Engineering Publish/Subscribe Systems and Event-Driven Applications

Dissertation

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

vorgelegt von
Helge Parzyjegla, geb. am 15.03.1980 in Berlin
aus Mühlenbecker Land

Rostock, 10. Dezember 2012

urn:nbn:de:gbv:28-diss2014-0028-3

Gutachter: Prof. Dr.-Ing. Gero Mühl
Universität Rostock

Prof. Dr. Hans-Ulrich Heiß
Technische Universität Berlin

Prof. Christof Fetzer, PhD
Technische Universität Dresden

Tag der Einreichung: 10. Dezember 2012
Tag der Verteidigung: 15. März 2013

Abstract

Driven by advances in information and communication technology, computing infrastructures continue to grow in size and complexity while people increasingly depend on them. To better master their inherent scale and complexity, modern computing systems are often designed to be self-managing. They are aware of their state as well as of their environment so that they can dynamically react on changes caused by internal or external events in order, for example, to adapt and optimize their configuration or to better support and assist their users.

Publish/subscribe systems provide a versatile basis for building distributed event-based infrastructures. Components communicate by publishing notifications about occurred events and by subscribing to those notifications of interest. The resulting characteristic loose coupling between participating components is both a big strength and a severe weakness. On the one side, the indirect communication decouples components in space, time, and control flow and, thereby, provides the flexibility and extensibility required by modern event-driven applications. On the other side, when systems grow in size, it becomes difficult or, in practice, even impossible to keep track of all effects and side-effects that may be caused by a published notification. In this thesis, we address the latter problem and develop structuring means to modularly design and engineer large-scale event-based infrastructures. We consider modularity and related engineering aspects at the level of the architecture, the middleware, and the application.

For publish/subscribe systems, we propose a broker architecture based on the concept of *features* and their *composition*. This way, middleware developers can easily modify, extend, or add individual broker features in order to tailor the overall functionality to actual application requirements and environment conditions. We show how features are implemented as pluggable broker components and discuss interfaces and guidelines for their composition.

With *scoping*, we present a module concept to structure publish/subscribe networks and event-driven applications. Scopes bundle related components to new application artifacts providing them an isolated environment for interaction. Therefore, the visibility of a notification is restricted to components of the same scope only, unless the scope interface allows its exchange with other components. Scopes can flexibly be arranged in hierarchies that help decomposing applications and infrastructure according to multiple aspects such as functional requirements or organizational responsibilities. Based on annotated and inherited attributes, application components are then automatically assigned respective scopes.

For application components, we introduce *event ports* as a new publish/subscribe interface for communicating relevant changes in the component's state to other interested parties. Components are then orchestrated either traditionally by subscribing their event ports to the notifications they have to process or alternatively by virtually connecting the event ports with compatible ports of other components. Further accompanying programming abstractions support developers and administrators to conveniently group components into scopes and subscopes as well as to customize their configuration.

Besides the conceptual work, the thesis contains a practical part. The presented composable publish/subscribe architecture forms the basis of our implementation of the REBECA notification service. The architecture's extensibility and flexibility is demonstrated by enabling REBECA brokers to be executed in a real network deployment as well as in a simulation environment. The latter is leveraged to prove the effectiveness and thoroughly evaluate the behavior of scoping in large-scale publish/subscribe networks under different application scenarios.

Kurzfassung

Der anhaltende Fortschritt auf dem Gebiet der Informations- und Kommunikationstechnologie lässt die Größe und Komplexität eingesetzter Computerinfrastrukturen beständig wachsen, während zunehmend mehr Menschen auf ihren Dienst angewiesen sind. Um Größe und Komplexität besser zu handhaben, werden moderne Computersysteme oft selbstmanagend entworfen: Sie sind sich über ihren Zustand und den ihrer Umgebung bewusst, so dass sie dynamisch auf interne und externe Ereignisse reagieren können, um beispielsweise ihre Konfiguration anzupassen oder Nutzer proaktiv zu unterstützen.

Publish/Subscribe-Systeme bieten eine flexible Grundlage zur Realisierung derartiger ereignisbasierter Infrastrukturen. Komponenten kommunizieren miteinander durch die Veröffentlichung von Notifikationen über aufgetretende Ereignisse und die Subskription der für sie relevanten Notifikationen. Die resultierende lose Kopplung ist sowohl Stärke als auch Schwäche von Publish/Subscribe. Einerseits entkoppelt die indirekte Kommunikation beteiligte Komponenten in Raum, in Zeit und im Kontrollfluss, wodurch sich ereignisgetriebene Anwendungen vielseitig einsetzen und erweitern lassen. Andererseits wird es schwierig und in sehr großen Systemen praktisch unmöglich, den Überblick über sämtliche Effekte, Seiteneffekte und Folgen einer veröffentlichten Notifikation zu behalten. In dieser Arbeit werden daher strukturelle Abstraktionen für den modularen Entwurf und Betrieb großer ereignisbasierter Infrastrukturen entwickelt. Modularität wird hierbei auf Ebene der Architektur, der Middleware und der Applikationen betrachtet.

Für verteilte Publish/Subscribe-Systeme wird eine Broker-Architektur vorgeschlagen, die eine möglichst freie *Komposition von Features* erlaubt. Einzelne Broker-Features lassen sich austauschen, modifizieren oder hinzufügen, um den Funktionsumfang des Brokers präzise an die Anforderungen der Applikationen und die Gegebenheiten der Netzwerkumgebung anzupassen. Die Implementierung der Features erfolgt in Form von Plugins. Schnittstellen und Regeln für deren Komposition werden diskutiert.

Mit *Scoping* wird ein Modulkonzept zur Strukturierung verteilter Publish/Subscribe-Systeme vorgestellt. Scopes fassen Applikationskomponenten zu neuen Artefakten zusammen und bieten diesen eine geschützte Interaktionsumgebung.

Die Sichtbarkeit von Notifikationen ist auf Komponenten des gleichen Scopes beschränkt, die Schnittstelle des Scopes regelt den Austausch von Notifikationen mit anderen Scopes. Scopes lassen sich hierarchisch sowie überlappend anordnen und erlauben so eine flexible Dekomposition des Systems nach verschiedenen Kriterien wie funktionalen oder organisatorischen Aspekten. Basierend auf vorhandenen, vererbten oder annotierten Attributen werden Komponenten automatisch ihren jeweiligen Scopes zugewiesen.

Für ereignisgetriebene Applikationskomponenten werden *Event-Ports* als Publish/Subscribe-Schnittstelle eingeführt, die den direkten Austausch relevanter Zustandsänderungen zwischen Komponenten ermöglicht. Die Orchestrierung von Komponenten und Notifikationsströmen erfolgt dann entweder traditionell durch Subskription der Event-Ports auf zu verarbeitende Notifikationen oder alternativ mittels logischer Verbindungen zwischen kompatiblen Event-Ports verschiedener Komponenten. Weitere vorgestellte Programmierabstraktionen unterstützen Entwickler und Administratoren bei der Gruppierung von Komponenten in Scopes und Subscopes sowie bei der Anpassung ihrer Konfiguration.

Im praktischen Teil der Arbeit wird die Tragfähigkeit der vorgestellten Konzepte untersucht. Die komponierbare Broker-Architektur bildet die Grundlage für die Implementierung des REBECA-Notifikationsdienstes. Die durch Komponierbarkeit gewonnene Flexibilität zeigt sich insbesondere durch die Möglichkeit REBECA-Broker sowohl in einem realen Netzwerk als auch in einer Simulationsumgebung ausführen zu können. Letztere wird genutzt um die Effektivität von Scoping in großen Publish/Subscribe-Infrastrukturen zu demonstrieren und das Systemverhalten unter verschiedenen Applikationsszenarien zu evaluieren.

Preface

Acknowledgements

This work would not have been possible without the support of various people. First of all, I thank my advisors Prof. Dr.-Ing. Gero Mühl and Prof. Dr. Hans-Ulrich Heiß for their support, guidance, patience, and wisdom during the all the time. I am very grateful to Prof. Christof Fetzer, PhD, for taking over the part of the third referee.

I also thank my dear colleagues at Berlin and Rostock: Anselm Busse, Daniel Graff, Dr.-Ing. Michael A. Jaeger, Nikolaus Jeremic, Dr.-Ing. Jan Richling, Jan Schönherr, Dr.-Ing. Jörg Schneider, Arnd Schröter, Enrico Seib, and Gabriele Wenzel – it was a pleasure to work with you!

And most of all, I thank my parents and my family for their love and support.

Publications

Parts of the thesis are based on papers already published during the work on it. Chapter 3 is based on joint work with Daniel Graff, Arnd Schröter, Jan Richling, and Gero Mühl [166] with accompanying publications [170, 171]. The theoretical background to the chosen simulation approach for publish/subscribe networks in Chap. 6 is based on work with Arnd Schröter, Gero Mühl, Samuel Kounev, and Jan Richling [200] that continued previous work [148]. Several publications [101, 102, 103, 169, 199, 201] are only distantly related to the presented work, while other papers [85, 86, 150, 167, 168, 198, 202, 203, 224, 225, 226] were not incorporated at all.

Contents

List of Figures

Chapter 1

Introduction

With the invention of the integrated circuit in the late 1950s [108], an unprecedented technical development began, which is far from being over. Meanwhile, the resulting advances in information and communication technology revolutionize the daily lives of billions of people. The Internet [119], for example, has changed the way how people communicate and interact with each other, how they work and conduct business, how they learn, play, and search for information, or how they produce, share and consume content and media. Thereby, the millions of interconnected computers and computer networks of the Internet as well as the services and applications running on them form a huge distributed and complex system. With microchips, sensors, and actuators pervading our environment and computing devices becoming smaller, faster, and cheaper as well as being always connected, it is to be expected that future computing infrastructures and their applications will continue to grow in *size* and *complexity* while people increasingly depend on them.

In order to master their inherent complexity and keep infrastructures manageable for their users, researchers and engineers work hard to design and render future systems intelligent and smart in the sense that they are able to autonomously manage themselves. By making components and systems aware of their own state and configuration as well as their environment, they can dynamically react on changes caused by internal or external events in order to adapt themselves accordingly or to better support and assist their users. In this thesis, we focus on such *event-based* systems. In particular, we develop and present means to modularly design and engineer large-scale event-based infrastructures. Thereby, we take a holistic approach that ranges from architectural guidelines for composable infrastructures over middleware concepts for bundling related components to programming abstractions for event-driven applications that help developers to conveniently leverage the provided concepts.

1.1 Motivation

The term *event*[1] may be generally characterized as a *happening of interest* [82]. Events usually occur asynchronously and may vary in size and complexity ranging from simple hardware interrupts to sophisticated database updates. Please note that the handling and processing of asynchronous events is a well-proven and established method leveraged in many computing domains where it supplements synchronous techniques or even represents the norm. In the following, we give examples from different domains to show the various applications of events as well as to illustrate inherent characteristics and resulting benefits.

Interrupts. Hardware interrupts [134, 192] are low level events by which, for example, an input/output device electrically signals the processor that it needs service. Usually, this is the case when the device has finished an operation and either has fetched input data that is now ready for further processing or it is awaiting new data to output next. The processor then acknowledges the interrupt signal and switches to a corresponding service routine that takes the necessary actions for handling the device. After successfully servicing the device, the processor returns from the interrupt routine and resumes its original thread of execution. Without interrupts, the processor would be stuck to periodically poll the device in order to check whether any action is required. Usually, this is very inefficient since in the majority of cases nothing has to be done and checking the device just wastes valuable processor cycles. Modern operating systems also leverage interrupts to implement *context switches* by which they efficiently handle multiple tasks and processes running concurrently. In this sense, interrupt events lead the way from former batch processing systems to those interactive systems as we know computers today.

Graphical user interfaces. Graphical user interfaces (GUIs) [115, 124] are inherently event-driven. Input data is provided asynchronously in form of user events such as key strokes, mouse clicks, or finger taps. Before being passed to the application, these events are usually preprocessed by the GUI library. For this purpose, GUI *toolkits* allow application developers to build versatile interfaces made up of prefabricated visual controls such as menu bars, input fields, or buttons. The library first checks to which of the controls the user event has to be applied, i.e., which menu item or input field was chosen or which button was clicked, respectively. Thereafter, a corresponding event handler is called that the application developer previously registered for this specific combination of event and control element. The handler then processes provided input data and executes the respective application function either by further delegating the call or by implementing it itself. This way, GUIs leverage events and event handlers to flexibly wire visual controls to application functions while both the GUI el-

[1] A more precise definition of *events* is given in Sect. 2.2.1.

ements as well as the application logic may remain encapsulated in their own components and, thereby, substantially improve their reusability.

Active databases. Active databases supplement regular database management systems with the capability to specify reactive behavior [54]. For this purpose, *event-condition-action* (ECA) rules are often used for specification [129, 172]. The event identifies the happening to which the rule responds while the condition allows to evaluate the context of the event's occurrence. If both match, the specified action is finally executed. In case of relational databases, an event may be, for example, the insertion or deletion of a tuple, the update of an attribute, or a transaction being started, committed, or aborted. Conditions consist of predicates over attributes and queries that put the tuples triggering the event in relation to other information stored in the database. Actions are usually not limited to database operations only and may also include the execution of external processes and applications. This way, it is, for example, easily possible to automatically reorder a particular item when the current stock level drops below a given threshold. Furthermore, executed actions may also trigger new events to be fired and corresponding rules to be processed. Hence, even complex and sophisticated business workflows requiring several processing steps can be appropriately mapped onto and implemented by active databases. Thereby, they allow to efficiently monitor the stored data for events of interest, and in case of their occurrence, to respond to them in a timely manner. Because of this economic relevance, nowadays, nearly all industrial-strength databases also feature active capabilities.

Publish/subscribe systems. Publish/subscribe [58] is a flexible group communication paradigm enabling distributed components to interact by publishing notifications about occurred events and subscribing to those notifications of interest. Since subscriptions usually specify the type or just the content of the notifications in which a component is interested, the communication is indirect requiring only a loose coupling between publishers and subscribers. In fact, publishers do not necessarily need to know their subscribers and vice versa making it easy to extend systems by flexibly adding new components. For this to work, it is usually the responsibility of a *notification service* to distribute a published notification to all interested subscribers. In large-scale scenarios, the notification service is often formed by a set of cooperating brokers that exchange published notifications and issued subscriptions [144]. Notifications are routed stepwise through the broker network, duplicated where necessary, and finally pushed to their subscribers. This way, it is ensured that event notifications are actively delivered in a timely manner while publishers are relieved from contacting each subscriber individually and subscribers do not need to periodically poll for updates. Hence, notification services and publish/subscribe systems are well suited and, therefore, widely used for applications that require the efficient dissemination of information to possible large numbers of clients and components.

As the given examples demonstrate, events are leveraged in a surprisingly broad field of computing including hardware and software systems as well as local and distributed applications. Requirements and objectives are manifold leading to substantially different implementations. But throughout the various application domains, the primary purpose of events is to efficiently notify system components about happenings and situations of interest and, thereby, allow the system to respond to these events in a timely fashion. With computing infrastructures growing in size and complexity, both aspects are gaining particular importance. On the one side, the growing number of data sources and computing nodes requires an efficient distribution of data making a periodic polling for updates no longer sustainable in large-scale scenarios. On the other side, infrastructures have to become more and more capable to manage and organize themselves as their increasing complexity often overwhelms users and administrators alike. Therefore, it is usually essential that systems and applications are able to quickly react on detected changes, for example, in order to adapt and optimize their configuration or to actively support and assist their users. This being said, it can, thus, be anticipated that event-based systems and applications will play a central role in future computing infrastructures.

In particular, publish/subscribe systems are ideally suited to flexibly connect event-driven application components and event-based subsystems within distributed infrastructures [161]. As all communication is performed indirectly by solely publishing and subscribing notifications about occurred events, there is only a *loose coupling* between interacting components as they only have to agree on a notification's content. On the one hand, it is, thus, always possible to easily add new components without the need to modify already existing ones and, thereby, naturally support computing infrastructures when they grow in size. On the other hand, the loose coupling makes it hard to identify the actual interactions between communicating components especially if use cases go far beyond unidirectional data dissemination. In practice, it is, for example, nearly impossible to determine all effects and side effects that a published notification may cause as any component within the system could be a potentially affected subscriber. Hence, without further structuring means, conventional publish/subscribe systems are only of limited use to cope with the increasing complexity resulting from interdependencies between system components.

In order to better manage component interdependencies, event-based infrastructures essentially require structural abstractions allowing publish/subscribe systems and event-driven applications to be designed and developed in a modular fashion. Although software engineering research knows the importance and benefits of structuring concepts such as modules [165], classes [21], and components [217], nevertheless, comparable concepts are apparently missing for event-based infrastructures. In this thesis, we exactly address the lack of *engineering means* to organize event-based infrastructures. However, developing those structuring concepts is challenging. In fact, they have to allow event-based systems to be engineered in a modular fashion while, at the same time, preserve the inherent benefits of a loose coupling between system components.

1.2 Goals

The term *event cloud* refers to the complete set of event notifications generated by one or more event-based systems [125]. The term is quite figurative illustrating the jumble of events independently produced by individual system components and addresses the problem to distinguish relevant from irrelevant information since usually all kinds of event notifications including low-level as well as high-level ones equally show up side by side within the publish/subscribe infrastructure. The sheer mass of event notifications then makes it hard to clearly identify those events of importance among the whole data noise. Moreover, it is even harder to determine their effects and side effects within the system because a published event notification can, intentionally or not, be received by any system component affecting its state and behavior.

To remedy the problems described above, there are basically two different options. On the one side, one may improve the expressiveness of tools and filter languages in order to more precisely specify and query for those event notifications and event patterns[2] one is interested in. On the other side, one may better structure and organize applications and publish/subscribe systems so that notifications are only visible within their scope of relevance, i.e., within respective subsystems and only to those application components they are intended for. In this thesis, we follow and emphasize the second approach by providing structural abstractions for event-based systems to ease their development and management alike. We also believe that, at the same time, a structured and organized event cloud significantly simplifies event queries and specifications.

More precisely, we aim for a systematic engineering approach that allows to modularly compose publish/subscribe systems and event-driven applications and to effectively orchestrate and control the event flows between their components. This way, we want to increase the reusability and extensibility of event-driven components and publish/subscribe systems, while keeping applications and infrastructure comprehensible and manageable, respectively. In the following, we address each of these aspects in more detail.

Modularity and composability. *Modularity* and *composability* are central ideas of many engineering approaches [12]. They allow to subdivide a complex project into simpler components and modules usually realizing a particular system function, design and implement this function in isolation, and finally compose and connect the individual modules to assemble the whole system. For distributed publish/subscribe applications, however, a modular development approach is only possible to a limited extent. If a particular application function requires the interaction of two or more components distributed in the system, any notification published in the context of this interaction automatically becomes globally visible in the whole infrastructure breaching any isolation boundaries. Thus,

[2] *Complex event processing* (CEP) is primarily concerned with specifying and detecting spatio-temporal patterns of events [125].

great care needs to be taken when deploying similar components or multiple instances of the same component in order to avoid their notifications to unpredictably interfere with each other. Therefore, our primary goal is to provide effective structuring means that, on the one hand, comply with the loose coupling and the flexibility of event-based systems and, on the other hand, allow to bundle related components to modules facilitating their systematic composition to profit from the benefits of a modular development approach.

Please note that this goal is not limited to event-driven applications only. It also includes the publish/subscribe infrastructure itself. Conventional publish/subscribe systems are usually build for specific types of applications in well-defined environments, for example, to disseminate business events within a company network minimizing the latency or to gather measurement readings within a wireless sensor network coping with very limited resources such as network bandwidth, computing power, and energy. Although they all leverage the publish/subscribe paradigm, their implementations often differ substantially and, thus, are incompatible to each other. However, with information and communication technology pervading our environments, it seems to be only a matter of time until the need arises to connect, join, and merge these different systems to a common infrastructure. A modular system architecture significantly eases this process allowing to flexibly compose and integrate features and protocols in order to make these heterogeneous publish/subscribe systems compatible.

Reusability and extensibility. *Reusability* and *extensibility* are key benefits of a modular design [17]. As components or modules usually implement a particular function in a self-contained fashion, they can often be reused without modifications. If a slightly different functionality is required, it is often sufficient to simply adapt or extent an existing component or module as needed. This way, a lot of programming effort is saved as the same or similar functions do not need to be developed from scratch over and over again. Due to the lack of effective module concepts for event-based infrastructures, publish/subscribe applications are the less reusable the more intertwined their interaction patterns are. But even single event-driven components are often not reusable or extensible at all. In particular, such a component must know to which event notifications it has to subscribe, which notification attributes are of specific interest, and how generated results need to be published. Therefore, event-driven components often contain a mixture of business logic, application context, and system configuration that drastically reduces their reusability and extensibility.

Likewise, publish/subscribe systems are usually designed to support a particular application type and, thus, equipped with a well-defined set of protocols and features optimized for the intended usage. Often this leads to powerful and efficient, but monolithic implementations. Moreover, basic assumptions may only be valid in specific application domains and computing environments making those publish/subscribe systems hardly reusable and extensible. With the growing diversity, heterogeneity, and interconnectivity of event-based infrastructures, however, the latter becomes increasingly important. Therefore, we advocate for a flexible and modular system architecture. This way, the integration of new and the adaptation of present features and protocols simply boils down to adding and modifying new and existing system modules, respectively. Hence, modularity is a necessary precondition for both sustainable system and application designs. But a functional decomposition in system modules and application components is not always sufficient. Additionally, it is also necessary to clearly separate application logic and system features from context information and system configuration in order to make modules and components inherently reusable and extensible and, thereby, profit the more from provided structuring means.

Comprehensibility and manageability. Especially the loose coupling of components makes it difficult to predict and comprehend the overall behavior of event-based systems. The behavior emerges from the interactions of all participating components resulting from the event notifications exchanged between them. Thereby, the flow of notifications is usually defined indirectly by the content of published notifications and the filter expressions of issued subscriptions. By matching every event notification against all active subscriptions, the set of receivers is determined dynamically and may potentially include any system component while changing from notification to notification. Hence, the effects and side effects of a published notification, intended or not, are hardly predictable without analyzing the complete publish/subscribe system and all application components. Moreover, adding, modifying, or removing any component, thus, requires a thorough reanalysis of the whole system. With the growing number of components and sophisticated interaction patterns, a system analysis becomes increasingly difficult making properties such as *comprehensibility* and *manageability* of event-based infrastructures the more important.

Against this background, structural abstractions facilitating modularity are the key to keep publish/subscribe systems and event-driven applications comprehensible and manageable. Modules help to control component interactions and, thus, allow to examine and analyze individual components and subsystems in isolation. But it is not sufficient to just provide the mere possibility to bundle components to modules as an additional feature. Instead, the module concept needs to be integrated into the programming language so that developers naturally leverage it to organize their event-driven applications. Moreover, it has to pay off by significantly easing the development process and helping programmers to better understand the application behavior. Likewise, the module concept must also support administrators to structure publish/subscribe systems by orchestrating application components and directing their notification flows. Thereby, organizational responsibilities as well as security concerns need to be addressed so that they can be mapped onto modules, too. In fact, structural abstractions and module concepts have to consider both the development process of event-driven applications at design time and the administration and operation of publish/subscribe systems at runtime in order to facilitate comprehensible and manageable event-based infrastructures.

1.3 Contributions

In this thesis, we focus on engineering means to build publish/subscribe systems and event-driven applications. Primarily, we propose a *scoping concept* that provides necessary structural abstractions to design and implement event-based infrastructures in a modular fashion. In particular, scopes bundle components and limit the visibility of exchanged event notifications. This way, it is possible to group components into modules and subsystems and precisely control their interactions. Scopes support both application developers and system administrators to easily create and implement reusable software artifacts and to actively organize and manage the publish/subscribe infrastructure, respectively. We analyze and evaluate scoping in depth covering the scope model and its formal definition, the integration of scopes into both the publish/subscribe middleware and the applications, and the effects of scopes on the system's performance. To ease the integration of scopes, we provide a *composable architecture* for publish/subscribe brokers as well as advanced *programming abstractions* for event-driven applications. On the one side, the presented broker architecture is based on functional modularity supporting middleware developers to adapt existing features and easily add new functionality such as, for example, scopes for system organization. On the other side, the derived programming abstractions help application developers to leverage scopes for their purposes. In the following, we discuss the contributions of the composable architecture, the scope concept, and the programming abstractions in more detail.

Composable publish/subscribe architecture. A middleware architecture for publish/subscribe brokers is developed that is based on the concept of features and *feature composition*. A feature represents a particular functional aspect such as a specific network protocol, a certain routing strategy, or an applied matching algorithm. Features are encapsulated and implemented as plugins that can be inserted into publish/subscribe brokers in order to add necessary functions. By composing features, i.e., selecting the right plugins, this kind of functional modularity enables middleware developers and system administrators to easily tailor the publish/subscribe infrastructure to actual requirements. To realize its functionality, each plugin is allowed to intercept and intervene the internal handling of event messages within a broker in order to modify, alter, or remove passing messages or to insert new ones. We show that this is sufficient to even implement sophisticated features such as scoping. Furthermore, we give guidelines to ease their composition. With the redesign and reimplementation of the Rebeca publish/subscribe middleware based on the proposed architecture, we prove the feasibility of our approach. Brokers are accompanied by diverse plugins that realize mandatory as well as optional publish/subscribe features and, thereby, demonstrate and test their composability. We exemplify the engineering benefits that result from the architecture's modularity and flexibility by extending and replacing several feature plugins in order to execute a Rebeca broker within a real network deployment as well as in a simulation environment.

Scoping. Scoping is introduced as an effective means to structure publish/subscribe systems and event-driven applications alike while complying with the loose coupling of components that is inherent to event-based infrastructures. Basically, scopes bundle related components and restrict the *visibility* of event notifications in order to provide a protected environment for interaction and to avoid unintended side effects. Formally, we define the semantics of scopes based on concepts of set theory enabling a hierarchical system decomposition in scopes and subscopes according to different criteria and aspects. However, to ensure flexible system designs, we allow components to be members of multiple scopes at the same time if several criteria apply. To support the modular engineering of scopes, we define scope interfaces that allow to precisely specify the event notifications that are exchanged with other scopes and components. Furthermore, we introduce scope attributes and their inheritance in order to annotate scopes and contained components with configuration data and context information and, thereby, ease their orchestration.

Scoping is embedded directly into the content-based routing layer forming *scope overlays* within the publish/subscribe system. This way, all major routing algorithms are supported while profiting from a close integration whose benefits are twofold. First, the scope structure is exploited for routing optimizations that stop the unnecessary dissemination of event notifications and subscriptions at scope boundaries as early as possible. Second, scope management operations such as creating or joining scopes are mapped to conventional publish/subscribe functions so that existing program logic and data structures can be reused.

As part of a thorough *evaluation*, we analyze the savings as well as the costs caused by scoping. For this purpose, we execute Rebeca brokers in a simulation environment that allows to conveniently study large-scale networks with arbitrary distributions of publishers, subscribers, and event notifications. For a better qualitative and quantitative interpretation of the results, we compare the measurements for scoping with those obtained for other routing strategies. We show that scoping significantly reduces the size of the brokers' routing tables as well as the message overhead within the network improving the overall system performance. In particular, scoping is especially beneficial for publish/subscribe systems with unequal distributions of clients forming local network regions with hot spots for certain notification types. Furthermore, we prove that our implementation of scopes scales with both an increasing number of brokers and clients independent of the ratio between publishers and subscribers. Finally, we measure the overhead caused by the management of scopes and identify the parameters it depends on as well as their influence on the overall costs.

Programming abstractions. Programming abstractions for content-based publish/subscribe are derived that support application developers and system administrators in orchestrating event-driven applications, organizing their components, and directing the notification flows between them. In particular, provided abstractions help to keep business logic apart from context and configuration data and, thereby, significantly increase the reusability and extensibility of components facilitating a modular development process. Most importantly, we introduce *event ports* as a novel interface that makes a set of annotated component fields visible and accessible to the outside in order to communicate significant state changes as event notifications, i.e., to publish or to get notified about them, respectively. To react upon those state changes, we also allow component methods to be conveniently annotated as event handlers and precisely specify the time and order of their execution. Separated from the application's business logic, components are then orchestrated by subscribing their event ports to the notifications they have to process. Following a data flow approach, we ease component orchestration by enabling developers to simply connect the out-ports of publishing components to the in-ports of subscribing components. Subscription and filter management is done automatically.

Our programming abstractions also support scoping. Event-driven components can be grouped into scopes and subscopes according to component attributes while established event connections remain unaffected. Furthermore, we allow developers to either configure and adapt provided default scopes as needed or to annotate a component to manage its own custom scope. In the latter case, scope-specific events such as members joining or leaving the scope are handled similarly to regular business events and processed in the same way. Nevertheless, to leverage all developed programming abstractions, publish/subscribe brokers require additional functions and services that, for instance, inspect components for annotations or call event handlers appropriately. Thus, based on REBECA's composable publish/subscribe architecture, we implement an advanced component container as a pluggable broker feature fulfilling these requirements and providing the necessary functionality.

1.4 Outline

The structure and organization of this thesis is illustrated in Fig. 1.1. Chapters 1 and 2 lay the foundation of the work and discuss its background. In particular, we introduce event-based infrastructures, elaborate on the basic concepts of publish/subscribe communication, and give an overview about existing notification services that support distributed event-driven applications. Furthermore, we pinpoint shortcomings and derive work objectives that primarily aim at improving the degree of modularity.

Chapters 3, 4, and 5 contain major contributions of the thesis addressing modularity on the level of the middleware design, the publish/subscribe infrastructure, and the event-driven applications, respectively. In Chap. 3, we propose a *composable architecture* for publish/subscribe brokers that enables middleware developers to easily modify, extend, or add individual broker features in order to tailor the overall functionality to actual application requirements and environment conditions. We discuss broker features and their encapsulation in plugins

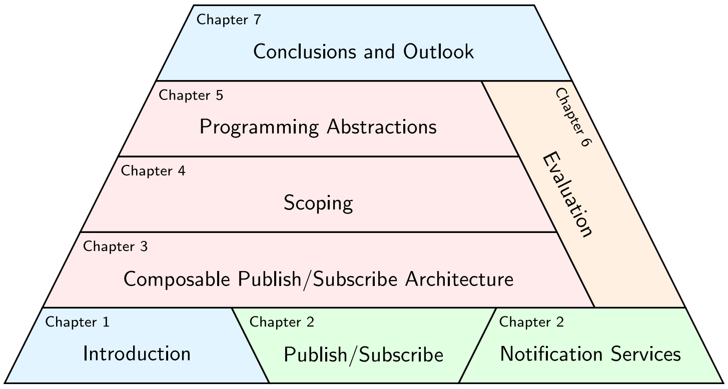


Figure 1.1: Structure and organization of the thesis.

as well as architectural rules and guidelines to facilitate their free composition within a broker. This is exemplified by our publish/subscribe middleware REBECA which builds upon the presented architecture.

Chapter 4 introduces *scopes* as a module concept for publish/subscribe systems. To better control the system interactions, scopes bundle related components to modules and limit the visibility of their event notifications. We formally define the scope model including scope interfaces for notification exchange, scope hierarchies that support a flexible system decomposition as well as scope attributes and their inheritance in order to annotate context information and configuration data. Furthermore, we integrate scoping into the content-based routing layer based on scope overlays and address management issues as well as implementation details.

Chapter 5 derives *programming abstractions* for event-driven publish/subscribe applications to ease the orchestration of their components and better leverage the modularity gained by scoping. After discussing common pitfalls and their remedies, we introduce event ports as component interface to easily communicate relevant changes in the component's state to other interested parties. We show how to connect the event ports of different components using publish/subscribe, how to react on and process received event notifications, and how to automatically group components into scopes and subscopes according to different criteria.

A comprehensive *evaluation* of the presented module concepts is given in Chap. 6. The primary focus is on the system performance and the effects of scoping while created simulation setups leverage the composable broker architecture and provided programming abstractions. We demonstrate the scalability of scoping, study the system behavior under different distributions of clients and notifications, and analyze the overhead introduced by scopes and their management.

Finally, Chap. 7 concludes the thesis. We summarize the work done and review our main results and their contributions for engineering publish/subscribe systems and event-driven applications. Furthermore, we pinpoint further questions newly raised by the thesis while giving an outlook of future research and the next steps to address these open issues.

Chapter 2

Distributed Event-based Systems

Contents

2.1 Introduction

Middleware [16] is an additional software layer introduced to facilitate and ease application-to-application programming. It is situated in the middle between the operating system below and the application layer above. In distributed systems, middleware usually supports developers by managing and handling the communication between interacting application components that reside on different computers within a network. Therefore, actual middleware implementations offer more convenient communication abstractions, functions, and services that go far beyond those provided by the network layer as part of the operating system. Moreover, by hiding cumbersome implementation details, the middleware relieves developers from tedious and time-consuming tasks helping them to better focus on the application's logic and purpose. With increasing complexity, middleware, thus, becomes an enabling factor to successfully engineer software systems [95]. This is especially true for the development of distributed event-driven applications for which the employed middleware usually plays a central role in managing and mastering the asynchronous and versatile interactions between their components.

In this chapter, we lay the foundation of this thesis and present its background. First, we define essential terms and introduce basic concepts on which event-based systems are built. Primarily, these are closely related to the publish/subscribe communication pattern and the content-based routing of event notifications in order to successfully decouple application components. Afterwards, we give an overview about existing middleware standards and implementations facilitating the development of event-driven applications. We identify the publish/subscribe concepts applied and analyze the strengths and weaknesses of their actual implementations. Based on the analysis, we finally conclude the chapter by discussing open challenges that developers still face when engineering publish/subscribe systems and programming event-driven applications.

2.2 Publish/Subscribe

Publish/subscribe is an appealingly simple, yet powerful group communication paradigm facilitating event-driven applications and architectures [58]. Components communicate by publishing notifications about occurred events and subscribing to those event notifications in which they are interested. This makes the communication asynchronous, indirect, and thus effectively reduces the coupling between participating components. In fact, notification publishers usually do not know their subscribers, while the receivers of a notification do not necessarily need to know which component published it. Because of this loose coupling, publish/subscribe is ideally suited for dynamic environments where components may spontaneously join or leave the network. It allows them to simply communicate any relevant change in state or environmental conditions as an event on which other interested components can react without obligation. This way, distributed and cooperative services can efficiently be realized.

2.2.1 Events and Notifications

In event-based systems, components cooperate and interact by exchanging notifications about occurred events [30]. An *event* is any happening that is of interest to a particular component [82]. It may vary in size and complexity and its cause may lie inside or outside the component. For example, a simple hardware interrupt, a new reading of an external sensor, or the completion of a sophisticated service request are events that may be of interest to a system component. Although these events are of different granularity, they all lead to a change in state. In particular, it is the state change by which we technically define an event in the following.

Definition 1 (event). *An* event *e is an observable change in state caused by an internal or external happening of interest.*

To inform about the occurrence of an event, the component creates a *notification* to describe the event in a way it can be automatically processed by other components. Therefore, the notification contains the data to precisely characterize the event itself and, where appropriate, the context and circumstances of its occurrence (e.g., time or location). Data models that are commonly used for notifications range from simple name/value pairs [31, 146] over semi-structured data based on XML [2, 35] to objects and classes [62, 61].

Definition 2 (notification). *A* notification *n_e reifies the event e in a way that it can be automatically processed. For this purpose, the notification n_e contains data describing the event e and, if appropriate, the context and circumstances of its occurrence.*

Often, the notification has to be sent through a communication network to reach and inform remote components about the event. In this case, the notification is serialized and transmitted in form of a message. Here, the *message* represents the data container for the serialized notification while it is in transit.

To clarify the meaning and notion of the terms event, notification, and message, Figure 2.1 visualizes their differences. The event e occurs at component C_1 which creates the notification n_e to describe the event itself as well as the context of its occurrence. In order to send the notification to the remote component C_4, n_e is serialized and wrapped into the message m_e, which is subsequently transmitted over the network. On the receiving side, the notification n_e is deserialized from the message's content and, finally, delivered to component C_4 for processing. The notification's serialization and deserialization as well as its transmission and delivery is usually the responsibility of a notification service, which is described in the next section.

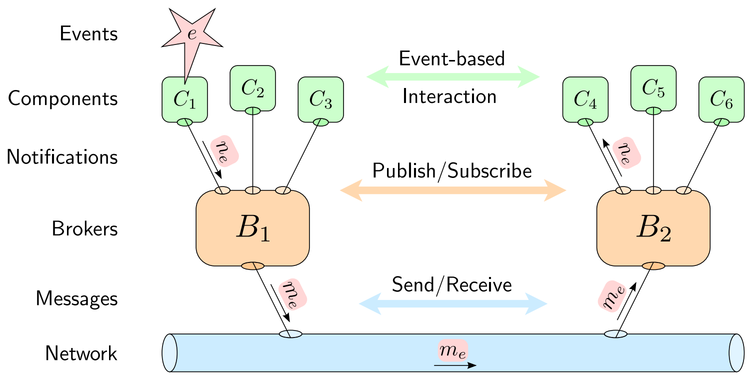


Figure 2.1: Distributed notification service.

2.2.2 Notification Service

Usually, a mediator is used to better decouple interacting, event-driven components. For this purpose, a *notification service* is interposed between them [193]. The notification service is responsible to take and deliver a notification to all receivers. Hence, it forwards the notification from its producer to all interested consumers. From the perspective of the notification service, the producers of notifications act as *publishers* and the consumers as corresponding *subscribers*. Altogether, they, thus, form a publish/subscribe system. Please note, that the role as publisher or subscriber is not static and immutable for a component's lifetime. In contrast, components can change them dynamically or even play both. For instance, the processing of a consumed notification may trigger an internal event inside a subscribing component, which, in turn, describes and publishes its change in state as a new notification on which yet other components can react.

There are different ways to implement a notification service. First approaches consisted of a central mediating component. Although easy to implement, these approaches are hardly scalable as a single instance is responsible for all event-driven components and has to match and deliver all published notifications to their subscribers. Today, the notification service is usually implemented in a distributed fashion by a set of interconnected brokers from which each broker is responsible for a number of clients [31]. This way, the load is efficiently shared among them. On the downside, the brokers have to exchange published notifications as well as information about the interests of their clients which may cause considerable overhead compared to the central solution in certain situations. Section 2.2.5 introduces several forwarding and routing strategies for reducing this overhead. Nevertheless, to deal with large-scale setups, a publish/subscribe system based on a distributed notification service is required.

Definition 3 (distributed notification service). *A distributed notification service consists of a set of interconnected brokers and has the following properties:*

(i) *Each broker services a set of exclusive clients and provides an interface to publish and to subscribe to notifications about occurred events.*

(ii) *Each broker allows a connected client to be a publisher producing notifications, a subscriber consuming published notifications, or even both at the same time.*

(iii) *The brokers cooperate and exchange notifications as well as information about client interests in order to ensure that a published notification is delivered to all interested subscribers independent to which broker a client is connected.*

Figure 2.1 visualizes a publish/subscribe system based on a distributed notification service [147]. The figure shows two of the interconnected brokers B_1 and B_2 each hosting three client components. Component C_1 detects event e and publishes the corresponding notification n_e at broker B_1. The brokers cooperatively implementing the notification service are now responsible to ensure that notification n_e reaches all subscribed components as, for example, C_4. Since components C_1 and C_4 are connected to different brokers, the published notification n_e needs to be forwarded from B_1 to B_2. Thus, n_e is serialized, transmitted over the network, and deserialized again. Thereafter, broker B_2 can finally notify component C_2 about the occurred event e by delivering its notification n_e.

2.2.3 Subscriptions and Advertisements

In event-based systems, components cooperate by publishing notifications and by subscribing to those notifications in which they are interested. Thereby, the latter consequently leads to the question how a component can specify its interest in certain notifications. Technically, this is done using a filter function. A *filter* simply determines whether the subscribing component is interested in a particular notification. By evaluating these filters on behalf of its clients, a broker can thus filter out and separate relevant notifications from those in which the clients are not interested at all.

Definition 4 (filter). *A* filter *F is a Boolean function applicable to a notification n to determine its relevance for a particular client. Only if $F(n) = true$, the notification is relevant for the client and needs to be delivered, otherwise the client is not interested in it.*

To inform a broker about its notification interests, a client creates a *subscription* out of a filter. The filter specifies which notifications are relevant for the client and is an inherent part of the subscription. Besides the notification filter, however, a subscription may also contain additional metadata. The metadata may include, for example, visibility constrains to restrict a subscription's validity range [74], time information to query for notifications about specific events in the past or the future [41], security credentials to prove the authorization for receiving certain information [15], or maximum delivery rates in order to avoid congestion and overload [149].

Definition 5 (subscription). *A* subscription *is a standing request to receive matching event notifications. It contains a filter specifying which notifications to receive as well as metadata to provide additional information or constraints, if appropriate or necessary.*

To inform a broker about its intention to publish notifications, a client creates an *advertisement*. Similar to a subscription, an advertisement contains a filter that specifies the kind of notifications the client is going to produce. Likewise, it can also contain metadata to provide additional hints or constraints. The brokers can use advertisements to their advantage, for example, to optimize the exchange of subscriptions within the broker network [146].

Definition 6 (advertisement). *An* advertisement *is an announcement to potentially publish notifications in future. It contains a filter specifying the kind of notifications to be produced as well as metadata to provide additional information or constraints, if appropriate or necessary.*

Please note that advertisements are sometimes considered as an optional feature. If advertisements are not supported by the notification service, it is simply assumed that every client may produce arbitrary notifications.

2.2.4 Notification Selection

Primarily, filters are used in subscriptions and advertisements to specify in which notifications a client is interested or what kind of notifications a client may publish, respectively. On the one side, it is in the interest of a client to specify these notifications as precisely as possible. Therefore, a comprehensive filter model is needed that provides the necessary expressiveness. On the other side, it is in the interest of a broker to determine as fast as possible whether a notification matches a given filter. Therefore, simple filters that are easy to evaluate are preferred in order to reduce a notification's latency and increase the broker's throughput. In fact, there is a trade-off between a filter's expressiveness and its matching complexity that needs to be considered in publish/subscribe systems [27]. In [234], Zeidler distinguishes five filter models according to their complexity. In the following, they are listed and discussed with increasing expressiveness.

Channel-based selection. When publishing a notification, it is sent into a *named channel*. Components have the possibility to subscribe to one or more channels and, thus, receive all notifications published therein. There is no further filtering of event notifications since individual interests of clients are not considered. This makes channel-based subscriptions to a very simple, yet efficient selection model that is easy to implement. The CORBA event service [157] is an example of this approach.

Topic-based selection. Each notification has an *associated topic* under which it is published. Topics are usually split into subtopics which can also be repeatedly subdivided into more specific categories. Thus, they form a hierarchy of topics into which each published notification is sorted. Considering a weather report, for example, the current temperatures in Berlin might be published under the topic weather.temperature.berlin. Subscriptions always refer to a particular topic and may also include further subtopics. However, topic hierarchies may be ambiguous. For example, another correct view of the weather report might be alternatively published under the topic berlin.weather.temperature. Please note that topic-based selection is also known as subject-based filtering [58].

Type-based selection. Notifications are represented as objects and filtered based on their *type* according to the corresponding class hierarchy [62]. Thus, subscriptions also include notifications of subtypes by default. So far, type-based filtering is quite similar to the topic-based approach, but allows for a closer and type-safe integration into the programming language and the middleware [57]. With multiple inheritance and comparable mechanisms, the selection of event notifications becomes even more flexible. To further increase its expressiveness, type-based selection is often combined with content-based filtering by specifying and evaluating additional constraints over object fields and attributes [179].

