraints. Moreover, it supports replay of past notifications and service factories. Finally, Rebeca has also served as the basis for the evaluation that was described in Chapter 6. As proof of concept two example applications had been implemented, a stock trading platform and an infrastructure for self-actualizing web pages. These applications show that the basic functionality of the prototype is working.

**Chapter 6: Experimental Results.** In Chapter 6, a detailed practical evaluation of the implemented content-based routing algorithms was presented giving new and detailed insights into the behavior of these algorithms. In contrast to previous evaluations, the one presented here focused on the inherent character-

istics of routing algorithms (routing table sizes and filter forwarding overhead) instead of system-specific parameters (CPU load etc.). Moreover, it is based on a working prototype (REBECA) instead of simulations, and several routing algorithms are included in the comparison. Besides the routing table sizes and the filter forwarding overhead, the effects of locality among the interests of the consumers was investigated and a preliminary evaluation of imperfect merging was carried out. The evaluation revealed a number of interesting facts: Firstly, advanced routing algorithms are useful in large-scale publish/subscribe systems. Secondly, the use of advertisements considerably improves the scalability. Thirdly, advanced routing algorithms operate efficiently in more dynamic environments than was previously assumed. Fourthly, the good behavior of the algorithms further improves if the interests of the consumers are not evenly distributed, which can be expected in practice. Finally, the evaluation of imperfect merging shows that it is suited to further reduce routing table sizes.

## Future Work

Large-scale content-based publish/subscribe systems exhibit a lot of interesting challenges. Today, the state of the art in most of the respective areas is not satisfactory. This is mainly because of the indirect addressing scheme that makes publish/subscribe far more complex than standard request/reply. Hence, a lot of interesting research problems still exist in different areas:

**Scalable Routing.** In the area of content-based routing a lot of open issues still exist. For example, imperfect routing algorithms should be investigated in more detail. In particular, this includes more advanced imperfect merging algorithms. It is also promising to use statistical on-line evaluation to adapt routing algorithms to changing environments. There is also room for new evaluations, but more information about the typical design and the actual work load of event-based applications should be gathered first.

**Formal Semantics of Publish/Subscribe Systems.** The work presented in Chapter 2 can be seen as a starting point for discussing the formal semantics of publish/subscribe systems. Future work should investigate variations of the specifications presented and should relate different specifications to each other. For example, ordering policies could be incorporated into the specification. Altogether, the area of formally treating the semantics of publish/subscribe systems offers a wide range of future research possibilities.

**Event Composition.** Event composition is about generating new events by detecting patterns (e.g., sequences) of occuring events. There has been a lot of work regarding composite events in the area of active databases [87] and in the area of monitoring and debugging of distributed systems [99]. One of the main problems with composite events in distributed systems is the absence of a global

time base [67]. Another problem is the efficient recognition of composite events which can only be achieved if it is properly supported by the underlying routing algorithms. Today, the relation between event composition and efficient routing is not clear at all, only some preliminary work on simple sequences exists [17].

**Fault-Tolerance.** In large-scale systems it is a severe restriction to assume that the system is fault-free. In a publish/subscribe system, clients (i.e., the producers and the consumers), event brokers, and the connecting network can be faulty. Currently, most work assumes fault-free systems and contains single points of failure. Hence, mechanisms for implementing fault-tolerant publish/subscribe systems should be investigated in more detail. The presented leasing-based scheme for self-stabilizing publish/subscribe systems can only be regarded as a first step.

**Quality of Service.** Quality of service includes, for example, reliability (e.g., best-effort or guaranteed) [11, 58], ordering [67] (e.g., causal), real-time constraints [55], and transactions [68, 69]. It is known from research on multicast and other related areas that realizing some of these features is non-trivial. It would be rewarding to identify useful quality of service attributes of publish/subscribe systems and show approaches to evaluate real systems with respect to these attributes.

**Heterogeneity.** A large-scale system, regardless whether it is based on request/reply or on publish/subscribe, must be able to cope with heterogeneity. This includes, for example, the support of various transport protocols (which have a direct impact on notification routing), as well as ambiguity in the syntax and the semantics of notifications [25, 26]. First approaches to tackle these problems exist [38, 42] but their practical impact is not yet clear.