Content-based selection. Instead of filtering notifications based on meta-data such as topics or type information only, *content-based subscriptions* allow the selection of notifications based on arbitrary aspects of a notification's content. In fact, the whole notification becomes subject to the filter functions contained in active subscriptions making content-based selection the most general and flexible filter model [143]. This way, subscribers are completely independent of any topic assigned or classification made by the notification's publisher [60]. The expressiveness of this approach is only limited by the data model used to describe the notification's content and the predicates available to specify appropriate subscription filters [27]. Several different content-based data and filter models are proposed in literature, for example, templates and regular expressions [46], XPath filters for XML-based notifications [2, 35], generic predicates on name/-value pairs [143], constraints on object attributes and properties [60], or filter functions contained in (mobile) program code [62] that may even be generated and compiled dynamically [55].

Concept-based selection. Although referring to the same event, notifications may vary to a great extent. In particular, in heterogeneous environments, events are often described differently. For example, synonyms may be used to refer to event attributes, measures may be given in different units, or locations and time data may be encoded as absolute or relative information. Because of these differences and inconsistencies, a filter may not match a particular notification although the subscriber is actually interested in the event the notification communicates. In this case, semantic translations on the notification's content must be performed first, in order to successfully match it afterwards. *Concept-based filtering* [42], therefore, includes semantic translations based on well-defined ontologies to deal with these kind of heterogeneity [40]. The gained flexibility, however, is traded off with a complex and costly processing of notifications often limiting the system's performance.

2.2.5 Routing Algorithms

In a distributed publish/subscribe system, brokers exchange active subscriptions and published notifications in order to ensure that every client receives the notifications in which it is interested regardless of the broker to which it is connected. The way how subscriptions and notifications are exchanged is specified by the *routing algorithms* that are applied in the broker network. In particular, advanced routing algorithms are able to exploit similarities between subscription filters in order to reduce the number of subscription messages that are needed to be exchanged. The exchanged subscriptions subsequently determine to which clients and brokers which notifications have to be forwarded. Besides the trivial flooding of notifications, Mühl et al. distinguish in [147] four more advanced publish/subscribe routing algorithms that may additionally be combined with event advertisements. In the following, we discuss these routing algorithms and variants in the order of increasing complexity.

Flooding. *Flooding* is the simplest routing algorithm available. Each notification is generally disseminated to all brokers within the complete network. Hence, the network is flooded with notifications. On the one side, as brokers receive every published notifications by default, it is not necessary to exchange subscriptions at all. A broker just needs to filter those notifications out of the message stream in which its clients are interested. On the other side, a broker also has to process those notifications in which none of its clients is interested. Even worse, as these are flooded into the network they are unnecessarily forwarded to other brokers and, thus, waste valuable resources by causing processing costs and clogging links. Hence, flooding is, in general, not scalable. However, it may be a good choice if client interests are homogeneous and similar subscriptions are uniformly distributed in the network [162]. Additionally, flooding can be combined with other routing algorithms to form *hybrid network configurations* [201]. Such hybrid strategies use flooding only in those parts of the network, where its simplicity is profitable and adaptively switch to more advanced routing algorithms as soon as flooding becomes inefficient [199], for instance, if subscriptions are increasingly selective or substantially differ in distinct parts of the network. Please note that flooding can be combined with all major algorithms listed below.

Simple routing. *Simple routing* [146, 142] floods subscriptions within the broker network. This way, every broker gains global knowledge about client interests. Thus, a broker needs to forward a notification only towards the directions where an interested subscriber is located. For this purpose, brokers maintain a routing table in which they store each subscription together with the neighboring broker or local client from which the subscription was received. Subsequently, notifications are matched against this routing table and are only forwarded to those neighboring brokers and local clients that have a matching entry. Hence, no notification is unnecessarily forwarded anymore. However, as each broker stores a copy of every subscription that is currently active in the network, the routing tables grow proportionally to the number of subscriptions in the system. The consequences are twofold. First, large routing tables induce high processing costs and make a broker's matching and forwarding decisions more complex. Second, when client interests change frequently, a significant fraction of the network traffic is required to propagate new and revoked subscriptions to all brokers in order to keep their routing tables up to date. Taken together, both reasons considerably limit the system's scalability.

Identity-based routing. *Identity-based routing* [146, 142] reduces the routing table sizes by exploiting filter similarities between active subscriptions. In fact, it aims at eliminating equivalent and duplicate routing entries. Therefore, the forwarding of a subscription towards a neighbor is suppressed if an identical subscription has already been sent to this broker. In this context, two subscriptions are said to be identical if their filters match the same set of notifications. Hence, forwarding just one of them is already sufficient. Please note that this does not only save a single subscription message for and a single routing entry at the neighbor broker, but also a message and an entry for each subsequent broker behind this neighbor.

Covering-based routing. Compared to identity-based routing, *covering-based routing* [29, 31] further reduces the routing table sizes. Here, the forwarding of a subscription towards a neighboring broker is suppressed if a more general subscription has already been transmitted. The general subscription is said to cover the more specific one if the set of matched notifications of the former is a superset of the notifications that are specified by the latter. In this case, it is sufficient to forward the covering one. Only if the covering subscription is revoked, the once covered filters and subscriptions have to be transmitted. There are imperfect variants of covering-based routing, too. *Subscription pruning* [19, 18], for example, simplifies filter expressions by omitting those filter constraints that are costly to evaluate but not very selective. In this context, the selectivity of a constraint determines how well it is suited to distinguish between subscribed notifications and those notifications in which no client is interested. Brokers, thus, forward a simplified and more general version of a received subscription whose filter is easier and faster to evaluate. However, the gained speed up of the matching process is traded off with a small number of notifications that are forwarded unnecessarily and have to be subsequently filtered out, at last, before notifications are delivered to a local client.

Merging-based routing. *Merging-based routing* [146, 142] enables brokers to create new subscriptions by combining active subscription filters to new filters that cover the original ones. Hence, filters are merged to new covering subscriptions that are forwarded instead. This way, routing table sizes are further decreased. Similar to covering-based routing, there are also imperfect variants of merging-based routing [143]. Likewise, they trade off a simplified matching process with false positives that must be filtered out before notifications are finally delivered to local clients.

Advertisements. *Advertisements* [29] are used to limit the forwarding of subscriptions. If publishers announce what types of notifications they are going to produce, a subscription just needs to be forwarded towards those publishers who have an overlapping advertisement, i.e., they are potentially able to publish a matching notification. On the one side, advertisements effectively limit the regions to which a subscription has to be propagated and, thus, reduce subscription table sizes. On the other side, brokers now have to forward, store, and maintain advertisements for which additional processing and new advertisement tables are required. Thus, the usage of advertisements is especially advantageous in systems, where the number of notification producers advertising their publications is significantly lower than the number of notification consumers issuing a corresponding subscription [142]. Advertisements can be combined with all routing algorithms above except flooding. Moreover, the very same routing algorithms can be applied to also forward advertisements while the choice which algorithm is used for routing which type of message is independent of each other. Hence, it is possible to use the same or two different routing algorithms to forward advertisements and subscriptions, respectively.

2.2.6 Broker Topologies

To make publish/subscribe systems scalable, they are implemented in a distributed fashion. Besides filter model and applied content-based routing algorithm, the network topology plays an important role as it determines how the brokers are connected with each other. For distributed publish/subscribe systems, tree-like topologies are predominant as they are very efficient for data dissemination. If network paths to different destinations share common links, it is, thus, sufficient to just send a single message over the shared segments and copy it only at those nodes where the paths split. This way, it is ensured that over each link just a single copy of the message is sent. Publish/subscribe systems apply different protocols and techniques to construct such *data distribution trees*. In particular, multicast transmissions, generic overlay networks as well as peer-to-peer routing substrates are often used for this purpose.

Multicast. *Multicast* is a group communication paradigm allowing a sender to reach several destinations by transmitting a message only once. It is the network ensuring that the message is automatically replicated where necessary to get delivered to each receiver. There is a multicast extension to the Internet Protocol (IP). IP multicast [212] uses specific multicast address blocks to indicate that a message is intended for multiple receivers. Receivers are required to join a multicast group associated with such an address. Once the receivers are joined, a multicast distribution tree is constructed that is subsequently used to disseminate messages that are sent to the group's address.

Implementing a publish/subscribe system on top of a multicast-capable network is not expensive. In particular, multicast networks are well suited for topic-based publish/subscribe variants. For each topic, it is sufficient to create an own multicast group which brokers can join if their clients are interested in. However, if the number of topics is large compared to a limited number of available multicast groups, this straight forward approach leads to a *channelization problem* [1]. Individual multicast groups become responsible for multiple topics. Hence, notifications are also sent to brokers that are not interested in them. It has been proven that it is NP-hard to find an optimal assignment of topics to groups that minimizes the number of notifications that are unnecessarily disseminated [1].

Regarding content-based publish/subscribe, it is the notification's content that determines to which brokers it is delivered. Thus, the number of receivers may vary for each notification published. In fact, for each notification, there are 2^n possible sets of potential receivers assuming that the publish/subscribe system consists of n brokers. Assigning multicast groups to all of them is not possible and inevitably leads to the channelization problem. Besides the number of required multicast groups, it is primarily the overhead to manage them that limits the applicability. Brokers may dynamically join and leave groups requiring multicast distribution trees to be updated and reconstructed. In order to decrease the management overhead, Opyrchal et al. [162] propose heuristics to reduce the number of necessary multicast groups. In particular, they suggest overbroad multicast groups that introduce a tolerable degree of imprecision, multiple transmissions of the same notification to different multicast groups as well as sending a notification over multiple hops with each hop using a multicast to forward the notification to a set of neighbors. Please note that forwarding a notification over multiple hops is usually applied in generic overlay networks, too. In this case, however, forwarding is often based on unicast communication only.

Overlay networks. Since IP multicast is a network feature, it needs to be implemented and supported by involved network elements such as routers and

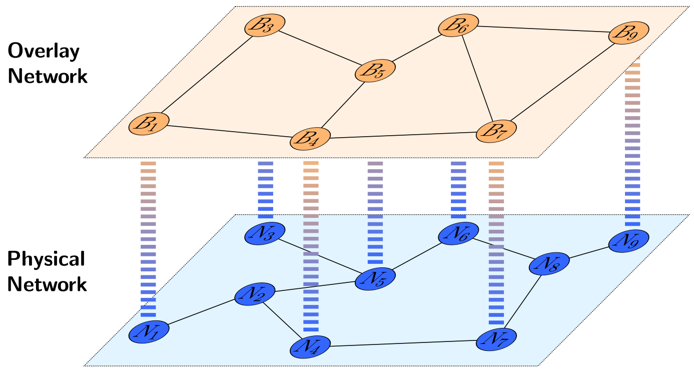


Figure 2.2: Overlay network on top of a physical communication network.

switches. In heterogeneous, interconnected networks, it thus has to be supported by all participating network providers. Alternatively, it is also possible to implement multicast features at application level solely using common unicast connections between end systems [39]. Nodes that are logically connected in this way are said to form an *overlay network*.

Definition 7 (overlay network). *An* overlay network *is a virtual network consisting of nodes and logical links that is built on top of an already existing network. Thereby, a logical overlay link is mapped to a path of nodes and physical links in the underlay network.*

Figure 2.2 illustrates the definition. It shows a logical publish/subscribe overlay network based on a physical communication network. All nodes of the physical network, except nodes N_2 and N_8, host a publish/subscribe broker and, thus, are also present in the overlay network. The brokers are connected with each other by logical overlay links. Although brokers may be directly connected in the overlay network, this does not imply that there is physical link between them in the underlay, too. Instead, notifications sent from one broker to a neighboring broker in the overlay may be required to get forwarded along a path of underlay nodes and physical links in order to reach their destination. For example, notifications from broker B_1 to broker B_3 have to follow the underlay path from node N_1 over nodes N_2 and N_5 to node N_3. As a consequence, several overlay links may share common underlay paths or segments thereof. Hence, when disseminating a message in the overlay network, it may happen that the message is sent over the same physical link multiple times even in opposite directions, for instance, when broker B_3 forwards notifications received from broker B_1 to broker B_5.

Because of message duplicates on physical links, an application layer multicast is not as efficient as a multicast strategy implemented in the network layer. Even worse, if the overlay's network topology is designed inappropriately, the induced overhead in the underlay becomes significantly large and may exceed the actually necessary dissemination costs multiple times. However, the overlay provides flexibility and freedom to implement customized multicast variants that are fine-tuned to the application's purpose. Publish/subscribe systems, for example, can directly integrate a content-based routing algorithm into their message dissemination strategy. Furthermore, arbitrary broker topologies can be created and virtually connected independent of the underlying physical network in a way that further supports and eases the system's functions as well as its maintenance. For publish/subscribe systems, hierarchical, acyclic, or generic overlay topologies may be used as well as any combination of the three [29, 31]. Please note that if not stated otherwise we assume an *acyclic broker overlay network* to form the basis for the publish/subscribe extensions developed and discussed in this thesis. In fact, this is no severe limitation as publish/subscribe systems based on more generic topologies usually construct tree-like structures to efficiently disseminate notifications, too [28, 32].

Peer-to-peer. If not operating in a hierarchical setup, publish/subscribe brokers exchange subscriptions and notifications as equal peers on behalf of their clients. However, a number of publish/subscribe systems additionally use a peer-to-peer routing substrate to organize their broker topology. In fact, these systems realize a notification service on top of a peer-to-peer network which itself is usually implemented as an overlay based on an existing network infrastructure such as the Internet. This way, even another layer is added to the implementation.

Basically, those peer-to-peer networks are organized as a large distributed hash table. A *distributed hash table* (DHT) [52] is a decentralized data structure to store and lookup key/value pairs. The hash function maps the key to a node which is responsible to store the associated value. In this context, a good hash function ensures that the load is uniformly distributed among the participating nodes. Moreover, the overlay topology formed by the nodes is constructed in such a way that a query for a particular key can be efficiently routed to the node responsible for its value. Assuming that the overlay network consists of N nodes, then several peer-to-peer systems (e.g., Chord [213], Pastry [194], and Tapestry [236]) are able to resolve the responsible node in at most $O(\log N)$ steps while each node has to maintain only $O(\log N)$ overlay links. In particular, these properties make this type of peer-to-peer systems exceedingly scalable. Furthermore, their overlays are also well suited to realize an application layer multicast.

In fact, peer-to-peer routing substrates can conveniently be leveraged to implement a topic- or type-based publish/subscribe system. For this purpose, the topic or type information is used as the key to forward notifications and subscriptions in the peer-to-peer overlay network. Eventually, corresponding notifications and subscriptions meet each other at the *rendezvous node* [34] to which their topic or type is mapped by the hash function. While being forwarded, subscriptions establish routing entries at each node they pass. Each routing entry points to the previous node from which the subscription was received. Together, all routing entries for a particular topic or type form a *core-based tree* (CBT) [13] where the rendezvous node is the root and the subscribers are the leaves. After arriving at the rendezvous node, this distribution tree is used to finally disseminate the notification to all interested subscribers. With advertisements, similar network paths can be set up that lead from the rendezvous node to all potential publishers. Distributing and installing additional content-based filters along these paths then allows to proactively reduce the number of notifications the rendezvous node has to process [179, 178].

2.2.7 Loose Coupling

One of the major strengths of publish/subscribe infrastructures and event-driven applications is the loose coupling between communication components. Reducing component dependencies, in general, allows systems to grow in size as well as to become more dynamic. In fact, publish/subscribe has been successfully proven to be well suited for large scale setups while supporting versatile interaction and cooperation schemes [144]. Furthermore, decoupling components results in increased flexibility. In particular, it allows applications, the infrastructure, and parts thereof to evolve freely, i.e., to get gradually adapted and extended whenever new requirements occur. Publish/subscribe is, thus, one of the key patterns for enterprise integration [93]. The integration of new components into existing infrastructures is facilitated as publish/subscribe systems fully decouple them in three dimensions: *space*, *time*, and *synchronization* [58].

Space decoupling. Publish/subscribe is an *indirect* and *anonymous* form of communication. Publishers as well as subscribers do not necessarily need to know each other, i.e., they are not required to address each other in order to exchange notifications. Instead, it is the notification service ensuring that a published notification is delivered to all interested subscribers. Hence, the publishing component neither knows whether the notification is received by none, one, or many receivers nor does it make any difference in sending. Likewise, a subscriber may also receive notifications from one or from multiple sources without any additional effort.

Time decoupling. Brokers are able to temporarily cache and store notifications [41]. This way, a notification is not lost if a subscriber is currently disconnected to the broker network at the time the notification is published. Instead, the notification is delivered at a later date when the subscriber eventually reconnects. At that time, conversely, the publisher may not be connected anymore. Hence, publish/subscribe does not necessarily require components to be present and active at the same time in order to interact with each other.

Synchronization decoupling. Publish/subscribe provides an inherently *asynchronous* communication decoupling the control flow of publishers and subscribers. In particular, a publisher is not blocked when sending a new even notification. The notification is simply passed to the local broker handling its forwarding and dissemination. Meanwhile, the publisher can continue its normal operation. On the other side, subscribers are usually notified about the event via a callback function that is started in an own runtime thread and takes the notification as parameter. This way, the received notification may be processed concurrently or queued to be handled by the component's main thread at an appropriate time. However, it is never required that the control flow of publishers and subscribers is synchronized in any form in order to exchange notifications.

2.3 Existing Notification Services

The need for adequate frameworks, tools, and software products that ease the development of distributed, event-driven applications has been addressed by academia as well as industry. As result, several middleware implementations and industry standards related to publish/subscribe communication and event-driven applications emerged that differ in requirements, follow distinct approaches, emphasize on various technical aspects, and pursue divergent goals. In this section, we give an overview about prominent representatives of each class. In particular, we first discuss industry standards for notification services analyzing their goals as well as their range of functions. Thereafter, we examine research prototypes of publish/subscribe systems and highlight the ideas and implementation approaches taken to realize distinct aspects and features. For both middleware standards and research prototypes we are especially interested in the question which of the publish/subscribe concepts presented in the previous section have been successfully applied in which way and to what extent.

2.3.1 Corba Event and Notification Service

The *Common Object Request Broker Architecture* (Corba) [160] provides an architectural framework and a middleware specification that enables distributed object implementations to transparently request and receive services from each other. Corba is developed and standardized by the Object Management Group (OMG) in an abstract platform- and language-independent fashion. Vendors are encouraged to implement the specification for concrete platforms and programming languages, while standardized concepts, mappings, and protocols ensure the interoperability of different products. This way, Corba significantly eases the development of distributed applications in heterogeneous computing environments. The specification's core deals with the interface, the functionality, and the structure of the *object request broker* (ORB) that is responsible to execute and mediate invocations of object methods that form the service requests.

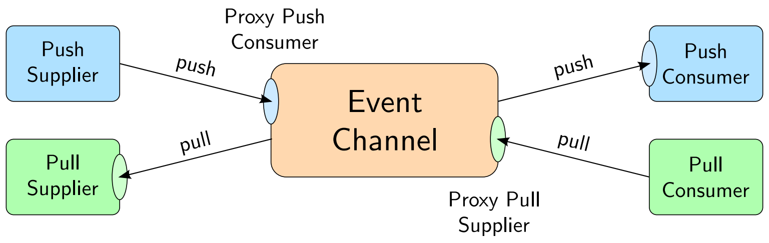


Figure 2.3: Combining push-based and pull-based mode of operation.

For this purpose, the ORB has to resolve object references to local or remote locations, to marshal and unmarshal method parameters as well as return values, and to send these over the network when invoking a remote method.

Corba is a mature middleware technology that probably has already passed its zenith [89]. Nevertheless, it conquered several domains such as financial and telecommunication services, where it is successfully deployed and still used today. Beyond that, Corba concepts influenced and inspired many younger middleware implementations that adopted, leveraged, and customized various of its ideas. In particular, one of Corba's strengths is its modularity and extensibility in terms of object services. These usually address advanced aspects and additional requirements that are common in distributed environments. Corba services, for example, range from transactional support [156] over persistent storage [154] to security functionality [155]. In the following, we discuss two of them, the Event Service and the Notification Service, that enable applications and components to interact with each other using publish/subscribe communication.

Corba Event Service. The Corba *Event Service* [157] implements a channel-based approach to decouple event suppliers and consumers. Acting as mediator, the *event channel* is placed between both parties. Hence, suppliers and consumers do not need to directly address each other in order to communicate. Instead, they only communicate with the event channel that receives events from suppliers and forwards them to connected consumers. To interact with the event channel, Corba's synchronous method invocations are used while the event channel itself supports different data models and communication modes.

Regarding the data model, the Event Service specification distinguishes between a *generic* and a *typed* event communication. In generic event communication, it is allowed to pass a single argument of an arbitrary type, while the typed model requires event suppliers and consumers to previously agree on a particular interface and its methods to exchange events. In the latter case, however, multiple arguments of the specified types may be exchanged with a single invocation.

Regarding available communication modes, the Event Service supports both a push-based and a pull-based mode of operation. In *push-based* mode, the event supplier notifies its consumer about the event and pushes the event data to them. In *pull-based* mode, however, it is the event consumer that requests the data from the supplier. The event channel interposed between suppliers and consumers behaves as proxy providing appropriate communication interfaces for each mode. This way, it is even possible to combine a push-based supplier and pull-based consumer or vice versa as shown in Fig. 2.3.

Corba Notification Service. The CORBA *Notification Service* [158] is seen as the successor of the Event Service as it addresses several shortcomings of the latter. Primarily, it extends ordinary event channels with comprehensive means for filtering notifications. For this purpose, *structured events* are introduced as a new lightweight form of typed events. These structured events consist of a header providing mandatory information about the event's domain, type, and name and of a body containing the event's opaque payload. Furthermore, both the header and the body can optionally be extended by an arbitrary number of filterable name/value pairs that carry additional meta-information, QoS hints, or data fields associated with the event.

Event consumers are then allowed to construct a filter expression based on these name/value pairs using the *Default Filter Constraint Language* in order to specify the events they are interested in. This way, it is possible to filter events by their content as well as by QoS constraints. Besides the Default Filter Constraint Language, which is required by the Notification Service specification, CORBA implementations may also support arbitrary additional and domain-specific constraint languages.

Furthermore, the Notification Service also defines an optional repository for the event types used in the system. If provided by the Notification Service implementation, it stores meta-information about each event type that is characterized by its name and domain as well as a set of properties which describe its name/ value pairs. This way, event consumers may use the repository to conveniently learn about available event types and the structure of their content in order to dynamically create appropriate and type-safe filter constraints.

2.3.2 Java Message Service

As part of the Java Enterprise Edition, the *Java Message Service* (JMS) [215] specifies how application components can asynchronously send, receive, and process messages. In particular, JMS defines the application programming interface, while vendors are encouraged to provide their own service implementations called JMS providers. In general, JMS supports two communication models: point-to-point and publish/subscribe communication. *Point-to-point* communication enables senders and receivers to interact using message queues that are administered and stored at the JMS server. To enforce the decoupling of clients, any form of direct communication between sender and receiver without an intermediary message queue is not supported. *Publish/subscribe* communication enables multiple senders and receivers to exchange messages based on *topics* that are managed on the server. Likewise, the JMS server decouples the message producers and consumers from each other.

JMS messages consist of a header and a body. Besides predefined header fields such as the message's destination, its unique identifier, its priority level, or its timestamp, the header can also contain any number of additional fields defined and supplied by the user. Regarding the body, several formats and data types are supported. Thus, the body may consist of a stream of bytes, characters, or Java primitive types as well as name/value pairs or serialized Java objects. However, this flexibility comes at a price. Filtering messages is only possible on header fields, the body cannot be evaluated. To restrict the number of messages delivered by the JMS server, consumers create a *message selector* that specifies the kind of messages they are interested in. The selector supports filter expressions that are based on a subset of the SQL92 [230] conditional expression syntax. In particular, these expressions may also include the evaluation of custom header fields defined and set by the user. This way, nonetheless, JMS offers a limited support for *content-based* filtering of messages besides selecting them by topic.

Furthermore, JMS provides several advanced features. In particular, persistent messages, durable subscriptions, and transactional sessions are supported. *Persistent messages* are logged to stable storage. This way, they are not lost even if the JMS provider fails and, hence, are guaranteed to be delivered exactly once. However, in comparison to regular messages that are volatile and might be lost in case of failures, persistent messages cause a considerable overhead that usually cannot be neglected. Similar to regular subscriptions, *durable subscriptions* allow consumers to specify the messages in which they are interested. Beyond that, they also remain active even if the subscriber disconnects from the server. Matching messages published in the meantime are kept on the server until the subscriber reconnects or they expire. With *transactional sessions*, a client can group the publication and consumption of a set of consecutive messages into a logical unit of work that is either carried out completely or not at all. When a transaction commits, all published messages are sent to the server, while consumed messages are acknowledged. When a transaction aborts and is rolled back, published messages are discarded while consumed ones are automatically recovered in order to get redelivered.

JMS has received wide industry support. Within companies, JMS is often used to integrate individual business components and services into reliable and event-driven infrastructures. In fact, by defining a common programming interface for messaging systems, JMS enabled Java applications to interact with a wide range of enterprise messaging products of numerous vendors. However, this comes at a price. In order to be easily and efficiently supported by various messaging products, the JMS specification largely dispenses with technical details and implementation aspects. Hence, these aspects are often implemented differently by individual vendors rendering their products incompatible to each other.

To facilitate the integration of enterprise applications, messaging interoperability is an essential precondition. The *Advanced Message Queueing Proto-*

col (AMQP) [164] is an alternative messaging specification that offers similar features and, moreover, describes the encoding, format, and semantics of the messages used to exchange information in form of bits and bytes across the network. This way, any two implementations that conform to the specified protocol are guaranteed to be interoperable independent of vendor and platform. In particular, applications as well as message brokers are not solely bound to the Java programming language anymore as it is the case with JMS.

2.3.3 Data Distribution Service

The *Data Distribution Service for Real-Time Systems* (DDS) [159] is an open middleware standard developed by the Object Management Group (OMG) focusing on publish/subscribe communication in real-time and embedded systems. It is designed to ensure the reliable dissemination of high volumes of data with minimal latency. To achieve this, DDS restrains from using separate brokers as intermediaries between publishers and subscribers that introduce additional processing delays and may even turn into bottlenecks or single points of failure. Instead, DDS applications entirely exchange their data in a peer-to-peer manner. Therefore, DDS introduces the notion of a distributed *global data space* that is shared between applications. To exchange information, application components can simply read and write data objects within the space, while DDS takes care to properly disseminate all data updates. When disseminating data, DDS pays much attention to Quality of Service (QoS) aspects in order to ensure real-time behavior. In particular, DDS allows an extensive control of QoS parameters to give reliability and ordering guarantees, adjust deadlines and priorities, and fine-tune bandwidth and other resource limits.

For applications, DDS offers two different layers of abstraction that each provide its own set of interfaces. The *data-centric publish/subscribe* (DCPS) layer describes the fundamental concepts and ideas used to efficiently disseminate the data, while the *data local reconstruction layer* (DLRL) aims to conveniently integrate the service into the application. In the following, both layers are described in more detail.

Data-Centric Publish/Subscribe (DCPS). The DCPS layer is responsible for data distribution. Data is published and subscribed in form of *topics* which are a triple made up of a unique name, an application-specific data type, and a set of assigned QoS policies. Data types are specified in a platform independent way using a subset of OMG's interface definition language (IDL) [160]. A custom data type may consist of an arbitrary number of fields that either are primitive types, template types, or constructed types created by nesting type definitions. One field or a combination of several fields is chosen as *key* whose value unambiguously identifies a topic instance. This way, multiple instances of the same data type can coexist in the global data space under the same topic at the same time. The key determines to which instance data updates belong, i.e., whether they are interpreted as successive values of the same instance or handled as data values of different instances. If no key is defined, then there is just one global topic instance comparable to the singleton pattern [81] in object-oriented programming languages. To better structure the global data space, topics are grouped into domains which can be subdivided into partitions. Applications have to explicitly join them first in order to produce or consume any data.

In DDS, a publisher is a middleware entity that is responsible to distribute produced data, while a subscriber is a middleware object for receiving the data updates. In order to ensure type safety, applications have to use a typed *data writer* and *data reader* generated from the topic's type definition in order to interact with a publisher or subscriber, respectively. Besides topics, publishers and subscribers as well as data writers and readers may have own QoS policies assigned. The QoS for disseminating produced data updates eventually results from a combination of the QoS policies of the entities and participants involved.

Associating a reader with a subscriber is interpreted as a subscription for the reader's topic, i.e., it signals the interest to receive corresponding data. To restrict the number of data updates delivered, topics can be filtered based on their content. Therefore, DDS provides filter expressions that are comparable to the WHERE clause of an SQL query and may include arbitrary data fields of the topic. For processing the data, DDS offers a notification-based and a wait-based style of interaction. The *notification-based* interaction mode uses listeners for registering callback functions. These callbacks are invoked by the middleware to asynchronously inform the application when new data is available, the communication status changes, or a QoS violation occurs. In the *wait-based* interaction mode, the application is blocked at a wait-set until an attached condition object is triggered, i.e., its condition evaluates to true, or else a timeout expires. This way, new data gets synchronously accessed and processed. Similar to the functionality offered by the asynchronous notification-based mode, condition objects can be created for status changes and QoS violations, too.

Data Local Reconstruction Layer (DLRL). The DLRL is an optional layer on top of the DCPS layer. It aims at seamlessly integrating the publish/subscribe service into the native language constructs of the application, i.e., to enable application developers to conveniently access the exchanged data in an object-oriented fashion. For this purpose, the DLRL allows developers to first define classes of shared DLRL objects including methods, data fields, and relations and to subsequently bind them to DCPS entities. Thus, when manipulating such a bound object, for example, by updating and changing several field values, any object modifications are automatically propagated via publish/subscribe to all parties possessing a local copy of the shared object. To ensure consistency, the DLRL introduces an *object cache* that is used to reconstruct the object's state from received updates.

In order to bind DLRL objects to the DCPS layer, DDS defines a structural mapping, an operational mapping, and a functional mapping. In particular, the *structural mapping* determines the relation between DLRL objects and DCPS data. It leverages concepts and ideas known from object/ relational mappings that are used to bridge object-oriented programming to relational databases [107]. In fact, DCPS topics can be seen as database tables, topic fields correspond to table columns, and data samples of topic instances constitute the table rows. The structural mapping then specifies which information about DLRL objects have to be considered and how they are stored in these kind of tables so that the object's state and its relations can successfully be reconstructed. The *operational mapping* defines how DLRL objects are bound to DCPS entities such as data writers and readers as well as publishers and subscribers. This also includes a fine-grained management of QoS policies, especially, if different object fields have different QoS requirements. Finally, the *functional mapping* specifies the translation of DLRL functions to DCPS functions. In particular, this comprises the management of the object cache, the application of incoming data updates on cached objects as well as the propagation of object modifications done by the application. Moreover, it also describes how the DLRL can inform the application about occurred modifications and vice versa.

2.3.4 Siena

The *Scalable Internet Event Notification Architecture* (Siena) [29, 31] is one of the first prototype implementations of a distributed content-based publish/subscribe system. As its name suggests, the research focus lies on architectures for a distributed notification service targeting at an Internet-scale deployment. For this purpose, an overlay network consisting of Siena broker servers is formed that offers connected clients the possibility to publish own event notifications and to subscribe to notifications published by others. Event notifications, subscriptions, and advertisements are exchanged between brokers via overlay links in order to ensure that a published notification is delivered to all interested clients holding a matching subscription. Thereby, Siena allows subscriptions to evaluate the whole content of a notification. Furthermore, it uses a covering-based routing algorithm while supporting several different server topologies.

Siena notifications consist of an arbitrary number of typed name/value pairs describing the event attributes. Subscriptions and advertisements contain filter expressions formed by a conjunction of attribute filters. Each attribute filter usually comprises a simple predicate that constrains a particular name/value pair, for instance, by a numerical comparison or a prefix/suffix test for an integer or a string value, respectively. Composed filter expressions are allowed to contain multiple attribute filters, i.e., predicates, for the same attribute. In this case, however, matching slightly differs for subscriptions and advertisements. While a notification matches a subscription only if all attribute filters are satisfied, an advertisement is already matched if at least one attribute filter for each name/ value pair is satisfied. On the one side, this simplifies the description of event notifications a producer is going to publish. On the other side, the distinction between subscriptions and advertisements makes matching and covering tests significantly more complex.

Considering the forwarding of advertisements, subscriptions, and event notifications in the overlay network, SIENA uses a *covering-based* routing strategy. Therefore, SIENA brokers store subscriptions and advertisements in a filter lattice in order to keep track of the relations between them. The lattice comprises each filter together with pointers to those filters that it covers. This way, the lattice resembles the *partially ordered set* (poset) induced by the filters' covering relation. On the one hand, maintaining the poset is costly because it needs to be updated whenever a new advertisement or subscription is received and, likewise, if an existing one is cancelled. On the other hand, the poset structure can be leveraged to significantly speed up the matching of notifications. For this purpose, the poset is traversed in depth first order starting at the root filters that are not covered by any other filter. If a filter does not match the notification, then no covered child filter can match either. Hence, it is not necessary to traverse the poset any further to evaluate them. However, in [33], Carzaniga and Wolf present an alternative, more efficient matching algorithm for SIENA. It is a predicate counting algorithm [231] that aims at evaluating simple attribute filters only once. The algorithm, thus, saves significant processing costs if complex filter expressions or different filters pose similar predicates on the same attributes. Since covering filters often feature similarities, the predicate counting algorithm is considered to be more efficient in the majority of cases.

To facilitate Internet-scale deployments, SIENA supports a variety of server topologies to form its overlay network. In particular, hierarchical topologies, acyclic peer-to-peer topologies, generic peer-to-peer, and hybrid topologies are considered. Figure 2.4 shows an exemplary setup for each topology type highlighting the differences. In *hierarchical topologies* as shown in (a), publish/subscribe clients are connected to SIENA brokers which themselves are connected to master brokers forming a hierarchical tree with a dedicated root broker at the top. To exchange subscriptions and event notifications, a hierarchical version of covering-based routing is used. In its hierarchical variant, clients and brokers use the same basic protocol to interact with their hosting broker or master broker, respectively. On the one side, a broker, thus, does not need to distinguish whether it interacts with a client or another, subordinated broker simplifying the routing logic. On the other side, both brokers cannot benefit from advanced routing optimizations that are applicable otherwise. For example, as the basic protocol does not propagate subscriptions to clients, they are only forwarded upwards in the hierarchy. As consequence, only the root broker possesses the complete set of active subscriptions and, thus, has to match every published notification. Therefore, each notification is always forwarded to the root broker, even if there is no interested subscriber in the whole network.

In contrast to the hierarchical topology, the peer-to-peer topologies as visualized in Fig. 2.4 (b) and (c) are not affected from protocol limitations. SIENA brokers interact as equal peers that mutually exchange their subscriptions. Hence,

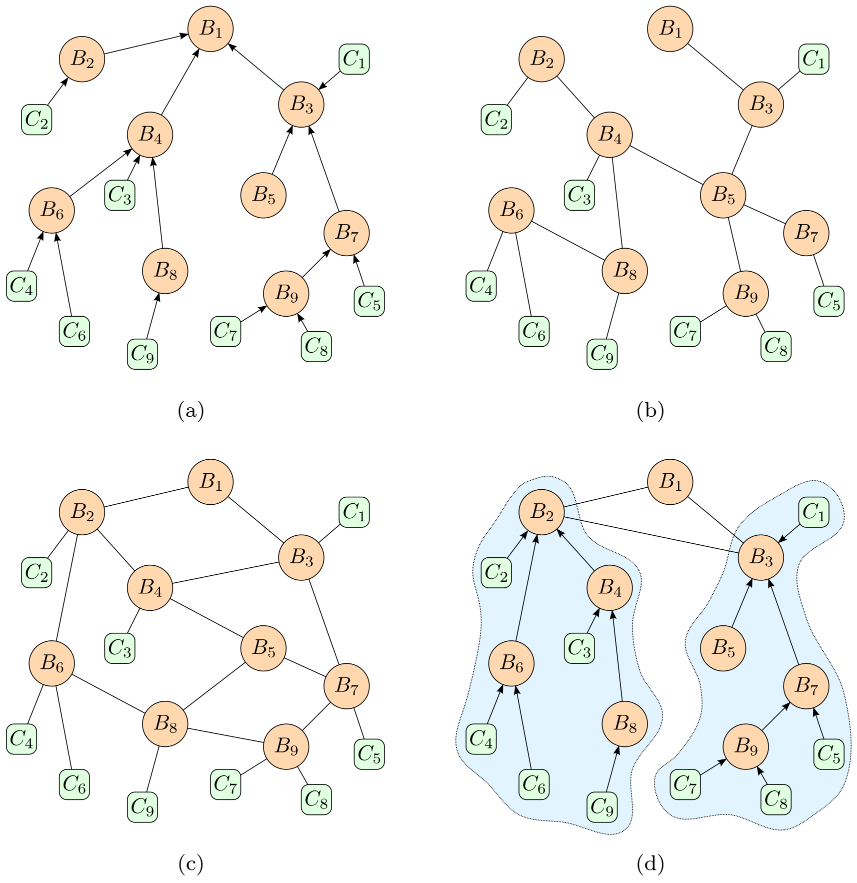


Figure 2.4: SIENA broker topologies: (a) hierarchical topology; (b) acyclic topology; (c) generic topology; (d) hybrid topology.

notifications without any subscriber are filtered out as early as possible. Additionally, advertisements can also be applied to further reduce the traffic. In the *acyclic peer-to-peer topology*, the network of SIENA brokers is still restricted to a tree structure, without a dedicated root though, while the *generic peer-to-peer topology* allows arbitrary connected overlay networks. However, in the latter case additional measures must be taken to avoid forwarding cycles on which notifications may loop indefinitely. Finally, *hybrid server topologies* are usually formed by merging other topologies to benefit from combined strengths and advantages. Figure 2.4 (d) shows two hierarchical clusters of SIENA brokers whose root brokers are members in a generic topology. This way, one can profit from the simplified broker management within each cluster and leverage the routing optimizations of the advanced protocols for inter-cluster communication.

2.3.5 Rebeca

The *Rebeca Event-Based Electronic Commerce Architecture* (REBECA) provides a publish/subscribe notification service originally designed for and targeting at distributed e-business applications such as, for instance, electronic markets and auctions or information dissemination and brokerage [22]. In order to support large-scale deployments, the notification service is implemented by a number of cooperating event brokers each hosting a set of clients, i.e., local publishers and subscribers. The brokers form an acyclic overlay network that is used to exchange published notifications and issued subscriptions. In general, the architecture is comparable to the SIENA publish/subscribe system as discussed in the previous section. However, there are important differences, too.

With ongoing development, the focus of the REBECA project shifted from event-based applications to middleware and integration aspects including the applied publish/subscribe communication protocols. In particular, content-based forwarding and routing strategies became a central subject of research [142]. As a result, REBECA features an extensible data and filter model as well as a flexible routing framework, which, in contrast to SIENA, are both based on formal specifications [144]. In fact, offering excellent customizability while ensuring a predictable system behavior makes REBECA a very appealing development and prototyping platform. Over the time, REBECA, thus, successfully served as publish/subscribe system to implement support for mobile clients and applications [235, 149] including data caches and histories [41], to study first visibility and structuring concepts [69], programming abstractions [218], and model-driven development approaches [167, 168], to evaluate reconfiguration mechanisms [169] and adaptive routing algorithms [201, 199] that enable self-organizing and self-stabilizing broker networks [103, 101], and to thoroughly validate the results of analytical and stochastic publish/subscribe models [148, 200].

REBECA's notifications consist of an arbitrary number of event attributes that hold the notification's data characterizing the event and the circumstances of its occurrence. Attributes are represented as name/value pairs with associated values being of simple or complex types. REBECA's subscriptions and the optional advertisements contain a filter expression made of any Boolean combination of predicates of which each predicate constrains a single event attribute. Both the data and the filter model are *customizable* in the sense that new data types as well as constraints and predicates can be added easily. It is the responsibility of the developer to ensure that new filter predicates are comparable to each other in order to leverage the optimizations of the more advanced routing algorithms that exploit filter similarities.

Besides trivially flooding all notifications into the broker network, REBECA also supports simple routing, identity-based routing, covering-based routing, and merging-based routing [145]. Moreover, the latter strategies can be applied independently from each other for forwarding subscriptions and advertisements, respectively. This way, REBECA allows various *combinations* of routing algorithms and configurations to be used within the same broker network. Sequential traces and linear temporal logic are used to specify the behavior of a correct publish/subscribe system. It has been shown that Rebeca's routing framework in general as well as the individual routing algorithms itself comply to the given specification [144, 147].

The idea of *scopes* [71, 74] has been introduced into Rebeca in order to structure publish/subscribe systems and organize their event-driven applications. Primarily, scopes allow developers to bundle related application components and to limit and control the visibility of their notifications. Several ways to implement scopes are presented in [68] that include, for example, collapsing scope graphs, scope addresses and dedicated scope brokers as well as scope overlays within the broker network. However, a reliable implementation of scopes allowing for a comprehensive evaluation of the concept is still missing.

Instead, *programming abstractions* have been derived to better integrate the publication, subscription, and processing of event notifications into object-oriented programming languages [218]. Therefore, existing language features such as overloaded operators and delegate functions are leveraged to enable developers to conveniently create type-safe filter expressions and to easily specify and pass event handlers, respectively. Moreover, it is also possible to pragmatically specify composite events and event patterns for correlation. Thereby, dynamic code generation is effectively used to fill in boilerplate code at runtime, for instance, to generate constructors and publication methods for event notifications based on plain interface definitions. Although presented concepts are not limited to any specific object-oriented programming language, their convenient usage strongly depends on built-in language features.

A revised scope concept and corresponding programming abstractions as well as their implementation and evaluation are major contributions of this thesis that exceed previous work. Please refer to Sect. 4.6 and Sect. 5.7 for a detailed discussion of improvements and differences.

Regarding mobility, Rebeca extensions provide support for *mobile clients* at the infrastructure as well as the application level [234]. Mobile clients are usually connected to the broker network via wireless links that often are fragile and may break down from time to time. Thus, brokers keep active subscriptions on behalf of disconnected clients and cache matching notifications until they can be delivered again. Moreover, roaming clients may reconnect to different brokers in the network. In this case, Rebeca transfers subscriptions as well as cached notifications to the new hosting broker without violating the ordering guarantees previously negotiated with the client [235]. Broker caches that temporarily store event notifications are, in general, comparable to Rebeca's *event histories* [41] that provide access to notifications that were published in the past. When receiving a new subscription, event histories republish stored notifications if they match the particular subscription as well as a mandatory replay specification restricting the time how long the notification's publication may lie in the past.

To support logical mobility, Rebeca offers *location-dependent filters* [70]. In particular, these allow clients to implement location-aware applications that subscribe to notifications of events happening in the vicinity of the client's current position. If the client moves, these subscriptions are updated or renewed accordingly. Movement graphs anticipating the client's direction can be exploited to optimize initial setup times, responsiveness, and notification delivery.

Recent work focuses on making Rebeca self-managing [100] which includes a self-organizing broker network, adaptive routing strategies, and a fault-tolerant implementation. For this purpose, reconfigurations of the broker network are a necessary precondition [169]. Please note that it is possible to split arbitrary complex network reconfigurations into a series of three basic operations that comprehend adding a new broker, removing an existing broker, and replacing an old overlay link with a new one. Algorithms are implemented that execute these operations without violating any ordering or service guarantees. Thus, they are well suited to provide the basis for a *self-organizing broker topology* that adapts itself to changing network conditions and event patterns [103]. In fact, Rebeca brokers aim at continually minimizing the lengths of the network paths between publishers and subscribers while trying to avoid instable and costly overlay links.