**Security.** Security is a problem that is difficult to tackle in any large-scale system. Especially, this is true for publish/subscribe systems where clients communicate only indirectly. For example, publishing and consuming of notifications should be controlled by security policies. Today, there is only few work on secure publish/subscribe systems [86, 117]. Considering that security is a special property with many intricate research questions, a lot of work also remains to be done in this area.

# Bibliography


[1] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 53–61, Atlanta, Georgia, USA, 1999.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

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

[4] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications. In *IEEE SDNE Services in Distributed and Networked Environments*, pages 148–155, Whistler, British Columbia, June 1995.

[5] J. Bacon, A. Hombrecher, C. Ma, K. Moody, and W. Yao. Event storage and federation using odmg. In D. S. G.N.C. Kirby, A. Dearle, editor, *9th International Workshop on Persistent Object Systems (POS-9)*, volume 2135 of *LNCS*, pages 265–, Lillehammer, Norway, Sept. 2000. Springer.

[6] J. Bacon, K.Moody, and W.Yao. Access control and trust in the use of widely distributed services. In *Middleware 2001*, volume 2218 of *LNCS*, pages 295–310. Springer, 2001.

[7] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, 2000.

[8] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–272, 1999.

[9] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information flow based event distribution middleware. In W. Sun,

S. Chanson, D. Tygar, and P. Dasgupta, editors, *ICDCS Workshop on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, Austin, TX, USA, 1999.

[10] J. Bates, J. Bacon, K. Moody, and M. Spiteri. Using events for the scalable federation of heterogeneous components. In P. Guedes and J. Bacon, editors, *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, Sept. 1998.

[11] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In J. Fabre and F. Jahanian, editors, *The International Conference on Dependable Systems and Networks*. IEEE Press, jun 2002.

[12] C. Bornhövd, M. Cilia, C. Liebig, and A. Buchmann. An infrastructure for meta-auctions. In *Second International Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'00)*, San Jose, California, June 2000.

[13] G. Bricconi, E. D. Nitto, A. Fuggetta, and E. Tracanella. Analyzing the behavior of event dispatching systems through simulation. In *In the Proceedings of the 7th International Conference on High Performance Computing IEEE*, 2000.

[14] G. Bricconi, E. D. Nitto, and E. Tracanella. Issues in analyzing the behavior of event dispatching systems. In *In the Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD-10)*, 2000.

[15] P. Buneman. Semistructured data. In *In Proc. of the 16th ACM SIGACT SIGMOD SIGART Sym. on Principles of Database Systems (PODS'97)*, pages 117–121, 1997.

[16] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, Apr. 1989.

[17] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.

[18] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In *Proceedings of the Third International Workshop on Software Architecture (ISAW '98)*, pages 17–20, Orlando, FL, USA, 1998.

[19] A. Carzaniga, D. Rosenblum, and A. Wolf. Content-based addressing and routing: A general model and its application. Technical Report

CU-CS-902-00, Department of Computer Science, University of Colorado, USA, 2000.

[20] A. Carzaniga, D. R. Rosenblum, and A. L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *Engineering Distributed Objects '99*, May 1999.

[21] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 219–228, NY, July 16–19 2000. ACM Press.

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

[23] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, Scottsdale, AZ, Oct. 2001.

[24] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390. SIGMOD, 2000.

[25] M. Cilia, C. Bornhövd, and A. P. Buchmann. Moving active functionality from centralized to open distributed heterogeneous environments. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 195–210, Trento, Italy, 2001. Springer-Verlag.

[26] M. Cilia and A. P. Buchmann. An active functionality service for e-business applications. *ACM SIGMOD Record*, 31(1):24–30, 2002.

[27] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Efficient query subscription processing in a multicast environment. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, 2000.

[28] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 261–270. IEEE Computer Society Press / ACM Press, 1998.

[29] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), 2001.

[30] G. Cugola, E. D. Nitto, and G. P. Picco. Content-based dispatching in a mobile environment. In *In Workshop su Sistemi Distribuiti: Algorithmi, Architectture e Linguaggi (WSDAAL)*, 2000.

[31] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[32] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[33] P. Eugster, R. Guerraoui, and C. Damm. Linguistic support for large-scale distributed programming. In *Proceedings of the Intl. Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 131–146. ACM Press, 2001.

[34] P. Eugster, R. Guerraoui, and J. Sventek. Type-based publish/subscribe. Technical Report DSC ID:200029, EPFL Lausanne, 2000.