Regarding the network's routing configuration, *hybrid routing algorithms* [201, 199] have been introduced that allow to employ different forwarding strategies within different parts of the broker network. This way, brokers are able to dynamically change the routing configuration to the strategy best suited for handling the current traffic pattern in the network region considered. Adapting both the network topology and the routing strategy helps to significantly reduce the general forwarding costs.

To increase the network's fault-tolerance at the same time, two different approaches are taken. Rigorously applying the concept of *soft-state* requires clients and brokers to periodically renew all subscriptions and advertisements [101]. This way, it is guaranteed that corrupted routing entries caused by transient network faults are purged within a determined time and cannot last forever. To tolerate broker crashes without notification losses, broker caches and acknowledgements are used. Basically, each broker temporarily stores the notifications it forwards until these are acknowledged by its direct neighboring brokers and their neighbors. Hence, each notification is always cached by two different brokers at the same time which is why the crash of one of them can be masked.

2.3.6 Hermes

Hermes [179] is a research prototype implementation of an event-based middleware platform. It is built on top of a distributed publish/subscribe system which itself is based on a peer-to-peer routing substrate formed by Hermes brokers. It is, thus, well suited for dynamic, large-scale, and data-rich deployments. Furthermore, Hermes aims at a tight integration into programming languages and applications. Therefore, it supports typed event notifications and facilitates the inheritance of event types while automatically type-checking publications and subscriptions. Additionally, HERMES provides a number of services to help and support application developers [178]. In particular, HERMES features composite events and allows for event correlation while also addressing Quality of Service (QoS) and security aspects.

The event model of HERMES is comparable to common data models of object-oriented programming languages. In fact, its similarity is an intentional design decision in order to minimize the semantic gap between both. Likewise, HERMES requires its event notifications to be typed. Hence, each published notification is seen as an instance of an *event type* that is described by a type schema. In particular, the type schema defines the type name and a set of event attributes formed by typed name/value pairs that contain the data associated with an event instance. In order to customize events and create more specific ones, the *inheritance* of event types is introduced. It allows developers to conveniently extend an existing type schema by adding new event attributes, while those defined by the parent type and its ancestors are inherited. This way, developers are able to create fine-grained type hierarchies for notifications. Supporting these familiar object-oriented concepts in the middleware's event model significantly eases the mapping onto programming language constructs. In this context, event notifications and type schema obviously correspond to object instances and class definitions in object-oriented languages, respectively.

Beyond that, HERMES requires advertisements and subscriptions to be typed, too. In fact, the type specification is the primary constraint by which notifications are matched. Thereby, subtype relationships are appropriately considered. Additionally, subscriptions may also contain a content-based filtering expression consisting of an arbitrary number of predicates over the notification attributes of the particular type. In order to avoid runtime type errors, HERMES automatically checks whether these predicates comply with the attributes' type information given in the event type's schema.

For the dissemination of event notifications, HERMES leverages a *peer-to-peer* routing substrate called PAN [180] that provides an extended implementation of a distributed hashtable (DHT) [52] applying the PASTRY routing algorithm [194]. This way, HERMES benefits from PASTRY's high scalability, its adaptivity to dynamically changing network topologies, and its excellent fault-tolerance properties and robustness. Based on the peer-to-peer overlay network, publish/subscribe dissemination trees for event notifications are build up. Therefore, the name of the event type is used as unique key being hashed to determine the HERMES broker responsible for its management. At this broker, the event type's schema definition is stored. Furthermore, advertisements as well as subscriptions are routed within the peer-to-peer overlay towards this broker acting as rendezvous node where they eventually meet each other. Along their way, they establish the routing entries and network paths that are subsequently used for the dissemination of notifications of the particular event type. Together, the network paths form a core-based tree [13] with the rendezvous node as root and the publishers and subscribers as leaves. Moreover, such a tree is created for each event type and is used for *type-based* routing in HERMES.

In order to also support the inheritance of event types, the dissemination trees of related types get connected with each other. This is done by additionally forwarding either subtype advertisements or supertype subscriptions from their rendezvous node towards the rendezvous node of the supertype or the subtype, respectively. Simply said, the subtype rendezvous node can be seen as another publisher of matching notifications, while the rendezvous node of the supertype acts as another subscriber that is interested in all publications of existing subtypes. To support content-based filtering besides the type-based selection of notifications, additional modifications of the routing strategy are necessary. In particular, subscriptions with predicates over notification attributes need to be further forwarded towards the publishers. Thus, they also follow the reverse network paths created by corresponding advertisements from the rendezvous node back down to the publishers. The intention behind this strategy is to place the predicates close to publishers in order to filter out notifications that do not match as early as possible. This way, HERMES features *type-* and *attribute-based* routing of notifications, too.

As an event-based middleware platform, HERMES provides an interface for service extensions. Specifically, extensions to avoid network congestion, to detect composite events, and to secure information by encrypting event notifications have been implemented. The *congestion control* mechanism [181] applies two different algorithms for avoiding congestion. The first algorithm is publisher-driven and uses a feedback loop to throttle the publication rate in case of congestion. The second algorithm is subscriber-driven and controls the number of retransmissions simultaneously requested after a failure allowing the system to recover properly and continue its normal operation.

The *composite event* service [182] enables users to specify tempo-spatial patterns of events and to get notified if such a pattern occurs. In particular, the provided composite event language allows to specify concatenated, ordered, and interleaved events, iterations and alternatives as well as timing constraints. The specified patterns are subsequently compiled into extended finite state automata that are used for detection. Furthermore, complex automata can be split into simpler ones and be distributed in the overlay network. This way, event detectors for subpatterns can be placed on brokers close to the respective event sources in order to save network traffic.

The *security* service extension [15] implements a role-based access control scheme. Based on their role membership, principals are granted individual privileges to distinct event types. Confidentiality constraints are primarily enforced by encrypting event notifications. In particular, it is even possible to individually encrypt event attributes using different encryption keys. As a consequence, principals possessing the complete set of keys are able to decrypt the whole content of a corresponding notification while others may only be able to read the supertype attributes. Further details of Hermes' access control model and implementation are discussed in Sect. 4.6.2.

2.3.7 Padres

The Padres publish/subscribe system [67, 98] is a research prototype designed to support the management of business workflows and to facilitate the integration of enterprise applications. Therefore, Padres provides a distributed content-based publish/subscribe middleware that allows for event correlations in order to detect complex event patterns. Moreover, it features a uniform access to historic and future data as well as effective algorithms for load balancing and failure recovery. On this basis, an ecosystem of tools and services is provided enabling developers and administrators to model, implement, and compose business workflows as well as to integrate, control, and monitor processes and applications [122].

Padres notifications consist of typed name/value pairs of which one attribute is mandatory and describes the notification's class. Notifications have to be advertised by publishers in a way comparable to database schemas. In particular, advertisements specify the type and may constrain the value ranges of each notification attribute. Likewise, subscriptions may contain similar predicates over notification attributes in order to specify in which events a client is interested. Padres brokers form an acyclic overlay network that is used to exchange notifications as well as subscriptions and advertisements. Usually, a *covering-based* routing strategy is applied although *merging-based* routing is supported, too.

Remarkably, Padres uses the Java Expert System Shell (Jess) [80] for matching notifications against subscriptions which is a full-fledged rule engine based on the *Rete* algorithm [78]. Therefore, the advertisement as well as the subscription table are maintained as Rete trees that store matching rules and matching states, while notifications are mapped to facts that are presented to the JESS engine. Leveraging the Rete algorithm, even composite subscriptions looking for tempo-spatial event patterns can be matched successfully. Combined with the deployment of data stores that keep recently published notifications, this also allows for the correlation of current and historic event data [120].

As Padres aims at enterprise deployments, load balancing strategies as well as fault tolerance mechanisms are considered, too. For both reasons, a routing extension is provided that drops the limitation to acyclic broker networks enabling efficient and fault-tolerant forwarding strategies for *general* broker topologies that, in particular, contain redundant paths and network cycles [121]. Basically, the idea is to let each advertisement construct its own dissemination tree that is marked by a unique identifier. Tagging subscriptions and notifications with tree identifiers of matching advertisements allows to subsequently detect and break existing forwarding cycles while profiting from redundant network paths used to better distribute the notification traffic.

Regarding the processing load caused by matching event notifications, Padres brokers are able to *offload* local subscribers to neighboring brokers in order to balance the number of clients for which a broker is responsible [38]. In particular, dedicated algorithms are available to independently balance the input utilization ratio, the matching delay, and the output utilization ratio or any combination of the three performance metrics among participating brokers.

Considering *fault-tolerance*, Padres is able to cope with broker failures [98], for instance, if the broker process or its server crashes or cannot be reached by others anymore. In this case, the neighboring brokers try to reestablish the overlay network by circumventing the faulty broker and connecting to each other directly. Thus, in order to tolerate n brokers failing at the same time, each node has to know all brokers in its neighborhood up to a distance of $n + 1$ hops. After reestablishing the network, the brokers start the recovery procedure by reconciling their routing tables. Additionally, the sequence numbers of sent and received event notifications are compared to each other in order to detect lost notifications which are then retransmitted automatically.

Leveraging its features and functionality, the Padres broker infrastructure provides a reliable and sound basis for service orchestration, workflow monitoring and management as well as enterprise application integration [98]. Publish/subscribe communication is used to implement an *enterprise service bus* (ESB) enabling developers and administrators to orchestrate and mediate the interactions between enterprise services and business applications. Primarily, the ESB offers communication interfaces compliant to web and middleware standards as well as adequate adapters that care for data transformation, service registration, and service lookup. This way, the ESB makes it easy to split complex business workflows into cooperating tasks and communication services as well as to distribute these across the infrastructure as needed. In [122], a methodology and runtime engine is presented that shows how break up processes specified in the business process execution language (BPEL) [163] into individual activities and map these activities to dedicated agents controlling their execution. Compared to central service orchestration engines, this allows for flexible and dynamic deployments in which parallel processes can profit most from their distribution.

2.4 Discussion

In this chapter, we took a look at the background of publish/subscribe systems and event-based infrastructures. First, we clarified basic terms and definitions such as events, notifications, subscriptions, and advertisements before discussing the different options and alternatives to build publish/subscribe broker topologies and to efficiently route, filter, and select notifications therein. Thereafter, we reviewed existing industry standards for notification services as well as research prototypes of publish/subscribe systems. While the former are often embedded in middleware environments such as Corba or Jee, the latter primarily focus on specific aspects of publish/subscribe systems such as dynamic broker topologies, routing strategies, or matching algorithms.

From a software engineering point of view, both is disadvantageous. Within established middleware environments, on the one side, there are best practices and proven design guidelines for developing systems and applications in general. However, as publish/subscribe is often seen as an additional middleware service for asynchronous messaging only, these practices and guidelines usually do not adequately address the loose coupling as well as the inherent characteristics of event-based infrastructures and applications. But, on the other side, solely concentrating on publish/subscribe techniques as done by many research projects does not help either when, at the same time, the development of applications is neglected completely.

As reviewed in this chapter, there are already approaches in industry and academia that aim to improve the engineering of publish/subscribe systems and event-driven applications. For a seamless integration into applications, for example, the Data Distribution Service (DDS) standard specifies how to map publish/subscribe entities to constructs and concepts of object-oriented programming languages. Unfortunately, the layer responsible for this mapping is only optional and, thus, not necessarily implemented by every vendor. The Hermes and Padres projects prove that publish/subscribe communication can itself provide the basis for an own middleware platform on which advanced services such as access and congestion control, the detection of complex events, or the management of business workflows can be realized. For these and similar services to work properly, however, additional functions and features must usually be embedded into the middleware which is often costly in terms of labor and time. Regarding a modular engineering approach, scoping has been shown to be applicable to event-based infrastructures, too. Here, scopes restrict the visibility of notifications to subsets of components and, thereby, provide necessary structuring means that help organizing publish/subscribe networks and event-driven applications alike. But because of a plethora of integration options and contradicting alternatives that each have individual strengths and weaknesses, there is no reliable and accepted implementation of scoping from which system administrators or application developers could actually profit.

From these examples, it becomes evident that isolated approaches may introduce new ideas and show the feasibility of concepts, but usually do not gather enough momentum to sustainably improve the engineering of systems and applications. Instead, a holistic approach is necessary which orchestrates actions and aligns engineering concepts and methods with both the middleware and the application layer. For this purpose, on the one hand, we present, revise, and evaluate a *scoping concept* for event-based infrastructures and, on the other hand, provide a *modular broker architecture* as well as *programming abstractions* to better integrate scopes into the publish/subscribe middleware and the event-driven applications, respectively. In the following chapters, we discuss each of our contributions in detail.

Chapter 3

Composable Publish/ Subscribe Architecture

Contents

3.1 Introduction

Publish/subscribe systems are becoming an integral part of modern communication infrastructures. In particular, they are well suited to provide the basis for event-driven architectures and applications. They enable information about relevant changes in state or environmental conditions to be distributed in a timely manner in order to trigger dependent actions and processes without latency. This agility is especially appreciated in a growing number of application domains. Consequently, the requirements posed to the publish/subscribe infrastructure also increase in numbers and variety. Such requirements include scalability, reliability, and security aspects as well as domain-specific extensions and constraints that have to be taken into consideration. Sometimes, they are even contradictory. This often leads to complex system designs and difficult implementations forcing developers to consider trade-offs and make compromises. From our publish/subscribe middleware Rebeca [166], we know that there is no 'one-size-fits-all' implementation. Instead of maintaining several different program versions, we aim for a *modular middleware* consisting of individual building blocks that are ready to be freely composed. This way, it is possible to tailor the system to the actual requirements and conditions by selecting and combining only those blocks that are relevant and needed. To facilitate this kind of functional modularity, we present a composable architecture in this chapter that enables developers to flexibly build, configure, and adapt customized publish/ subscribe systems.

First, we introduce the concept of features to address and represent functional aspects in Sect. 3.2. We discuss their composition as well as implications and challenges for an architecture to support functional composability. In Sect. 3.3, we present a modular publish/subscribe architecture based on the concept of feature composition. Thereby, each feature is implemented as individual plugin that can be added to a broker. Thus, composing features basically boils down to selecting and applying an adequate set of plugins. Section 3.4 gives an overview about available feature plugins for our publish/subscribe middleware Rebeca, which is based on this architecture. Presented plugins address mandatory processing and publish/subscribe features as well as additional aspects such as manageability, security, adaptability, and fault tolerance. In Sect. 3.5, we highlight similarities and differences to related approaches that aim at building modular publish/subscribe systems or provide composable architectures for other domains. We conclude the chapter with a final discussion in Sect. 3.6 summarizing our main contributions and the advantages of a composable architecture in terms of flexibility gained by free composition of features.

3.2 Feature Composition

After successful software systems have been developed, shipped, and deployed, they are usually maintained, updated, adapted, and extended as existing requirements change and new ones arise over time. This process is called *software evolution* for which a set of common behaviors has been observed. These observations are also known as Lehman's laws [117, 118]. The most important observations are that actively maintained software systems are subject to continuing change as existing functions are improved and new ones are added. As they evolve over time, their complexity steadily grows while, to the same degree, they loose structure and organization. This happens until a point is reached where deliberate actions such as refactoring or redesigning need to be taken to thoroughly revise and streamline the software system reducing its complexity again. Regarding publish/subscribe software, the REBECA middleware is no exception to Lehman's laws. It has grown over the years and, in order to stay maintainable, a radical redesign was necessary.

As a research prototype, REBECA does not need to support all features and functions at the same time that have been developed so far. In fact, when investigating, testing, or evaluating a particular aspect of publish/subscribe communication, too many additional features may be distracting or even counterproductive. Often, just a specific combination of a limited set of functions is needed. Thus, the primary design goal is to facilitate the flexible composition of features. However, before presenting REBECA's new design, we first introduce the ideas and principles it is based on. In the following, we first clarify terms and concepts we have used intuitively so far and discuss the challenges for building a modular publish/subscribe middleware supporting the free composition of features.

3.2.1 Features and Composition

In software engineering, a *feature* is usually a distinguishable characteristic or property of a software artifact [3, 106]. For example, it may refer to characteristics such as performance, portability, or functionality. In this chapter, however, we focus on system functionality and define the term feature accordingly.

Definition 8 (feature). *A* feature *is a well-defined aspect of a software artifact's functionality.*

This definition has the advantage that it allows us to specify the whole functionality of a software system in terms of features. Thereby, each feature represents a specific software function or functional aspect. Enumerating all features, thus, yields in an overall functional specification. Moreover, by incrementally adding and integrating features, we usually get more sophisticated systems with higher functionality. Often, this process is called *feature composition* [183, 184].

Definition 9 (feature composition). Feature composition *is the process of combining features and associated software artifacts to a new artifact that unites these features.*

This definition emphasizes the process of combining artifacts and features, but leaves open what exact features the resulting artifact exhibits. The reason lies in a possible interference or *interaction of features* [23, 25] of which there are three types: none, positive, and negative interference. If no feature interference occurs, i.e., the features to compose are independent and orthogonal to each other, the resulting artifact will also exhibit all the features that are brought into the composition. This is the easiest and a very convenient case as it allows to freely combine those features without any restrictions. With positive interference, the resulting artifact may also show additional features that originate from the composition itself. In particular, this case is often desired since the composition result will be more than the sum of its parts. However, the new feature is no longer independent and its interdependencies need to be considered in subsequent compositions. In contrast to the previous cases, negative interference is usually undesired. Here, individual features and functions that perfectly work in isolation hamper and countervail each other when combined. Thus, in order to be composable, they need to be modified and adapted first.

Feature composition is an appealing concept. Starting with a core system without significant functionality, the system can be incrementally extended by composing new features. This way, it is possible to build highly customized systems tailored to specific purposes, requirements, and needs. For example, a basic publish/subscribe middleware can be equipped with just the filter and routing algorithms that are subject to a thorough analysis omitting irrelevant and distracting features in this context. Because of its adaptability and flexibility, such a feature-oriented design approach is advantageous especially for the development of prototypes.

3.2.2 Architecture and Composability

So far, we have focused on functionality only and neglected structural aspects. Obviously, each feature has an implementation, which is usually encapsulated in a module or a component. Thus, composing features also includes a structural composition, for example, by assembling modules or connecting components. For this to work, however, it must be ensured that the individual structural elements and pieces fit together.

Definition 10 (software architecture). *The* software architecture *determines the set of structures a software system is build of comprising software elements, their relations, and their properties.*

In literature, manifold definitions of the term *software architecture* are given. In [43], Clements et al. gather and list several prominent examples. They all are centered around structures, software elements, and the relations among them. Often they are also supplemented with additional aspects relevant in the particular context of the respective authors. In this respect, our definition is no exception as we focus on structures for building software systems. In fact, we require the architecture to determine the elements and their relations within the software system and, thus, the rules by which it is build and composed. Thereby, the architecture can prescribe the design of system elements as well as their interfaces and, hence, ensure that they fit together. We call such an architecture composable. Accordingly, we also define *composability* as a property of a software architecture and not as a property of a particular system or its components.

Definition 11 (composability). *A software architecture is* composable *if it enables its elements to be combinable and recombinable in a predefined manner to form new and usually more complex architecture elements or system instances.*

The definition covers the intuitive meaning of composability as a systematic construction principle that is applicable to architecture elements as well as systems thereof and which allows for flexible and variable system configurations. Most importantly, it requires the architecture to determine the manner how system elements are composed, i.e., the composition operation itself. In literature, several stronger and more formal definitions also exist that consider composability with respect to a particular system property, e.g., timeliness or testability [112, 111]. Here, the goal is to specify the composition operator in a way that the particular property is guaranteed to stay invariant under composition or that it is verifiably reached and established by composition. For our purposes, however, such restrictive definitions are not necessary.

3.2.3 Challenges

After discussing the terminology in the previous sections in order to clarify the meaning of a composable publish/subscribe architecture, we now focus on major aspects and challenges of its design. Primarily, the architecture has to facilitate and support the free combination of publish/subscribe features and functions. Hence, it must provide design guidelines and mechanisms to plug chosen features into a broker in order to flexibly extend its functionality. This is challenging because of several reasons. First, features must be encapsulated in components to make them pluggable. Second, interfaces must be designed and provided that allow a free composition of those components as well as their integration into the broker. And third, the plugged features need to work together respecting functional dependencies while avoiding negative interference. Having a reasoned and sustainable feature management is, therefore, a necessary precondition to allow systems to further evolve in future. In the following, we discuss each challenge in detail.

Features vs. components. Features just describe functional aspects. In order to make them composable, they need to be encapsulated in pluggable components. However, *functional modularity* in terms of features and *structural* *modularity* in terms of pluggable components are two different and orthogonal concepts. Both need to be combined in a sound and reasonable way so that components have a common structure to be uniformly plugged into a broker while they must still be able to fulfill the functions they are designed and intended for.

Composability. The broker architecture must be designed in a way that allows the free composition and recombination of feature components. On the one hand, the broker has to provide interfaces enabling the necessary access to event messages and data structures for implementing a particular feature. On the other hand, the broker architecture has to ensure that event messages are processed in a controlled and organized manner. The latter is especially important in order to make features composable, i.e., allowing feature components to be freely combined and recombined. Primarily, well-defined *interfaces* and a precise *processing scheme* allow a feature component to be easily replaced by other components with different properties.

Feature interaction. Features may interact and interfere with each other in manifold ways. On the one hand, positive feature interaction is desired or sometimes even necessary if features depend on each other. Hence, the broker architecture must allow plugged components to interact with each other. On the other hand, certain feature combinations may also interfere with each other in a way that they hamper or even countervail their actions. Unfortunately, preventing such negative interferences by design is impossible as, in general, it cannot be distinguished from the positive case. However, the broker architecture can enforce *interaction patterns* to make feature interactions visible and explicit in order to simplify and ease the elimination of undesired side effects.

3.3 Architecture

Functional modularity is the key concept for enabling a fine-grained composition of features. In particular, we consider every function of a publish/subscribe broker, even trivial and elementary ones, as being subject to a feature composition. Consequently, this idea is also reflected by Rebeca's new architecture [166]. The revised architecture comprises two different main types of elements only: the *broker* and its *plugins*. Basically, the broker is just a simple plugin container only able to manage inserted plugins, while the actual publish/subscribe functionality, however, is provided by these plugins that each implement a specific feature. In this context, feature composition, thus, boils down to inserting the desired combination of plugins into the broker. This section elaborates on the structures of brokers and the details of plugins that make this approach work. Furthermore, we discuss the connection of publish/subscribe clients and application components to brokers and give further implementation guidelines and blueprints to support the development of custom feature plugins.

3.3.1 Broker

REBECA's new architecture is based on functional modularity in order to make the composition of features as free as possible. Therefore, brokers do not possess any built-in publish/subscribe functionality. Instead, all publish/subscribe logic is encapsulated in pluggable components that can be inserted at runtime. The broker itself only has limited functionality to manage plugged components and organize the flow of event messages to be routed and processed. In particular, it supports the concept of *message channels* and *processing stages* into which the components are plugged. There, the plugged components are responsible to realize and implement the desired publish/subscribe features.

In general, when a broker processes an event message, three different phases can be distinguished. First, the message is received. Thereafter, it is handled and routed appropriately. And finally, it is send to neighboring brokers and clients. For each phase, we have reserved an own processing stage to execute the individual processing steps required in the particular phase. The *input stage* corresponds to the first message handling phase and starts with the receipt of the event message from a neighboring broker. Typical processing steps in this stage are, for example, message decryption and message deserialization. Thereafter, the second phase and the *main stage* for message handling follow. Here, the publish/subscribe routing decisions are made. The deserialized notification, for example, is matched against the subscription filters stored in the routing table to determine, to whom it has to be forwarded, while a received subscription gets incorporated into the routing table and is also forwarded, if not already covered otherwise. Finally, the *output stage* is passed in the third phase, where notifications to be forwarded are prepared for transmission, for instance, by serializing and encrypting them again.

The stages have different contexts in which event messages are handled and processed. Input and output stage have a separate message channel for each incoming and outgoing connection, respectively. Thus, message processing is limited to the context of the respective *connection* here. The main stage, however, has just a single channel that is used to globally handle messages in the context of the *broker*. This is required, for example, to globally administer the routing tables as well as to make routing decisions. The complete message processing scheme is illustrated in Fig. 3.1 visualizing processing stages, message channels, and feature components plugged in. An event message is received from an incoming connection and traverses the broker from left to right. Thereby, it is forwarded from plugged-in component to plugged-in component that are chained one after the other in the message channels.

As there is just a single message channel in the main stage, event messages must be multiplexed and demultiplexed when being transferred from the input to the main stage and from the main to output stage, respectively. Multiplexing is realized by first storing all event messages in a queue that arrive at the end of the input stage. Thereafter, they are individually dequeued and handled one by

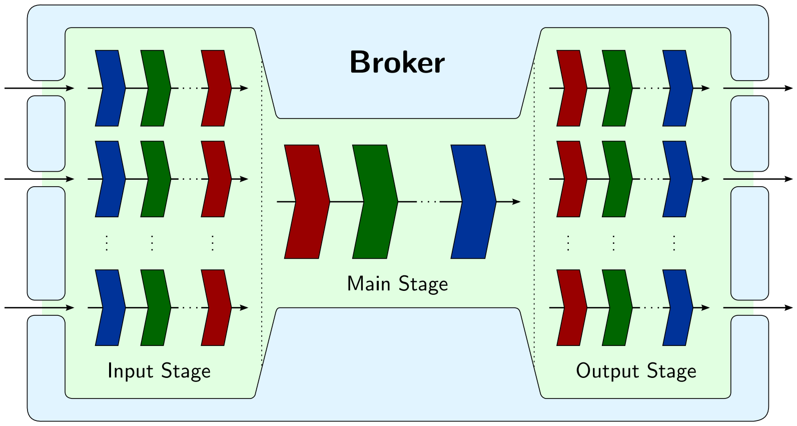


Figure 3.1: A broker's processing stages with plugged components.

one in the main stage. At the end of the main stage, the handled event message is demultiplexed again. Depending on the destinations, the event message needs to be forwarded to, it is copied to the respective output channels. In particular, this includes cases in which the message is copied to multiple output channels, transferred to just a single channel, or not forwarded at all. In any case, however, it is ensured that each output channel works on its own message copy independent of other channels.

Publish/subscribe functionality and other advanced broker features are realized by manipulating event messages and message streams within the broker's message channels. In particular, a component plugged into a channel has the possibility to arbitrarily alter any passing event message, for instance, by marking the message, by adding or changing message attributes, or even by transforming the whole content. Beyond that, a passing message can also be deferred or removed and new event messages can even be created and inserted into the message stream. Depending on the channel and the position where the component is plugged in, the stream of event messages can, thus, be globally processed and altered within the main stage or directly modified after receiving or before sending in the context of the incoming or outgoing connection, respectively.

Usually, this provides sufficient options for a component to implement a particular feature. When several of those feature components are plugged into the broker, the message channels, the positions, and the sequence, in which they are traversed by event messages, become important as all of them may intercept and manipulate the message stream. Therefore, the *broker configuration* exactly specifies in which channel which component is plugged and how they have to be chained in order to work as desired. In fact, it is the broker configuration that thereby defines the feature composition and, thus, determines the broker's overall behavior.

3.3.2 Plugins

Broker plugins encapsulate features to make them dynamically addable when needed. Thereby, each plugin ideally implements a single feature. For advanced features, however, more complex message handling strategies are needed that often require intercepting and manipulating event messages in a coordinated way at different stages in a broker's processing scheme. Hence, a single component only plugged into one of the broker's message channels is not sufficient. Therefore, Rebeca plugins are allowed to comprise multiple components in order to implement the message handling strategy needed to realize a particular feature. To simplify plugin management, we even require a plugin to provide a component for each processing stage and message channel by default. Accordingly, the components depicted in Fig. 3.1 that have the same color also belong to the same feature plugin.

Components plugged into the main stage have a distinguished position in the message processing scheme. Within the global message channel, they have the possibility to access and modify all event messages the broker receives. Thus, they usually contain essential data structures as well as a major part of the logic that drives the event processing to realize the particular feature. To emphasize their importance, we refer to these components as *broker engines*. Accordingly, they are also depicted larger in the figures than the components of the input or output stage. Routing engines are a representative example as all routing decisions are made in the broker's main processing stage. Here, the routing engine provides and administers the broker's routing tables and also implements the particular routing algorithm operating on these tables to determine whether a received event message has to be forwarded and, if necessary, to which neighboring broker. Preprocessing in the input stage or postprocessing in the output stage is usually not required for event routing.

Certain features, however, are easier to be implemented in other stages. Serialization and encryption, for example, are usually realized in a connection context. Characteristically, those features also involve two inverse operations from which one is applied in the input stage while the other is executed in the output stage. Moreover, when composing those features their respective operations are applied one after the other in one stage, while their inverse operations, however, must also be carried out in inverse order in the other stage. If an event message, for example, is first serialized and subsequently encrypted before being transmitted, it must first be decrypted and then deserialized after receiving before it can be processed further. As consequence, the input stage and the output stage of a broker are usually symmetric to each other. Hence, combining input and output stage and arranging both vertically as shown in Fig. 3.2 leads to a layered design.

Layered architectures are known from network protocol stacks and also used by many middleware implementations [197] where they have been proven to be functional and beneficial. In particular, higher layers allow to abstract from the processing details of lower layers and, thus, help to organize the message

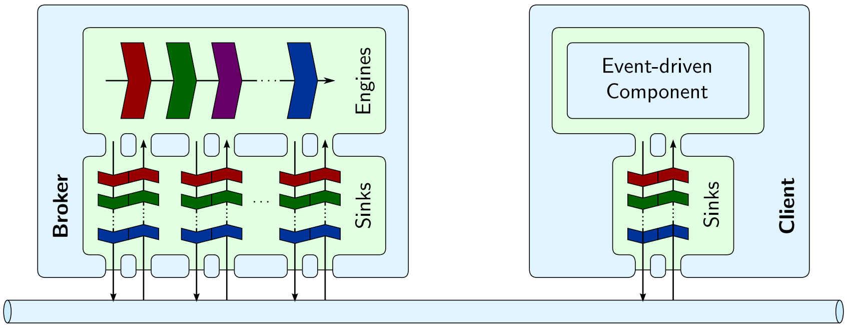


Figure 3.2: Comparison of client and broker architecture.

handling and to manage its complexity. Therefore, we combine each pluggable component of the input stage with its corresponding component of the output stage to form an *event sink* representing an own processing layer. Event sinks are stackable and can be directly plugged into a broker connection in which context they process and modify event messages that are received or send. Thereby, received event messages are passed upwards during the input stage, while forwarded messages travel downwards during the output stage. An event sink, thus, allows to conveniently implement a feature that requires to intercept the message stream at a specific point in both stages.

Rebeca plugins, to state more precisely than above, always comprise a broker engine as well as an event sink whose instances are plugged into connections to clients or other brokers. This way, it is ensured that the plugin can intercept event messages of interest in all three processing stages. If this is not needed, engine and sinks simply pass the event message along the processing stage without modifying it. Ideally, a Rebeca plugin implements a single feature, works transparently and autonomously, and is self-contained and independent of other plugins. This way, arbitrary combinations of plugins can be added to brokers enabling a free combination of features.

As features, however, may depend on each other, plugins may not always be autonomous and self-contained. On the contrary, plugins implementing dependent features often need a way to interact with each other. On the one hand, this can be realized by modifying passing event messages or inserting new control messages in the message stream by which these plugins communicate. Advantageously, this solution requires no further architectural changes. On the other hand, the Rebeca architecture does not prevent plugins from implementing additional interfaces to coordinate their actions or share common data structures. For example, advertising the event notifications a publisher is going to produce can help to significantly reduce the number of subscriptions that must be exchanged between brokers. For this purpose, the advertisement plugin has to determine all those subscriptions managed by the routing plugin that overlap with a particular advertisement. Instead of storing each subscription twice, both plugins may share a single subscription table. As advertising publications always require a filter-based routing algorithm to work, the advertisement plugin can, thus, assume that a subscription table is already provided by the routing plugin. Rebeca brokers allow plugins to query for other plugins' engines, interfaces and versions. This way, a plugin can ensure that all its requirements and preconditions for a successful feature composition are met.

3.3.3 Clients

For publish/subscribe applications and clients, Rebeca brokers provide a programming interface to advertise and disseminate own event notifications as well as to specify their interest in notifications of others. The interface is intentionally designed to be simple yet comprehensive having clear semantics that are independent of the actual features such as filter algorithms or routing strategies used within the broker network. On the one hand, this eases application development and ensures application compatibility in the long term. On the other hand, it makes middleware implementations more complex. The reason is that brokers have to distinguish whether they interact with a client or communicate with a neighboring broker. In case of a client, brokers have to support the simple programming interface, while, in case of other brokers, they use more advanced and efficient protocols for exchanging information. Supporting both by a single, combined middleware implementation, however, makes the message processing logic lengthy, more difficult to write, understand, and maintain, and thus often error-prone. Moreover, resulting dependencies also hamper the free composition of individual broker features.

In order to simplify broker implementations and clarify their processing logic, we drop the necessity to distinguish between client and broker connections as well as to support the traditional simple application interface. From a broker's point of view, there are only connections to other neighboring brokers that use advanced protocols. As consequence, clients have to behave as regular brokers now making their implementations more complex. To master the new client complexity, we build on the same concepts that we already applied successfully on the broker side: feature composition and high modularity. Likewise, we allow clients and application components to be extended with pluggable *client sinks*.

Figure 3.2 schematically compares client and broker architecture highlighting both chains of event sinks. On the client side, additional event sinks are transparently plugged into the connection between application component and broker. They form a chain of event sinks that is equally structured as the one on the broker side. For each sink on the broker side, there is also a corresponding sink at the same level on the client side. Both sinks belong together and are responsible to realize a particular feature. For this purpose, client sinks can also modify, insert, remove, or delay event messages that are passed up- or downwards the sink chain. Hence, clients have same preconditions and possibilities to implement the advanced broker protocols for information exchange. While the application component itself can still use the traditional simple publish/subscribe interface, the client sinks plugged into the connection transparently ensure the compatibility with the advanced broker protocols. Thus, changing or adapting the publish/ subscribe application, their components and interfaces is not necessary unless the application wants to explicitly use and leverage plugged new features.

With pluggable client sinks, a part of the event processing logic is shifted from the broker to the client. This is beneficial for several reasons. First, it cleanly separates client logic from broker logic preventing client handling issues to get intertwined with event processing implementations. Second, it simplifies the implementation because each client sink just handles a single application component while brokers would have to care for multiple client connections otherwise. Third, it is usually sufficient for client sinks to implement a subset of the functionality of advanced broker protocols in order to just make them compatible while ignoring more sophisticated optimizations. This way, client sinks are often compatible to several broker plugins of the same type (e.g., routing plugins, advertisement plugins) and can, thus, be reused without changes.

However, there are also drawbacks. Considering performance, the layered sink architecture has penalties. In particular, if clients and broker are running on the same host or even in the same process context, the serialization and transport mechanisms of the sinks in lower layers become a burden and cause significant and unnecessary overhead. In this case, nevertheless, the modular structure of the architecture offers an elegant solution. It allows to simply omit plugging in those sinks on the client as well as on the broker side that are not needed in a particular setup. This way, it is possible to efficiently support local publish/subscribe components, for which event delivery does not require any serialization, as well as remote clients connected over network, which depend on appropriate data representations and transport mechanisms.

3.3.4 Implementation

Rebeca's broker architecture is designed to facilitate the fine-grained and flexible composition of publish/subscribe features. This is also reflected by the broker's implementation. All publish/subscribe functionality is bundled and encapsulated in pluggable components while the broker itself is stripped down to a simple *plugin container* managing the feature components plugged in. Each of those plugins consists of a *broker engine* and respective *connection sinks*. While the engine usually contains the major and global part of the functionality and logic to realize a particular feature, the sinks allow additional message manipulations in the context of a single broker connection just after a notification is received or before it is send. On the client side, a corresponding client sink is required to connect application components. Client sinks are quite similar to broker sinks and are developed side by side with the respective broker plugin to ensure their compatibility. The Rebeca framework already provides a number

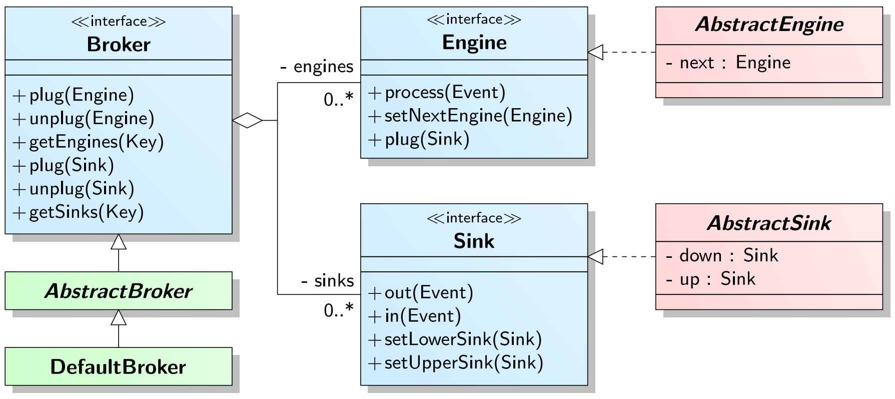


Figure 3.3: Overview about the primary architectural elements broker, engine, and sink as well as their relations.

of extendable base classes that significantly ease the development of custom broker plugins in order to implement new features. Figure 3.3 shows a UML class diagram giving an overview about their interfaces, properties, and relations. In particular, it highlights how broker, engines, and sinks are related to each other.

As all publish/subscribe functionality has been made modular and pluggable, a broker just needs to implement an adequate plugin container to manage the feature modules plugged in. The Broker interface ensures that the container provides methods to plug and unplug new broker engines and sinks. A broker engine is plugged or unplugged to add or remove a particular publish/subscribe feature, respectively. A broker sink is usually plugged by an active transport engine when a new connection to a client or neighboring broker has been established. The broker is then responsible to build up a complete sink chain for this connection and integrate it into the event processing scheme. The sink and its chain is unplugged again when the particular connection is closed.

Furthermore, the broker allows to query and list plugged engines and sinks. The getEngines method and the getSinks method return all engines and sinks, respectively, that are associated with a given key. Usually, engines register the additional interfaces they implement. This way, a dependent feature plugin can easily query for required interfaces and, thus, determine whether all plugins are present that are needed to provide its functionality. Broker sinks, however, are usually registered with the connection they are plugged in. In particular, the destination is often used as key for the topmost sink in a connection's sink chain. Hence, when queried for a destination, the first sink of the chain is conveniently returned that starts forwarding the event message to this destination. If no specific key is provided, the methods getEngines and getSinks simply list all engines and sinks, respectively, that are currently plugged. The Broker interface is implemented by the AbstractBroker class which, as abstract base class, is a starting point providing basic functionality and default logic for all further or custom implementations. This also includes REBECA's DefaultBroker class.

Each feature plugin contains a broker engine by which it is represented and accessed. Although broker engines usually implement the major part of the functionality to realize a particular feature, the general interface is plain and simple. The most important operation prescribed by the Engine interface is the process method for handling event messages. It allows engines to arbitrarily manipulate and modify a passed event message before it is passed further along the processing chain. The processing chain itself is set up using setNextEngine method which specifies the subsequent broker engine to continue processing the message. Furthermore, engines have a plug method, too. It is used to set up the sink chain for every connection established to a client or neighboring broker. After a connection is established, the method is called on every registered broker engine allowing it to plug in own logic and extend the connection's sink chain by adding a sink instance of an appropriate type. Please note, that there is no corresponding unplug method. If the connection is closed, the whole sink chain is deactivated and their resources are freed. Beyond these general operations, broker engines may also provide arbitrary additional methods that are required to implement a particular feature. As these operations are feature-specific, however, they are not part of the default interface. The AbstractEngine class provides a basic implementation of the interface and, thus, is well suited as extendable base class for the development of own, custom broker engines.

The Sink interface specifies the operations a broker sink has to implement. Most prominently, this are the in and out methods for message processing by which the sink can modify, transform, or manipulate a passed event message. Therefore, the in method is called for incoming incoming event messages just received, while the out method is executed on outgoing messages ready to send. After method completion, the event message is passed to next upper or lower sink in the connection's sink chain, respectively. The sink chain itself is set up using the setUpperSink and setLowerSink methods that specify the next upper and lower sink layer in the chain, respectively. The AbstractSink class provides a default implementation of the interface. It supports the construction of sink chains and its processing methods simply forward every passed event message further along the chain in the respective direction. Thus, the AbstractSink class is well suited as extendable base class for custom broker sink implementations. Regarding client sinks, they also follow the presented interface. But in this case, the AbstractComponentSink class, which itself is a subclass of an AbstractSink, is used as starting point by convention.

3.4 Feature Plugins

The last sections introduced the idea of feature composition as well as an architecture for publish/subscribe brokers based on this concept. The major benefit of such a composable architecture is its high degree of configuration freedom, adaptability, and flexibility. To demonstrate and underpin this statement, we give an overview about the various *feature plugins* in this section that are available in the REBECA middleware, show their interactions, and highlight sensible compositions to leverage synergies. The feature plugins are grouped into mandatory plugins required for brokers to work, plugins directly related to publish/subscribe functions, and plugins implementing diverse and miscellaneous features. In a case study, we also discuss a more complex scenario in which the REBECA middleware and its plugins are integrated into an environment for discrete event simulations. In particular, this requires to correctly manage simulation time, which is a non-functional property and, as cross cutting concern, equally affects the whole middleware and all plugins. Although REBECA's architecture is primarily designed for the composition of functional properties and features, its modular structure proves to be advantageous in this case, too.

3.4.1 Mandatory Features

Feature composition is the primary and fundamental design concept in REBECA. Even elementary functionality, which is usually built in other publish/subscribe brokers by default, is designed to be modularly pluggable in REBECA. On the one side, this allows to exchange the whole broker logic and to easily replace basic features and functions with customized implementations individually tailored to the respective purpose and area of application. On the other side, brokers still depend on a number of *mandatory plugins* to be available in order to work at all. In particular, this includes plugins responsible for configuration, event processing, and message transport. We discuss them in the following.

Configuration. Configuration plugins belong to one of the few plugin types that are not responsible to process received event messages. Instead, they contain a broker's *configuration* regarding which other plugins to load and how to instantiate, initialize and interconnect them. In particular, the configuration plugin assembles the processing chain of broker engines and instantiates the corresponding chain of engine sinks to be plugged into newly established connections to other brokers or clients. Please note that it is left open to the plugin developer and existing application requirements how the configuration data itself is stored. For example, it can be hardcoded into the implementation, fetched from a file or a remote server, individually negotiated when connecting to a present broker network, or any combination of the different options.

Processing. The processing plugin fulfills two major functions. First, it drives a broker's processing stage. Its engine is the first and the last engine in the processing chain. Therefore, it is responsible to queue the event messages received from different connections, initiate their processing one by one, and finally copy the message to the appropriate sink chains that lead to the respective forwarding destinations. Second, the plugin, thereby, links a broker's different processing stages by connecting its processing chain with its sink chains. For this purpose, the plugin's sink instances reside on the topmost layer of each sink chain and can, thus, liaise directly with the plugin's engine. Custom processing plugin's are often used to adapt a broker's *queueing strategy*. This way, it is easily possible to provide fair or weighted strategies, strategies based on priorities for messages, or any combination thereof.

Transport. The transport plugin is responsible to receive event messages from and deliver them to neighboring brokers, remote clients, or local application components. Furthermore, it establishes new connections and manages the established ones. Depending on the type of the connection, different *transport mechanisms* are used. Connections to neighboring brokers or remote clients are usually based on network protocols such as TCP or UDP [212], while for local components more efficient inter-process communication can be used. Thereby, each connection has exactly one transport sink, which implements the protocol logic and marks the bottom end of the connection's sink chain. The plugin's engine does not process event messages. Instead, the transport engine establishes new connections and manages the existing ones. For each new connection, a transport sink instance is created that is subsequently extended with other engine sinks to form the connection's sink chain. Assembling the sink chain is actually directed and controlled by the configuration plugin. An appropriate configuration provided, it is also possible to have multiple transport plugins loaded, for example, to support remote clients and brokers as well as local application components at the same time.

3.4.2 Publish/Subscribe Features

According to the concept of feature composition, a broker's publish/subscribe logic has also been rendered to be dynamically combinable and pluggable. However, publish/subscribe plugins are not considered mandatory as they do not contribute in making the broker operational itself. Instead, they add, adapt, and fine-tune *forwarding strategies* for event messages, which, nevertheless, are required for any advanced publish/subscribe functionality. In particular, the publish/subscribe plugins address aspects of matching event notifications, routing subscriptions, advertising publications, and partitioning the system into individual event scopes.

Matching. *Matching* refers to the process of comparing client and application subscriptions to received event notifications in order to determine who is actually interested in which notification. Therefore, the broker engine of the matching plugin manages a filter table in which subscriptions are stored together with their issuers. A received event notification is then matched against stored subscriptions to determine the set of interested receivers to which it needs to be forwarded and delivered. Matching plugins can be differentiated by the kind of algorithm applied, the algorithm's efficiency, and the expressiveness of subscriptions that is supported [64]. The Rebeca framework supports arbitrary Boolean expressions based on constrained notification attributes.

Routing. Publish/subscribe *routing* usually refers to the strategies applied in a broker network to exchange information about issued event subscriptions. Thereby, routing strategies can be differentiated by the degree of which they are able to exploit similarities between the filter expressions of issued subscriptions in order to reduce the amount of exchanged information. The Rebeca middleware provides plugins that support simple, identity-based, and covering-based routing strategies [128] for disseminating event subscriptions. Thereby, the routing plugins use the filter table implemented and shared by the matching engine in order to avoid storing issued subscriptions twice.

Advertising. Advertisements provide additional means to further reduce the message overhead induced by distributing issued subscriptions within the publish/subscribe network [146]. For this purpose, *advertising* requires all application components, remote clients, and other information producers to announce the kind of notifications they are going to publish. This way, a broker can restrict forwarding an issued subscription only towards those brokers that also host a potential publisher. Therefore, the plugin's engine manages another routing table in which advertisements are stored together with the brokers they were received from. Using this table, the plugin's outgoing sink can easily suppress the forwarding of a subscription if no overlapping advertisement is stored from the particular destination and indicates a potential publisher. Advertising, thus, effectively avoids forwarding a subscription into network regions where no publisher evidently exists that is going to produce a matching notification. Hence, depending of the distribution of potential publishers, advertising can save a large amount of subscription overhead [142]. For disseminating advertisements in the broker network, the same routing strategies as for subscriptions are available. The Rebeca framework provides advertisement plugins featuring a simple, an identity-based, and a covering-based routing strategy.

Scoping. *Scoping* provides effective means to structure and organize publish/subscribe networks as well as their event-driven applications. Primarily, this is done by delimiting the visibility and, thus, the dissemination of event notifications in the network. This way, scoping enables system administrators as well as application developers to create individual visibility domains and hierarchies thereof. These are well suited, for example, to direct and control event and information flows, to reflect organizational structures within the network, or to model entire application domains and restrict their interactions. Scoping requires brokers and clients to first join a particular scope in order to receive event notifications that were published therein. While the plugin's engine manages scope membership, the plugin's sinks enforce visibility constraints by simply dropping an event notification if it leaves the scope while being forwarded. Exceptions can be made by specifying scope interfaces that corresponding notifications are allowed to pass. Chapter 4 introduces the concept of scopes, describes its integration into publish/subscribe routing, and also elaborates on management and implementation details. Please note that Rebeca's scope plugin reuses existing routing and advertisement logic and, thus, requires both other plugins to be present and active.

3.4.3 Optional Features

Besides the essential message processing and forwarding logic, publish/subscribe middleware implementations usually offer numerous additional features that, for example, are only relevant in certain environments, improve the quality of service (QoS) under certain conditions, or make using particular middleware functions more convenient. For those *optional features*, the concept of feature composition is ideally suited. It allows to tailor the system to the actual application requirements and environment conditions. Features are composed only when actually required and omitted otherwise ensuring comprehensive, yet light-weight middleware implementations without too much overhead. The Rebeca framework contains several plugins implementing additional functionality to complete a broker's set of features. In particular, available, optional plugins consider monitoring and management issues, address security aspects, and improve the QoS in dynamic network environments.

Management. Using publish/subscribe in production environments usually requires advanced broker interfaces to remotely monitor, control, and manage a broker in order to enable administrators to identify and diagnose problems in real time and take immediate counter actions. This way, arising problems can often be solved before they cause an interruption in the operation of the broker network. Rebeca provides a *management plugin* based on the Java Management Extensions (Jmx) technology [216]. It enables administrators to inspect a broker's configuration as well as to load and unload additional plugins at runtime. For each loaded broker engine, an MBean instance is created reflecting the engine's interface and exporting its operations. This way, it is possible to query statistics and gather monitoring information, read and change attributes and properties as well as call and execute public methods and operations on the engines. For this purpose, various connectors are available that provide access to individual MBeans based on protocols such as SNMP [210], HTTP [77], or Java RMI [228].

Security. Secure communication is ensured by encrypting exchanged event messages. Therefore, Rebeca's encryption plugin uses the Java Secure Socket Extension (JSSE) [153] to provide a transport sink that encrypts the whole data stream within a connection. For each connection, *encryption* can be individually configured and activated. This way, it is easily possible to encrypt the data when communicating to brokers and clients over insecure networks and switch encryption completely off otherwise. Please note that REBECA's composable architecture also provides the preconditions to implement more fine-grained encryption plugins that are based on alternative approaches which even allow the encryption of selected event notifications or individual attributes only [15, 209].

Adaptivity. Dynamic environments are often characterized by changing network conditions as well as varying distributions of publications and subscriptions. Sometimes, these dynamics gain such a momentum that they easily overload individual brokers or links. However, by adapting the broker topology to present network conditions and current notification interests, it is often possible to avoid those overload situations. Moreover, continuous adaptation and self-optimization may help to ensure an adequate level of performance. The REBECA framework contains two plugins following different approaches to adapt the broker network in order to reduce the message overhead and improve its quality of service. The first plugin is based on a heuristic [103] that continually reorganizes the *broker topology* to minimize the number of forwarding hops between publishers and subscribers. Thereby, it also tries to replace expensive or instable overlay links with better suited connections. The second plugin does not change the broker topology but adapts the routing algorithm used within the network. It saves traffic by switching individual network links to the most efficient *routing strategy* available under the current conditions [199, 201]. Furthermore, both plugins can be used simultaneously in order to compose both heuristics to adaptively optimize the publish/subscribe infrastructure [171].

Fault Tolerance. REBECA's publish/subscribe infrastructure is based on an acyclic broker network. Hence, the failure of a single broker or overlay link may cause the network to get partitioned. To improve the network's robustness, REBECA features a *recovery plugin*. The implemented *recovery algorithms* ensure that the network gets reconnected again while bypassing the failing link or broker [170]. Hence, they render the network self-stabilizing provided that no other fault occurs during the time of recovery.

3.4.4 Discrete Event Simulation

Simulations provide adequate means to test and evaluate new broker features. They are often used where deployments of real broker instances are not suited or applicable. In particular, simulations offer fine-grained control of present load, network, and environment conditions and, thus, make experiments and their results easily reproducible. Furthermore, simulations also allow to study a feature's scalability in large network instances that may significantly exceed the resources available for real deployments. Therefore, simulations have been proven to be especially useful for the design, development, and analysis of distributed systems [214].

For publish/subscribe systems, discrete event simulations are often used to analyze routing algorithms and evaluate their performance. In particular, *discrete event simulations* model the system as chronological sequence of events, which are stored in an event queue and processed one after the other. Thereby, each simulation event occurs at a specific instant in time and represents a happening relevant to the system's state, for example, a published notification received by a broker. The event is then processed according to the protocols and rules of the simulated system which potentially cause new and future events to be scheduled and triggered. In case of the received notification, the message is forwarded to neighboring brokers and clients with matching subscriptions causing new simulation events to be scheduled for the notification's propagation and delivery. After the event has been processed completely, it is removed from the event queue. Furthermore, the simulation time is advanced to the next chronological simulation event which is then handled appropriately. This is repeated until the event queue is empty or a predefined end time is reached.

The Rebeca framework supports discrete event simulations. In particular, Rebeca brokers can either run as independent processes in a distributed deployment or alternatively be executed in the PeerSim simulation environment [139]. The latter, however, requires specific adaptations and changes that affect individual plugins to a varying extent. These changes include functional as well as non-functional aspects. Functional aspects primarily deal with communication. Brokers need a way to exchange notifications and subscriptions within the simulated network environment. Therefore, we provide a new transport plugin that connects a broker's sink chain with the simulated network enabling brokers to communicate with each other. Network characteristics such as transport and network protocols or the latency of links and their bandwidth are determined and controlled by the simulation environment.

The non-functional aspects mainly affect the simulation's *notion of time* as well as its style of execution. In fact, as the simulation is based on discrete events, it has no continuous time model anymore. Discrete events are just instants in time which itself have no duration. Hence, when modeling an action or process that takes time, for example, a broker forwarding a notification, it is necessary to schedule at least two discrete events: the first to represent the start of the process and, after an appropriate delay, the second to mark its end. All events are processed sequentially one after the other. For this purpose, a single thread is, thus, entirely sufficient to drive the processing. On the contrary, concurrent threads are rather harmful as they may mix up the deterministic order of event execution and, thus, limit the reproducibility of simulation runs. When synchronizing threads to ensure reproducible runs, however, there is often no advantage left anymore compared to the single thread model of execution. Instead, sophisticated thread synchronization is an additional source of complexity and potential errors.

To support discrete event simulations, the non-functional aspects cause the majority of implementation changes for brokers and their plugins. Primarily, there are two actions that must be taken. First, all timing aspects must explicitly be scheduled as discrete events. Therefore, all long-running tasks and processes need to be broken into individual events representing the achieved progress. Delays, durations, and runtimes must be explicitly specified and corresponding simulation events have to be scheduled appropriately. Second, all independent threads need to be deactivated making the simulation the only thread that is executed. Nevertheless, the tasks the deactivated threads originally fulfilled must still be carried out and, thus, have to be explicitly scheduled and triggered by simulation events just created for this purpose. Against this backdrop, required adaptations seem to be tremendous at first glance. In fact, changing the time and execution model affects the whole implementation. However, necessary changes usually follow the same pattern. The idea is to provide a *wrapper* for each broker plugin that encapsulates the plugin's engine and intercepts all calls that would, otherwise, lead to the creation of independent threads. Replacing these threads, the wrapper has now to ensure that, nonetheless, required tasks and operations are still triggered by scheduling corresponding simulation events. As a consequence, the wrapper is also responsible to handle them appropriately. To actually carry out the triggered operation, however, the wrapper may simply call the corresponding function of the original plugin.

For every broker plugin presented so far, the REBECA framework contains a corresponding plugin wrapper enabling its usage in the PEERSIM simulation environment. Often, a single wrapper implementation can be used for a whole class of plugins. For example, there is just one wrapper for all routing plugins and another one for all advertisement plugins. Furthermore, the plugin wrappers allow to uniformly set and specify plugin parameters as well as simulation values using a single configuration. In Chap. 6, we especially use PEERSIM simulations in order to study and evaluate the publish/subscribe algorithms and structuring means that are presented in the following chapters.

3.5 Related Work

In literature, numerous publish/subscribe systems are described and presented including commercial products as well as research prototypes [177]. Usually, they are tailored to a specific application domain and, thus, exhibit a well defined and fixed set of selected features. Only very few systems are designed to be reconfigurable and adaptable enabling features to be flexibly combined and composed. In the following, we first discuss some of these publish/subscribe systems that are open source and highlight their feature management. Thereafter, we present approaches and architectures from other domains that were intentionally developed to be composable and facilitate a free combination of features.

3.5.1 Publish/Subscribe Architectures

The Reds [48] framework is a reconfigurable event dispatching system designed for *mobile ad-hoc networks* (MANETs). In this context, reconfigurations primarily refer to adaptations of the dispatching broker network when the MANET topology changes dynamically. To facilitate the flexible application and evaluation of new reconfiguration strategies, the Reds framework has a modular architecture separated by concerns. Thus, it provides interfaces to plug in custom message formats and filters, specify matching algorithms and routing engines, change the underlying transport protocol, and provide specialized topology managers and reconfiguration strategies. Allowing these features to be composed freely, Reds is well suited to analyze the behavior of publish/subscribe infrastructures in dynamic environments under various conditions. In particular, the large configuration space makes the framework versatilely usable in many application domains. Regarding design concerns unrelated to reconfigurations, however, the integration of novel features is not supported to the same extend. In fact, the framework does not provide a way to define new interfaces and plug in functionality different from those it was originally designed for. This is the most important difference and a significant disadvantage compared to the more general Rebeca architecture [166] presented in this chapter.

The Padres project [67] leverages publish/subscribe for *enterprise application integration*. The project focuses on a distributed content-based publish/subscribe routing substrate that is accompanied by a whole ecosystem of tools and services dedicated to workflow management and monitoring [123]. In particular, the latter control and monitor the execution of business processes that are composed out of numerous individual business activities specified in the Business Process Execution Language (BPEL) [163]. Thereby, each activity is controlled and executed by a dedicated agent. In order to cooperatively implement the specified workflows, the agents communicate with each other using the content-based publish/subscribe infrastructure. This way, no central instance is necessary that triggers the execution of each activity making the architecture exceedingly scalable and extendable. In fact, workflows can easily be extended by adding new activities and created by freely composing activities and existing processes.

Concerning the publish/subscribe routing substrate, Padres brokers have a modular architecture, too. Similar to our approach, these brokers have input and output queues as well as core components encapsulating particular aspects of their functionality [98]. For example, Padres supports composite subscriptions that allow for event correlations as well as historic queries which also include events already published and delivered in the past. Although Padres components often have very efficient implementations, they are, nevertheless, quite coarse and heavy. For instance, the Java Expert System Shell (Jess) [80] is a full-fledged rule engine based on the Rete algorithm [78]. Padres leverages this engine to match and correlate events as well as make routing decisions. Hence, regarding publish/subscribe functionality, a really fine-grained and flexible composition of features is not possible. In contrast, for example, Rebeca considers the matching of events, the routing of subscriptions, and the forwarding of advertisements as individual features that are subject to composition.

3.5.2 Feature Composition

Besides publish/subscribe systems, other application domains profit from functional modularity, too. In particular, several architectures consider composability as fundamental principle and primary means. In the telecommunications domain, *Distributed Feature Composition* (DFC) [96, 232] is a virtual architecture for composing telecommunication services. It allows a feature-oriented specification and a modular implementation of services as so called feature boxes [233]. When establishing a connection (e.g., telephone call, Voice-over-Internet-Protocol (VoIP) call) from the caller to the callee, a component chain out of applicable feature boxes is dynamically assembled determining the composed and provided service. For this to work, each feature box must be transparent, autonomous, and independent as well as implementing just a specific functional aspect. The latter includes, for example, blocking calls from particular addresses or suppressing them at quiet times as well as forwarding calls to different receivers or the voice mail when the line is busy. The overall service is finally determined by the composed chain of feature boxes.

In REBECA, a broker's functionality is also defined by the feature plugins inserted into its processing stages forming a chain of broker engines and sinks, respectively. These chains are once assembled when the plugins are inserted and remain unchanged for all event messages processed afterwards. DFC, however, dynamically assembles an individual chain of feature boxes for every connection that is established. It may also modify a particular chain, for example, by forking it and adding new boxes when caller or callee issue another simultaneous call to a third person in order to start a telephone conference. A chain of feature boxes is finally torn down when one of the participants hangs up. In particular, this dynamic composition, modification, addition and removal of features make DFC an exceedingly flexible architecture for implementing advanced and versatile telecommunication services [20].

Feature composition and composability is not limited to functional aspects only. It can also be applied to non-functional properties. Thereby, a composable architecture usually ensures that a non-functional property such as timeliness or testability once established on subsystem level stays invariant under composition. Thus, the composed overall system also exhibits this property. For example, the *Time-Triggered Architecture* (TTA) [113] offers a versatile infrastructure for the development of fault-tolerant distributed real-time systems. In particular, it comprises distributed algorithms for communication, clock synchronization [114], and group membership [110] as well as their implementation in hard- and software. Each node providing a real-time service can be developed independent of other nodes and integrated constructively into the system without disturbing the operation and timeliness of other nodes and services.

In [186], Richling takes the concept of composability one step further and investigates properties that besides staying invariant or changing its value may also emerge or vanish. The idea is to provide an architecture including elements and composition rules that, when applied, lead to correct systems with desired properties. For the domain of embedded real-time systems, this is demonstrated by an architecture for *Message Scheduled Systems* (MSS) [185, 187]. As the architecture is formally proven [188, 189], the timeliness of an actual system is automatically guaranteed if it is build according to the architecture's composition rules. For Rebeca's architecture, however, we cannot give any similar guarantee. Nevertheless, its modularity has also been proven to be very advantageous when addressing non-functional properties, in particular, when dealing with discrete time due to the integration into a simulation environment.

3.6 Discussion

Publish/subscribe systems increasingly often play a key role in modern communication infrastructures. In particular, they provide the basis on which flexible event-driven applications and architectures are usually built. With the large variety of application domains, many different requirements are posed to a publish/subscribe middleware from which some may even contradict each other. Thus, it is hardly possible to consider all requirements in one implementation to the same degree. Instead, we focused on functional modularity that allows developers and administrators to tailor the publish/subscribe infrastructure to actual requirements and environment conditions. Therefore, we introduced the concept of features and *feature composition*. Based on this idea, we presented a composable publish/subscribe architecture on which our middleware Rebeca is built. Each feature representing a particular functional aspect is encapsulated in an individual plugin that can be inserted into a Rebeca broker. This way, brokers can be equipped with just those features that are actually required while superfluous functions are simply omitted.

When inserted into a broker, we allow *feature plugins* to intervene the internal stream of event messages at three different processing stages: after a message is received, while it is processed, and before it is send. This way, a plugin can modify and alter passing messages as well as remove them or even insert new ones in order to realize its functionality. Regarding its implementation, a plugin consists of an engine and an event sink. While there is just one engine per plugin inserted into the global processing stage of a broker, an individual sink instance is created and plugged into each connection to a neighboring broker or client. Plugins and their features are composed by simply chaining their engines and sinks within the broker. To unify and ease the development, a corresponding sink chain is also used on the client side. Based on this flexible architecture, we implemented the Rebeca publish/subscribe middleware as prototype accompanied by a diversity of feature plugins to test and demonstrate their composability. Some of these plugins are mandatory as they configure and drive a broker's internal processing stages, different plugins implement publish/subscribe features including diverse matching, routing, and advertisement strategies, and other plugins address management and security aspects or render the infrastructure adaptive and fault-tolerant. Furthermore, it is even possible to execute a REBECA broker and its plugins within a simulation environment.

The presented architecture forms the basis for the work done in the following chapters. Its functional modularity and composability significantly simplifies the integration of scopes as described in Chap. 4 that limit the visibility of events and, thereby, help structuring the publish/subscribe broker network as well as the event-driven applications built on it. Likewise, the encapsulation of functional aspects in separate plugins makes it exceedingly easy to revise and adapt just a specific feature. For example, by extending REBECA's component plugin that enables brokers to host local publish/subscribe components to a full-fledged component container, we are able to adequately support the programming abstractions for event-driven applications discussed in Chap. 5. Furthermore, we exploit the possibility to run REBECA brokers in a simulation environment in Chap. 6 in order to thoroughly analyze and evaluate the presented algorithms under various load and network conditions. We are convinced that a modular architecture facilitating a combination of features is an invaluable help for the design and implementation of future publish/subscribe systems.

Chapter 4

Scoping

Contents

4.1 Introduction

A major strength of event-driven applications based on publish/subscribe is the ease of integrating new components. By just adding them to the infrastructure, new components are already able to interact with existing ones by simply publishing and subscribing to relevant notifications. Further adjustments and modifications, especially on the side of already present applications, are usually not required. With increasing system size and complexity, however, this strength turns into a severe weakness as the probability of unexpected side effects when adding new components is growing quickly. Moreover, the indirect and loose coupling of components makes it eminently hard to identify all potential interactions and analyze their effects and side effects. Without structuring means, any part of the system may be affected requiring that the system as a whole needs to be reconsidered whenever a single component is added, modified, or removed. For large-scale event-based infrastructures, this is practically not feasible anymore.

Scopes precisely address this problem by limiting the visibility of notifications and, hence, restrict interactions to a comprehensible subset of components. Thereby, scopes support building modular event-driven applications and help structuring publish/subscribe systems at the same time [69]. For the former, application developers may leverage scopes to bundle related components, to hide their internal communication from outside interference, and to, thus, design reusable artifacts that can be shared and instantiated by different applications. For the latter, system administrators may use scopes to subdivide the publish/subscribe infrastructure into parts and subsystems that are independently administered, controlled, and supervised. Thus, even large publish/subscribe systems become and also remain manageable.

In this chapter, we build on the work of Fiege [68] and introduce scoping as a technique to structure publish/subscribe systems and to organize event-driven applications. We first lay the foundations in Sect. 4.2 by defining terms and notations, explaining the scope concept, and exemplifying its application to build customized scope hierarchies. Furthermore, scope attributes and notification mappings are discussed. Section 4.3 shows how to integrate scopes into distributed publish/subscribe infrastructures. We leverage scope overlays and specify the routing of notifications, subscriptions, and advertisements therein while preserving the freedom to employ arbitrary routing algorithms and configurations. Section 4.4 discusses the management of scope overlays and explains how components join or leave a scope while Sect. 4.5 provides further implementation details. We conclude this chapter with an overview of related work and a final discussion in Sect. 4.6 and Sect. 4.7, respectively, in which we compare the presented scope concept to other approaches, highlight similarities and differences, and point out individual strengths and weaknesses.

4.2 Scopes

Scopes primarily constrain the visibility of event notifications to a well defined set of application components and are, thus, well suited to manage and control their interactions. In the following, we define scopes based on sets of components and specify further scope features such as interfaces, attributes, and mappings. We show how to utilize these features to compose valid scope hierarchies that augment components with additional information, organize event-driven applications, and structure the publish/subscribe network. In general, we follow the terminology introduced by Fiege [68] but provide much simpler definitions based on set theory that lead to slightly different semantics and substantially ease the subsequent integration of scopes into distributed publish/subscribe systems. Please refer to Sect. 4.6.1 for a more detailed discussion of similarities and differences.

4.2.1 Specification

A *scope* simply bundles related components including publishers as well as subscribers and provides a shielded environment to interact with each other without interference from outside the scope.

Definition 12 (scope). *A* scope *S is an isolated environment for interaction given by the set $S \subseteq \mathcal{C}$ of event-driven components that communicate by publishing and subscribing notifications.*

A scope, thus, is primarily defined by the set of components that it encloses. Therefore, it is important how the components are chosen that belong to a particular scope. Basically, a Boolean *component selector* function can be used to determine scope membership.

Definition 13 (component selector). *The* component selector *of scope S is a Boolean function* $\mathrm{sel}_S : \mathcal{C} \to \mathbb{B}$ *on the set $\mathcal{C}$ of all components that evaluates to true for any component $C \in S$ and returns $false$ otherwise.*

By bundling the selected components to a scope and isolating them from the remaining system unexpected side effects are prevented effectively. This is done by silently discarding all notifications that cross the scope boundary by either being published by a component from inside the scope and delivered to outer scope components or vice versa. Components inside the scope, however, are able to receive all notifications from each other. Thus, their interaction is not restricted in any way while any communication with components of the remaining system is suppressed completely. In order to allow inner scope components to communicate with the outside system by publishing and subscribing notifications, we need to define *scope interfaces* that selected notifications may pass unhindered.

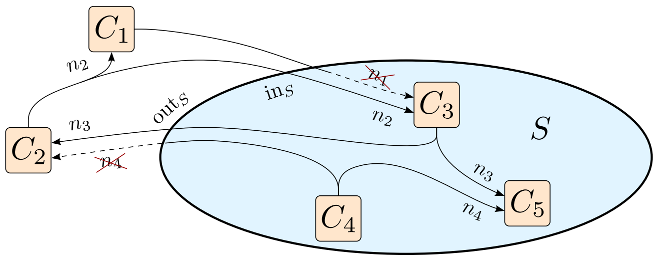


Figure 4.1: Bundling event-driven components in a scope.

Definition 14 (scope interface). *A* scope interface I_S *of scope S declares a pair of Boolean functions* $\text{in}_S : \mathcal{N} \to \mathbb{B}$ *and* $\text{out}_S : \mathcal{N} \to \mathbb{B}$ *on the set $\mathcal{N}$ of event notifications that are applied as follows:*

(*i*) *A notification $n \in \mathcal{N}$ published outside S is delivered to an inner scope component $C_1 \in S$ if* $\text{in}_S(n) = \text{true}$ *and discarded otherwise.*

(*ii*) *A notification $m \in \mathcal{N}$ published inside S is delivered to an outer scope component $C_2 \notin S$ if* $\text{out}_S(m) = \text{true}$ *and discarded otherwise.*

By filtering notifications when entering or leaving a scope, the interaction of inner scope components with the remaining system becomes manageable and controllable. Unexpected side effects are not completely prevented anymore, but thoroughly avoided as all incoming and outgoing notifications passing the scope boundary need to be explicitly specified by the filter functions of the scope's interface. Figure 4.1 exemplifies the definition. Components C_3, C_4, and C_5 are assembled within scope S while components C_1 and C_2 are not. Published notifications are filtered using in_S and out_S when they cross the boundary of S. Thereby, notifications n_1 and n_4 are filtered out by in_S and out_S, respectively, while notifications n_2 and n_3 are allowed to pass. In fact, scope S allows only components C_2 and C_3 to communicate across its boundary while components inside S as well as those outside S may freely interact among each other.

Scopes in publish/subscribe systems are comparable to module and package concepts known from many programming languages. Likewise, they encapsulate related components and hide their internal communication from the outside view. Furthermore, system interdependency is shifted from an uncertain set of components onto a well defined interface that specifies valid notifications and, hence, determines allowed component interactions.

4.2.2 Hierarchies

Scopes as defined above provide structuring means for publish/subscribe systems and event-driven applications. They allow developers to bundle related

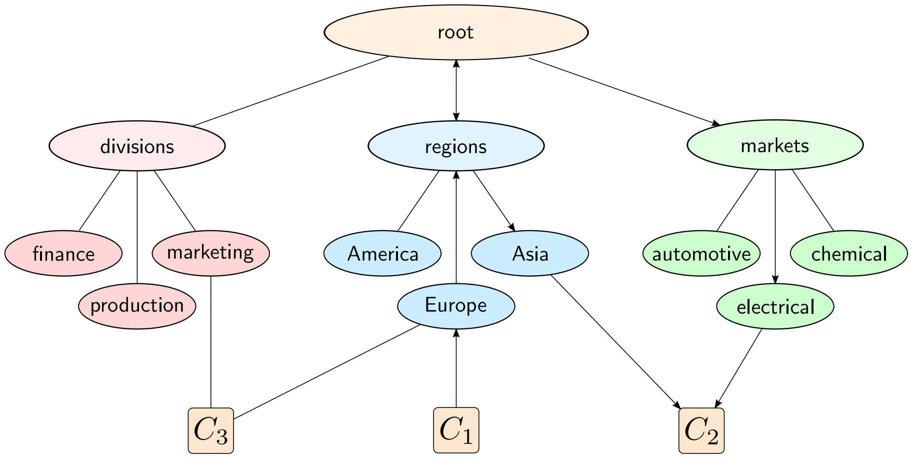


Figure 4.2: Scope hierarchy of a world-wide operating company.

components to reusable modules and application artifacts while administrators can split complex systems into simpler subsystems and units of a comprehensible size. To facilitate such modularization, scopes can be aggregated in *superscopes* as well as divided into *subscopes*.

Definition 15 (subscope/superscope). *Let S and T be two scopes. S is called* subscope *of T ($S \subseteq T$) and T is called* superscope *of S ($T \supseteq S$) iff for each component $C \in S \Rightarrow C \in T$.*

Using superscopes and subscopes event-driven applications and publish/subscribe systems can be organized according to different aspects and criteria. This way, a hierarchy of scopes is established. In order to ease the maintenance and administration a *scope hierarchy* must meet several conditions to be valid.

Definition 16 (scope hierarchy). *A set of scopes is called a valid* scope hierarchy *if it meets the following requirements:*

(i) *There is one* root scope *that contains all components.*

(ii) *Every scope, except the root scope, has exactly one distinguished superscope as direct* parent scope*.*

(iii) *Scopes whose parent scope is the root scope are called* top level scopes*.*

(iv) *Scopes that do not have further subscopes are called* leaf scopes*.*

(v) *A component can be a member of* several *scopes at the same time.*

Requirements (i) and (ii) ensure that the scope hierarchy is a tree with a single root scope. The root scope is, according to (iii), divided into top level scopes that each reflect a different criterion to organize the system. Top level scopes can successively be subdivided into subscopes to refine the criteria into aspects of interest and concern. Leaf scopes are not divided any further as requirement (iv) states. To ease system decomposition, requirement (v) additionally allows components to be assigned to several scopes if multiple aspects apply. Thus, a component is usually a member in a set of scopes composed by assigned leaf scopes along with all their superscopes. Those *scope sets* are called *valid*.

Definition 17 (valid scope set). *A set of scopes $\mathcal{S}$ is called a* valid scope set *iff for each scope $S \in \mathcal{S}$ (except the root scope) follows that also its parent scope P_S belongs to the set $P_S \in \mathcal{S}$.*

To illustrate the definitions, Fig. 4.2 shows an example scope hierarchy of a worldwide operating company that uses a publish/subscribe network to interconnect its event-driven applications. There are many ways to organize the company and its publish/subscribe infrastructure that, for example, could be structured according to the company's different divisions such as *finance*, *marketing* or *production*. Alternatively, the structure may correspond to world regions including *America*, *Asia*, and *Europe*, where company offices are located, or it may be divided into the different business areas such as the *automotive*, *chemical*, or *electrical* market in which the company is active. The scope hierarchy allows to flexibly combine these organization alternatives and to define dedicated top level scopes for *divisions*, *regions*, and *markets* each. Thereafter, each top level scope is independently subdivided and refined as described above. Application components and instances are then assigned to the appropriate leaf scopes. Here, application component C_1 is just member of the Europe region and its superscopes while components C_2 and C_3 are assigned to multiple scopes on different branches of the scope hierarchy. The arrows in the figure are used to determine the visibility of notifications to components as discussed in the following section.

4.2.3 Visibility

Within the same scope, application components can communicate freely with each other. In complex systems, however, there are usually components of different scopes which need to interact, too. This is only possible if their notifications are *visible* to each other.

Definition 18 (notification visibility). *Let C_1 and C_2 be two components. A notification n_1 published by C_1 is* visible *to C_2 if the following conditions hold:*

(*i*) *For all scopes S with $C_1 \in S \wedge C_2 \notin S$ is $\mathrm{out}_S(n_1) = \mathrm{true}$.*

(*ii*) *For all scopes T with $C_1 \notin T \wedge C_2 \in T$ is $\mathrm{in}_T(n_1) = \mathrm{true}$.*

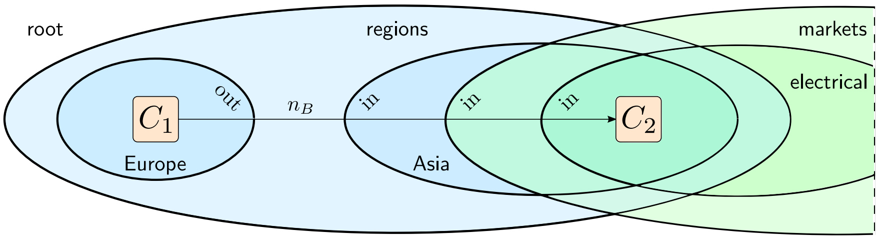


Figure 4.3: Set diagram identifying scope boundaries to cross.

The definition is straightforward and simply states that the appropriate in and out filters are applied when notifications cross scope boundaries. If, after all filtering, a notification is eventually delivered to the final subscriber then it is, thus, visible to the particular component. By using a set diagram as shown in Fig. 4.3, it is easy to graphically determine which scope boundaries need to be passed in which direction and, hence, which filters to apply. For this purpose, the diagram contains the notification's publishing component C_1 and its subscribing component C_2 together with their surrounding scopes known from the example scope hierarchy of the previous section. Drawing a straight line from the publisher C_1 to the subscriber C_2 identifies which scope boundaries the notification n_1 has to cross. Whenever a scope is left, the scope's out filter is applied and, likewise, whenever a new scope is entered, the corresponding in filter gets evaluated.

Alternatively, the scope hierarchy as shown in Fig. 4.2 can also be used to determine the notification's visibility. Publisher C_1 and subscriber C_2 have been connected to assigned leaf scopes. Furthermore, all paths are marked by an upwards arrow that lead from the publisher C_1 to the root scope. Additionally, all paths leading from the root scope to the subscriber C_2 are shown with a downwards arrow. Scopes that are part of an upwards and a downwards path include both components while those that are only on one type of path contain either C_1 or C_2. A notification to be delivered, thus, must leave scopes with just the publisher C_1, stay in scopes that include both components, and enter those that only contain the subscriber C_2. Hence, when leaving a scope the out filter and when entering a scope the in filter gets applied, respectively.

4.2.4 Attributes

Notifications describe occurred events and their context as perceived by the publishing component. Additionally, many publish/subscribe systems allow processing elements to attach further attributes to a notification that provide metadata or complement information not known to the publisher. For example, additional attributes may include timestamps, priorities, or system identifiers. Likewise, *scope attributes* may also be used to associate further information with a scope.

Definition 19 (scope attribute). *Let $S \in \mathcal{S}$ be a scope. A* scope attribute *$S.a$ characterizes a property of S by associating data under the name a with the actual scope instance S.*

Scope attributes may flexibly be used for different purposes. They help annotating scopes with arbitrary information, for example, to document a scope's creation, configuration, usage, or maintenance. These information may be human-readable or as well be intended to be automatically processed by the publish/subscribe system. Moreover, scope attributes may also carry application-specific data. As scopes bundle related applications and components, it is possible to factor out common properties and store their values as scope attributes only once. Attaching scope attributes to published notifications may also help to better specify the context in which the notification was produced. This is especially useful if the notification leaves the scope it was published in.

Assume a company using a publish/subscribe system to interconnect its production sites. Each production site has its own scope to ensure that application instances from different sites do not interfere with each other. Selected notifications, however, are allowed to leave the scopes in order to be centrally monitored, aggregated, and evaluated. Therefore, it is essential to know a notification's origin. This is easily accomplished by automatically attaching a scope attribute to the notification that identifies the publishing production site. Using this technique, it becomes possible to extend notifications with arbitrary information and especially with those information that were not available at the application's design time. Thereby, scope attributes provide an elegant way to incorporate deployment, configuration, and administration data and, thus, ease the engineering of event-based systems and applications.

4.2.5 Inheritance

Scope attributes characterize scopes as well as components and notifications therein. Subscopes may further partition a scope's components into more specific groups and, thus, need to define additional attributes or refine existing ones. Therefore, subscopes *inherit* the attributes of their parent scopes as well as their values by default, but do also have the possibility to override them or add new ones as needed.

Definition 20 (attribute inheritance). *Let S be a scope and $S.a = v$ an attribute of S with assigned value v. Furthermore, let $T \subseteq S$ be a subscope of S. Then attributes of S are* inherited *as follows:*

(i) Subscope T automatically possesses an inherited attribute $T.a = v$ by default requiring no explicit declaration or assignment.

(ii) Subscope T may override the value of inherited attributes by explicitly setting a new value $T.a = w$.

(*iii*) *Subscope T may also declare additional attributes $T.b = x$ with arbitrary assigned values x.*

Part (i) of the definition states that an attribute declared in a superscope remains an attribute in all derived subscopes and gets the declared value assigned by default. Statements (ii) and (iii), however, allow developers and administrators to override the inherited attribute value or to extend the set of attributes to better characterize the derived subscope.

For example, consider an international company whose offices and production sites are grouped into world regions such as America, Asia, and Europe. The company's publish/subscribe infrastructure may be organized similarly with each region scope declaring an attribute *location* that is set appropriately. When subdividing a region scope into different country scopes to form smaller groups, it makes sense to refine the location attribute. In case of Europe, the region scope could be divided into Great Britain, Denmark, and Germany with locations set to London, Copenhagen, and Berlin, respectively, assuming that the company's offices are situated in the countries' capitals. Additional attributes may also be declared such as *currency* which is set to pound (GBP), krone (DKK), and euro (EUR), respectively, and may help interpreting price information.

Although very useful to better characterize and refine a subscope, overriding inherited attributes may also lead to subtle ambiguities. This is due to the possibility that according to point (v) of definition 16, components may be members of several scopes at the same time. So let us assume that the offices in Copenhagen and Berlin from the example above use the same application to automatically process prices and, thus, the application's components are members of the Danish as well as the German scope. Hence, the question arises which currency, kroner or euros, now applies to these components. The best answer is both since the application has to deal with Danish as well as German prices. Therefore, we support such ambiguities by enabling the assignment of *multiple values* to attributes in order to characterize components.

Definition 21 (multi value attribute). *Let S_i, $1 \leq i \leq n$, each be a leaf scope with an attribute $S_i.a = v_i$ and component $C \in S_i$ as member. The* multi value attribute *$C.a$ is derived from the scope attributes $S_i.a$ by combining their values to the set $\{v_1, \ldots, v_n\}$.*

Component metadata can be augmented by deriving component attributes from scopes. Thereby, sets of values may get assigned to single attributes. It is, however, left open to application developers and system administrators to interpret these sets and to leverage and exploit them to their advantage.

4.2.6 Mappings

Complex event-based systems usually comprise numerous applications and components of which many may feature inherently different event models, representations, or semantics. System integration has to deal with this heterogeneity and, thus, is often a cumbersome and tedious task. Scoping supports system integration by enabling administrators to bundle those components that share common models and semantics. While these may interact seamlessly, however, certain interoperation with subsystems is usually required that use different event representations. For this purpose, it is possible to *transform* notifications when they enter or leave a scope.

Definition 22 (notification mapping). *A* notification mapping *is a function* $\mathrm{map} : \mathcal{N} \to \mathcal{N}$ *on the set* $\mathcal{N}$ *of event notifications that transforms a notification* $n \in \mathcal{N}$ *into a possibly different notification* $m \in \mathcal{N}$*.*

Please note that notification mappings subsume the filtering of notifications at scope boundaries, too. This is the case when using the identity function for all notifications that are allowed to pass and mapping the remaining ones to the empty notification ϵ to be filtered out. Thus, Def. 14 specifying scope interfaces can be generalized by replacing the two Boolean filter functions by notification mappings to determine not only which notifications may enter or leave but also how they are represented inside and outside the scope.

Notification mappings are an important design concept as they allow to centrally specify how to convert a notification from one data model into another. Even if data conversions are not necessary, notification mappings may, nevertheless, help to adjust and fine-tune the (meta) information a notification is carrying by replacing or removing some of its attributes. Within a subsystem, for example, it may be of concern to identify the concrete component that published a given notification, while in the remaining system, it may be just sufficient to know the subsystem from which the notification originated. A notification mapping can, thus, change the source attribute as required when the notification leaves the subsystem scope. Depending on whether sophisticated model conversions or simple attribute adjustments are conducted, the complexity to implement notification mappings in publish/subscribe systems differs significantly. Thus, different implementation approaches are possible that have individual strengths and weaknesses.

4.3 Routing

There are several ways to integrate visibility and scope concepts in event-based systems and applications [68]. Some of them simply add scoping as an additional layer on top of a publish/subscribe infrastructure. Thereby, either publishing components have to explicitly restrict the set of notification receivers or consuming components are required to filter out non-visible notifications before these are passed to applications. Alternatively, when embedding scoping directly into publish/subscribe routing layer, more efficient solutions are feasible as described in this section.

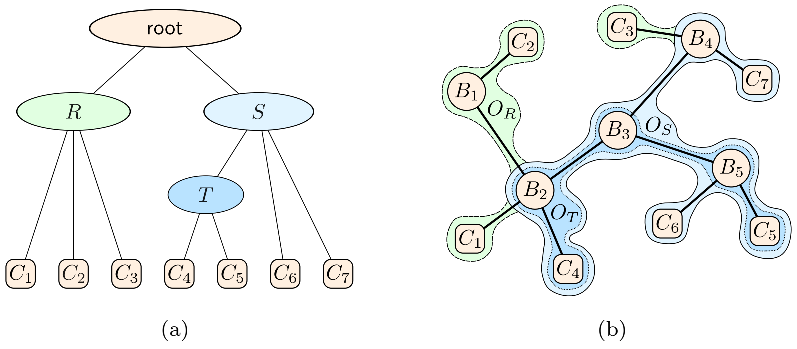


Figure 4.4: Scope overlays in a publish/subscribe system: (a) given scope hierarchy; (b) overlays formed within the broker network.

4.3.1 Scope Overlays

Considering scopes within the publish/subscribe routing layer leads to a differentiation of the network. Distinct scopes may be active in different parts of the network and, thus, form so called *scope overlays.*

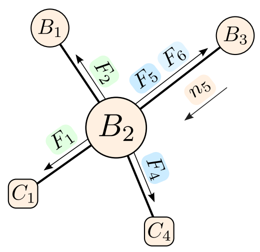
Definition 23 (scope overlay). *The* scope overlay O_S *of scope* S *comprises the minimal subset of brokers in the publish/subscribe network that is required to interconnect all components belonging to* S*.*

Figure 4.4(a) illustrates an example scope hierarchy comprising the top level scopes R and S, the subscope T, and seven components $C_1, \ldots, C_7$ assigned to them. When the components are connected to the publish/subscribe brokers $B_1, \ldots, B_5$, as shown in Fig. 4.4(b), scope overlays are formed within the broker network. Thereby, the shape and size of an overlay is determined by the position of the brokers that host the components belonging to the corresponding scope. Thus, the overlay of scope S contains the brokers B_2, B_4, and B_5 that host components of S or its subscope T as well as the broker B_3 that is needed to interconnect the others. Furthermore, an overlay of a superscope always encloses the overlays of its subscopes as it is the case for O_S and O_T. However, the overlays of subscope siblings or of scopes from different branches of the hierarchy may intersect and overlie each other as, for example, O_R and O_S do. Please note that since all components are members of the root scope by definition, its overlay conforms to the complete publish/subscribe network and is, thus, not emphasized in the figure. In fact, the root scope corresponds to a conventional publish/subscribe infrastructure without scopes.

As brokers usually host components of several scopes (e.g., B_2 hosts C_1 from R as well as C_4 from T) they, thus, have to manage and keep track of visibility

F_1	$\{R\}$	C_1
F_2	$\{R\}$	B_1
F_4	$\{S,T\}$	C_4
F_5	$\{S,T\}$	B_3
F_6	$\{S\}$	B_3

(a)



(b)

Figure 4.5: Routing with scopes: (a) extended routing table of broker B_2; (b) network environment of broker B_2 with routing entries.

constraints. Therefore, they first need to know from which scopes a message originates. Thus, all messages—notifications, subscriptions, and advertisements if supported—are tagged with the appropriate set of scopes they belong to. Furthermore, the brokers additionally need to know to which scopes a message is allowed to be forwarded. Hence, it is also necessary to *extend* a broker's routing table to store additional scope information.