[35] P. T. Eugster and R. Guerraoui. Content-based publish/subscribe with structural reflection. In *In 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, 2001.

[36] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In T. Sellis and S. Mehrotra, editors, *Proceedings of the 20th Intl. Conference on Management of Data (SIGMOD 2001)*, pages 115–126, sanat Barbara, CA, USA, 2001.

[37] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha. Efficient matching for content-based publish/subscribe systems. Technical report, INRIA, 2000.

[38] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 309–333, Malagá, Spain, June 2002. Springer-Verlag.

[39] L. Fiege and G. Mühl. Webevents. http://www.gkec.informatik.tu-darmstadt.de/rebeca/demo.html.

[40] L. Fiege and G. Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.

[41] L. Fiege, G. Mühl, and A. Buchmann. An architectural framework for electronic commerce applications. In *Informatik 2001: Annual Conference of the German Computer Society*, 2001.

[42] L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.

[43] L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 2002. accepted for publication.

[44] G. Fitzpatrick, T. Mansfield, S. Kaplan, D. Arnold, T. Phelps, and B. Segall. Augmenting the workaday world with elvin. In S. Bødker, M. Kyng, and K. Schmidt, editors, *Proceedings of the Sixth European Conference on Computer Support Cooperative Work (ECSCW-99)*, pages 431–450, Dordrecht, NL, Sept. 12–16 1999. Kluwer Academic Publishers.

[45] M. J. Franklin and S. B. Zdonik. "Data In Your Face": Push Technology in Perspective. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 516–519. ACM Press, 1998.

[46] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1), Mar. 1999.

[47] F. C. Gärtner. *Formale Grundlagen der Fehlertoleranz in verteilten Systemen*. PhD thesis, TU Darmstadt, 2001.

[48] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[49] P. Godfrey and J. Gryz. Answering queries by semantic caches. In *Database and Expert Systems Applications (DEXA) LNCS Vol. 1677*, pages 485–498. Springer, 1999.

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

[51] O. M. Group. CORBA Messaging Specification. OMG Document orbos/98-0505, May 1998.

[52] T. O. Group. *DCE 1.1: Remote Procedure Call*. Technical Standard C706. The Open Group, Cambridge, MA, USA, 1997.

[53] A. Y. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29, Dec. 2000.

[54] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *19 ACM SIGMOD Conf. on the Management of Data, Atlantic City*, pages 271–280, May 1990.

[55] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems,*

*Languages and Applications (OOPSLA-97)*, pages 184–200. ACM Press, 1997.

[56] R. Hayton, J. Bacon, J. Bates, and K. Moody. Using events to build large scale distributed applications. In *Proceedings of the ACM SIGOPS European Workshop*, 1996.

[57] D. Heimbigner. Adapting publish/subscribe middleware to achieve gnutella-like functionality. In *Coordination Models, Languages and Applications, Special Track at 2001 ACM Symposium on Applied Computing (SAC 2001)*, 2001.

[58] Y. Huang and G.-M. Hector. Exactly-once semantics in a replicated messaging system. In *Proc. of the 17th International Conference on Data Engeneering (ICDE)*, 2001.

[59] IBM. Gryphon: Publish/subscribe over public networks. Technical report, IBM T. J. Watson Research Center, 2001.

[60] J. Kaiser and M. Mock. Implementing the real-time publisher/subscriber model on the controller area network (can). In *2nd Int'l Symposium on Object-Oriented Distributed Real-Time Computing Systems*, 1999.

[61] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[62] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, Mar. 1977.

[63] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1995.

[64] M. Langheinrich, F. Mattern, K. Römer, and H. Vogt. First steps towards an event-based infrastructure for smart things. In *Ubiquitous Computing Workshop (PACT 2000)*, Philadelphia, PA, USA, Oct. 2000.

[65] D. Lee and W. W. Chu. Conjunctive point predicate-based semantic caching for wrappers in web databases. In *Workshop on Web Information and Data Management*, 1998.

[66] C. Liebig, B. Boesling, and A. Buchmann. A notification service for next-generation it systems in air traffic control. In *GI-Workshop: Multicast-Protokolle und Anwendungen*, Braunschweig, Germany, May 1999.

[67] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of the 4th Intl. Conference on Cooperative Information Systems (CoopIS '99)*, Sept. 1999.