Definition 24 (extended routing entry). *An* extended routing entry *is a 3-tuple consisting of a filter F, a scope set $\mathcal{S}$, and a destination which is either a component C or a neighboring broker B.*

In a subscription table, a filter expression describes the notification's content in which a subscriber is interested. In an advertisement table, a filter specifies the notifications an event producer is going to publish. The scope set contains the scopes the subscribing or publishing component is associated with and, thus, the scope combination for which the filter is active. The destination determines the component itself or the next hop in the broker network to which matching notifications or overlapping subscriptions need to be forwarded.

Figure 4.5 gives an example. Subfigure (b) shows the network environment of broker B_2 from the previous illustration in more detail including its connections to neighboring brokers and hosted components. Subfigure (a) depicts the broker's extended routing table with additional scope information. The first entry, for instance, represents an active subscription from component C_1. As C_1 belongs to scope R the subscription's filter F_1 is only active in the corresponding overlay. The entry, thus, ensures that notifications that are visible in scope R and match the filter F_1 are forwarded towards component C_1 which is indicated by a corresponding arrow in (b). Please note that since the root scope is always contained in a valid scope set, it is not explicitly denoted within an extended routing entry in the table.

4.3.2 Forwarding

With scopes, it becomes more difficult for brokers to make routing decisions and forward notifications. Particularly, this is the case if subscription and notification are from different scopes. Considering scope interfaces, the notification may, nevertheless, be visible in the subscription's scope and, thus, has to be forwarded if it matches the subscription's filter stored in the routing table. The usage of advertisements or the application of advanced routing schemes such as identity- or covering-based routing algorithms further increase the complexity as filter expressions from potentially different scopes need to be compared to determine and exploit their similarities [146]. Matching notifications as well as comparing filters works best when done in a *set of common superscopes.*

Definition 25 (greatest common superscope set). *Let $\mathcal{S}$ and $\mathcal{T}$ be two valid scope sets associated with a notification or a filter. The* greatest common superscope set *(GCSS) of $\mathcal{S}$ and $\mathcal{T}$ is given as $\mathcal{S} \cap \mathcal{T}$.*

Considering two components within a scope hierarchy, the greatest common superscope set (GCSS) represents the part of the hierarchy that is shared by both components. This is observable when comparing the scope sets of the routing entries for filters F_1, F_4, and F_6 in Fig. 4.5(a) with the position of the components C_1, C_4, and C_6 in the scope hierarchy in Fig. 4.4(a), respectively, which issued the corresponding subscriptions. Since C_1 and C_4 only share the root scope, the GCSS of their subscriptions is $\{R\} \cap \{S, T\} = \emptyset$, while the GCSS for subscriptions of C_4 and C_6 is $\{S, T\} \cap \{S\} = \{S\}$ as S is superscope of T. If issued notifications, subscriptions, and advertisements are visible and active in these scope sets, forwarding decisions can be made there. But for this purpose, notifications and filters must first be stepwise brought *upwards* the scope hierarchy into the GCSS for what they need to pass scope interfaces and may get transformed.

Definition 26 (upward transformation). *Let $\mathcal{S}$ be a scope set and $S \in \mathcal{S}$ be a scope having no further subscope in $\mathcal{S}$. When raising a message from $\mathcal{S}$ to the superscope set $\mathcal{S} \setminus \{S\}$ the following* upward transformation *is applied:*

(*i*) *A notification n stays unchanged if* $\mathrm{out}_S(n) = true$*, otherwise it is replaced by the empty message ϵ.*

(*ii*) *The filter F of a subscription s is narrowed to $F' = F \wedge \mathrm{in}_S$. If $F' =$* false *for all notifications, s is replaced by the empty message ϵ.*

(*iii*) *The filter G of an advertisement a is narrowed to $G' = G \wedge \mathrm{out}_S$. If $G' =$* false *for all notifications, a is replaced by the empty message ϵ.*

Notifications that are not visible in the common superscope set become the empty notification and need not to be handled. Subscription and advertisement filters are combined with the interface filters of the subscopes that are not contained in the superscope set. Their filter expressions are combined to a new conjunctive filter that narrows the set of matching notifications as both original filter constraints and imposed interface constraints need to be met. For subscriptions, the interface's in constraints are used, while the out constraints are applied on advertisements. If the resulting filter expression becomes unsatisfiable (i.e., the constraints are contradictory) it needs not to be handled anymore.

Figure 4.5 exemplifies the forwarding process. The notification n_6 is received by broker B_2 and needs to be forwarded according to its routing table. Since n_6 was relayed from B_3, only the first three routing entries for broker B_1 and components C_1 and C_4 need to be considered as eligible forwarding destinations. Furthermore, assuming that n_6 was originally published by component C_6, cf. Fig. 4.4, it carries the associated scope set $\{S\}$. As the scope sets of notification and eligible routing entries differ, an upward transformation to the respective GCSS needs to be carried out before the notification can be matched against an entry's filter. Considering the third entry for destination C_4, the $\mathrm{gcss}(\{S,T\},\{S\})$ is $\{S\}$ and, thus, only the entry's filter F_4 needs to be transformed. As it is a subscription filter it is narrowed to $F_4 \wedge \mathrm{in}_T$ against which n_6 is matched afterwards. Considering the first two routing entries for destinations C_1 and B_1, however, the $\mathrm{gcss}(\{R\},\{S\})$ is just the root scope. Therefore, both the notification as well as the subscription filters need to be transformed. First, the notification is tested against the out filter of scope S and only if $\mathrm{out}_S(n_6)$ succeeds, n_6 is matched against the narrowed subscription filters $F_1 \wedge \mathrm{in}_R$ and $F_2 \wedge \mathrm{in}_R$, respectively. Finally, if n_6 matches a narrowed subscription filter it is forwarded to the corresponding destination.

Please note that every broker first performs upward transformations to the GCSS for notifications and filters of different scopes. Subsequently, arbitrary matching and routing algorithms can be applied without modifications to test notifications and compare filters. In particular, further changes to support scoping are not necessary. Transformations of notifications and filters are usually temporary to just make forwarding and routing decisions. Thereafter, the original messages with their associated scope sets are forwarded to components and neighboring brokers. However, if a message is definitively leaving the scope overlay of one of the scopes it is associated with, i.e., it reaches network regions where a particular scope overlay is not existend anymore, upward transformations may become permanent for notifications and especially for filters that got narrowed. Furthermore, the respective scope is removed from the scope set the message is carrying. This is necessary as brokers outside the overlay of a particular scope may not know the scope's definition and interface and may, thus, not be able to perform an upward transformation to the GCSS for forwarding and matching. In the example of Fig. 4.5 and Fig. 4.4, this is the case if notification n_6 is forwarded by broker B_2 to broker B_1 or component C_1 whereby the notification finally leaves the overlay of scope S. As S gets removed from the set of associated scopes, the remaining scope set will only consist of the root scope.

4.4 Management

Besides considering scope boundaries when routing event notifications, the integration of scopes into event-based systems and applications also requires comprehensive means for their management. In particular, mechanisms must be provided to create and open new scopes as well as to close unneeded ones, to assign scopes to application components, and to allow components and brokers join and leave a scope.

4.4.1 Scope Components

To facilitate a modular design each scope is managed by a dedicated component. *Scope components* are connected to the publish/subscribe system similar to publishers or subscribers. In fact, any event-driven component may create new scopes and manage them if it complies to the following definition.

Definition 27 (scope component). *The* scope component C_S *managing the scope S is a regular event-driven component C that additionally fulfills the following requirements:*

(*i*) *C_S provides the specification of scope S including the scope's name, its component selector and interface as well as optional scope attributes and mappings.*

(*ii*) *C_S is a registered member of the parent scope of S and maintains a directory by itself for scope components of direct subscopes to register.*

(*iii*) *C_S provides access control by determining which components and brokers are allowed to join the scope.*

To instantiate a new scope it is first necessary, as stated in (i), to provide the scope's specification. Furthermore, as scope components are regular components they are scope members, too. Thus, a scope component must join the parent scope, as (ii) requires, in order to get its scope established. At the same time, the scope component itself must allow others to create and register further subscopes. However, statement (iii) grants the scope component the possibility to control which other components are allowed to join its scope. Thereby, the component can manage the membership of its scope and, thus, enforce access control restrictions.

Letting regular components manage their own scopes has several advantages. First, it allows for reusing the functionality already present in publish/subscribe systems without the need to provide completely new broker interfaces. Second, it relieves brokers by partly shifting the efforts of scope management to the components. In fact, brokers provide just the mechanisms for scoping while the components implement actual policies, e.g., the brokers extend a scope's overlay so that it includes new members but only the managing components decide whom access is granted. Third, this separation of policies from mechanisms provides more flexibility and allows for a tighter integration of scope concepts in event-driven applications. The following sections discuss the mechanisms how scope managing components and brokers together implement the assignment of scopes as well as management of scope overlays within publish/subscribe systems.

4.4.2 Scope Assignment

Grouping components into scopes helps organizing publish/subscribe systems and event-based applications, whereby different requirements and criteria usually need to be considered. Application developers, for example, may want to ensure at design time that all components of a distributed application are assembled in a common scope while administrators may want to customize scope assignments at deployment or runtime to reflect the company's organization structure and ensure information policies. Thus, a flexible and scalable *scope assignment* approach is needed that meets the different requirements and is able to cope with large numbers of scopes and components.

Definition 28 (scope assignment). *Given an inactive event-driven component that gets connected to a broker. The component is activated and scopes are assigned to it as follows:*

(*i*) *Each broker has a set of default scopes $\mathcal{B}$ configured by its administrator. These scopes are imposed on the component when connecting to the broker.*

(*ii*) *Each component has a set of default scopes $\mathcal{C}$ specified by its application developer. Joining these scopes is explicitly requested by the component when connecting to the broker.*

(*iii*) *The set $\mathcal{M}$ of mandatory scopes is given by the union of $\mathcal{B}$ and $\mathcal{C}$ together with all direct and indirect superscopes of these.*

(*iv*) *The set $\mathcal{O}$ of optional scopes is determined by all direct and indirect subscopes of $\mathcal{B}$ and $\mathcal{C}$.*

(*v*) *A scope S from the scope sets defined above is* assignable *to the component if the scope's component selector* sel_S *evaluates to* true.

(*vi*) *All assignable scopes from the mandatory and optional scope sets $\mathcal{M}$ and $\mathcal{O}$, respectively, are applied on the component.*

(*vii*) *The component is activated if all mandatory scopes $\mathcal{M}$ have been successfully applied, otherwise the component stays inactive.*

The scope assignment as described above is an adaptable and customizable process as it allows to consider aspects of application design as well as system

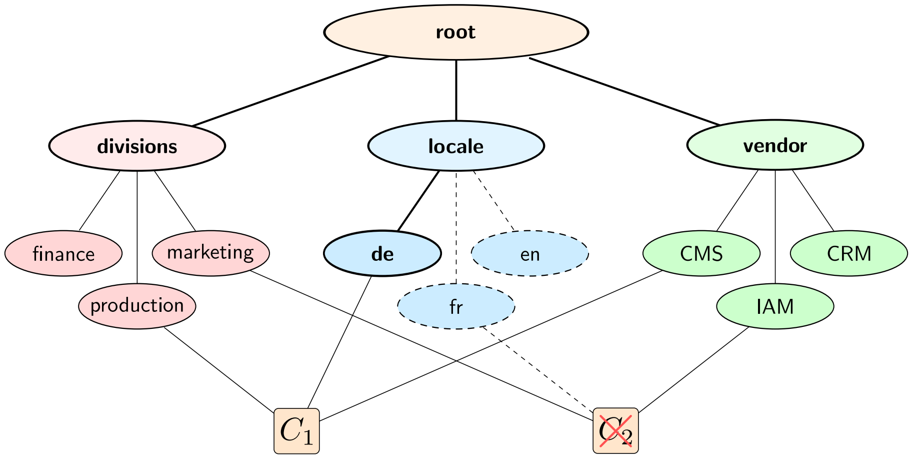


Figure 4.6: Scope assignment process.

administration. Both application developers and system administrators may specify mandatory scopes ensuring that their design criteria and organizational requirements are incorporated. However, instead of enumerating all required scopes it is sufficient to only name mandatory top level scopes. Subscopes are dynamically tested using their component selectors whether they are assignable and, if so, all assignable subscopes get automatically applied on the component. Thereby, the assignment process stays extendable and maintainable as it is later possible to refine the scope hierarchy at any time, whereby all changes can be conducted without the need to modify any application component.

Figure 4.6 shows an example scope hierarchy of an international company to illustrate the assignment process and demonstrate its flexibility and advantages. The hierarchy consists of scopes defined by the company's administrators to organize the publish/subscribe infrastructure as well as scopes specified by the developers of the software vendor whose applications the company uses to monitor and control its business workflows. Under the top level scope *company*, the system administrators have created an own subscope for the divisions *finance*, *marketing*, and *production* to reflect the company's organizational structure. Furthermore, the top level scope *locale* contains subscopes for the different regional conventions and settings that are employed at the company's international offices. Thus, there are subscopes *de*, *en*, and *fr* for the German, English, and French conventions and settings, respectively. According to step (i) in the assignment process, each broker of the company's publish/subscribe system has a set of configured default scopes $\mathcal{B}$ that are applied on all connected components. In this example, the default scope set consists of the top level scope *company* together with the locale's subscope *de* when assuming that the particular broker is situated in Germany. According to step (ii), connected application components may also request a set $\mathcal{C}$ of default scopes. Therefore, the application's vendor provides the top level scope *vendor* and subscopes for each application and software product it is selling. The company uses a content management system (*CMS*), an application for customer relationship management (*CRM*), and software for identity and access management (*IAM*) offered by the vendor. However, independent of which product an application component belongs to, the application developers configured it to just request the *vendor* top level scope in this example.

Based on the assumptions and configurations above, the mandatory scopes $\mathcal{M}$ and the optional scopes $\mathcal{O}$ can be determined according to step (iii) and (iv) of the definition, respectively. In the figure, mandatory scopes are printed with bold font and borders while optional scopes have regular borders. Broker default scopes $\mathcal{B}$ and component default scopes $\mathcal{C}$ together with their superscopes are mandatory so that, in this case, $\mathcal{M}$ consists of the scopes *company*, *de* and its superscope *locale*, and *vendor*. All subscopes of the default scopes are optional so that $\mathcal{O}$ contains the company divisions *finance*, *marketing*, and *production*, as well as the vendor's products *CMS*, *CMR*, and *IAM*. The locales *en* and *fr*, however, are neither mandatory nor optional as they are just siblings to the broker default scope *de*. Thus, they are printed with dashed borders in the figure. Step (v) and (vi) test mandatory and optional scopes whether they are assignable. The scope's selector is used to evaluate a component's attributes and, if successful, the scope is applied. Thereby, it is ensured that a component is only grouped into those scopes whose specifications it fulfills. In the example, the scopes *production*, *de*, *CMS*, and their superscopes are, thus, assigned to component C_1 while component C_2 is grouped into the scopes *marketing*, *IAM*, and respective superscopes.

After all assignable scopes are applied, a component is only activated in step (vii) if all mandatory scopes are covered. In the figure, this is the case for component C_1 while component C_2, however, stays inactive and is crossed out because the German locale *de* is missing. Although C_2 may comply to the French locale *fr*, as depicted by the dashed line, the component is not activated as the German locale is explicitly configured as a mandatory default scope of the broker. In fact, the system administrators, thereby, enforce that this particular broker may only host components that comply to the German locale. Regarding company divisions and vendor products administrators and developers, respectively, use a different approach. Here, just the top level scopes are are configured to be mandatory defaults so that grouping the components into subscopes is solely based on the scopes' selector functions and the components' attributes. This ensures the flexibility to adapt, refine, and change subscopes as needed at any time without modifying components or adapt broker configurations.

4.4.3 Joining Scopes

Implementing the scope assignment procedure as described in the last subsection is challenging in a distributed publish/subscribe system. Scope components may be scattered over the network and their scope specifications may, thus, not be directly available at the broker a new component is connected to. Moreover, scope overlays may need to be expanded to cover new components and be shrunk if the components are removed again. However, much of the functionality inherent to publish/subscribe systems can be reused for this purpose. Advertisements, for example, announce new publishers in the network and, likewise, they can also inform brokers about available scopes. Such advertisements are called *scope advertisements.*

Definition 29 (scope advertisement). *The* scope advertisement a_S *of scope* S *announces the publication of scope control messages for* S*, contains the scope's specification as well as its component selector, and is visible within the superscope of* S*.*

Figure 4.7 exemplifies the process of creating and joining scopes. After connecting to broker B_5 and joining scope R, component C_S creates and opens a new subscope S which it manages. Thus, only broker B_5 and component C_S are members of the new scope in the beginning. This subscope is advertised in Fig. 4.7(a) in order to let other brokers and components know about its existence. Therefore, component C_S creates a scope advertisement a_S containing scope specification and component selector. Advertisement a_S is subsequently disseminated within its superscope R and, thus, also reaches brokers B_1, B_2, and B_3 as well as connected components. However, it is not forwarded towards broker B_4 as B_4 is not a member of the parent scope R. Brokers store the scope advertisement in their routing tables and create an advertisement entry A_S pointing into the direction of scope S's managing component. Moreover, by applying S's component selector on the attributes of newly connected components, a broker can, thus, determine whether a component fits in the scope. If so, the broker must join the scope's overlay O_S and become a member unless it already is. Using subscriptions components and brokers usually express the interest in certain notifications and, likewise, corresponding *scope subscriptions* may also serve as a request to join the scope.

Definition 30 (scope subscription). *The* scope subscription s_S *requests the membership of scope* S*, contains the identities of the requesting components and brokers, and is visible in* S *and the superscope of* S*.*

In Fig. 4.7(b) the component C_1 wants to join the new subscope S and, therefore, issues a corresponding scope subscription s_S. Similarly, a broker may also issue a scope subscription for any hosted component. As scope subscriptions are visible in their enclosing superscope, they can be compared to available scope advertisements and routed appropriately. Thus, brokers B_2, B_3, and B_5 forward subscription s_S along the reverse path of the corresponding scope advertisement towards scope S's managing component. Each broker stores the subscription's filter F_S in its routing table and creates a routing entry pointing into the di-

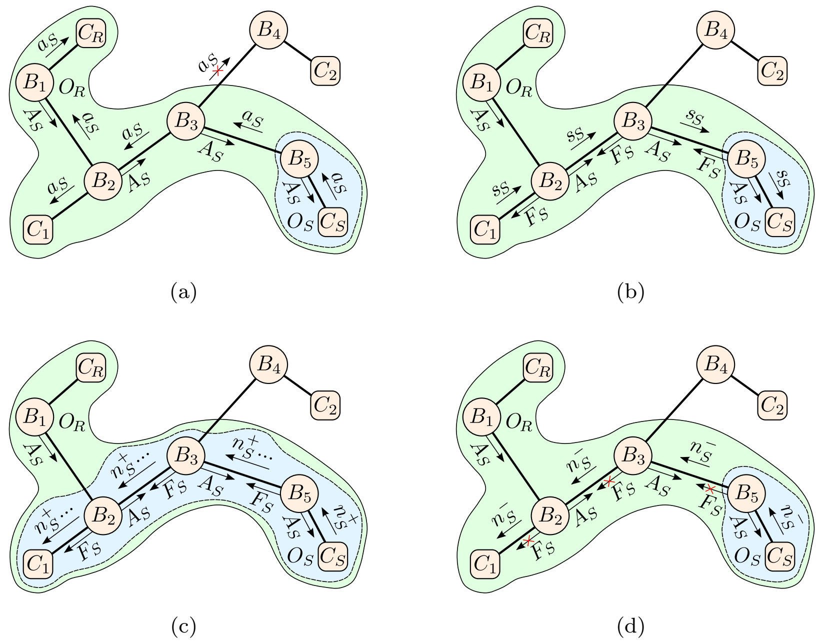


Figure 4.7: Creating and joining scopes: (a) C_S advertises its new subscope T; (b) C_1 subscribes scope T to join; (c) C_S approves the join request; (d) C_S denys the join request.

rection of the requesting component. Moreover, along its way, the identities of passed brokers are added to the subscription that are not members of the scope yet. When arriving at managing component C_S, scope subscription s_S, thus, carries the extended request to also admit brokers B_2 and B_3 to the scope in order to join component C_1. Based on these information, managing component C_S then decides whether the request is approved and the scope's overlay O_S is expanded. If so, an approval is published, otherwise a denial.

Please note that several advanced routing algorithms such as identity-based routing or covering-based routing do not require all subscriptions to be necessarily forwarded to the advertising component if filter similarities can already be exploited and leveraged along the path. In this case, an identical or covered scope subscription would not reach the managing component anymore which, thus, cannot admit the requesting component or broker to the scope. Nevertheless, there are two different solutions. The first and simpler one is to make scope subscriptions unique so that they are not identical to or covering each other. Since there are no similarities to exploit anymore, all scope subscriptions would reach the managing component again. The price to pay, however, are larger routing tables and a decreased efficiency as advanced routing algorithms degrade to the basic simple routing strategy.

The second, more efficient solution is to distribute the admission of new scope members and let brokers assist in the process. Therefore, the scope managing component defines and publishes requirements and policies for new members to join the scope that may, for example, require new components to provide particular certificates or new brokers to be situated in certain network segments. Based on these policies brokers that already are scope members may then admit new members to the scope on behalf of the managing component. Although advanced routing algorithms are able to exploit their optimizations and realize their full potential, there are other drawbacks regarding this approach. Primarily, brokers must be able as well as trusted to strictly follow specified policies for joining new components and brokers while, furthermore, the responsible scope component looses control and flexibility about the scope's management. Nevertheless which approach is employed, a scope subscription requesting to join a scope is, in both cases, replied with a *scope notification* published by either the managing scope component or a broker on behalf of the scope component that approves or denies the request.

Definition 31 (scope notification). *A* scope notification n_S *contains a control message for members of scope S supporting the management of its overlay. A scope notification may especially approve (denoted as n_S^+) or deny (denoted as n_S^-) the scope membership of requesting components and brokers.*

Published scope notifications follow the reverse path of matching scope subscriptions back to the members of the scope overlay. In case of a membership approval it is forwarded towards the requesting component or broker expanding the overlay along its way. Furthermore, all subscriptions and advertisements that now became visible to new scope members are also forwarded along the path. Figure 4.7(c) shows the admission of requesting component C_1 to scope S. The managing component C_S responsible for S publishes the scope notification n_S^+ in order to approve the join request of C_1. The approval n_S^+ is forwarded according to the brokers' routing entries F_S established by the preceding scope subscription. Thus, it follows the reverse path of the scope subscription over brokers B_5, B_3 and B_2 towards the component C_1 and successively expands scope S's overlay along the way. As a consequence, advertisements and subscriptions belonging to scope S are now visible to the new members or, at least, they are not narrowed by the scope's interface filters anymore. Thus, they are also forwarded to update the respective routing tables of the brokers along the path. This is depicted by the scope notification n_S^+ followed by dots indicating that other control messages such as newly visible advertisements and subscriptions are sent immediately afterwards.

In case of a denial of the scope membership, a scope notification is also routed towards the requesting component or broker and is informing about the rejection. Additionally, the unsuccessful scope subscription needs to be removed from the routing tables of the brokers along the path. This can be accomplished by letting the requesting component or broker simply unsubscribe the scope after it received the denial. Using this approach, only regular publish/subscribe functionality is exploited, but another unsubscription message is necessary and needs to be routed towards the scope managing component to cancel the superfluous routing entries. Alternatively and more efficiently, the scope notification itself may immediately remove the routing entries. This, however, requires to process a negative scope notification differently to regular event notifications as a broker's routing table is also affected and modified. Figure 4.7(d) illustrates the second approach. Managing component C_S responsible for scope S publishes the scope notification n_S^- in order to reject the join request of component C_1. The denial n_S^- is routed towards the requesting component C_1 and removes the corresponding scope subscription filters F_S from the routing tables of brokers B_5, B_3, and B_2 along the path.

4.4.4 Leaving Scopes

When application components are removed and disconnected from brokers they automatically leave the scopes they are members of. This is done by simply *unsubscribing* the scopes in hierarchical order starting with the most specific subscopes. If no scope subscription for a particular scope is left in a brokers routing table, i.e., there is no other application component or neighboring broker requiring the scope anymore, then the broker can also leave the scope. As consequence, it will no longer receive or forward notifications published in this scope and, hence, it can remove their routing entries that may still exist in its routing table. However, there is one exception. If notifications comply to the scope's interface, they are allowed to pass the scope's boundary and are also visible outside. Likewise, subscriptions and advertisements are combined with the scope's interface filters and may be relevant outside, too. Therefore, the broker has to conduct an upward transformation (cf. Def. 26) of every extended advertisement and subscription routing entry whose scope set contains the scope to leave. Thereby, the entry's filter gets narrowed by the scope's in and out interface, respectively. Only if the narrowed filter becomes unsatisfiable, the entry is safely removed.

When scope components are removed and disconnected from brokers, their provided and managed scopes get closed. This is done by *revoking* the scope's advertisement. Brokers receiving a scope unadvertisement process it like any other regular unadvertisement, remove the routing entry pointing to the scope managing component, and delete corresponding scope subscriptions if no other scope component has advertised to also manage the scope. Thereby, application components are forced to leave the scope and its subscopes which is accomplished as described in the paragraph above. Especially, an upward transformation is conducted for every advertisement and subscription issued within the scope to leave. Moreover, in case of a mandatory scope, the application component is deactivated with all remaining advertisements and subscriptions being revoked. Otherwise, the application component may continue, but has to renew its subscriptions and advertisements as these are not restricted by filters of the scope interface anymore. However, it is not recommended to close and unadvertise a scope if there are still application components active inside as this may lead to race conditions. Notifications published inside the scope may still be in transit within the broker network while subscribed components are already forced to leave the scope and, thus, do not receive them anymore. To avoid this, a scope managing component may publish a scope notification in advance that is delivered to all scope members in order to announce the closing of the scope. Thereby, application components get the opportunity to prepare and execute precautionary actions.

4.5 Implementation

Scoping has been implemented and integrated into our publish/subscribe middleware REBECA. This section discusses important considerations and implementation aspects. Its purpose is twofold. First, we proof the feasibility of efficiently embedding scoping inside the middleware layer. While previous sections lay the conceptual foundations of scopes and describe their integration and management in general, here, we provide further insights into middleware interdependencies, highlight several implementation details, and open the possibility to a quantitative evaluation. Second, by implementing scoping as a pluggable module, we demonstrate the advantages and benefits of REBECA's plugin architecture. The emphasis on flexibility and modularity notably eases the whole integration process. In the following, we give an overview about the scope plugin's structure, explain necessary extensions to REBECA's filter framework, and elaborate on handling of application and scope components.

4.5.1 Scope Plugin

A REBECA plugin is usually self-contained and primarily consists of two parts: i) the *plugin engine* providing the main processing logic and ii) the *plugin sinks* allowing to modify or transform event messages before they are sent or after they are received. For each plugin, there is only one global engine in the broker's main processing stage, whereas a sink instance is created for every connection the broker maintains to a component or another broker. Additionally, the scope plugin also depends on employed routing algorithms and their respective plugins as it needs access to advertisement and subscription tables in order to modify stored entries and filters if the scope membership changes. Figure 4.8 shows a UML class diagram visualizing the plugin structure and its dependencies.

ScopeEngine and ScopeSink inherit from their respective base classes all necessary properties and operations to be seamlessly plugged into a broker's processing infrastructure. Each broker engine offers a general process method for handling event notifications and keeps a reference to the next engine in a broker's engine chain to let the following plugin proceed the processing. Each broker sink may

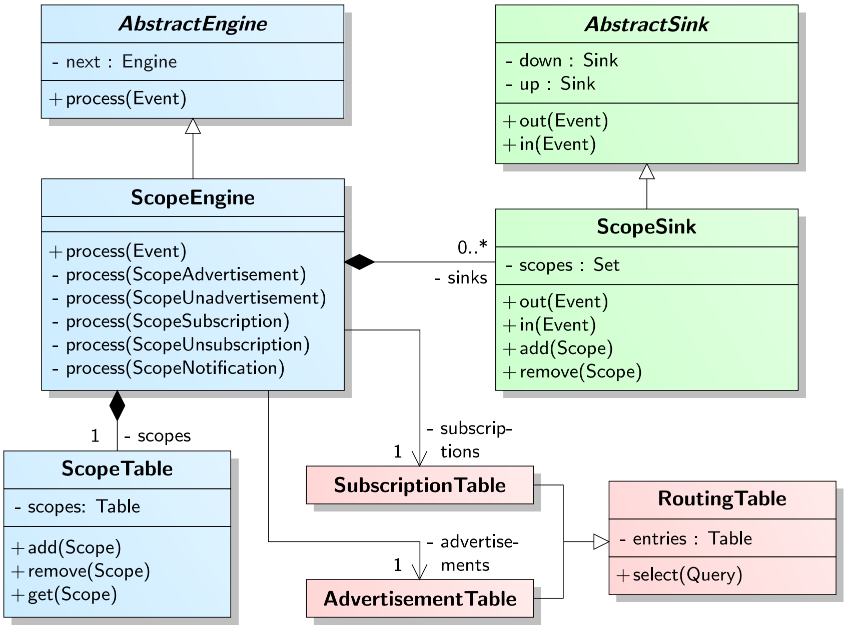


Figure 4.8: Overview about REBECA's scope plugin.

be passed by event messages in two opposite directions: *downwards* after being processed and to finally get transmitted to a neighbor broker or *upwards* after being received from a neighbor to get finally processed. The sink, therefore, provides the methods out and in, respectively, to transform or modify event notifications on their way, if needed. References down and up point to the next lower and upper sink in the sink chain, respectively, that the event message is going to pass next.

Regarding scope functionality the ScopeEngine features dedicated private processing methods for scope control messages (i.e., scope advertisements, scope subscriptions, and scope notifications) introduced in Sects. 4.4.3 and 4.4.4 to manage scope membership and scope overlays within the broker network. When handling a ScopeAdvertisement, the scope specification contained in the advertisement is extracted and stored in the scope table for later lookup and usage. Forwarding the advertisement, however, is done by the advertisement engine as part of one of the broker's routing plugins. When processing ScopeSubscriptions, the broker's identity and credentials are added to the subscription for joining the scope. Again, forwarding the subscription is actually carried out by the responsible routing plugin. ScopeNotifications may contain approvals or denials for a scope membership. In case of an approval, the connection sinks and their scope sets are updated to reflect the now extended scope overlay. In case of a denial, the corresponding scope subscription received by the join request is removed from the subscription table. In both cases, the scope notification is, thereafter, forwarded towards the requesting component.

ScopeUnadvertisements and ScopeUnsubscriptions are also handled by dedicated variants of the engine's overloaded process method. Although message forwarding is conducted by the routing plugins as well as maintaining the advertisement and subscription table in general, the ScopeEngine still needs to update the scope and overlay membership and conduct an upward transformation of stored filter entries if a scope is left. For the latter, the routing tables (i.e., the SubscriptionTable and the AdvertisementTable) provide a select method allowing to query for specific entries, for example, for advertisement or subscription filters issued in a particular scope. By iterating through the returned set of routing entries, the table can, thus, be updated by narrowing affected filters one by one.

Supporting scope management, the ScopeSinks maintain and administer the broker's membership in the overlay networks for subscribed scopes within the publish/subscribe infrastructure. Therefore, each ScopeSink manages a set of scopes containing only those scopes whose overlay network includes the connection the sink belongs to. The union of the scope sets of all connections a broker maintains determines the overall set of scopes in which the broker is a member and is able to receive corresponding notifications. To expand or to diminish a scope overlay the ScopeSink provides the methods add and remove that extend or shrink, respectively, the overlay of a particular scope along the edge within the broker network that corresponds to sink's connection. Additionally, the ScopeSink is responsible to enforce visibility constrains. The overridden in and out methods of the sink ensure that only those notifications and control messages are forwarded along the sink's connection that are visible in scope overlays the connection belongs to. Therefore, filters may be narrowed, notifications may be transformed, or messages may even dropped if necessary.

4.5.2 Scoped Filters

In order to embed scoping directly into the publish/subscribe infrastructure, we require, as described in Sect. 4.3.2, that both event notifications and filters are brought to the GCSS before any forwarding or routing decision is made. Therefore, we additionally demand that all messages are correctly tagged with the scopes they belong to and that routing entries are extended to store scope information along with filter expressions. As side effect of REBECA's functional modularity, the forwarding and routing logic as well as the management code for subscription and advertisement tables are scattered over several classes in different plugins. Adapting these to support scoping is, hence, costly, tedious, and time-consuming. However, REBECA provides an extendable *filter framework* that is easy to customize and, thus, more promising to integrate and implement scoping.

REBECA's filter framework has been designed with extensibility and customizability in mind. It eases the task for developers to subsequently add new filter classes for novel event notifications or to supplement existing filters with tailored versions that, for example, allow to better specify certain event notifications or

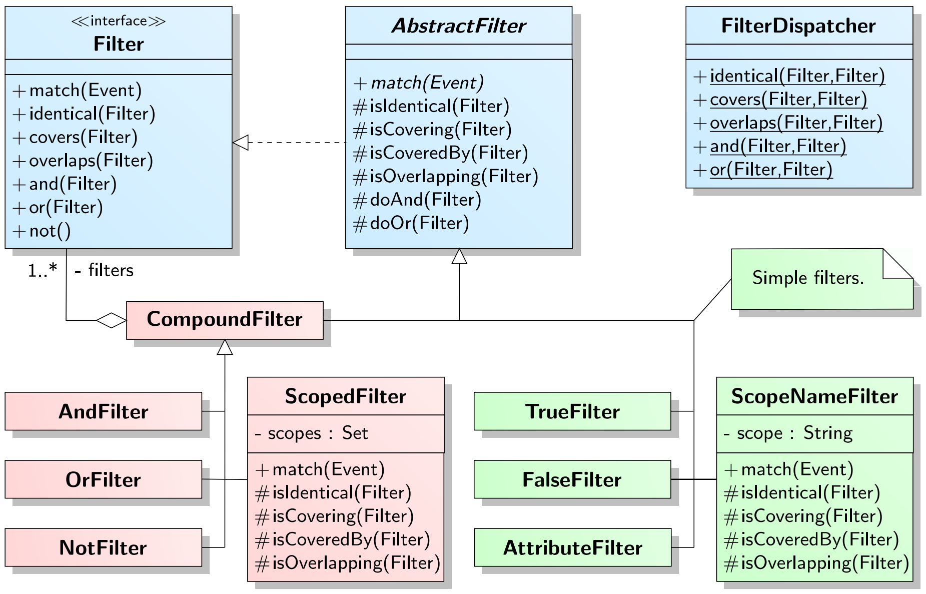


Figure 4.9: Customized filter framework supporting scopes.

that feature more efficient implementations. Figure 4.9 shows a UML diagram of REBECA's filter framework giving an overview of the framework's most important classes. In REBECA, every filter class has to implement the Filter interface. This interface specifies the method match to determine whether a given notification fulfills the filter constraints, the methods identical and covers to test two filter expressions for equivalence or whether one subsumes the other, respectively, as well as the method overlaps to determine whether two filters may have common notifications that both match. Furthermore, the operations and, or, and not enable the Boolean composition of filter expressions. The class AbstractFilter provides a starting point for new filter implementations. Even REBECA's basic filter classes have been derived from it. Except for matching of event notifications, AbstractFilter features safe default implementations of the methods required by the Filter interface and contains a part of REBECA's advanced filter dispatching logic which is discussed later. In order to implement a new filter, a developer may, thus, simply extend the AbstractFilter class and override or overload just those methods that are specific to the new filter type.

REBECA's filters can be grouped into two categories: simple filters and complex compound filters. *Simple filters* usually match a single constraint on an event notification. The AttributeFilter, for example, allows to specify a value range on a single name/value pair of the notification, while the Boolean TrueFilter and FalseFilter even always return true and false, respectively, independent of the tested notification. Thus, the latter ones are primarily for internal usage. *Complex filters* are compound filters usually resulting from Boolean compositions of simple or other complex filters. For example, the AndFilter, the OrFilter, and the NotFilter are the results of the respective Boolean operations, if the resulting expression cannot be minimized to a simple filter again. For constructing complex CompoundFilters, the filter framework follows the well known composite design pattern [81]. Thereby, it is possible to dynamically assemble sophisticated and arbitrary complex filter expressions.

When extending REBECA's filter framework, the existing filter classes are usually unable to handle added filters appropriately and, thus, return safe default values that do not allow for any routing optimization or filter minimization. Developers are, therefore, encouraged to also provide filter logic to determine similarities between filter expressions of new and old classes as well as to simplify compositions thereof. To integrate this logic into the existing framework a double dispatch mechanism is needed that chooses the appropriate method based on the runtime type of the two filters to be compared or composed, respectively. Unfortunately, Java [84] in which the current version of REBECA is written does not provide multimethods and a multiple dispatch mechanism [151] as a built-in language feature as programming languages like Common Lisp [211], Clojure [90], or Groovy [109] do as default. Thus, we have to emulate it.

REBECA's filter dispatching logic is divided into the AbstractFilter and the FilterDispatcher class. AbstractFilter introduces the new methods isIdentical, isCovering, isCoveredBy, isOverlapping, doAnd, and doOr that actually contain the logic to perform similarity tests or filter compositions, respectively, while the methods identical, covers, and, and or declared by the general filter interface just start the dispatching process by calling the correspondent methods of the FilterDispatcher class. Based on the runtime types of the passed filters the FilterDispatcher uses a function table to look up which filter class or super class provides the most appropriate method to execute. Thereby, more specific methods, e.g., methods with filter logic declared in filter subclasses for a restricted set of filter types, are preferred over more general methods. At last, if no other suitable methods are available, the basic implementations of AbstractFilter are chosen to guarantee a safe default behavior. Hence, to integrate new filter logic, a developer just needs to override or overload the isIdentical, isCovering, isCoveredBy, and isOverlapping methods as well as the doAnd and doOr methods in the derived filter class. Thereby, it is sufficient to only provide implementations for those methods that handle new and anticipated combinations of filter types. The FilterDispatcher ensures that these new methods are executed only when appropriate or, otherwise, safe default values are returned.

Please note that since the covering relation is not symmetric, the two methods isCovering and isCoveredBy need to be dispatched for testing whether one filter subsumes the other or vice versa. Contrarily, to determine a filter's logical complement no sophisticated dispatching is necessary as negation is an unary function. Thus, there is no need to introduce an additional doNot function. Likewise, the match function is not dispatched which determines whether a given event notification fulfills the filter constraints. As filters are usually defined for just a particular type of notifications, the dispatching mechanism causes too much overhead in the majority of cases. Using reflection [66, 127], it is possible to automatically build up the function table at system startup that is used by the FilterDispatcher class. As several routing algorithms especially aim at exploiting filter similarities or simplifications, an efficient dispatching process is crucial. Therefore, it is notably beneficial to manually optimize the order in which filter classes are stored and dispatched in the table or to even hard code the main dispatching logic if performance is of outmost importance. As the main dispatching logic is completely enclosed in the FilterDispatcher class, the framework remains maintainable.

By leveraging compound filters as well as the dispatching mechanism of Rebeca's filter framework it is easy to embed scoping into publish/subscribe routing. The main idea is to store required scope information within the filter itself. Therefore, we introduce the ScopedFilter as a new compound filter that encapsulates another, arbitrary filter and additionally holds its accompanying set of scopes. By wrapping each filter in a broker's routing table in a ScopedFilter, it is, thus, possible to augment the routing entries with scope information without the necessity to change or adapt the table's management logic. Furthermore, the ScopedFilter class overrides all inherited methods that are responsible to match event notifications or test for filter similarities. Wrapped filters as well as event notifications are first brought to the GCSS and transformed, if necessary, before being evaluated or tested as usual. Although ScopedFilters may contain arbitrary complex filters, they are not subject to any filter composition itself. The filter dispatching logic ensures that the new, overridden methods are applied whenever ScopedFilters are involved. Thereby, it is guaranteed that scope constraints are considered automatically whenever event notifications are matched or routing decisions are made.

Furthermore, in order to support the management of scopes the ScopeNameFilter class is added to the filter framework. A ScopeNameFilter is a simple filter directly derived from the AbstractFilter class. Its purposes is twofold. First, it uniquely identifies a particular scope and, hence, is used in scope advertisements and scope subscriptions to clearly specify which scope is offered or requested, respectively. Second, it matches all control notifications published by the component which is managing the particular scope. Thereby, it is ensured that membership approvals or denials as well as other management information reaches requesting components and scope members.

To sum up, Rebeca's filter framework eminently eases the integration of scoping. Furthermore, it is flexible enough to even allow the gradual adoption of scoping within the publish/subscribe infrastructure. Event-driven applications with scopes and those without may run in parallel without interfering with each other. Notifications and filters without scope information are implicitly assumed to simply belong to the root scope.

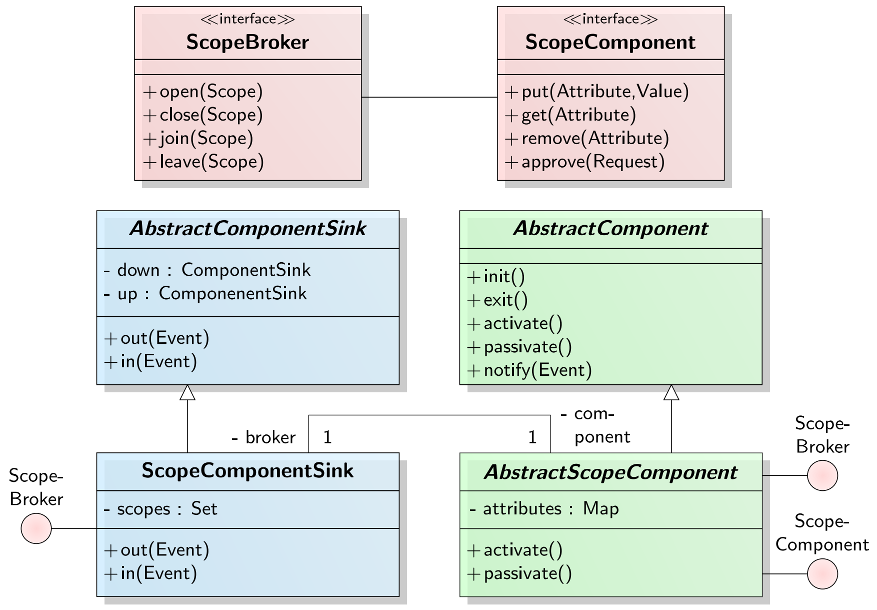


Figure 4.10: Implementation of scope broker and scope component interface.

4.5.3 Scope Components

Beyond just being passive scope members, in REBECA, regular application components may also actively define, create, and manage their own scopes. To support components in administering their scopes REBECA's scope plugin provides a dedicated component sink with elementary management logic including often needed, basic scope operations for the components to use. The ScopeComponentSink belongs to the middleware's client library and is plugged into the component's connection to its hosting broker. Its purpose is twofold. First, it transparently tags passing event notifications, subscriptions, and advertisements with appropriate scope information. Thus, even legacy components and applications may be deployed and seamlessly integrated in publish/subscribe infrastructures with scopes. Second, it provides components that depend on scoping the necessary interface to access management functions and scope operations. Figure 4.10 shows a UML diagram giving an overview about the ScopeComponentSink, implemented interfaces and related classes as well as offered functions and operations.

The ScopeComponentSink class extends the AbstractComponentSink class and inherits properties and logic to be easily plugged into the connection between component and hosting broker. Methods in and out allow to modify and transform event notifications after being received from or before being send to the broker, respectively. Similar to the broker-side, component sinks can be chained, too. Thus, up and down point to the next upper and previous lower sink in the sink chain, respectively. Custom components usually extend the AbstractComponent class and override inherited methods as needed. Methods init and exit allow components to perform initializations after component creation and to free resources before destruction, respectively, while methods activate and passivate are called for starting and stopping component threads, respectively.

To facilitate scope management, however, components and their sinks need to implement additional interfaces. The ScopeBroker interface declares methods for scope creation and membership management and is implemented by the ScopeComponentSink. Methods join and open allow components to explicitly request the membership in an existing scope or to define and create an own new scope with corresponding overlay network, respectively. Methods leave and close are, contrarily, used to quit a scope membership and to revoke a previous scope declaration, respectively. Furthermore, the ScopeComponentSink is responsible to implicitly assign available scopes based on configuration settings, scope selector functions, and component properties as described in Sect. 4.4.2. Thereby, the sink ensures that the component is not activated until it has joined all mandatory as well as available optional scopes and subscopes.

To perform the scope assignment process, the sink must be able to access a component's attributes and properties. The ScopeComponent interface, therefore, provides the methods put, get, and remove to add new or modify existing attributes, to query their values, and to eventually delete them again, respectively. Finally, the approve method allows a component managing its own scopes to decide which other components are admitted as scope members. For each request reaching the managing component, the approve method is called and the identity as well as provided credentials of requesting components and brokers are passed. A scope membership is granted if the method finishes normally. It is denied if an exception is thrown. The exception's cause is added to the scope notification that is subsequently generated and send to the requesting component in order to explain why the scope membership is denied.

The AbstractScopeComponent class bundles all necessary operations and interfaces to conveniently manage a component, its attributes, and scopes. While the ScopeComponent interface is actually implemented, the class, however, does not contain own logic to realize any ScopeBroker method. Instead, it acts as proxy which forwards all calls to the respective ScopeComponentSink instance. Nevertheless, the AbstractScopeComponent class is intended to be a starting point for custom components. Developers are encouraged to extend this class and add or override methods as needed to efficiently implement advanced components that leverage and exploit scoping to their advantage.

4.6 Related Work

Scoping is a concept familiar to every computer scientist. In high-level programming languages that are commonly used today, a *scope* determines the context in which a certain name, e.g., a variable, is associated with a particular entity, e.g., a value. Thereby, the scope[1] limits the visibility of the variable and ensures that its value cannot be accessed throughout the whole program. For example, a local variable can only be accessed within its defining function as outside the function block it has no meaning. Scoping is, thus, an essential prerequisite for structural programming [50] facilitating elementary concepts such as information hiding [165] and encapsulation [21] that help us mastering the complexity of modern software systems.

Although scoping has been introduced in programming languages more than half a century ago [6] and is deeply ingrained today, it is all the more surprising that the majority of publish/subscribe systems does still not provide adequate abstractions and comprehensive means to easily manage and control the visibility of components and their event notifications. Nevertheless, several approaches exist in research literature, some of which directly deal with scoping, others of which focus on different aspects casually affecting visibility. In this section, we give an overview about related approaches and summarize their ideas.

4.6.1 Visibility

The need for an engineering methodology to build structured event-based systems and applications was first expressed by Fiege et al [71, 73]. Engineering requirements are derived to facilitate the design of modular systems considering *visibility* as central abstraction to achieve such modularity [72]. Consequently, scopes are introduced as primary engineering means to bundle components and limit and control the visibility of their event notifications. In [74, 75], a more formal discussion of event-based systems with scopes is given accompanied by trace-based specifications and implementation schemes that enable scoping on top of a simple, flat notification service without structuring means. In [68], Fiege further explores the design space for implementing scopes and engineering event-based systems and delivers a comprehensive as well as detailed insight into the field. Advanced concepts like scope interfaces and attributes, transmission policies, and notification mappings are elaborated while the benefits and drawbacks of different implementation approaches are discussed. The latter include collapsing scope hierarchies into extended filter expressions, mapping scopes onto multicast groups or dedicated event brokers as well as scope overlay networks formed by integrating visibility constraints into the publish/subscribe routing layer. This work is summarized in [147] while important aspects are also highlighted in [69].

In our work, we pick up many ideas from Fiege, most prominently, we also consider visibility as the major abstraction to facilitate modularity and scopes as adequate structuring means to control its extend. However, there are decisive differences caused by dissenting intentions, too. From a software engineering

[1] More precisely, the *lexical* scope of a statically bound variable whereas dynamic binding leads to an indefinite scope. See Moreau [140] for a distinction of both.

point of view, scopes fulfill a twofold role. First, a scope is a unit for abstraction that bundles a set of related components, offers higher-level functions that are jointly implemented by its members, provides a common interface to access these functions similar as components do, and may, thus, itself become subject to a recursive composition process. Second, a scope is also a unit for encapsulation that provides a shielded environment for components to interact by constraining the visibility of event notifications that are published or subscribed by its members. While Fiege primarily concentrates on the former role of scopes we, on the contrary, focus on the latter. Scopes are well suited to bundle components for orchestration, to filter, direct, and fine-tune the information flow between them, and to provide environment as well as context information. Focusing on these goals allows us to simplify many aspects in scope handling.

Furthermore, Fiege investigates and evaluates several different approaches and architectures to integrate scoping whereas we just consider distributed publish/subscribe infrastructures based on a network of cooperating brokers as target platform. Reducing the number of architectural choices enables us to tightly integrate scoping with publish/subscribe routing and to provide optimized and more efficient implementations. In the following, we pinpoint the similarities and differences in detail.

As scopes bundle components to new software artifacts that itself can recursively be bundled in superscopes, Fiege, thus, represents a structred event-based system by a directed graph that denotes the component/scope and scope/superscope relationship, respectively. Based on this scope graph, Fiege subsequently introduces component interfaces along the graph's edges, describes upward and downward delivery paths for notifications, and finally defines the visibility of components as well as event notifications. However, by introducing scopes as simple *sets of components* as done in Sect. 4.2, we are usually able to give more natural and straightforward definitions of these terms and concepts based on set theory. Likewise, it is possible to decompose a system into component sets and more specific subsets. In fact, the resulting set hierarchy, or scope hierarchy as we call it in Sect. 4.2.2, resembles Fiege's scope graph. Both describe and represent the system's structure and can often be mapped onto each other.

Usually, components are bundled in scopes that are somehow related and share common properties, e.g., the kind of service they jointly provide, the subsystem where they are deployed, or the organizational unit to which they belong. Thus, it is possible to factor common properties out and attach them as *attributes* to enclosing scopes as described in Sect. 4.2.4. This is especially useful if these properties are not yet known at design time, for instance, if the organizational unit responsible for a certain set of components is not assigned until deployment. Scope attributes provide an effective and convenient way to flexibly add and incorporate environment and context information as well as configuration data. Furthermore, we allow scope attributes to be inherited, to get overridden, and to transparently augment components and event notifications in order to better characterize and classify subscopes and published events therein.

Consequently, we primarily use *notification mappings* as introduced in Sect. 4.2.6 to control the visibility of their attributes when crossing scope boundaries, for instance, by removing certain attributes or adapting their values. Fiege also introduces scope attributes as well as notification mappings [68]. However, both concepts serve different purposes. Scope attributes are used to annotate the scope graph to support system composition by labeling interfaces and services that are either provided or required by a particular scope and its components. Notification mappings address heterogeneity issues occurring at composition by transforming event notifications at scope boundaries, e.g., from a data model used by components inside the scope to a data representation only valid outside and vice versa.

Regarding the implementation of scoping, Fiege investigates a variety of approaches and platforms while we primarily focus on distributed publish/subscribe systems whose brokers correspond to the modular architecture presented in Chap. 3. This has several advantages of which a tight integration into the publish/subscribe routing layer and a high reusability of existing publish/subscribe functions are the most important. According to our modular broker architecture, we provide scoping as a *pluggable feature* that is compatible with all major routing algorithms. Furthermore, all operations required for scope management are mapped onto standard publish/subscribe features so that components may easily advertise own scopes, subscribe to others and are notified on updates. This is in contrast to proposed solutions of Fiege that also foster the combination of routing and scoping but still require extensive changes to existing routing algorithms and tables as well as the addition of new data structures and message types. In particular, each broker needs to maintain a separate routing table per hosted scope which may internally reference other tables according to scope interfaces and mappings defined on the edges of the scope graph.

This does not only introduce a lot of management overhead but may also lead to the duplication of notifications send to a single neighbor or component if multiple delivery paths over different tables exist. Thereby, duplication may occur inside a single broker or on different brokers of the network and, unfortunately, both cases are quite costly to counter. As we, however, store all advertisements and all subscriptions together with their scope affiliation in a single routing table each, we are able to efficiently avoid duplicates. Nevertheless, there is a computational price to pay. Notifications and filter expressions may need to be temporarily transformed or narrowed, respectively, before they can be matched or compared in a common superscope. But, if possible, just the original notification or filter is forwarded to a neighbor broker or component. It is only duplicated or permanently transformed when absolutely necessary, e.g., if send to two distinct destinations in different scopes. Hence, there is a trade-off between compact routing tables requiring computational overhead on the one hand and expanded routing tables causing a high management overhead to counter duplicates on the other hand.

4.6.2 Security

In general, *information security* includes all measures to protect the confidentiality, the integrity, and the availability of data that is stored, processed, or transmitted within a system [227]. Securing information assets in a publish/subscribe system is especially challenging due to the loose coupling of components and their indirect style of communication. For example, it is hard to guarantee the confidentiality of a published notification if neither its receivers are known a priori nor it is even desired to find out their identities. Further security considerations for publish/subscribe systems are given by Wang et al. [222].

Nevertheless, information security in publish/subscribe systems is an active field of research offering approaches and solutions that, similar to scoping, allow to define, configure, and administer in detail what information, e.g., which notification, subscription, or advertisement, is visible to whom, e.g., which component or broker. For instance, Miklós [137] shows how to define detailed and fine-grained access control and visibility restrictions that even consider individual notification attributes as well as their values.

Role-based access control (RBAC) [195] schemes are well suited to enforce visibility constrains in publish/subscribe systems. Roles specify particular system or application functions and serve as intermediate between *principals* and *privileges*. On one side, roles are assigned all privileges that are required to fulfill their function, for example, a service provider is granted the right to subscribe to client requests and publish replies while clients are allowed to issue service requests and receive the results. On the other side, roles are associated with principals that fulfill the specified functions, for example, all components that actually implement the particular service and those components that are allowed to use it, respectively. By introducing roles it is, thus, possible to effectively manage and control access restrictions and visibility constrains while preserving the inherent loose coupling of components in publish/subscribe systems.

To our knowledge Belokosztolszki et al. [15] were the first who integrate RBAC into publish/subscribe systems. The prototype implementation is based on the Hermes publish/subscribe middleware [178, 179, 180] and the Open Architecture for Secure Internetworking Services (Oasis) [10, 11] contributing the access control logic. In order to publish notifications and subscribe to events, application components need to provide credentials to Hermes brokers that prove their membership in a role which has the necessary privileges. Thereby, privileges may be individually granted for distinct event types. Likewise, a broker must be authorized by the owner of the event type to handle corresponding notifications, subscriptions, and advertisements. To prove its authorization to neighbors and clients, a broker needs a valid X.509 certificate [44] that is signed by the owner of the event type or that is part of a certificate chain with the owner as root.

Bacon et al. [7, 8] and Pesonen et al. [173, 174, 175, 176] introduce a decentralized administration of roles, refine applied encryption mechanisms, and employ advanced key management techniques to extend the work to *multi-domain* environments and to efficiently manage those. By encrypting notifications and filters confidentiality is even ensured if messages are routed over brokers that are not fully trusted to handle the particular event type. Moreover, by individually encrypting event attributes it is, thus, possible to practically adjust and fine-tune the level of information visibility and required trust at will. Singh et al. [205, 206, 207, 208] present a comprehensive case study that employs publish/subscribe with access control in collaborative healthcare environments, e.g., homecare environments where a patient's well-being is remotely monitored. While health information is sensitive and must be protected for privacy reasons, it must, at the same time, also be shared between involved care providers such as doctors, care nurses, specialists, pharmacies, or technical support as appropriate to afford a proper treatment.

Although access control as well as scoping limit the visibility of events and information in publish/subscribe systems, both have different intentions. Access control mechanisms manage and monitor component privileges in order to enforce the confidentiality, integrity, and availability of data in the system while scoping bundles components to structure and organize the infrastructure and its applications. Thus, for a component within a scope, there are no further access restrictions preventing it to interact with any other component of the same scope or to publish and subscribe to arbitrary event notifications. However, access control restrictions may be in place as described in Sect. 4.4.3 when a component requests to join a scope. Furthermore, notifications are primarily filtered or transformed at scope borders for engineering purposes enabling brokers to exploit scope structures for optimizations. Contrarily, access control restricts the visibility of notifications or notification attributes to ensure confidentiality. Usually, this is accompanied with encryption on infrastructure level, which poses a considerable overhead on brokers, in particular, if brokers do not completely trust each other. In this case, scoping may help to organize domains with different levels of trust.

4.6.3 Context

In general, *context* may be seen as any useful information to better characterize the situation of an entity such as a person or an object of interest [53]. In particular, context may provide information about location and time as well as ongoing activities and identities of persons and objects around. Many novel applications increasingly depend on context information to adapt and tailor their functionality to the present situation and existing environment conditions. Such context-aware applications often make use of publish/subscribe as communication paradigm as it enables publishers to easily add context attributes to event notifications and, moreover, additionally allows subscribers to conveniently filter on those. Thereby, it is possible to restrict the visibility of events to those notifications that are relevant for a particular situation. Similar to scoping, other notifications are hidden.

First publish/subscribe systems and algorithms explicitly designed to support context-aware applications primarily focused on aspects of *location* and *mobility*. Meier and Cahill [132, 133], for example, propose a proximity-based event model for mobile ad hoc networks (MANETs). The dissemination of event notifications is simply restricted to a limited geographical area around the publisher. Eugster et al. [59] enable publishers as well as subscribers to define a geographical range around their current position as publication or subscription space, respectively. As a consequence, notifications and subscriptions are only matched against each other if both the publisher and subscriber are located within each others' space. To support location-based services in pervasive environments, Chen et al. [37] augment event notifications with location data and allow subscribers to filter on these information using spatial predicates. Although the evaluation of predicates is offloaded to clients, the approach follows a central architecture. Instead, Cugola and de Cote [45] consider a distributed broker network dispatching events and subscriptions associated with geographical scopes. Brokers gather and maintain location information of clients and other brokers to limit the forwarding of notifications and subscriptions to those within relevant scopes. On the one side, this saves forwarding traffic, but on the other side, it requires brokers to keep the gathered location data up-to-date which may also be quite costly.

Symmetric subscription systems as introduced by Rjaibi et al. [191] are a general extension of the regular publish/subscribe model that is not limited to location data or context information. In *symmetric publish/subscribe* , both event notifications as well as subscriptions carry attributes and predicates at the same time. Thus, they only match if the predicates of each side are fulfilled by the attributes of the other. Although originally not intended by the authors, symmetric publish/subscribe systems are well suited to encode context information into system messages and bind their visibility to arbitrary constraints. Cugola et al. [47] take up the idea but draw a clear distinction between a message's content and its context and the predicates that refer to each part. This is a requirement in order to implement the efficient forwarding and routing strategies proposed by the authors for a distributed broker network. Beyond that, Frey and Roman [79] present an enriched context-aware publish/subscribe model that, besides filtering on a publisher's or subscriber's context, allows to define a context of relevance for each publication as well as a context of interest for each subscription. Thus, for a positive matching, the publisher's and subscriber's context as well as the context of relevance and interest must overlap. Although the publish/subscribe model itself is generic, the described implementation solely focuses on location-awareness in a MANET environment.

In contrast to the related work above, scoping approaches the idea of context and context-awareness from a different angle. A scope bundles components and provides a protected environment for interaction limiting the visibility of event notifications to those relevant in the scope's context. Hence, publishers and subscribers do not need to augment their notifications with context attributes and provide additionally context constraints, respectively, in order to define which events are relevant. Instead, it is the scope specification that determines the components relevant in the particular situation. As described in Sect. 4.2.1 and Sect. 4.4.2, the component selector of a scope evaluates component attributes to identify those publishers and subscribers the scope is applicable to. Although top level scopes must explicitly be requested by components or brokers for management reasons, their subscopes are assigned automatically on availability and relevance. Moreover, it is even possible to characterize the interaction context itself by defining scope attributes as introduced in Sect. 4.2.4 that are passed on to all members. Exploiting this feature components can be made aware of their environment, e.g., by providing scope attributes describing their current location or ongoing activities. However, since creating and closing scopes as well as joining and leaving them causes a considerable management overhead for brokers and components, more light-weight approaches as presented above may be preferred in environments that are highly dynamic. Nevertheless, scoping provides a competitive alternative for domains of moderate dynamics.

4.7 Discussion

In this chapter, we introduced scoping as a module concept to structure publish/subscribe systems and organize event-driven applications. Scopes bundle related components and provide a protected environment for interaction. Therefore, scopes restrict the *visibility* of events and limit the communication with components outside the scope to those notifications specified in the scope's interface. Scopes can be nested enabling a hierarchical system decomposition according to different criteria and aspects. However, to facilitate the design of versatile and flexible infrastructures, components may be members of several scopes at the same time if multiple criteria apply. Moreover, scope attributes help annotating scopes and the components therein with arbitrary information. They allow system administrators and application developers to efficiently provide and incorporate configuration data or context information and, thereby, ease component orchestration and system organization. Formally, we defined the semantics of scopes based on concepts of *set theory* and showed the applicability and usefulness of these definitions by illustrative examples. Likewise, we introduced scope attributes and inheritance as well as basic notification mappings.

Regarding the integration of scoping into distributed publish/subscribe infrastructures, we presented a solution based on *scope overlay* networks that directly embeds scopes into the content-based routing layer. Primarily, this allows to exploit the scope structure for routing optimizations. Thus, it is possible to really restrict the forwarding of event notifications, subscriptions, and advertisements within the broker network according to defined scope boundaries instead of just filtering out non-visible notifications before passing these to application components. Although tightly integrated into the routing layer, our approach is compatible with all major content-based routing algorithms. Moreover, we were also able to map necessary scope management functions to conventional publish/subscribe operations reusing existing data structures and middleware logic. For example, the creation of a new scope is advertised within the network so that brokers and components may subscribe for membership and get notified on updates and changes. Finally, we discussed concrete implementation details in order to realize scoping as a *pluggable feature* according to the broker architecture presented in Chap. 3.

Scoping is an essential building block for engineering publish/subscribe systems and event-driven applications. In particular, with increasing system size and complexity scoping helps to keep applications comprehensible and infrastructures manageable. To further underpin this statement, we discuss, among other things, the important role of scopes for component orchestration in Chap. 5. Furthermore, Chap. 6 evaluates and proves the positive effects of scoping regarding system scalability.

Chapter 5

Programming Abstractions

Contents

5.1 Introduction

In computing, *programming abstractions* are commonly used today. In fact, every high-level programming language abstracts from details of the programmed machine, offers effective control and data structures, uses syntax elements that are often close to natural languages, and may also provide runtime support, for example, by garbage collection. As consequence, writing programs and developing applications becomes easier and convenient as well as more efficient and productive. Looking at publish/subscribe middleware implementations and event-driven applications, however, little has changed over the years. Programming support is often limited to a generic interface that simply allows to publish and subscribe event notifications. While many research projects and approaches focused on issues of event routing and filtering, a better integration of publish/subscribe into programming languages has been neglected so far, aside from a few exceptions [61, 218]. In this chapter, we develop and present programming abstractions that ease the development of event-driven applications and, in particular, increase the reusability of their components.

First, we analyze common pitfalls when using the publish/subscribe paradigm in event-driven applications and discuss possible remedies in Sect 5.2. Often, application components mix business logic with context information and configuration data making them hard to adapt and maintain. With the provided programming abstractions, we aim at improving the separation of concerns. In Sect. 5.3, we introduce event ports as novel interface for event-based communication allowing developers to primarily focus on the business logic of their components while directing event streams is factored out and left for configuration. Section 5.4 discusses the role of scopes for applications and how to define and customize them to the application's context. Section 5.5 then covers all aspects of system configuration. We provide effective means to orchestrate applications and components by connecting their event ports and grouping them into scopes. The integration of presented concepts into the architecture of a publish/subscribe broker is discussed in Sect. 5.6 along with implementation details. The chapter concludes with an overview of related work in Sect. 5.7 following comparable approaches and a final discussion in Sect. 5.8 summarizing main contributions and giving a short outlook of open problems.

5.2 Publish/Subscribe

Publish/subscribe is a successful interaction paradigm well suited for event-driven applications. In graphical user interface (GUI) toolkits, for example, it is a proven method to separate the underlying data model from its actual presentation on one hand and to keep both consistent on the other. If the data in the model changes, all dependent views are notified to update their presentation. In fact, this is a recommended design pattern also known as the *observer* *pattern* for building reusable software elements [81]. Its primary advantage is the abstract and minimal coupling between participants as both sides do not need to know their exact identities. Instead, it is sufficient to know that one side provides an interface to subscribe for updates, while the other offers an interface to get notified on changes.[1]

In distributed dynamic environments, only very few assumptions can usually be made about available communication partners or infrastructure services. A paradigm inherently featuring a loose coupling should, therefore, eminently pay off making publish/subscribe a dominant communication technique in dynamic domains such as pervasive or ubiquitous computing. However, this is not the case, although a strong potential is seen by many researchers. In fact, there are still many drawbacks and pitfalls that hamper the adoption of publish/subscribe and, in particular, make it hard to write reusable event-driven components for distributed environments. In the following, we first discuss these major pitfalls in order to propose corresponding remedies afterwards.

5.2.1 Pitfalls and Remedies

Today's software development is primarily object-oriented. However, the object-oriented style of communication with which the majority of application developers is familiar differs considerably from the event-driven way publish/subscribe communication is used in distributed systems. This is also known as the *object/ event impedance mismatch* [219, 220]. In object oriented languages, developers are used to synchronously call a specific method on a particular object and may simply get the result as return value. However, publish/subscribe middleware implementations usually provide just a single generic method to asynchronously publish arbitrary types of events that are delivered to potentially many anonymous subscribers or even to no one at all. Hence, there is also no designated way to get a result back from a subscriber other than switching roles and having the former subscriber publishing an event as reply. Developing components for distributed dynamic environments often requires a mixture of both styles: object-orientation within the component itself and publish/subscribe to communicate with other remote components in a loosely coupled fashion. This is ambitious and challenging, but does not completely explain why so many publish/subscribe components are hardly reusable in a different application context.

Events vs. context. Events and event notifications are the primary communication objects when using publish/subscribe. Thereby, a notification reifies an event and describes what happened as well as the circumstances of its occurrence. Especially the latter is important to distinguish similar events published by components of the same type. Therefore, attributes characterizing publisher (e.g., component, device, user) and context (e.g., location, time) often need to be

[1] However, as the notified view in a GUI has to interpret the model data for presentation, it does usually know more details about the subject it observes.

added to a notification, although they may not be directly related to the component's actual purpose. Nevertheless, it is usually the developer writing the component's business logic who is also responsible to provide these additional attributes before publishing the notification. As this is cumbersome and tedious, it is often neglected or, worse, hard-coded in the component just for the current application context. Hence, the component's reusability is strongly limited.

A clear *separation of concerns* is the first step for mitigation in order to identify which notification attributes really constitute the actual event, which properties are simply inherited from the publishing component, and what further context aspects need to be added. Second, the middleware can support developers by annotating some of these attributes (e.g., component id, timestamp) automatically. For the others, the middleware may provide interfaces to plug in custom logic to intercept and augment event notifications after they have been published by the component but before they are actually transmitted. Chapter 3 presents a pluggable publish/subscribe architecture designed for feature composition that enables developers to easily integrate such interfaces.

Development vs. deployment. Software engineering clearly distinguishes between component development and component deployment. While development comprehends implementing, testing, and documenting the component and its functions, deployment includes all activities such as packaging, installing, and activating that are necessary to make the component available for use, for instance, in a production environment. Unfortunately, it is the publish/subscribe paradigm that interferes with the clear distinction between the two. As components are required to subscribe to event notifications they are interested in, it is usually the developer who also has to provide an appropriate filter expression or construct a corresponding filter object for event selection. Therefore, the developer is forced to already make extensive assumptions about the deployment environment at design time. This often leads to a mixture of code and configuration hampering the component's reuse in different environments.

To improve component reusability, the role of an administrator responsible for configuring event-driven components and organizing the publish/subscribe infrastructure needs to be firmly integrated into the development process. Consequently, middleware implementations have to provide adequate tools and means to support administrators in their work. In Chap. 4, we introduced scoping as structuring means to bundle related components and organize the broker network. Scoping allows administrators to externally restrict the visibility of event notifications to those relevant in the deployment context and, thus, enables developers to issue more reusable, generic subscriptions. Additionally, scopes provide the possibility to centrally define attributes valid for all scope members. This feature may also be exploited to provide configuration data and settings. However, modern middleware implementations usually feature a *component container* that provides a managed runtime environment offering common services and functions for components. In particular, the component container is the preferred source for configuration data and context settings and should, therefore, enable administrators to configure and adapt component attributes, subscriptions, and advertisements as needed. Unfortunately, only few, if any, provide this level of control for publish/subscribe components.

Events vs. objects. Name/value pairs are a general and common data model for notifications to represent occurred events and their attributes. However, processing and manipulating them directly is tedious, cumbersome, and error-prone. Thus, publish/subscribe middleware implementations aim at providing representations that are more handy and better integrated into programming languages used by application developers. Accordingly, objects have been suggested to represent events [62]. The benefits are appealing. Object fields correspond to event attributes and are easy to access and manipulate. Moreover, checking events and attributes for type safety is done by the programming language's compiler as natural byproduct. At the same time, however, the class hierarchy of object oriented languages is also the biggest drawback limiting a component's reusability. Assume, for example, an on-line statistic analysis has to be done for particular stock quotes in one application scenario and for certain temperature sensor readings in the other. As stock events and sensor events belong to different application domains, it is usually sensible to model both by unrelated event classes that feature different event attributes, e.g., price and temperature, respectively. Consequently, two different components each dedicated to a single event type are required to analyze stock prices and temperature data. However, the component logic itself will be identical as the calculation is the same in both application domains.

Many other publish/subscribe middleware implementations often use diverse forms of dictionaries to represent events and store their attributes and associated values. Although several programming languages (e.g., C# [88], Lua [94], Python [126]) provide array-like syntax constructs to make accessing attribute values more convenient, dictionaries do not provide much abstraction. Before being able to publish new event notifications, for instance, they usually need to be manually populated by the developer. This is tedious, cumbersome, and prone to the pitfalls discussed above. However, regarding the statistic example from above, by simply renaming the attributes within a dictionary, the same component could have been (re)used to analyze stock prices as well as temperature readings. But to our knowledge, there is no current publish/subscribe middleware that both provides convenient programming abstractions for event-driven applications and facilitates a flexible reusability of components.

5.2.2 Components and Events

When designing event-driven publish/subscribe applications, developers have to consider two different aspects at the same time. On one side, there are the event-driven components containing the actual business logic. On the other side, there are the event flows connecting the components. Hence, meaningful programming abstractions for publish/subscribe applications have to adequately support both aspects. But, as the discussion of pitfalls and remedies in the previous section shows, no current approach or middleware succeeds equally well on both sides leading to the mentioned problems of mixing business logic, configuration data, and context information.

However, engineering domains such as digital signal processing or control theory have developed practical and proven design methodologies to model, analyze, and implement complex systems by decomposing them into individual components and explicitly defining the *dataflow* between them. Thereby, each component implements an operator or function block that consumes input data, applies its operator function, and, thus, produces new output data. Dataflow programming languages [104] are often used, in particular in visual variants [91], to determine which exact components are assembled and how their outputs and inputs are connected for passing the data from one component to the next. As components are generally treated as black boxes, they are simply interchangeable and, thus, easy to reuse provided that the data types at inputs and outputs correspond to those they are connected with.

Looking at publish/subscribe systems from a dataflow perspective, many similarities become evident. Likewise, event-driven components consume data in form of event notifications, process them, and may produce new event notifications that are send out to other components. However, emerging event flows are more diverse and complex, but yet do they follow the publish/subscribe pattern. In the remainder of this chapter, we derive and present programming abstractions that aim at easing the design of publish/subscribe applications by modeling and implementing them in terms of event-driven components and event flows between them. Therefore, our immediate intention is twofold: to increase the *reusability* of components and to simplify their *orchestration*. Moreover, from the pitfalls discussed in the previous section, we draw the following conclusions as general guidelines to improve the development process:

- Designing, implementing, and testing the business logic of involved components is an application developer's major objective when developing a publish/subscribe application. Configuration and context data and detailed event flows are secondary. Moreover, the more the business logic is mixed with such data, the less reusable the component becomes.

- Orchestrating components and directing event flows as well as providing context and configuration data is part of an application's deployment done by the system administrator. Thereby, components are adapted to the local broker infrastructure, organizational conditions, and already existing publish/subscribe applications.

- The publish/subscribe middleware has to support application developers and system administrators alike by providing an adequate component container and runtime environment. For developers, the container offers frequently used services accessible from within components easing the implementation. For administrators, the container allows to configure and adapt components as well as to provide necessary context data.

5.3 Event-driven Components

Distributed event-driven applications are made of components that produce, consume, and react to event notifications. By publishing a notification, a component informs about a significant change in state or conditions allowing subscribed components receiving the notification to react appropriately, for instance, by adapting their own behavior. Every component may be a publisher, a subscriber, or both making event-driven applications flexible and agile. When designing programming abstractions for event-driven components we, thus, have to consider and support both roles alike. Essentially, we introduce the concept of event ports as configurable novel interface through which components receive subscribed events and publish their own in a uniform and standardized fashion. Subsequently, we discuss how event ports relate to event handlers, component threads, and filter expressions.

Abstractions and concepts presented in the following are not bound to any particular programming language although we exemplify them in Java [84]. In fact, we do not use any language construct unusual for object-oriented programming languages or even introduce novel ones. However, we often make use of Java *annotations* to conveniently mark and label component fields and methods that are of relevance for the component container providing the component's runtime environment. Alternatively, appropriate comments and a preprocessor or a detailed configuration file could have been used instead.

5.3.1 Event Ports

In general, an event is an incident of interest that may occur inside or outside a computer. When detected by a component, however, it causes a change in the component's state that is subsequently published to notify other components. Thereby, the component fields constituting the new state are often included as notification attributes to characterize the occurred event. Creating the notification and publishing it may take place at arbitrary positions inside a component, likewise, creating a corresponding filter expressions and subscribing to it. Directing and arranging the event flow between components, however, becomes much harder if a component's publish and subscribe operations are buried somewhere inside the component's code. Possibilities and options to externally configure notifications and filters adjusting flows are limited. Furthermore, it is risky since the component's developer may not have taken such a customization into account. For this purpose, we need a more uniform and standardized way how components produce and consume event notifications that is better accessible for external configuration and adaptation. In fact, we aim at providing publish/ subscribe communication without the basic publish and subscribe operations. Therefore, we introduce *event ports* as novel interface to replace both.

Definition 32 (event port). *An* event port *exposes selected fields of a component to become subject to publish/subscribe communication making them either to a source or a sink for event notifications. There are two types of event ports:*

(*i*) *An* out-port *contains component fields whose values are published as new event notification with corresponding attributes. An* out-port*, thus, defines the source of an event flow.*

(*ii*) *An* in-port *contains component fields whose values are set to corresponding attributes of a received event notification. An* in-port*, thus, defines the sink of an event flow.*

An event port simply consists of one or more component fields that together with other fields determine a component's state. Hence, an event port constitutes a part of the component's state. The idea behind an *out-port* is to bundle exactly those fields that subsume the part of the component's state that is published as new notification to inform others about an occurred event. Likewise, an *in-port* contains exactly those fields constituting the part of component's state that is affected by a received event notification. Instead of requiring the component itself to create a new notification with appropriate attributes or inspecting a received notification for data, this can now be done by the component's runtime environment, i.e., the component container. Hence, on every notifiable state change the component container automatically creates a new event notification deriving attributes and values from the component's out-port fields. Similarly, on every received event notification, the container automatically inspects the event attributes and sets the values of corresponding in-port fields.

Figure 5.1 shows the source code of an event-driven component written in Java. Event ports are defined via meta data using the Java annotation mechanism. Thereby, the annotations `@InPort` (lines 2 and 3) and `@OutPort` (line 5) each declare an event port of corresponding type. However, as Java does not allow multiple annotations of the same type to be added to the component class, the two in-port declarations must be enclosed in one `@InPorts` annotation (line 1). We provide such simple wrapper annotations whenever necessary, for instance, a similar `@OutPorts` annotation is needed to define components having two or more out-ports. An event port declaration always contains two required elements. The `event` element names the event notifications produced or consumed by this port for later reference. The `attributes` element enumerates the notification's attributes as a comma-separated list. Hence, considering the event port declarations of the component altogether, the component has two in-ports and one out-port. It consumes the event notification `e` with attributes `a` and `b` as

```
 1 @InPorts( {                                          // wrapper with
 2   @InPort( event = "e", attributes = "a, b" ),       // in–port declarations
 3   @InPort( event = "f", attributes = "x" )
 4 } )
 5 @OutPort( event = "g", attributes = "b, x" )         // out–port declaration
 6 public class EventDrivenComponent {
 7   @Attribute int a;                                  // event port fields
 8   @Attribute int b;
 9   @Attribute( "f.x, g.x" )                           // field aliases
10   Object c;
11   private int x;                                     // private component field
12   ...
```

Figure 5.1: Component with event port declarations.

well as the event notification f with attribute x. Furthermore, it publishes event notification g with attributes b and c.

So far, the attributes are not bound to any component field yet. This is done using the @Attribute annotation which is simply added to appropriate fields (lines 7 and 8). It associates notification attributes with annotated component fields of the same name, for instance, attributes a and b of event notification e are bound to the component's integer fields a and b. As internal and external names and representations may differ the @Attribute annotation supports aliases provided as a comma-separated list (line 9). For example, the component field c is bound to attribute x of event notifications f and g. The private component field x, however, is not considered or affected as it is not annotated. The association between component fields and event attributes is not exclusive. A component field may be bound to attributes of different event notifications and vice versa. Furthermore, component fields are not limited to primitive data types only. Arbitrary complex types may be used as demonstrated by field c. However, the component container must be able to serialize and deserialize the field's value.

Event ports are a key concept to facilitate the orchestration of event-driven components and their event flows. Their benefit is twofold. First, event ports provide a new uniform interface replacing present publish and subscribe calls scattered throughout the component. They relief developers from manually creating and inspecting event notifications before sending and after receiving them, respectively. Moreover, they centrally bundle, specify, and document all component fields that either constitute a new event notification or are affected by a received one. Thereby, they specify a well defined start point or end point for event flows making the component ready for orchestration. Second, the publication of and subscription to event notifications is shifted from the component towards the runtime environment provided by the component container. This opens the possibility to incorporate configuration data more easily, augment event notifications with context information, and adapt subscriptions as needed. Thereby, system administrators are enabled to effectively organize and control event flows between orchestrated components.

5.3.2 Event Handlers

Event-driven components react to event notifications received on in-ports, process the data carried by notification attributes, and may also produce new event notifications published on out-ports. Thereby, processing is done by *event handlers* that usually contain the component's business logic.

Definition 33 (event handler). *An* event handler *is a method of an event-driven component that is called in reaction to an event notification that is received or send on an event port.*

Basically, the definition above states that received event notifications are processed by event handlers. The reasons why event handlers may also be called when event notifications are published as well as resulting consequences are discussed later. Figure 5.2 continues the source listing of an example component started in Fig. 5.1 and shows how component methods are annotated as event handlers. Since event ports and associated component fields are used for data exchange, event handlers must not have any parameters or return values. A component method is simply turned into an event handler by adding the `@OnEvent` annotation in front of its method declaration (line 13). Thereby, for example, the `validate` method gets called on every event notification received. Often, it is necessary to restrict event handlers to process notifications received on specific ports only. Therefore, the events a handler is responsible for can be specified by a comma-separated enumeration provided as argument to the `@OnEvent` annotation (lines 20 and 25). Hence, the methods `compute` and `transform` are only called on event `e` and on events `f` and `g`, respectively.

For each received event notification, there can be multiple handlers available for processing and they all are called one after the other. However, to guarantee a controlled processing, developers need a way to define the sequence in which event handlers are executed. For this purpose, priority values are assigned to event handlers using the `@Priority` annotation (lines 14 and 26). Priorities are integer values ranging from -2^{32} to $2^{32}-1$ with higher values coming first in the execution order. For convenience, we have defined meaningful constants such as `HIGH`, `NORMAL`, or `LOW` for different priority levels. For a fine-grained differentiation within each level, developers may simply write `HIGH-1` or `NORMAL+1` for example (line 26). Based on event ports, event handlers, and assigned priorities, we can now define in detail how event notifications are processed by components and their container.

Definition 34 (event handling). *Event notifications received on a component's in-port are handled and processed as follows:*

```
  ...
  @OnEvent                                    // event handler
  @Priority( Priority.HIGH )                  // always called first
  void validate() {
    // validate all fields
    ...
    if (!valid) throw new AbortException();   // abort event handling
  }
  @OnEvent( "e" )                             // event handler
  void compute() {                            // for in–port e
    // compute field c
    c = ...
  }
  @OnEvent( "f, g" )                          // event handler
  @Priority( Priority.NORMAL + 1 )            // for in–port f
  void transform() {                          // and out–port g
    // recode field c
    c = ...
} }
```

Figure 5.2: Component with event handler annotations.

(*i*) *On receiving an event notification on an in-port, corresponding component fields are set to the values of associated notification attributes.*

(*ii*) *All event handlers are determined that are responsible to process the notification data on the particular event port.*

(*iii*) *The set of responsible event handlers is sorted according to priority beginning with the highest value. The execution order of event handlers having the same priority is undefined.*

(*iv*) *Event handlers are executed one after the other. The whole execution of handlers stops if any event handler aborts the processing. In this case, all event port fields are restored to their original values.*

(*v*) *The component's out-port fields are inspected after the last event handler finished execution. State changes, i.e., new field values, are published as new event notifications. Publication order corresponds to the order of event port declarations.*

(*vi*) *If an event notification is published on an out-port with registered event handlers, event processing continues at step* (*ii*) *for the particular out-port.*

The usage of event ports also affects the way event notifications are processed. There are several important specifics. Obviously, event handlers do not have any arguments or return values as notification data is conveniently exchanged on event port fields. However, this has disadvantages, too. Attribute values of notifications are copied by the component container to the component's in-port fields without checking their validity. Hence, if event handlers fail invalid data remains on in-port fields and contributes to the overall state of the component rendering it potentially invalid, too. To avoid this problem, we use a transaction scheme as described by step (iv) of the event handling process above. If an event handler aborts its execution, subsequent handlers are prevented from running and the component's event port fields are restored to their original values. Therefore, the component container always keeps a copy of the last notifications successfully processed. However, private component fields that are not associated with an event port cannot be restored. The listing shown in Fig. 5.2 exemplifies the application. The event handler validate (line 15) is assigned a high priority to ensure that it runs first whenever a notification is received on any in-port. It checks all event port fields and throws an AbortException when unsuccessful. Thereby, it is ensured that subsequent event handlers such as compute or transform (lines 21 and 27) always run on valid data.

After all event handlers finished execution, the component container inspects the component's out-port fields and compares them to previous values. State changes are automatically published by the container as new event notifications whose attribute values are taken from associated out-port fields. This is convenient as well as sufficient in many cases. Nevertheless, there is at most one event notification published per out-port. Thus, components cannot create multiple notifications of the same type because of a single event anymore. This is a serious drawback limiting the number of interaction and communication patterns for which event ports are applicable. However, step (vi) of the event handling process addresses the issue and alleviates its consequences. Event handlers can also be registered on out-ports and are called after notifications have been successfully derived and published from associated fields. Their execution follows the same rules. In fact, a single event handler may be registered for events on an in-port as well as an out-port at the same time. Thereby, the handler starts processing a received notification on the in-port and produces results to be published on the out-port. Thereafter, the handler is called again and can continue the processing to produce results which are published next.

Figures 5.1 and 5.2 provide an example. The event handler transform (line 27) is executed on in-port events f as well as out-port events g (lines 3 and 5). Its function is to transform the computed or received event attribute x stored in field c from one representation into several others depending on the component's configuration and settings. On receipt of event f the transformation starts and produces the first new representation of x which is subsequently published as attribute of event g by the component container. Afterwards, transform is called again to recode x producing the next representation to publish. This cycle is repeated until the handler finishes without recoding x. Hence, the out-port field c is not modified and does not require the publication of any state change. However, as event handlers for published notifications are rather unusual, we require developers to explicitly annotate those handlers responsible for out-ports.

Therefore, the validate handler (line 13) without any port declarations is only called once for the received event f at the begin of the transformation cycle.

5.3.3 Active Components

The majority of event-driven components is reactive, i.e., they react to received event notifications, process them, and may produce new notifications for other components to react on. For this to work, however, there must be sources which create initial event notifications. In general, we call these components *active* as they autonomously produce new notifications. Thereby, we distinguish two types of active components. Some components periodically produce new notifications, e.g., the latest sensor readings published every minute. Other components, in contrast, only sporadically publish new events, e.g., the latest user input just entered. Both types require different programming abstractions.

Definition 35 (time-triggered component). *A* time-triggered component *is a component whose handler methods are executed periodically triggered by the global progression of time.*

Supporting *time-triggered* components is easy and inexpensive in terms of required changes to the regular event handling process as described in the previous section. In fact, instead of calling event handlers on receipt of notifications, the handler methods of a time-triggered component are simply executed on timer alerts. If a timer expires, an internal notification is created that causes associated handlers to be executed. Please note that we do neither guarantee nor aim at ensuring any real-time behavior. Hence, annotated priorities are considered only to decide which handler to run first if two or more handlers are associated with the same timer. The component container runs one handler after the other and even considers event handlers that are triggered by published notifications on out-ports before processing the next timer event in order of their expiration.

Figure 5.3 shows a component listing exemplifying the usage. The component is responsible to monitor a sensor collecting environmental data. The method sample (line 11) is annotated to be triggered every quarter of an hour (line 10). It determines the current sensor reading and uses the setValue method (line 7) to map the absolute reading into a relative value ranging from zero to one. This value is assigned to the component's single out-port field which is inspected by the component container afterwards. Usually, the value would be published as attribute of event e only on a change, i.e., if the current value is really different to the one calculated from the previous sensor reading. On the one hand, this corresponds to the definition that an event embodies a significant change in state. Moreover, it saves many notifications to be send. On the other hand, there are numerous applications that rely on a continuous stream of event notifications. For example, the receipt of a new sensor reading also proves independent of its current value that the particular sensor is still alive and working. In order

```
@Active                                             // active component
@OutPort( event = "e", attributes = "value",        // with out-port and
          policy = OutPort.PUBLISH_ALWAYS )         // publishing policy
public class ActiveComponent extends Thread {
  @Attribute float value;                           // out-port field
  private int min, max;
  synchronized void setValue(int reading) {         // synchronized access
    value = (float)(reading-min) / (float)(max-min);
  }
  @OnTimer( "900s" )                                // time-triggered
  void sample() {                                   // handler method
    ...
    setValue(reading);
  }
  synchronized void calibrate(int low, int high) {
    min = low; max = high;
    sample();                                       // inform container
    notify();                                       // about new value
  }
  @Override void run() {                            // own thread
    while( true ) {                                 // of execution
      ...
      calibrate(low, high);
} } }
```

Figure 5.3: Active component featuring a time-triggered handler method as well as an own thread of execution.

to support such kind of applications, we thus enable components to specify the precise publishing policy to follow for each out-port separately. Likewise, event handlers that process received notifications may also be called for each notification received or only if the received notification contains attributes with different data values.