[68] C. Liebig, M. Malva, and A. Buchmann. Integrating notifications and transactions: Concepts and X$^2$TS prototype. In *Proceedings of EDO 2000*, Nov. 2000.

[69] C. Liebig, M. Malva, and A. Buchmann. X$^2$TS: Unbundling active object systems. In J. Sventek and G. Coulson, editors, *Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, volume 1795 of *LNCS*. Springer-Verlag, Apr. 2000.

[70] L. Liu, C. Pu, W. Tang, and W. Han. Conquer: A continual query system for update monitoring in the wwww. *International journal of Computer Systems, Science and Engineering, Special issue on Web semantics*, 1999.

[71] C. Ma and J. Bacon. COBEA: A CORBA-based event architecture. In J. Sventek, editor, *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pages 117–132, Santa Fe, NM, USA, 1998. USENIX Association.

[72] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

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

[74] G. Mühl. Inhaltsbasierte Benachrichtigungsdienste. In *Informatiktage 2001*. Konradin Verlag, 2001.

[75] G. Mühl and L. Fiege. Supporting covering and merging in content-based publish/subscribe systems: Beyond name/value pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.

[76] G. Mühl, L. Fiege, and A. P. Buchmann. Evaluation of cooperation models for electronic business. In *Information Systems for E-Commerce, Conference of German Society for Computer Science / EMISA*, pages 81–94, Nov. 2000. ISBN 3-85487-194-5.

[77] G. Mühl, L. Fiege, and A. P. Buchmann. Filter similarities in content-based publish/subscribe systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *Lecture Notes in Computer Science*, pages 224–238, Karlsruhe, Germany, 2002. Springer-Verlag.

[78] G. Mühl, L. Fiege, F. C. Gärtner, and A. P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe

systems. In A. Boukerche and S. Majumdar, editors, *The Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, TX, USA, October 2002. IEEE Press.

[79] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *SIGMOD Record*, 2001.

[80] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Version 2.3. Object Management Group, Framingham, MA, USA, 1998.

[81] Object Management Group. Corba notification service. OMG Document telecom/99-07-01, 1999.

[82] Object Management Group. CORBA event service specification, version 1.0. OMG Document formal/2000-06-15, 2000.

[83] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus—an architecture for extensible distributed systems. In B. Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 58–68, Asheville, NC, United States, Dec. 1993. ACM Press.

[84] O. M. G. (OMG). Complete uml 1.4 specification, 2001.

[85] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In J. Sventek and G. Coulson, editors, *Middleware 2000*, volume 1795 of *LNCS*, pages 185–207. Springer-Verlag, 2000.

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

[87] N. W. Paton, editor. *Active Rules in Database Systems.* Monographs in Computer Science. Springer, 1999. ISBN 0-387-98529-8.

[88] J. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha. Publish/subscribe on the Web at extreme speed. In A. El Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 627–630, Cairo, Egypt, 2000. Morgan Kaufmann Publishers.

[89] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In O. Etzion and P. Scheuermann, editors, *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, volume 1901 of *LNCS*, Eilat, Israel, 2000. Springer-Verlag.

[90] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In J. Bacon, L. Fiege, R. Guerraoui, H.-A. Jacobsen, and G. Mühl, editors, *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, July 2002. IEEE Press. Published as part of the ICDCS '02 Workshop Proceedings.

[91] P. R. Pietzuch. Event-based middleware: A new paradigm for wide-area distributed systems? In *6th CaberNet Radicals Workshop*, February 2002.

[92] D. Platt. The COM+ event service eases the pain of publishing and subscribing to data. *Microsofts Systems Journal*, September 1999.

[93] D. S. Platt. *Understanding COM+*. Microsoft Press, 1999.

[94] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[95] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[96] D. Rosenblum, A. Wolf, and A. Carzaniga. Critical considerations and designs for internet-scale, event-based compositional architectures. In *Workshop on Compositional Software Architectures*, Monterey CA, USA, 1998.

[97] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In J. Crowcroft and M. Hofmann, editors, *Third International on Networked Group Communication (NGC 2001)*, volume 2233 of *LNCS*, pages 30–43, London, UK, 2001. Springer-Verlag.

[98] M. Schneider. Self-stabilization. *ACM Computing Surveys (CSUR)*, 25(1):45–67, 1993.

[99] S. Schwiderski. *Monitoring the behaviour of distributed systems*. PhD thesis, University of Cambridge, April 1996.