Definition 36 (port policies). *The following* port policies *specify for each event port separately when to derive and publish new event notifications and when to call associated event handlers:*

(*i*) *The* publish-on-change *policy creates a new event notification from out-port fields only on a state change, i.e., assigned values differ from those published in a previous notification. This is the default policy.*

(*ii*) *The* publish-always *policy creates a new event notification from out-port fields whenever all event handlers finished their execution independent of any state change.*

(*iii*) *The* call-on-change *policy triggers event handlers associated with an event port only on a state change, i.e., attribute values contained in the event notification differ from those previously assigned to the port fields.*

(*iv*) *The* call-always *policy triggers event handlers associated with an event port independent of any state change. This is the default policy.*

Port policies enable component developers to precisely specify for each event port separately when to create new event notifications or to trigger associated event handlers. Thereby, developers can customize the container's event handling process and adapt it to their specific needs. However, publish-always and call-always policies have to be combined cautiously. Together with an event handler associated with an out port, processing loops may be created that do not terminate and stop producing new notifications. In contrast, the default policies publish-on-change and call-always do not cause this risk while being suited for the majority of event-driven applications. In the listing shown in Fig. 5.3, PUBLISH_ALWAYS is specified as policy for the component's out-port e (lines 2 and 3). This makes sure that a new notification containing a fresh sensor reading is published by the component container whenever the time-triggered sample method (line 11) was executed. For other port policies, similar symbolic constants, namely PUBLISH_ON_CHANGE, CALL_ON_CHANGE, and CALL_ALWAYS, have been defined as bitmasks that can be combined where appropriate.

As demonstrated by the example above, time-triggered handlers and event port policies are well suited to conveniently implement periodic tasks that regularly produce new event notifications. However, certain components just sporadically publish new events as they, for instance, depend on user interaction that is hardly predictable. Usually, those components have own threads of execution that produce or acquire the data to publish. Hence, they are really *active* in the narrow sense of the word.

Definition 37 (active component). *An* active component *is a component that features own threads of execution.*

Because of concurrent threads and processes, active components are more complex to host than other event-driven components. Primarily, component threads and component container have to inform each other about available data that is ready to be published or to be processed as well as to synchronize their access to corresponding event port fields. For both, we use the standard synchronization and coordination mechanisms provided by Java. Besides featuring time-triggered handlers to periodically publish sensor data, the component listed in Fig. 5.3 also manages its own thread of execution. In fact, it directly extends the Java Thread class (line 4) and, thus, is annotated as being @Active (line 1). This informs the container to first acquire the component's monitor lock before accessing any event port fields. Likewise, the component has to use synchronized methods or program blocks to read from or write to those fields, e.g., the method setValue (line 7) is synchronized[2] because it converts a raw sensor reading and assigns the result to the out-port field value in order to get published (line 8).

The component's thread continually executes the main loop (lines 21–24) within the run method which is, among other things, responsible to recalibrate the sensor every once in a while. This may be necessary, for example, to adapt to changing environment conditions or may simply be carried out when requested by a user. The calibration process readjusts the lower and the upper bound for valid sensor readings which are subsequently set by the calibrate method (line 15). Furthermore, the calibrate method requests a fresh sensor sample (lines 11 and 17) whose absolute reading is mapped to its relative value using the new minimum and maximum bounds. Since the new sample is taken outside the time-triggered regular schedule, the container needs additionally to be notified that there is a new sensor value to publish. For this purpose, Java's notify operation is used that wakes the container thread waiting at the component's default monitor lock (line 18). The container thread inspects the out-port fields at the next opportunity and asynchronously publishes a corresponding notification. Meanwhile, the component thread continues execution as it does not wait for the container to complete the value's publication. Nevertheless, the container notifies the component in the same way when the notification was successfully sent. Thereby, it is possible to ensure that out-port fields were published before the component thread assigns new values. In our case, however, this is not necessary as we just want a fresh sensor reading to be published at all. Please note, for the opposite direction, regular event handlers can be used to inform component threads about available data to be processed on in-ports. The handler is called by the component container on receipt of a new notification and can then signal the responsible component thread in any appropriate way.

5.3.4 Dynamic Subscriptions

To facilitate component reusability, business logic needs to be separated from configuration data. As the exact system configuration is usually not known until deployment, it is, thus, primarily the responsibility of the system administrator to organize event flows and subscribe components to those notifications they are supposed to process. Yet as always, there are exceptions to this rule. If components, for instance, are closely related to each other and jointly provide a particular service they are usually bundled in a common scope. In this case, it is rather the developer who organizes the communication within the scope, while the system administrator configures and customizes the scope as a whole and deploys the component bundle at once. In other cases, components even autonomously manage their own subscriptions. Many *context-aware* components

[2] Method calibrate is also synchronized, but for a different reason. It may, otherwise, modify the minimum and maximum bounds for valid sensor readings just in the moment another thread is converting a sample leading to undefined results.

adapt or refine their subscriptions according to the current situation or present environment conditions. A location-aware component, for example, may be interested in events occurring in its proximity. Hence, its subscription depends on its current position and needs to be updated whenever the component moves. In this example, subscription management is actually an inherent part of the component's logic.

As these examples above demonstrate, there are, in fact, many reasons why developers want to actively manage and organize the subscriptions of their components. We thereby distinguish between *static* and *dynamic subscriptions.*

Definition 38 (static vs. dynamic subscriptions). *A subscription is called* dynamic *if its filter expression determining matching event notifications is modified and updated during the runtime of the subscribing component. Otherwise, the subscription is* static.

Many publish/subscribe systems allow or require components to specify the notifications they are going to publish. Depending on the employed routing algorithm this either increases system efficiency or is even a precondition for matching published notifications against active subscriptions at all. However, components may change the kind of notifications they publish during their lifetime. Likewise, we can, thus, distinguish between *static* and *dynamic advertisements.*

Definition 39 (static vs. dynamic advertisements). *An advertisement is called* dynamic *if its filter expression specifying event notifications to be published is modified and updated during the runtime of the producing component. Otherwise, the advertisement is* static.

Static subscriptions and advertisements are simpler to manage and to handle than their dynamic counterparts. Basically, it is sufficient to add a filter expression to an event port that specifies the notifications being received or sent. If the filter is added to an in-port, it is interpreted as a static subscription determining the event notifications in which the component is interested in. If the filter is associated to an out-port, it serves as static advertisement describing the notifications the component produces on the particular port. For the component container, however, there is not much difference whether the filter expression is annotated by the developer during implementation or it is provided by the system administrator at deployment. On component creation, the container inspects component annotations as well as configuration files for subscription and advertisement filters. Thereby, annotated filters can be subsequently refined or restricted by configuration settings.

The management and handling of dynamic subscriptions and advertisements is more complex. In particular, components need a way to dynamically create and modify the filter expressions that are used for subscriptions and advertisements. For this purpose, we extend the event port concept that associates component

```
 1 @InPort( event = "e" , attributes = "position , info" )  // event ports
 2 @OutPort( event = "f" , attributes = "info , distance",  // with filters
 3          filter = "distance > 0 && distance < " + MAXDISTANCE )
 4 public class LocationAwareComponent {
 5   public static final double MAXDISTANCE = 50.0d;       // filter constant
 6   @Attribute Position position;                         // event port fields
 7   @Attribute String info;
 8   @Attribute double distance;
 9   @Filter("e")                                          // dynamic in–
10   Filter proximity;                                     // port filter
11   @OnEvent("e")                                         // event handler
12   void calcaluate() {                                   // for in–port
13     // calculate distance
14     distance = ...
15   }
16   @OnTimer("60")                                        // time–triggered
17   void update() {                                       // handler method
18     // update proximity filter
19     proximity = ...
20 } }
```

Figure 5.4: Location-aware component managing a dynamic subscription.

fields with event attributes. Likewise, we also store the filter expression specifying the notifications to receive or to publish on an event port in an associated component field. Thus, the filter can be conveniently accessed and modified by the component. Moreover, this approach seamlessly fits into the regular event handling process as described in Definition 34 in Sect. 5.3.2. After the last event handler successfully finished its execution in step (iv) and before the first out-port field is inspected for new values to publish in step (v), all component fields with filter expressions are also checked for changes by the component container. In the event of a change, the corresponding subscription or advertisement is updated depending on whether the filter belongs to an in-port or an out-port, respectively. However, just a few publish/subscribe systems support the direct update of active subscriptions distributed throughout the broker network. For the others, we map the filter update to a sequence of subscribe or advertise operations, respectively. To update an in-port's subscription, the new filter is first subscribed before the subscription of the old one is revoked. Similarly, to update an out-port's advertisement, we first advertise the new filter before revoking the old one. Thereby, the completeness of notifications received from or published for other components remains guaranteed.

Figure 5.4 shows the listing of a location-aware component managing its own dynamic subscription. The component has two event ports. On its in-port (line 1) it receives events that provide information (attribute info) about as well as the exact location (attribute position) of occurred happenings and incidents within the component's proximity. The component calculates the distance (attribute distance) between its current position and the incident's location. The result is then republished on the component's out-port together with the incident's description (line 2). The surrounding area for which incidents are reported is limited by a maximum distance (constant MAX_DISTANCE) defined within the component (line 5). Thus, we can exploit this fact to better specify the event notifications going to be published by the component as we know that the values of the distance attribute (line 8) must lie between zero and this constant. For this purpose, event port annotations provide a filter element to add predicates restricting and refining the range of valid attribute values (line 3). Please notice the usage of the constant MAX_DISTANCE in the component's out-port annotation in order to create the filter expression for distance values. On component activation, the component container creates a static advertisement for this out-port based on the annotated filter expression when supported or required by the employed publish/subscribe routing algorithm. If out-ports are not annotated with a filter, nonetheless, a static advertisement will be created. However, the advertisement will just list the notification attributes without restricting their values. Likewise, static subscriptions are created for in-ports.

To receive notifications about incidents, the component subscribes to all events occurring within a given radius around its current position. Hence, the radius must be greater or equal to the maximum distance for which incidents are republished. Furthermore, a static subscription is not sufficient anymore as the filter must be updated whenever the component moves. Therefore, the @Filter annotation allows to mark component fields that contain notification filters to be used for dynamic subscriptions. In the component listing in Fig. 5.4, the proximity field (lines 9 and 10) is thereby turned into a dynamic subscription for events e to be received on the component's in-port (line 1). For this to work, the component field must either be a character string containing a valid filter expression or implement the infrastructure's Filter interface. In the latter case, it is, thus, even possible to use customized and advanced filters provided that they are supported by the underlying broker network. The component's proximity filter is updated by a time-triggered handler that is called every minute (lines 16 and 17). The handler determines the current position and creates a new location-based filter for the component's proximity (line 19). After the handler finished execution, the filter is inspected by the component container. If it is different to the previous one, the component's subscription is updated.

5.4 Scope Management

As we allow event-driven components to dynamically create and administer their own scopes, we have to adequately support them in doing so. Brokers, or more precisely their scope plugins, provide a uniform management interface that bundles all necessary scope operations and management functions as described in

```
 1 @Scope( name = "finance", superscope = "divisions",   // scope specification
 2         selector = "division == 'finance'",           // with component
 3         in = "true", out = "false",                   // selector, event
 4         attributes = "info,context" )                 // filters, attributes
 5 public class FinanceDivision {
 6   @Attribute String info;                             // attribute fields
 7   @Attribute("context")
 8   Table parameters;
 9   @Filter(Scope.OUT)                                  // dynamic filter
10   Filter monitoring;
11   ...
```

Figure 5.5: Component with annotated scope declaration.

Sect. 4.5.3. However, the mere presence of a scope management interface does not automatically lead to a good and reasonable application design and structure. Developers need to be encouraged to properly use scopes to organize their event-driven applications when these grow in size and complexity. In the following, we, therefore, present programming abstractions that ease the definition and handling of scopes as well as facilitate their intended application to benefit most from them. Similar to the approach taken for regular event-driven components, we enable developers to conveniently annotate and label those parts of code that directly address scopes and their management. The provided programming abstractions include means for specifying and defining new scopes, for administering scope members as well as for instantiating further subscopes.

5.4.1 Scope Specification

For each scope there is a component responsible that is responsible to provide the scope's specification, to register the scope at a broker in order to open and to close it as well as to manage scope membership and members. Thereby, such a scope component may be responsible for several scopes. In fact, the number of scopes a component may open and manage is not limited. With regard to modularization, however, scope components are usually dedicated to just a single scope. We foster this practice and enable developers to conveniently specify a scope together with its managing component. Figure 5.5 gives an example showing the source code of a scope component with corresponding *scope specification*. In this example, the component defines and manages the scope finance which encloses all event-driven components that belong to the finance division of an international company. Thus, the scope may be part of the example scope hierarchy introduced in Sect. 4.2.2 and depicted in Fig. 4.2.

The purpose of the @Scope annotation (line 1) is twofold: first, it declares and specifies a new scope and, second, makes the annotated component responsible for managing it. The @Scope annotation consists of several elements each addressing an individual aspect and part of the scope's specification. The name and superscope elements refer to the name of the scope and its parent scope, respectively. Together, they uniquely identify the new scope in the hierarchy. In this example, thus, the complete name is divisions.finance. Please note that the superscope element is optional. In order to separate code from configuration it can be set later or overridden at deployment. However, if it is not provided, the root scope is assumed as default superscope.

The selector, in, and out elements allow the specification of a Boolean filter expression. The selector element (line 2) defines the scope's component selector which brokers apply on the attributes of hosted components to determine whether the scope is assignable. Please refer to Definition 13 and 28 for details about the component selector function and the scope assignment process, respectively. In the provided example, the selector filter evaluates to true for all components having the attribute division set to finance. Hence, all finance components are automatically incorporated into the finance scope. Elements in and out (line 3) refer to the corresponding filter functions of the scope's notification interface. Only matching event notifications are allowed to cross the scope border and enter or leave the scope, respectively. Please see Definition 14 for further details. Here, all external notifications are allowed to enter the scope without any restrictions while own notifications published by scope members must not leave the scope. Elements selector, in, and out are optional and are assumed to be false by default when not provided.

Scope attributes that further characterize scopes and their members need to be declared in the scope specification. This is done by the optional attributes element (line 4) which enumerates all associated attributes separated by comma. In the given example, the defined scope has two attributes: info may provide a short description of the scope while context may contain parameters and settings applicable to all scope members. Please note that only attribute names are listed within the @Scope annotation. Type information and attribute values are taken from corresponding component fields. Similar to event ports, scope attributes are bound to component fields using the @Attribute annotation. Thus, the scope's info attribute is bound to the component field of the same name (line 6) while the context attribute is associated with the parameters field (lines 7 and 8). Moreover, the component container handles scope attributes similar to out-port attributes. Hence, associated fields are regularly inspected for changes, for example, when components join or leave the scope or when notified by a component thread. On a detected state change the new attribute values are published as scope notification that is send to all scope members for update.

The update mechanism is not limited to scope attributes only. Similar to dynamic subscriptions and advertisements even the scope's interface filters and component selector can be updated dynamically. For this purpose, the filter needs to be stored in a component field that can be inspected by the component container. Furthermore, it has to be marked appropriately using the @Filter annotation. In the example listing, the monitoring field is labeled this way (lines 9 and 10). The constant OUT indicates that the field contains the filter for the scope's outbound interface, for example, to allow certain components and their notifications to be monitored externally. Likewise, constants IN and SELECTOR refer to the scope's inbound interface and component selector, respectively. Alternatively, the fields itself may be simply named out, in, and selector provided that there is no name clash with other attributes or event ports. Please note that the outbound filter is defined twice in this example. First, it is given as static filter expression within the @Scope annotation (line 3) and, second, it is dynamically set from the monitoring field marked by the @Filter annotation. However, the dynamic filter prevails over the static one unless the filter is not valid (e.g., the field is not initialized or contains an inappropriate filter type). In this case, the static filter is used as fallback and stops all notifications when leaving the scope.

5.4.2 Scope Membership

Event-driven components that manage their own scopes are also responsible to administrate scope membership and members. They approve or decline the admission of new scope members including components as well as brokers. This way, it is possible to restrict the dissemination of event notifications to eligible components and even implement sophisticated access control policies. We provide programming abstractions to support developers in managing their scopes and, in particular, to conveniently handle and administer new and existing components *joining* and *leaving* the scope, respectively. The central idea is to consider requests to join the scope as well as notifications about leaving components as events to be processed. For these, however, we can build on available concepts such as event ports and event handlers that we introduced in Sect. 5.3. Figure 5.6 exemplifies the approach and continues the listing from Fig. 5.5.

Although we build on event ports for passing parameters we do not have to explicitly declare one. This is implicitly done when making a component responsible to manage a scope by adding a scope annotation. Thus, we can directly annotate component fields to hold information about joining or leaving scope members. For example, the component field member (line 13) contains identities and, if provided, credentials of components and brokers that want to join or leave the scope. Please note that a regular @Attribute annotation together with the constant MEMBER_INFO is used to mark the field appropriately (line 12). This way, arbitrary other scope relevant as well as context information can be injected into the component.

The injected data and information are required by event handlers to approve or deny a component request to join the scope. In the example, the methods check and add (lines 16 and 22) are annotated as event handlers responsible to process join requests. For this purpose, regular OnEvent annotations are used combined with the constant JOINING (lines 14 and 21) reserved for scope

```
11    ...
12    @Attribute( Scope.MEMBERINFO )                     // info about joining or
13    MemberInfo member;                                 // leaving scope member
14    @OnEvent( Scope.JOINING )                          // high priority handler
15    @Priority( Priority.HIGH )                         // to check a new member's
16    void check() {                                     // authorization
17      // check component's authorization
18      ...                                              // abort and deny access
19      if (!authorized) throw new ScopeException();     // via exception
20    }
21    @OnEvent( Scope.JOINING )                          // handler for joining
22    void add() {                                       // scope members
23      // add component to members
24      ...
25    }
26    @OnEvent( Scope.LEAVING )                          // handler for leaving
27    void remove() {                                    // scope members
28      // remove component from members
29      ...
30  } }
```

Figure 5.6: Component with annotated scope management handlers.

management. By assigning the check method a higher priority (line 15) than the add method[3] the execution order of both handlers is determined unambiguously. The check method is called first and verifies if the requesting component as well as intermediate brokers along the network path are authorized to join the scope. If the component or at least one broker is not authorized the event processing is aborted by throwing a ScopeException (line 19). The component container catches the exception and interprets it as a denial to join the scope which is published as a negative scope notification and routed towards the requesting component. Hence, the subsequent add method is only executed if the authorization check was successful. In order to keep track about scope members the add method maintains a list of joined components to which the requesting component is added. Please note that maintaining an own member list is not required for managing a scope. It just demonstrates a possible application and benefit of having multiple event handlers processing a join request. If all associated event handlers terminated successfully the component container publishes the approval to join the scope as scope notification which is then routed towards the requesting component.

The method remove (line 27) is responsible for components leaving the scope. It is made an event handler using the OnEvent annotation together with the scope-specific constant LEAVING (line 27). Contrary to the add handler discussed

[3] The default priority of event handlers is NORMAL when not annotated otherwise. Please refer to Sect. 5.3.2 for details.

above, remove simply deletes the component leaving the scope from the list of members. For this to work, the field member (line 13) contains the identities of the component as well as intermediate brokers this time that together unsubscribed the scope membership. Please note that, contrary to a join request, it is not possible to stop components or brokers in leaving the scope by raising an exception. In fact, the particular component may not be present or available anymore, e.g., if it has crashed and the scope unsubscription is issued by the hosting broker on behalf of its component.

5.4.3 Scope Instantiation

Scopes provide flexible means to structure publish/subscribe systems and organize event-driven applications. They enable application developers and system administrators to adequately model and represent architectural, functional, or organizational structures within applications and the broker network to ease their development and maintenance. For example, a distributed event-driven application may be split into several functional modules each embedded into a separate scope. Likewise, a company may be divided into different divisions with each of them enclosed in an individual scope. Thereby, it is often the case that many scopes do not substantially differ from each other. In fact, module scopes may have the same basic structure and division scopes may share the same attributes that just differ in assigned values. Nevertheless, each scope needs to be defined individually so far which includes interface filters, scope attributes as well as their values. This is tedious and cumbersome. Instead, it is beneficial and much more convenient to have some kind of scope classes or templates defining a scope's structure that can be easily *instantiated* and *configured* on demand. Moreover, this can even be done automatically. In fact, we have already established necessary foundations and prerequisites by allowing scope specifications to be annotated to the Java classes of their managing components. Hence, we just need to precisely define how annotated scopes can be instantiated and configured appropriately.

Definition 40 (scope instantiation). *Based on the scope annotations added to scope components the component container is able to automatically create a new scope instance when (i) a scope component is connected to its hosting broker, (ii) another scope is created, or (iii) an application component is added. Requirements, conditions, and details are as follows:*

(*i*) *A scope component is connected to its hosting broker and successfully joins the parent scope as determined by provided annotations. Furthermore, there is no other active scope of the same name advertised within the parent scope. Then the component container creates and opens the corresponding scope and activates the newly connected scope component.*

(*ii*) *A new scope is opened which has another scope previously configured to be its subscope. Furthermore, the configuration identifies the subscope's man-*

```
@Scope( name = "divisions",                   // scope declaration with
        selector = "division == '*'",         // component selector and
        attributes = "info" )                 // one scope attribute
class Division {
  @AutoConfigure                              // attribute automatically
  @Attribute String info;                     // configured by container
  @Autoconfigure("out-filter")                // outbound filter also auto-
  @Filter(Scope.OUT) Filter monitoring;       // matically configured
  @AutoCreate                                 // default constructor for
  Division() {                                // automatic instantiation
    // initialize component
    ...
  }
  @AutoCreate("division")                     // constructor for automatic
  Division(String division) {                 // instantiation of subscopes
    // initialize component                   // for new divisions
    ...
} }
```

Figure 5.7: Annotated constructors to automatically create subscopes.

aging component providing a marked constructor for instantiation. Then the component container creates an instance of the subscope's managing component, sets the appropriate parent scope, and connects it to the hosting broker. Subsequently, the particular subscope is derived according to (i).

(*iii*) *An application component successfully joins a scope that is previously configured to group all members into subscopes based on a particular component attribute. The constructor to instantiate subscopes is provided and there is no active scope that corresponds to the value of the components attribute yet. Then the component container creates a new scope component using the attribute value as scope name and connects it to the broker. Subsequently, the particular subscope is derived according to (i).*

In case (i) of the definition above the scope specification is simply extracted from component annotations when the scope component is connected to the broker. The scope component itself, however, needs to be previously instantiated. Cases (ii) and (iii) go beyond that. Here, the component container creates the new scope as well as its managing scope component provided that an appropriate configuration and constructor functions are given. The listing shown in Fig. 5.7 revises the finance scope specification introduced in Sect. 5.4.1 to make the scope component automatically instantiable on demand.

An international company may be divided into divisions, but may also be structured according to the regions and markets in which it is active. In fact, a company's organization is usually based on a mixture of these criteria. Thus, when creating a new company scope, it makes sense to create subscopes these organizational aspects, too. The annotation @AutoCreate (lines 9 and 14) marks constructors that can be safely used by the component container to create new component instances on demand. In order to create a scope that contains all company divisions the no-argument constructor (line 10) applies. It instantiates a scope component specifying a scope named divisions (line 1) that is responsible for all component members having a division attribute regardless of its value (line 2). Whenever a new company scope is established, it makes also sense to automatically create a scope responsible for the company's divisions. In fact, this scenario corresponds to case (ii) of the scope instantiation process.

The general scope covering all company divisions is usually further subdivided so that each division has its own subscope. Moreover, such subscopes may be instantiated automatically by the component container whenever a new division is established within the company. For this purpose the second constructor is used whose @AutoCreate annotation is constrained by the division attribute (lines 14 and 15). If an application component has a division attribute for which no corresponding subscope exists yet, a new scope component and subscope is instantiated. Therefore, the attribute's value is passed as argument to the constructor of the scope component. Furthermore, the newly instantiated subscope is named after the attribute and the application component is grouped into it. For example, if the application component has its division attribute set to finance for which no subscope already exists a new one named divisions.finance is created with the application component as member. In fact, annotating constructors this way allows for automatically grouping application components into own subscopes according to a particular attribute. This is enabled by case (iii) of the scope instantiation process.

Automatically created subscopes have the same structure. For instance, all division subscopes have an info attribute (line 6) as well as a dynamic filter for monitoring (line 8). However, they may differ from each other by the actual values and expressions assigned to attributes and filters, respectively. In fact, the info text for the finance division may be different to the text of the marketing division. When instantiating a new scope component, therefore, the name of the scope to be created is also passed to the constructor so that scope attributes and filters are initialized appropriately. However, this usually leads to a mixture of code and configuration which we effectively want to avoid. Hence, an improved approach is needed in which the component container additionally configures the scopes and scope components it instantiates. The @AutoConfigure annotations (lines 5 and 7) are intended for this purpose. They mark component fields and values that are specific for each individual scope instance and are, thus, subject to configuration. Here, the scope attribute info and the monitoring filter for outgoing notifications are configured this way. After creating a new scope component the component container looks up the marked fields in its configuration for this component class and sets their values appropriately. Please note that for the scope's outbound filter the identifier out-filter (line 7) provided by the annotation is used instead of the field's name.

Definition 41 (field initialization). *The component container assigns initial values to annotated fields of automatically created components that are looked up in a name/value list within the container's configuration. The lookup key is formed by the complete name of the scope in the hierarchy extended by the annotated identifier or the field's name if the former is not given. The list may contain multiple entries matching the key from which the most specific is taken.*

It is rather unusual to have multiple entries matching a single key. But it allows for flexible configurations in which one entry applies to multiple scopes and components. Thereby, it is possible to provide a general default value that can be overridden by a more specific entry. In particular, we may use the wildcard symbol ? to denote an arbitrary name component, while * denotes an arbitrary part of the scope hierarchy. Then, `*.divisions.?.contact` denotes a general contact address for all division subscopes while `*.divisions.finance.contact` overrides the address for the finance scope as the entry is more specific (i.e., more name components are specified without wildcards). Please note that the usage of wildcards is just one possible way of implementation. The definition does not prescribe any specific implementation and explicitly leaves open how the lookup is realized in detail. Naming and directory services as well as databases may be used for this purpose. Alternatively, simpler internationalization and localization techniques are also applicable that manage key/value pairs and provide adapted and translated resources (e.g., error messages, menu items) according to the language, country, and region of the user. Please further note that the `@AutoCreate` and `@AutoConfigure` annotations are not limited to scopes and scope components only. They are also applicable to regular event-driven components in order to let these automatically instantiated and configured by the component container.

5.5 Component Orchestration

In the previous sections, we have introduced programming abstractions that ease the development of event-driven components as well as the management of scopes to organize systems and applications. Based on these foundations, we are now ready to put together the various building blocks to form and design comprehensive solutions. In particular, this comprehends creating and customizing event-driven components, grouping them into scopes and subscopes, and orchestrating their event flows. Collectively referred to as *component orchestration*, this is an important and essential part of the system configuration.

System configurations are usually provided in external files or, sometimes, in databases that are parsed or queried on start up, respectively. Thereby, internal data structures are build up that hold and represent the configuration from then on. Alternatively, it is also possible to offer an interface for manipulating these data structures directly. This way, developers are able to program the system configuration, too. In fact, this has several practical benefits, for instance, the same tools and development environments used for implementation can also be

```
@Configuration                                          // configuration class
class AlertingConfiguration {
  @Configuration                                        // configuration method
  void configureComponents() {
    Component c = Component                             // proxy reference for
      .of(Alerter.class)                                // alert component
      .where(Value.of("id").is("42"));                  // specified by its id
    c.set(
      Value.of("application").to("alerting"),           // adding/changing
      Value.of("type").to("alert") );                   // component attributes
    c.subscribe(                                        // subscribing in-port
      Port.in("items").with("type = invoice")           // to invoice events
          .mapping("article,price")                     // renaming attributes
          .to("item,value") );                          // article and price
    c.advertise(                                        // advertising out-port
      Port.out("alert").with("type = alert")            // alert notifications
          .mapping("text").to("description") );         // renaming attribute text
  }
  ...
```

Figure 5.8: Customizing component attributes and event ports.

reused for configuration. Additionally, the compiler checks the syntax of the configuration for free and provides meaningful error messages where necessary. In this section, we follow the latter approach to present the configuration's primary elements and concepts. Besides the benefits already mentioned above there are two main reasons. First, it saves us from introducing an own configuration format or configuration language. Second, the interface operations also allow us to write complex configurations in a compact and concise style and form that is still quite intuitive to read and understand.

5.5.1 Customizing Components

When connecting prefabricated components and gluing them together to meaningful applications, it is often necessary to *customize* and *adapt* some components first in order to make them fit. Primarily, this includes changing component attributes and subscribing their event ports to the event notifications to process. This is a major part of the system configuration for which we also provide programming abstractions. Instead of introducing a new configuration language or format, we provide a programming interface enabling developers to directly manipulate the component container's configuration. The listing in Fig. 5.8 exemplifies how such a programmed configuration looks like. The idea is to realize an alerting service that monitors a company's accounting system and warns in case of abnormal or suspicious expenses.

The whole system configuration can be split up into multiple class files. In particular, we recommend to create one per application. The @Configuration annotation (line 1) is used to mark those classes that contain configuration logic which itself can be subdivided into several configuration methods. Likewise, configuration methods need to be appropriately annotated, too (line 3). Similar to event handlers (cf. Sect. 5.3.2), it is possible to additionally assign priorities to enforce a particular execution order. However, this is not necessary if configuration methods do not override each other's settings. They are executed by the configuration container when a corresponding configuration object instance is provided together with a new application component.

Within a configuration method, it is first necessary to specify the component to configure. This is done by obtaining a proxy reference using the static of method of the Component class (lines 5 and 6). The returned proxy initially refers to all instances of the specified Alerter class. Hence, all settings made when using it are valid for all component instances of this class and its subclasses. However, it is possible to narrow the proxy reference to individual instances. Therefore, the where method allows to specify additional constraints (line 7). Here, the reference for the Alerter class is narrowed to a single instance specified by the given id attribute.

Having a proxy reference, it is then easy to change existing or add new component attributes using the set method (line 8). The method takes one or more name/value pairs as arguments that are added as component attributes replacing already existing ones. Name/value pairs can be constructed using the Value class. The attribute's name is provided to the static of method while the attribute's value is given to the is or to method. Of course, there are alternative and easier ways to create name/value pairs of which providing a simple constructor taking both arguments is one of them. Introducing the Value class, however, allows for more verbose and literate configurations that we want to make as self-explaining as possible. Here, the component attributes application and type are set to the values alerting and alert, respectively (lines 9 and 10).

Besides setting component attributes, the configuration also specifies event flows between components. Basically, this means subscribing components to event notifications they have to process while advertising those they produce. For this purpose, corresponding subscribe and advertise methods (lines 11 and 15) are provided that take as argument one or more in-port or out-port specifications, respectively. In particular, it is, to be more precise, the event ports of the component that are subscribed or advertised. An event port specification is created using the Port class. The static in (line 12) and out (line 16) methods create a named in-port or out-port specification, respectively, that refers to the component's port of the same name. The with method provides the notification filter or the filter expression used for component subscription or advertisement. Here, the component's in-port item is subscribed to invoice notifications identified by their type attribute (line 12). Notifications from the component's out-port alerts are advertised as having the type alert (line 16).

```
19    ...
20    @Configuration                                   // configuration method
21    void configureScopes() {
22      Scope s = Container.create(                    // create scope instance
23        Scope.of(DefaultScope.class) );              // and obtain proxy
24      s.set(
25        Name.to("alerting"),                         // customize name,
26        SuperScope.to("company.finance"),            // superscope,
27        Selector.to("application = 'alerting'"),     // component selector,
28        InFilter.to("true"),                         // interface filters
29        OutFilter.to("type = 'alert'"),              // and add a new
30        Value.of("loglevel").to("debug") );          // scope attribute
31  } }
```

Figure 5.9: Customizing a scope specification.

However, when building applications from various components, we cannot expect that a consistent naming scheme is used by all of them. This is especially true as components may come from different vendors whose developers do not know the application context in which their components are deployed later. Thus, event port attributes do often not correspond to the attribute names of the notifications they have to process and, therefore, need to be mapped and renamed appropriately. Attribute mappings are specified using the mapping method combined with the to method. Attributes listed as argument to the mapping method are mapped, in the same order, to the names given to the to method. Thus, in the example of Fig. 5.8, the notification attributes article and price are mapped to the in-port attributes item and value (lines 13 and 14) while the attribute of published alert notification on the component's out-port is renamed from text to description (line 17).

5.5.2 Customizing Scopes

Similar to application components, it is often necessary to *customize* and *adapt* scopes, too. This usually includes defining their name and position within the scope hierarchy, adding or changing scope attributes as well as specifying and fine-tuning component selectors and interface filters. In fact, the whole scope specification is configurable. The listing in Fig. 5.9 gives an example continuing the application configuration started in Fig. 5.8. All settings regarding scopes are bundled within a specific configureScopes method (line 21).

First and foremost, we have to specify the scope we want to configure. Similar to application components, this is done by using the static of method of the Scope class to obtain a proxy reference to the appropriate scope (line 23). More precisely, it is a reference to the class of the scope's managing component that contains the annotated scope specification and still needs to be narrowed to address a single scope instance only. Therefore, we may add further constraints using the where method/clause to uniquely identify the scope component of interest. In fact, we have already taken this approach to successfully configure the application's Alerter component in the previous section. Nevertheless, we still had to create the component instance manually. Alternatively, the component container can do this automatically. For this purpose, the Container class offers the static create method (line 22) taking a component class as argument and returning a proxy reference for the created instance for further customization. In the example, we thus configure the component container to create a new instance of the DefaultScope, which contains no application components yet and does not allow any notification to pass its boundary. Please note that automatic instantiation is only possible if component and scope constructors are appropriately marked with @AutoCreate annotations (see Sect. 5.4.3 for details).

Having the proxy reference, we are able to adapt the DefaultScope specification as needed. Thereby, the set method (line 24) allows us to address and override each part of the scope's specification separately. However, the set method takes one or more name/value pairs as arguments and adds these as component attributes as described in the previous section. To address parts of the scope specification, we, therefore, need specific name/value pairs for which we provide own classes and constructor functions. In particular, the ScopeName, SuperScope, Selector, InFilter, OutFilter classes (lines 25 to 29) refer to the name of scope, its superscope, its component selector function, and the interface filters for incoming and outgoing notifications, respectively. All of them provide a static to method to create an appropriate name/value pair with the value set to the given argument. To add or change an ordinary scope attribute, the Value class and its methods (line 30) are used as before.

In the example listing, the created default scope instance is extensively customized and adapted. First, it is named alerting (line 25) and grouped under the top level scope finance (line 26). Furthermore, it is configured to encompass all components that belong to the alerting application. Therefore, its component selector checks whether the component has an application attribute that is set appropriately (line 27). The scope's interface is configured to make all external, incoming event notifications visible to scope members (line 28) while only alert notifications published by members are visible outside the scope (line 29). Finally, the scope attribute loglevel is added and set to debug (line 30) to make all scope members to also log debug information besides warnings and errors.

5.5.3 Grouping and Connecting Components

The programming abstractions introduced to customize components and scopes are sufficient to create and compose arbitrary system configurations. However, the more complex applications and infrastructure, the more extensive and comprehensive the system configuration becomes. In particular, it turns out to be

```
 1 @Configuration                                  // configuration class
 2 class DivisionConfiguration {
 3   @Configuration                                // configuration method
 4   void configure() {
 5     Scope division = Container.create(          // create divisions
 6       Scope.of(DefaultScope.class) );           // scope with subscopes
 7     division.set(Name.to("divisions"));         // for each division
 8     division.groupBy("division");
 9     Component monitor = Container.create(       // create monitor
10       Component.of(StockMonitor.class) );       // component and set
11     monitor.set(                                // division attribute
12       Value.of("division").to("sales") );       // to be assigned to
13     Container.insert(monitor).into(division);   // the sales subscope
14     Component manager = Container.create(       // create manager
15       Component.of(ProductionManager.class) );  // component and set
16     manager.set(                                // division attribute
17       Value.of("division").to("production") );  // to be assigned to the
18     Container.insert(manager).into(division);   // production subscope
19     Container
20       .connect(Port.out("warnings").of(monitor))  // connect event ports
21       .with(Port.in("demand").of(manager))        // of both components
22       .mapping("item,quantity,date")              // and rename several
23       .to("article,units,deadline");              // attributes
24 } }
```

Figure 5.10: Grouping components into scopes and connecting their event ports.

tedious and error-prone to manually define and manage filters and filter expressions to determine scope membership as well as the visibility of event notifications. To ease component orchestration, we, therefore, provide additional configuration methods that abstract from filter details and allow to directly specify how components are *grouped* and *connected*. The listing in Fig. 5.10 gives an example by automatically grouping two business components in their appropriate scopes and directing the flow of exchanged event notifications by simply connecting their event ports. The presented application implements an agile and event-driven production planning service that decides what article to produce on which production line based on current sales and the stock level in storage.

First, a top level scope is configured to be automatically created by the container from the default scope class (lines 5 and 6). It is used to structure the publish/subscribe system according to the company's divisions. In particular, the groupBy method is applied for this purpose (line 8). It groups application components according to their division attribute into corresponding subscopes. If a matching subscope does not exist yet, it is automatically created. This is possible if the scope component provides an appropriate constructor marked with the @AutoCreate annotation as described in Sect. 5.4.3. Thereafter, the container is configured to instantiate two application components. One component is created from the StockMonitor class (lines 9 and 10). It is responsible to watch the stock level in storage and publish an appropriate warning before a particular article is running out of stock. The monitor component belongs to the sales division. Hence, its division attribute is set accordingly (lines 11 and 12). The other component is created from the ProductionManager class (lines 14 and 15). It is responsible to manage the company's production lines. In fact, it plans and controls when which article is manufactured where and in what quantities. Therefore, the component belongs to the production division and has its attributes set accordingly (lines 16 and 17). After instantiation, both components are simply placed into the divisions scope using the configuration methods insert and into provided by the component container (lines 13 and 18). Moreover, as the scope is configured to group members according to their division attribute, the two components get automatically assigned to their corresponding subscopes. If necessary, a new sales or production subscope is created on demand.

Finally, communication between the stock monitor component and the production management component needs to be established. In particular, it has to be ensured that published warnings about low stock levels are delivered to the production manager in order to adept the production schedule and restock affected articles. Therefore, the configuration allows developers to directly connect the out-port of one component to the in-port of another component. In fact, such component connections are usually a central abstraction in dataflow approaches that ease application development and increase component reusability. For this purpose, we offer the configuration methods connect and with that take appropriate out-port and in-port specifications as arguments, respectively (lines 20 and 21). Port specifications are created using methods of the Port class. Methods out and in specify the port's type and name while method of determines the component instance or class. However, as event port attributes may not correspond to each other on both sides of the connection, an additional attribute mapping is often needed. This is defined per connection using the methods mapping and to (lines 22 and 23). Attributes listed as arguments to the mapping method are renamed to the identifiers given to the to method in order of appearance. In the example, the out-port warnings of the stock monitor is connected to the in-port demand of the production manager. Thereby, the attributes item, quantity, and date are mapped to article, units, and deadline, respectively.

Grouping scopes and components as well as connecting their event ports significantly eases their orchestration. In fact, the presented configuration methods above abstract from filters and filter management and, thus, enable developers to directly focus on the data flows between components. Behind the scenes, however, data flows are completely directed by subscriptions, advertisements and their filters. Moreover, all configuration methods eventually implement their functionality by using or adapting filter expressions. For example, the configuration methods insert and into that are used to directly assign components to a given scope in reality just manipulate the scope's component selector and expand its filter expression. In more detail, the implementation comprises two different parts. First, when obtaining a proxy reference for configuration, it is ensured that the referenced components are uniquely identifiable. Either the constraints provided by the `where` clause are used or a hidden `id` attribute is added if the component is automatically created by the container. Second, the provided constraints or the added `id` are incorporated into the filter expression of the scope's component selector. The same is done for the selectors of any parent scope. Hence, the component is grouped into the particular scope by the regular scope assignment process as described in Sect. 4.4.2.

A similar approach is used for connecting components and establishing event flows. First, we make event ports and their notifications identifiable by associating an additional `id` attribute to each port. This `id` is added to every notification published by the port. Thus, each event flow within the system can be subscribed using the `id` of the port it originates from. Second, we exploit this feature for implementing the configuration methods `connect` and `to` by subscribing the in-port of the receiving component to the notifications of the out-port of the publishing component using the `id` of the latter. Furthermore, scope interface filters need to be considered if both components are situated in different scopes. In particular, for each scope of the publishing component in which the receiving component is not a member the filter expression for outgoing events is adapted to allow notifications of the flow to leave the scope. Likewise, for each scope of the receiving component to which the publisher does not belong the filter expression for incoming events is adapted to allow the notifications to enter the scope. In both cases, the added port `id` specifies the relevant notifications.

Please note that connecting components this way also has side effects. When allowing notifications to pass scope boundaries, they also become subject to subscriptions of other scopes. Thus, besides the component to which the notification flow is connected, there may be other components that receive all or some of the notifications, too. Contrarily, it is also possible that notifications do not reach the receiving event port although publishing and subscribing component are connected to each other. This is the case if publisher or subscriber may dynamically be assigned to other scopes whose interface filters have not been adapted yet. In fact, only those scopes are considered that are addressed within the system configuration and from which is known that at least one of the components is a member, for instance, by explicitly assigning the scope using the configuration methods `insert` and `into`. Thus, great care has to be taken when mixing dynamic filters annotated to scopes and components with static configurations.

5.6 Implementation

The programming abstractions presented in the previous sections aim at easing and simplifying the development and orchestration of event-driven components and applications. Primarily, they relieve developers from cumbersome and boilerplate code often required to interact with the publish/subscribe middleware.

Introduced event ports, for instance, save developers from the necessity to manually inspect received notifications and extract the relevant data to process. Moreover, processing results get automatically and conveniently published by this approach, too. In fact, a large proportion of middleware-specific code is thereby factored out from application components. Thus, developers can better focus on the component's business and application logic. However, the functionality factored out from components does not vanish. It is shifted into the middleware where the presented programming abstractions need to be implemented. In this section, we give an overview about the general implementation, show its integration into the middleware architecture, and highlight relevant details how particular abstractions are realized. The implementation, thereby, is split into two different parts: a *component container* and a *component wrapper*. While the former provides general services and convenient functions, the latter ensures that application components can leverage the offered functionality. Both are discussed in the following.

5.6.1 Component Container

REBECA brokers are able to locally host event-driven components. This feature is implemented as plugin which can be added to brokers when required. Unlike other plugins, hosting components does not change the way how notifications need to be forwarded by the broker. Thus, the component plugin does not intervene or participate in the broker's message processing chain as other plugins such as the routing plugin or the scope plugin do. Nevertheless, the component plugin follows the general design pattern common for all REBECA plugins to be compatible with the broker's architecture. In order to support the programming abstractions presented, we revised the component plugin and extended its functionality. Following the idea of a tailored feature composition [166] brokers are free to use the regular component plugin or the revised version for supporting normal event-driven components or the full set of programming abstractions, respectively. The UML diagram in Fig. 5.11 gives an overview about the component plugin and its extensions.