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

[101] W. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia, September 1997.*, 1997. http://elvin.dstc.edu.au/doc/papers/auug97/AUUG97.html.

[102] N. Skarmeas and K. Clark. Content-based routing as the basis for intra-agent communication. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent*

*Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555 of *LNAI*, pages 345–362, Berlin, July 04–07 1999. Springer.

[103] D. Sturman, G. Banavar, and R. Strom. Reflection in the gryphon message brokering system. In *In Reflection Workshop of the 13th ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, 1998.

[104] Sun Microsystems, Inc. Java 2 Platform Enterprise Edition. http://java.sun.com/j2ee/.

[105] Sun Microsystems, Inc. Java AWT delegation event model.

[106] Sun Microsystems Inc. Java RMI.

[107] Sun Microsystems, Inc. JavaBeans. http://java.sun.com/products/javabeans/.

[108] Sun Microsystems, Inc. *Network Programming*. Sun Microsystems, Mountain View, CA, Mar. 1990.

[109] Sun Microsystems, Inc. Distributed Event Specification, 1998.

[110] Sun Microsystems, Inc. JavaSpaces Service Specification version 1.1, 2000.

[111] Sun Microsystems, Inc. Java Message Service Specification 1.1, 2002.

[112] W. Tang. *Scalable Trigger Processing and Change Notification in the Continual Query System*. Oregon Graduate Institute, 1999.

[113] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[114] TIBCO, Inc. TIB/Rendezvous. White Paper, 1996. http://www.rv.tibco.com/.

[115] J. D. Ullman. Information integration using logical views. In *6th Int. Conference on Database Theory; LNCS 1186*, pages 19–40. LNCS 1186, Springer, 1997.

[116] J. Vitek, N. Horspool, and A. Krall. Efficient type inclusion tests. *ACM SIGPLAN Notices*, 32(10):142–157, Oct. 1997.

[117] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security issues and requirements for Internet-scale publish-subscribe systems. In *Proceedings of the Thirtyfifth Hawaii International Conference on System Sciences (HICSS-35)*, Big Island, Hawaii, Jan. 2002.

[118] World Wide Web Consortium (W3C). Extensible markup language (XML) 1.0 (second edition), 2000.

[119] World Wide Web Consortium (W3C). XML path language (XPath) version 1.0, 2000.

[120] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, 1994.

[121] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination. In *The 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'01)*, June 2001.

# Lebenslauf

| | | |
|---|---|---|
| Name: | Gero Mühl | |
| Geburtsdatum: | 18.10.1973 | |
| Familienstand: | ledig | |
| Geburtsort: | Lüdenscheid | |
| Staatsangehörigkeit: | Deutsch | |
| Schulbildung: | 1980-1984 | Grundschule Rothenstein Meinerzhagen |
| | 1984-1992 | Evangelisches Gymnasium Meinerzhagen |
| Zivildienst: | 01.08.92-31.10.93 | Kreiskrankenhaus Lüdenscheid Hellersen |
| Studium: | 01.10.92-15.01.98 | Informatik mit Nebenfach Elektrotechnik an der FernUniversität in Hagen (Abschluss Diplom-Informatiker) |
| | 01.10.93-28.07.98 | Elektrotechnik an der FernUniversität in Hagen (Abschluss Diplom-Ingenieur) |
| Promotionsstudium: | 01.10.98-30.09.02 | Stipendiat/Kollegiat im Graduiertenkolleg „Infrastruktur für den elektronischen Markt“ am Fachbereich Informatik der Technischen Universität Darmstadt |
| Arbeitsverhältnisse: | 02.01.95-31.01.96 | studentische Hilfskraft am Lehrgebiet Praktische Informatik VI der FernUniversität in Hagen |
| | 01.02.96-31.08.98 | studentische Hilfskraft am Institut für Systementwurfstechnik der GMD - Forschungszentrum Informationstechnik GmbH in St. Augustin |
| | 01.10.01-30.09.02 | wissenschaftliche Hilfskraft am Fachgebiet „Datenbanken und verteilte Systeme“ des Fachbereichs Informatik der Technischen Universität Darmstadt |
| Sonstiges: | | Stipendiat der Studienstiftung des deutschen Volkes |

# Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades „Dr.-Ing.“ mit dem Titel „Large-Scale Content-Based Publish/Subscribe Systems“ selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 19.08.2002 Gero Mühl