The central part of the *component plugin* is the ComponentEngine. From its abstract parent class, it inherits all necessary properties and operations to be seamlessly plugged into a broker's processing infrastructure. As the ComponentEngine, however, does not need to modify any notifications, we simply stick to the inherited default logic. In fact, the default implementation of the process method immediately passes any message to the next engine in the broker's processing chain. Regarding the hosting of components, the ComponentEngine offers methods to plug and unplug local components. After plugging a component to the broker the broker's output is locally connected to the component's input and vice versa. Using this connection, component and broker can exchange subscriptions and matching event notifications. Unplugging the component disconnects broker and component again.

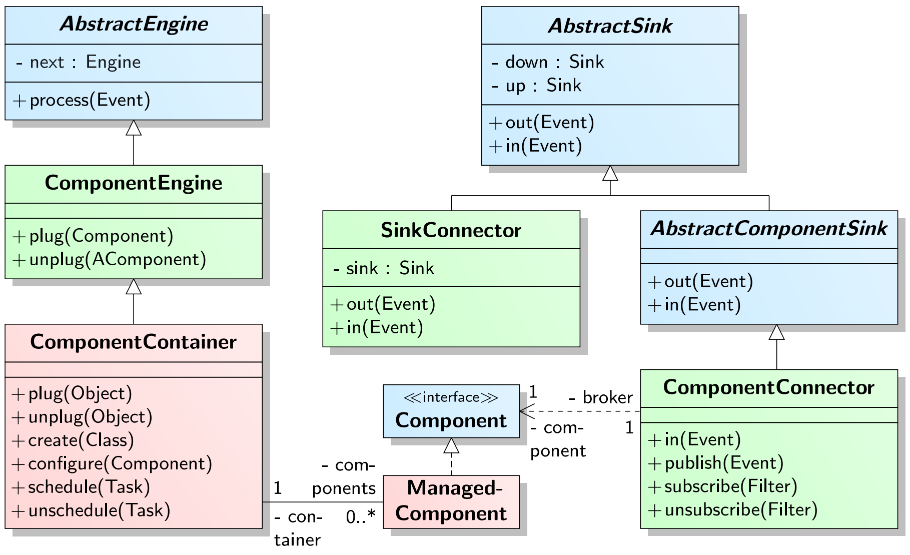


Figure 5.11: Overview of REBECA's component plugin and its extensions.

For each connection, broker and component possess an own sink chain that event notifications have to pass immediately after being receipt from or before being sent to the opposite side. Plugins may insert own logic into the sink chain on either side of the connection to modify or transform passing notifications. This way, for example, the serialization plugin converts event notifications into an appropriate format for network transport or the scope plugin enforces its visibility constraints. For plugin developers, the REBECA framework provides the abstract base classes AbstractSink and AbstractComponentSink as starting point. They simply pass incoming notifications (method in) upwards and outgoing events (method out) downwards to the next sink in the chain. Plugin developers are encouraged to extend these classes and overwrite their methods as needed. The component plugin itself ships with two new event sinks: the SinkConnector on broker side and the ComponentConnector on component side. The SinkConnector is responsible to directly connect the broker sink to the component sink. In particular, as components are hosted locally, it thus allows to bypass the serialization/deserialization of event notifications as well as their network transport, which both are usually quite expensive and costly. The ComponentConnector finally connects the component sink with the component itself. It eventually delivers incoming event notifications to the component and, in return, offers a basic broker interface with which the component can publish own events or subscribe and unsubscribe the notifications of other publishers.

For regular event-driven components, the publish/subscribe interface offered by the ComponentConnector is usually sufficient. To support all programming abstractions presented in this chapter, however, the component plugin yet lacks essential features and services. To add the required functionality we extended the plugin's engine to a full-fledged *component container* that manages its components and provides common services and functions. The ComponentContainer class first overloads its inherited plug and unplug methods to also accept those components that do not fulfill the conventional component interface, but use the introduced annotations to declare event ports and event handlers. Moreover, executable configurations that orchestrate these components are plugged this way, too. Additionally, the ComponentContainer also provides the operations create and configure to instantiate new components and configure these subsequently. Thereby, the new component is inspected and existing class annotations are evaluated in order to determine the part of the container configuration that is relevant for the respective component. This part gets attached to the component itself making costly reinspections unnecessary. Besides the lifecycle of components, the container also manages their repeating and periodic tasks. Therefore, the container method schedule enables components to register own time-triggered handlers and methods to be executed periodically while unschedule stops the respective timers again. Bundling timer management within the component container especially allows for more efficient implementations than having each component individually scheduling its own tasks.

Event-driven components using services and functions provided by the container are said to be *managed*. Although they also implement the common Component interface to publish and receive event notifications they depend on the component container for supporting all programming abstractions. In the next section, the ManagedComponent class is discussed in more detail.

5.6.2 Managed Component

Leveraging the presented programming abstractions, application developers can better focus on the component's business logic. In particular, it is usually sufficient to annotate methods and fields contributing to the applications functionality in order to just describe when and for which purpose they have to be used. Event ports, for example, relieve developers from writing boilerplate code to manually create notifications for the events to publish as well as to tediously inspect received notifications for relevant data. Instead, a notification's content and data are automatically taken from and provided by event port fields, which are directly accessible from within the component and, thus, significantly ease the development. However, those business components are not able to use a publish/subscribe middleware on their own. Instead, they have to be *managed* and *wrapped* in order to be compatible to the middleware's architecture and functionality. In REBECA, the ManagedComponent class is used for this purpose. Figure 5.12 shows the corresponding UML class diagram giving an overview about its structure and relations to other classes.

The ManagedComponent class extends the AbstractComponent class and, thereby, inherits properties and logic to interact with the ComponentConnector in order

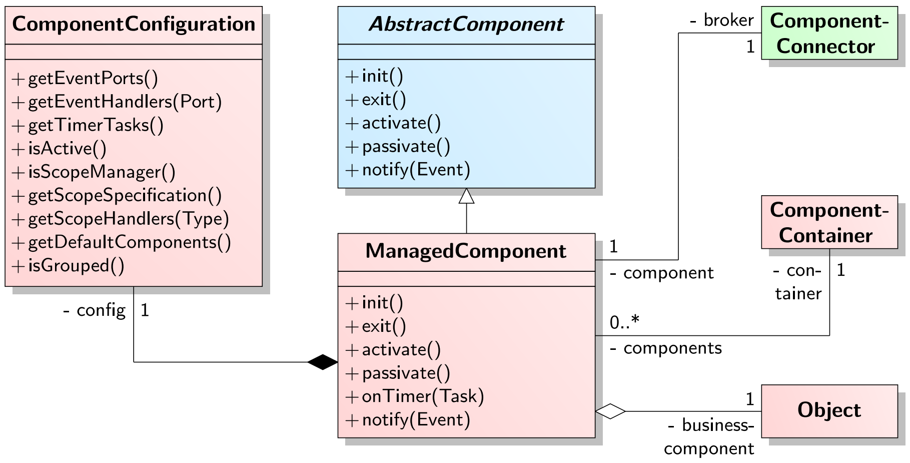


Figure 5.12: Managed component wrapping a business component.

to publish own events and receive subscribed ones. It also maintains a reference to the ComponentContainer whose services and functions are required to support some of the more advanced programming abstractions. Each ManagedComponent wraps an application component, which contains the actual business logic. Thereby, the application component can be of arbitrary type provided that fields and methods are annotated appropriately. Thus, the class diagram just shows the Object class as it is the root of the whole Java class hierarchy. Furthermore, each ManagedComponent also includes a ComponentConfiguration for the enclosed application component that contains prepared information about available event ports and existing event handlers. The ComponentConfiguration is generated from provided annotations once by the ComponentContainer when the application component is plugged. Hence, application component and its annotations need not to be extensively reinspected again, for example, whenever a received event notification has to be delivered to the responsible event port.

The ManagedComponent class overrides many inherited methods. The lifecycle operations init and activate are adapted to create necessary data structures and start the component's event processing. Among other things, this includes creating subscriptions and advertisements for event ports and scopes, issuing these and schedule time-triggered component tasks as well as finally start component threads. Contrarily, the methods passivate and exit stop component threads and timers, revoke subscriptions and advertisements, and free allocated resources. The operation onTimer is a callback method required by the container to inform that a time-triggered task is ready to be executed. Calling onTimer instead of immediately executing the task itself is necessary for synchronization reasons. This way, the ManagedComponent class ensures that no time-triggered task may interfere with an event handler concurrently processing a received notification. Please note that we primarily aim at hiding concurrency issues to ease application development rather than guaranteeing any realtime behavior. Finally, the notify method delivers received event notifications to appropriate event ports of the enclosed application component and manages its processing by calling the responsible event handlers.

In order to deliver notifications to event ports and subsequently call their event handlers for processing, the ManagedComponent depends on the prepared and compiled information provided by the ComponentConfiguration class. Therefore, the ComponentConfiguration offers operations to conveniently access and query details about event ports and event handlers as well as scope specifications in case that the application component defines and manages its own scope. In particular, getEventPorts returns all in-ports and out-ports as well as associated filters that are declared by the application component or defined in the system configuration, respectively. For each event port, getEventHandlers provides the list of associated business methods sorted by priority that need to be called for processing the data. Similarly, getTimerTasks returns a list of all time-triggered business methods, while isActive tells whether application component possesses own threads of execution. In this case, additionally concurrency and synchronization issues have to be considered as described in Sect. 5.3.3 making the event processing significantly more complex.

Regarding the management of scopes, the ComponentConfiguration also contains the detailed scope specification if the enclosed application component defines its own. Whether the application component manages an own scope can be determined by the isScopeManager method, while the specification itself is accessed using getScopeSpecification method. Furthermore, getScopeHandlers then returns a list of scope management methods declared by the application component that need to be called whenever a component member joins or leaves the defined scope, respectively. The list of scope management handlers is filtered by type of operation, i.e., whether the component joins or leaves the scope, and is sorted by priority starting with the handler of the highest priority which is executed first. Besides components that dynamically join and leave, a scope may also be configured to always contain a set of default members. Classes and constructors are returned by the getDefaultComponents method, so that component instances can be created by the ComponentContainer on demand. Finally, the isGrouped method tells whether component members get automatically grouped by their attribute values into corresponding subscopes. This is just a convenience method as all grouping details are a part of the whole scope specification.

5.7 Related Work

Event ports are the central element of the programming abstractions presented in this chapter. They provide a high-level interface for publish/subscribe components that separates reusable business logic from configuration and context information. This way, developers are enabled to easily create event-based applications by orchestrating prefabricated components, connecting their event-ports, and directing the event streams between them. As there are many similarities to dataflow architectures, we split the discussion of related work into two parts. In the following, we first discuss existing publish/subscribe abstractions before giving an overview about programming approaches based on dataflow concepts.

5.7.1 Publish/Subscribe

In recent years, event-driven programming has gained major importance and popularity. For Graphical User Interfaces (GUIs), for example, it even is the norm. Accordingly, programming abstractions that support and ease event-driven techniques are welcomed and appreciated. In [136], Meyer discusses general design aspects and considerations for asynchronous, event-driven interaction. For this purpose, he presents an *event library* written in Eiffel [135] and compares implemented concepts to those of other programming languages and frameworks. Although the event library is based on advanced language features such as multiple inheritance, constrained generics, and agents it still impresses by its simplicity.

The event library primarily consists of a single class and publishing of as well as subscribing to events actually boils down to a simple method call each. In particular, arbitrary methods to be executed can be subscribed to events by using Eiffel *agent expressions* which simply wrap a routine ready to be called into an object that can subsequently be passed as argument to other methods. This scheme, thereby, allows to make any method and component event-driven including those components that were originally not designed this way. Especially developers benefit from the high degree of reusability as it enables them to start from existing programs and systems without the need to write complex and lengthy glue code to connect the pieces. Moreover, agents may have open and closed arguments. While *open arguments* must be provided for each call anew as part of the communicated event, *closed arguments* are fixed and given once when the agent is defined itself. Thus, the latter are well suited to incorporate configuration data and settings when subscribing a component (i.e., defining an agent) to process a particular type of events. In fact, enabling components to get externally configured and subscribed to events is also a key concept of our approach. In [4], Arslan et al. provide further insights into the event library, its details, and its application.

While the event library is just designed for local interaction, Eugster [56] provides a set of programming abstractions for publish/subscribe in distributed environments. The overall goal is to neatly integrate content-based publish/subscribe into an object-oriented programming language such as Java while ensuring type safety. Therefore, Eugster et al. [62] advocate for using regular objects to represent the events to communicate. This is especially emphasized by coining the term *obvents* as abbreviation for such event objects. For exchanging events *Distributed Asynchronous Collections* (DACs) are introduced as an object-oriented abstraction that allows to subsume different publish/subscribe interaction styles as well as to express various Quality of Service (QoS) aspects [63]. Thereby, DACs adopt the basic idea of a *tuple space* as known from the coordination language Linda [83, 26]. In that sense, a DAC is a container to which obvents are added by one process and from which contained obvents are read by other processes. In contrast to the original tuple space, however, a DAC is inherently distributed and asynchronous allowing processes to be notified when new obvents are added on a host somewhere else in the network. To ensure type-safe operations, each DAC only handles obvents of the type it is created for. Additionally, it is possible to specify QoS aspects such as delivery and ordering guarantees for each DAC individually.

In [60], Eugster and Guerraoui extend DACs by filter objects to inspect the content of newly added obvents in order to determine which application components to notify. In addition to selecting obvents by type, this enables components to also specify constraints on an obvent's data and, thus, to engage real content-based publish/subscribe communication. Thereby, filtering obvents is based on structural reflection. For this purpose, the application developer specifies methods to be invoked on obvents together with expected return values to be matched. This way, arbitrary constraints can be formulated without the need to introduce a dedicated event language while preserving object encapsulation at the same time. However, invoking methods via reflection is not statically type-safe. With JAVA$_{PS}$ [62], Eugster et al. present a different approach that extends Java by embedding the two communication primitives *publish* and *subscribe* directly into the programming language. In particular, the subscribe primitive combines the subscription to an obvent type with two closure declarations that contain the content-based filter expression and, for the case of a match, the event handler to be executed, respectively. A dedicated precompiler is used to subsequently transform all publish/subscribe communication primitives into middleware calls and interactions. In [56] and [57], both approaches for type-based publish/subscribe are elaborated and compared to each other discussing their individual strengths and weaknesses.

Ulbrich et al. [218] present programming abstractions for content-based publish/subscribe as part of the .NET version of REBECA aiming to make writing event-driven applications as convenient as possible. Although abstractions and concepts themselves are not limited to any particular programming language or environment, their convenient application, however, strongly depends on inherent features of C# [88] and the .NET runtime [138]. On the subscriber side, operator overloading is elegantly used to enable the construction of subscription filters in a statically type-safe as well as easily readable fashion. In addition, the delegate mechanism is leveraged to specify corresponding event handlers. On the publisher side, the creation of event notifications going to be published is significantly simplified. It is sufficient to just provide an interface with a respective publishing method taking relevant event attributes as arguments. The actual implementation is generated and compiled on demand at runtime. Besides publishing methods, it is also possible to specify *template methods* that allow the developer to set certain event attributes once for all subsequent publications. In fact, this is a first step in order to separate code and business data from configuration data incorporated during system initialization. A major part of the work also deals with the pragmatic specification of event patterns and composite events, which, however, lies out of our focus here.

5.7.2 Dataflow

Research into dataflow was originally driven by the motivation to efficiently exploit massive parallelism. Conventional processors based on *von Neumann architectures* were criticized to not be suited for parallel computing because of their global program counter and global memory updates becoming a bottleneck. To avoid these bottlenecks, the *dataflow architecture* [51] was proposed which operates on local memory only and executes instructions as soon as their operands become available. Thus, a dataflow computation can be represented as a directed graph with the data to be processed flowing along the arcs towards the nodes containing the processing instructions. Several hardware designs were proposed, implemented, and studied. To program those machines more efficiently, dedicated dataflow programming languages such as LUCID [5] and SISAL [65] were developed. Although researchers were confident and optimistic, dataflow architectures never superseeded von Neumann machines as the fine-grained instruction level parallelism posed an unmanageable challenge to the hardware available then. However, dataflow and stream processing concepts have found their way into the design of modern processor architectures. In fact, with the current shift from single-core to multi- and many-core systems, the question how to efficiently exploit the offered degree of parallelism is all the more pressing and relevant today. Johnston et al. [104] provide a more detailed overview about the history of dataflow programming than the brief summary above.

Beyond parallel computing, *dataflow programming languages* and, in particular, their visual descendants have gained considerable importance in software engineering [91]. As domain-specific languages, they are quite successful in several application domains such as signal processing and image processing which themselves are inherently well suited for dataflow approaches. In fact, for tasks essentially dealing with data manipulation, they provide specialized and adequate solutions. Moreover, visualizing the dataflow graph lowers entry barriers for novice programmers as the visualization helps to better understand which data is when processed in which way. Based on these concepts, several successful commercial products have been developed that are widely used today.

LABVIEW[4] [196], for example, enables the design and construction of virtual instruments used to analyze and process laboratory data obtained from monitored experiments. Similar to conventional analog experiments, where sensors are wired to amplifiers and measuring instruments, LABVIEW programs connect data sources to the inputs of virtual instruments by arcs along which the

[4] LABVIEW is an acronym for Laboratory Virtual Instrumentation Engineering Workbench.

data is propagated. Each virtual instrument represents a function block that is applied on its input data and may produce output data, which can be further connected to other virtual instruments. Thereby, a virtual instrument starts processing as soon as all required input data is available. Another successful product example is SIMULINK [49], which also uses connected graphical function blocks for modeling and simulating technical, physical, and financial mathematical systems. From a software engineering point of view, such function blocks are characterized by a high degree of flexibility and reusability making dataflow approaches attractive and interesting for many other application domains, too.

Morrison [141] proposes a component-based dataflow approach for the development of business applications that is called *flow-based programming* (FBP) in order to better distinguish it from existing work dealing with dataflow programming languages and dataflow hardware architectures as discussed above. In FBP, an application consists of several component instances each driven by an own asynchronous thread of execution. Components have data ports on which they either receive input or produce output. To exchange data between components out-ports are connected to in-ports by bounded buffers, thus, forming a component network comparable to a pipes and filter architecture [204] or a Kahn process network [105]. Furthermore, FBP facilitates the hierarchical composition of component networks. Therefore, it is possible to bundle exchanged data to complex information packets as well as to reuse and instantiate existing component networks as subnets in bigger applications. Compared to the programming abstractions provided in this chapter for event-driven applications, many similarities become evident.

In fact, with FBP, we share many ideas, concepts, and intentions regarding component-orientation, reusability, and component orchestration. Likewise, we define event ports to exchange data between application components. As these ports are subject to publish/subscribe communication, however, a large variety of interaction patterns can be flexibly implemented. With scopes, we also provide a composition mechanism that bundles a set of related components to higher-level building blocks facilitating their reuse. Beyond that, scopes additionally provide a clear interface by filtering event notifications, may have own attributes containing configuration data, and are dynamic structures that allow member components to join and leave while enforcing access control restrictions.

5.8 Discussion

In this chapter, we presented programming abstractions for content-based publish/subscribe. Therefore, we first analyzed common pitfalls when building applications based on the publish/subscribe paradigm. In particular, many middleware implementations provide just a low level interface for publishing own event notifications and subscribing to those of other components. A further integration into the programming language that exploits advanced language features is often missing. It seems that middleware designers rather concentrate on efficiency issues regarding notification filtering and routing than on middleware usability. Hence, many aspects have to be explicitly considered by the application developer who, thus, has to usually write a significant amount of glue code to connect the business logic to the middleware. Moreover, if not done with care, this frequently entraps application developers to mix up business logic and application data with context information and configuration data making the application component hard to maintain, adapt, and reuse. Besides making the development of application components more convenient and productive, we exactly addressed this problem with the developed programming abstractions that aim at keeping business logic apart from context and configuration.

With *event ports*, we introduced a novel interface by which event-driven components can publish and receive notifications. Event ports consist of a well defined set of component fields, which are made visible and accessible to the outside in order to communicate significant state changes as event notifications to interested components. To react upon those state changes, we allowed methods to be annotated as event handlers so that they are called appropriately. Additionally, we also added support for time-driven handlers as well as active components featuring own threads of execution. Orchestrating components, i.e., subscribing their event ports to the notifications to process, is part of the configuration of an application, which was separated from the components' business logic. To further ease component orchestration, we also defined configuration methods that abstract from filter management and content-based filter expressions. Following a dataflow approach, we enabled developers to simply *connect* a publishing component's out-port to the in-port of a subscribing component. Likewise, components can be easily *inserted* into scopes and *grouped* into subscopes according to component attributes while established event connections remain unchanged. The filter expressions of affected subscriptions, scope interfaces, and component selectors are adapted appropriately and updated automatically. Regarding scopes, we provided means to either configure and adapt a basic scope as needed or to write a component managing its own custom scope. For the latter, it is sufficient to just annotate the scope definition as well as to specify component methods to handle scope-specific events such as members joining or leaving the scope. To support all presented programming abstractions, we revised the limited functionality of REBECA's component plugin that enables brokers to host local components. We extended the plugin to a full-fledged component container, discussed implementation details, and showed its integration as pluggable feature into the broker architecture described in Chap. 3.

The presented programming abstractions ease the development of event-driven applications. In particular, they separate business logic from context and configuration data and, thus, significantly increase the *reusability* of components. This is a further step towards enabling developers to build new applications by simply composing and connecting prefabricated components and artifacts. For this purpose, event ports play an important role since they provide comprehensive and convenient means to establish, direct, and process the event flows between these components. Moreover, as event ports only consist of a set of component fields, it is not difficult to also generate respective setter and getter methods to access them via a conventional request/reply scheme. Combining request/reply interaction and event-driven communication in a uniform way that both styles mutually support each other, however, is left for future work.

Chapter 6

Evaluation

Contents

6.1 Introduction

There are several ways to evaluate a software system. Probably, the most realistic results are obtained by instrumenting a real deployment of the system and measuring its performance. At the same time, however, this is usually the most costly variant even if a working prototype already exists. In case of distributed systems, there are a couple of additional drawbacks, too. In order to evaluate the system's scalability, a real deployment has substantial resource requirements in terms of computing nodes and network bandwidth. Thus, the available computing and network infrastructure usually limits the system size to be evaluated to small- or medium-sized installations. Furthermore, system parameters may have to be set to values that are compatible with or even tailored to the infrastructure the system is deployed on. While on the one hand, this precisely ensures valid and reliable results, on the other hand, it also restricts the configuration space that is covered by the evaluation. Regarding parameter coverage, simulations provide an alternative and better way for system evaluation. In particular, they offer a fine-grained control to arbitrarily vary any system parameter. Given a limited amount of computing resources, it is often possible to trade off system size against simulation detail. Hence, with deductions in accuracy, simulations are capable and well suited to analyze large-scale systems. In case of Internet-scale algorithms and systems, they often are the only feasible approach.

In this chapter, we apply simulations to thoroughly evaluate the concepts and ideas developed in this thesis. In particular, we focus on scoping as central concept to modularly build and extent large-scale event-based systems. Qualitative and quantitative results substantiate the claims about the usefulness of scopes that we previously made in Chap. 4. Scoping is analyzed as part of our publish/subscribe middleware REBECA in which it is integrated as pluggable feature according to the composable architecture presented in Chap. 3. As compromise between accuracy and parameter coverage including large-scale networks, we execute REBECA's original routing logic within a simulated network environment. In the following, we first provide an overview about the simulation environment discussing features, protocols, and experiment setups in Sect. 6.2. Thereafter, we proof the scalability of scoping by increasing the number of brokers and clients in Sect. 6.3, whereas we analyze the positive effects of different client distributions in Sect. 6.4. The costs of scoping measured in terms of generated overhead are evaluated in Sect. 6.5, before we conclude the chapter with a summarizing discussion in Sect. 6.6.

6.2 Simulation

To evaluate the publish/subscribe extensions presented in the previous chapters, *discrete event simulations* [116] are adequate and sufficient. The operation of the broker network including its clients and applications is modeled as a sequence of simulation events. In this context, for example, the receipt of a new subscription or the forwarding of a notification are considered as events to be simulated. A simulation event occurs at an instant in time and is instantly handled according to the simulated protocols and algorithms. Thereby, the simulation event may cause a change in state that affects the system's subsequent behavior. The receipt of the subscription, thus, leads to the creation of a new routing entry that, together with other already existing entries, determines the forwarding of subsequent notifications. Furthermore, handling a simulation event may also schedule causally related simulation events to be triggered in future. Hence, forwarding a matching notification over a network link inevitably causes a simulated receipt event to be triggered when arriving on the other side. Please note, that simulation events have no duration. Therefore, system processes that take time to finish have to be modeled by a series of simulation events that mark the start and the end of the process and, if necessary, its progress in between.

In the following, we give an overview about the simulation environment and its architecture. In particular, we discuss the underlying network model, the broker overlay as well as the protocols considered and abstractions made. Furthermore, we describe the default setup of the experiments conducted. This includes chosen parameters, their role in the simulation as well as their envisaged effects.

6.2.1 Environment

Several simulation environments are available that allow a realistic modeling and evaluation of network environments. There are, for example, NS-3 [190], OMNeT++ [221], and OPNET [36] just to name a few. Using elaborate configurations and setups, achieved simulation results are quite accurate and often comparable with measurements gained from real deployments. For this accuracy, however, there is a price to pay. To reach a high degree of precision, the whole network protocol stack as well as physical propagation models need to be considered and simulated. This is complex and costly. As a consequence, the applicability of those realistic simulations is limited to networks of moderate size only. Simulating a dynamic overlay network on top of a large physical network is out of scope. Therefore, we have chosen a different simulation environment that is particularly designed for the evaluation of overlay networks.

PEERSIM [139] is a discrete event simulator designed to analyze the routing in large-scale *peer-to-peer* overlay networks. To be that scalable, on the one side, PEERSIM abstracts from details of the lower network layers in the physical network. Furthermore, it also provides an additional coarse-grained simulation mode in which time passes in global rounds instead of individual events. To facilitate meaningful evaluations, on the other side, PEERSIM is very customizable allowing developers to conveniently integrate new protocols and refine existing ones in order to include all aspects of interest. In fact, it is this side of which we make heavy use to evaluate the publish/subscribe features implemented in the REBECA middleware.

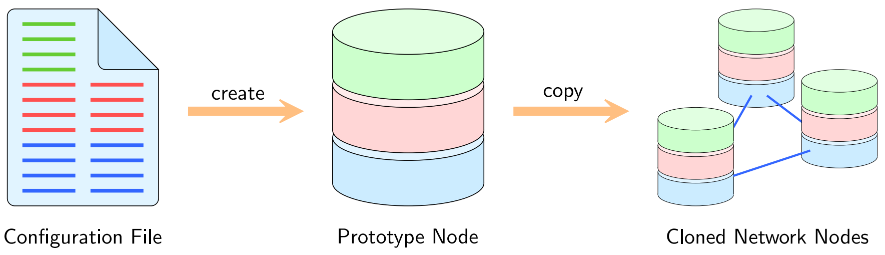


Figure 6.1: Simulation setup in PeerSim.

In PeerSim, a network consists of a set of nodes of which each node itself is a simple container for protocol instances. First, it is the task of the developer to provide protocol implementations that may leverage the PeerSim framework to schedule protocol events to be processed in future. The simulator framework then ensures that the particular protocol instance is called at the addressed node at the appropriate time and additional context information are provided in order to handle the current event. Second, the developer has to create a configuration file that describes the simulation's setup. Besides general information such as random seeds, network size, and simulation end time, the configuration includes the list of protocols to instantiate as well as further protocol parameters and settings. PeerSim uses the *prototype* pattern [81] to populate the network. Thus, one prototype node is created according to the configuration that includes instances of all listed protocols. Thereafter, the prototype node is cloned as often as needed. Special control protocols that are implemented as singletons may be used to differentiate network nodes as well as to monitor the simulation. While initializing protocols are called only once after setup, normal control protocols are called regularly and, thus, suited for periodic tasks.

Figure 6.1 visualizes PeerSim's simulation setup. Based on a configuration file, a prototype node is created that contains instances of the protocols listed therein. Afterwards, the prototype node is cloned with each generated copy becoming a new network node. In this process, the protocols are also cloned so that all instances of the same protocol now form an individual layer in the network stack. However, it is left to the protocol implementations how data is exchanged between different nodes. Even the network topology, i.e., which pair of nodes is connected with each other, must be set up as part of a PeerSim protocol. Usually this is done on the lowest network layer. In the following section, we exemplify the usage of PeerSim protocols by describing the protocol stack and the publish/subscribe features considered in the simulations.

6.2.2 Protocols and Features

PeerSim simulations are based on tailored protocol implementations. These resemble the protocol logic and emulate its execution by scheduling appro-

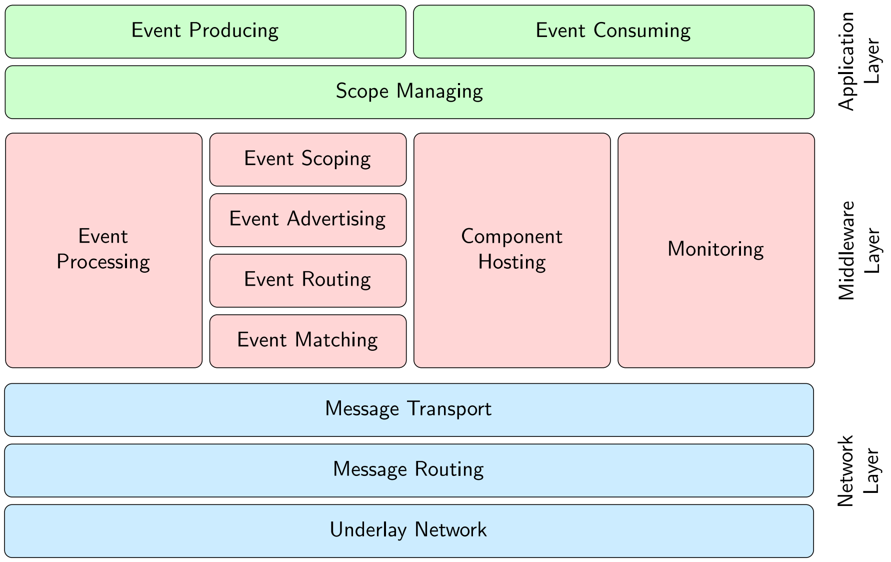


Figure 6.2: Stack of simulation protocols used for evaluation.

priate simulation events. In order to evaluate the REBECA middleware and its extensions—in particular, we are interested in the effects of scoping—we, thus, have to provide corresponding simulation protocols that implement the required publish/subscribe logic and associated features. Advantageously, REBECA's modular architecture eases the protocol implementation. Since REBECA encapsulates each publish/subscribe feature in a separate plugin, this basically boils down to reusing and adapting the plugin's logic and wrapping it appropriately to make it fit in the PEERSIM environment. The approach in general as well as individual actions to be taken are discussed in Chap. 3.4.4 as part of REBECA's modular and composable architecture. Besides the middleware, however, we have to consider the network layer and the application layer in the simulations, too. For a thorough evaluation, these layers and their protocols are essential as they determine the context in which simulation results have to be interpreted. Figure 6.2 gives an overview about the whole protocol stack evaluated. Layers and protocols are explained in the following.

Network. The bottom three protocols in the simulation stack are directly related to network functions. They determine the topology of the underlay network, are responsible for the routing of messages therein, and take care for the establishment of logical connections between broker nodes by which the overlay network is created. In particular, the *underlay* protocol provides the network topology including latency and bandwidth information. The BRITE topology generator [130] is used to create Internet-like network topologies that consider both the interconnectivity between Autonomous Systems as well as the links at router level within a single domain. BRITE offers several models for topology generation while allowing to flexibly combine different aspects. We use a heavy-tailed distribution of nodes [131] combined with the Generalized Linear Preference (GLP) connection model [24] to reflect characteristic path lengths, clustering coefficients as well as power-law properties of node degrees.

The *message routing* protocol takes care of forwarding messages within the underlay network. Routers as well as links are modeled as $M/M/1$ queueing systems according to the system model published in [200] that, besides simulations, also allows a thorough analytical analysis. When a message is send between two nodes in the underlay network, it is routed along the shortest network path between them. In order to speed up simulations, this path is precomputed only once at the beginning of the simulation and stored for subsequent use. The underlay nodes hosting publish/subscribe brokers are chosen randomly and logical overlay connections are established between them that are based on the shortest paths computed. The publish/subscribe overlay network is then formed by a minimum spanning tree of overlay links that connects all brokers. The *message transport* protocol is responsible for setting up and managing overlay links as well as to control the transmission of data over them. It is used by the brokers in order to exchange their event messages in the context of the simulations.

Middleware. To resemble a broker's publish/subscribe functionality, a couple of PEERSIM simulation protocols are required. In this context, the *brokering* protocol plays a central role as it manages and coordinates all remaining broker protocols that each resemble the functionality of a single broker plugin. This way, REBECA's composable architecture is reflected and leveraged in simulations, too. Primarily, there are four middleware responsibilities to care for in simulations: the strategy for processing and handling received event messages, providing the publish/subscribe functionality in the strict sense, supporting client components, and gathering statistics for evaluation.

Similar to routers in the underlay network, brokers are modeled as $M/M/1$ queueing systems. The *event processing* protocol ensures that received event messages such as notifications, subscriptions, and advertisements are first queued and subsequently handled one after the other. For the latter, the *event matching*, the *event routing*, the *event advertising*, and the *event scoping* protocol are used according to the message type to be handled, respectively. Matching received event notifications against stored subscriptions is one of the brokers core functions on which all other publish/subscribe extensions build on. In order to evaluate the effects of scoping, the usage of an advanced content-based routing algorithm combined with advertisements is required, too. In particular, the routing logic is needed to forward scope subscriptions as well as scope advertisements when clients subscribe to existing scopes or advertise their own, respectively.

The *component hosting* protocol enables simulated brokers to support and host local client components that produce and consume event notifications and, thus, create the event flows for evaluating the system. In this context, the *monitoring* protocol is responsible to detect and observe the event flows as well as to gather meaningful statistics as simulation results to be analyzed afterwards. Therefore, the monitoring protocol is implemented as singleton within the middleware layer. The singleton pattern allows to have a global view on the state of all simulated brokers and, thus, simplifies collecting statistical data. Furthermore, the central position within the protocol stack allows to easily trace notifications and created copies when they are received, processed, forwarded, or delivered.

Application. The application protocols create and shape the event flows necessary for system evaluation. They generate a synthetic application load used to measure the middleware's performance in the simulations. In this context, the *scope managing* protocol constrains the visibility of events by defining separate scopes for individual notification types. It controls the creation and placement of scope managing components of which each managing component opens an individual scope and administers its members, i.e., it processes join and leave requests of event publishers and subscribers. Publishers and subscribers are simulated and managed by the *event producing* and the *event consuming* protocol, respectively. On the one side, the producing protocol determines which publisher is responsible to generate what kind of notifications. Therefore, publishers are modeled as Poisson processes that each continually produce new event notifications of a fixed type randomly chosen beforehand. On the other side, the consuming protocol defines which client subscribes to which notification type.

All application protocols, i.e., the scope managing, the event producing, and the event consuming protocol, allow for sophisticated simulation setups. In fact, it is easily possible to simulate unbalanced, dynamic, and changing workloads. For this purpose, each protocol allows the definition of several component profiles that determine how exactly a single component behaves, for example, what scopes a component joins or administers, which notifications it publishes and subscribes, or how this pattern may change over time. Furthermore, each protocol sets a birthrate by which new components are created, connected to the system, and assigned one of the defined behavioral profiles. A probability distribution specifies how likely it is that a new component is assigned a certain profile and connected to a particular broker. By limiting a component's lifetime, it is ensured that old application components are eventually replaced by new ones. Thus, changing the probability distribution over time allows to simulate any arbitrary global behavior of the clients, too. In particular, we use these configuration features to simulate dynamic workloads and unbalanced client distributions leading to so-called component hotspots.

6.3 Scalability

A working prototype marks a major milestone when developing a hardware or software system. However, it does not automatically imply and guarantee that the final system also functions properly. Regarding distributed systems, prototypes are usually of a very limited size containing just a few devices that are easy to manage, while final deployments may comprise up to thousands of computing nodes or even more. Hence, implemented algorithms, strategies, and concepts must be efficiently applicable to large installations, too. *Scalability* is the ability of a system to adequately cope with increasing system size and load. For event scopes as presented in this thesis, scalability is of particular importance. Since they allow a modular development of publish/subscribe systems and event-driven applications, administrators and developers precisely benefit most when engineering large-scale systems. In the following series of experiments, we primarily evaluate the scalability of the scope implementation. Therefore, we increase the system size and load in terms of brokers and clients while measuring the system performance.

6.3.1 Brokers

Scopes restrict and control the visibility of event notifications as well as subscriptions and advertisements. In large publish/subscribe networks, they prevent subscriptions and advertisements from being forwarded needlessly into network regions that do not contain any other scope member such as a corresponding publisher or subscriber. This way, scopes help to reduce the overall network traffic. Regarding event notifications, however, this does not apply to the same extent. Since the regular routing algorithm usually filters out superfluous notifications as early as possible, it is already ensured that notifications are only forwarded if there is at least one interested subscriber.

In order to prove the scalability of the scope implementation and evaluate its performance, we increase the *system size* in a first experiment by successively adding new brokers to the publish/subscribe network. Although publishers and subscribers dynamically connect and disconnect to random brokers, the overall system load remains nearly constant over the series of measurements. The system performance is evaluated by measuring the overall number of subscriptions and advertisements forwarded in the publish/subscribe network as well as the average size of the brokers' routing tables. Parameters and experiment details are given in the following description.

Experiment 1 (increasing system size). *System scalability is evaluated by increasing the system size determined by the number of publish/subscribe brokers* $N_B = 10, \ldots, 250$ *within the network. The birth rate for new publishers and subscribers is set to* $\lambda_P = \lambda_S = 5$ *each having an expected lifetime of* $\tau_P = \tau_S = 200s$ *which leads to an average number of* $N_P = N_S = 5000$ *publishers and subscribers in steady state. There are* $M_T = M_S = 250$ *event types and corresponding scopes organized in a hierarchy with* $l = 3$ *levels and a branching order of* $b = 5$*. Each publisher and subscriber randomly chooses a dedicated event type, connects to a random broker, and joins the scope responsible for its type. The publication*

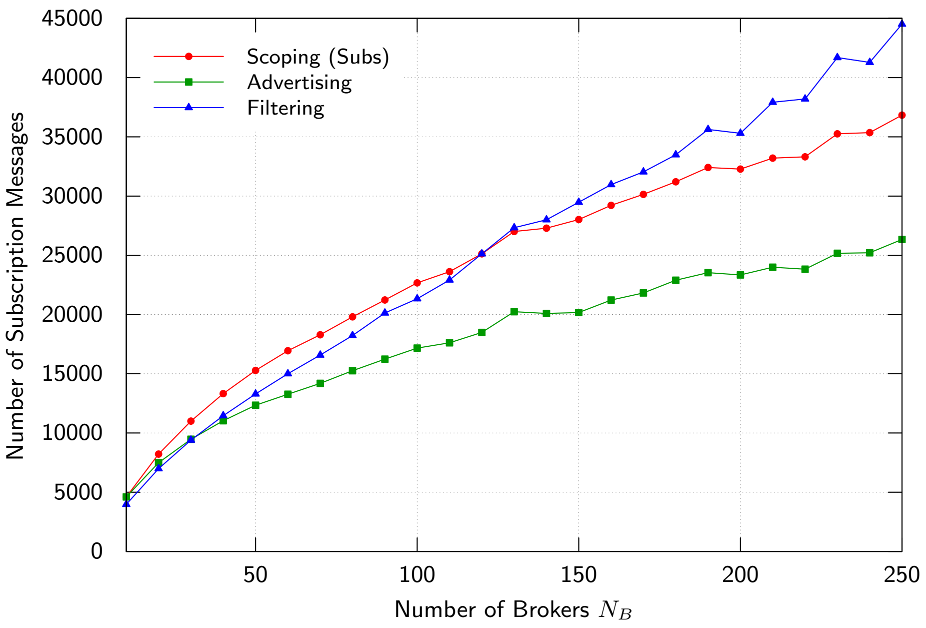


Figure 6.3: Subscription forwarding overhead versus network size.

rate of notifications is set to $\lambda_n = 5$. The system performance is determined by measuring the overall number of forwarded subscriptions and advertisements as well as the average size of the brokers' subscription and advertisement tables.

To better interpret the effect of scopes on the system's performance, we repeat the experiment with different combinations of routing extensions. Thus, we can compare the results and pinpoint characteristic properties and differences. In particular, we use a pure identity-based routing strategy (filtering) as base version, add advertisements (advertising) and scopes (scoping subscriptions), respectively, and finally apply both extensions (scoping advertisements) at the same time. Please note that we restrict ourselves to identity-based routing in order to prevent cross-scope optimizations to bias some of the measurements making it difficult to compare the results. Otherwise, more advanced routing strategies such as covering-based routing or merging-based routing may exploit similarities between filters in some setups although their subscriptions belong to different scopes in the other cases. Measured results are plotted in the diagrams of Figs. 6.3, 6.4, 6.5, and 6.6.

Figure 6.3 shows the overall number of forwarded subscriptions as a function of the network size. Brokers exchange active subscriptions with each other in order to ensure that a matching notification is delivered to all its subscribers independent of where it is published. Thus, the more brokers there are in the publish/subscribe network, the more subscription copies need to be exchanged making all curves grow with increasing network size. However, the extent of growth differs depending on the ability of the particular routing variant to limit the forwarding of subscriptions to relevant network regions. With pure filtering, the identity-based routing algorithm only suppresses the transmission of a subscription, if an identical one has already been forwarded on the particular link. Therefore, its graph does not grow exactly linearly. Nevertheless, for large networks comprising 200 and more brokers, it clearly causes the most overhead. This overhead is reduced significantly by using advertisements. In this case, the forwarding of subscriptions is strictly limited towards brokers that definitely host corresponding publishers. In large networks, therefore, advertising marks a lower bound for the subscription forwarding overhead. Finally, the overhead caused by scoping lies in between advertising and filtering. Although scope borders restrict the forwarding of subscriptions, scope overlays usually contain more brokers including those that, compared to advertising, do not lie on the direct path between publisher and subscriber. For instance, scope overlays also comprise brokers that only host subscope components. Additionally, subscriptions are also forwarded towards other subscribers since, without advertisements, all components within a scope have to be handled as potential publishers.

Considering small broker networks with less than 20 brokers, the results are different. Surprisingly, pure filtering performs better than advertising and scoping. This is due to publishers and scope members dynamically joining and leaving the network. With pure filtering, a subscription is only disseminated once in the broker network and then stored in the brokers' routing tables throughout the subscriber's lifetime. Contrarily, with advertising and scoping, it requires an advertisement or an acknowledged scope request, respectively, to trigger the forwarding of the subscription. The subscription is stored in the routing tables of the brokers on its way as long as the respective publisher or scope member is present, too. If there is no publisher or scope member in the particular network region anymore, the subscription is removed from the routing tables. However, it is resend when the next publisher or scope member arrives. Thus, during the lifetime of the subscriber, its subscription may be send multiple times over the same links causing the overhead compared to filtering. While the dynamic overhead caused by advertising is negligibly small, it is significant for scoping. In the experiment, it is not countervailed by savings from scope overlays until a network size of 120 brokers.

Please note that, for the sake of clarity, the results of combining advertisements and scopes are not explicitly plotted in the diagram. In fact, this is not necessary because, for the dissemination of subscriptions, advertisements are more restrictive than scopes and, thus, dominate their routing behavior. Hence, measured results do not observably differ from the advertising curve that is already shown. For the same reason, when evaluating subscription overhead in subsequent experiments, we will also plot just one curve for the routing variants with advertisements regardless if scopes are also used or not.

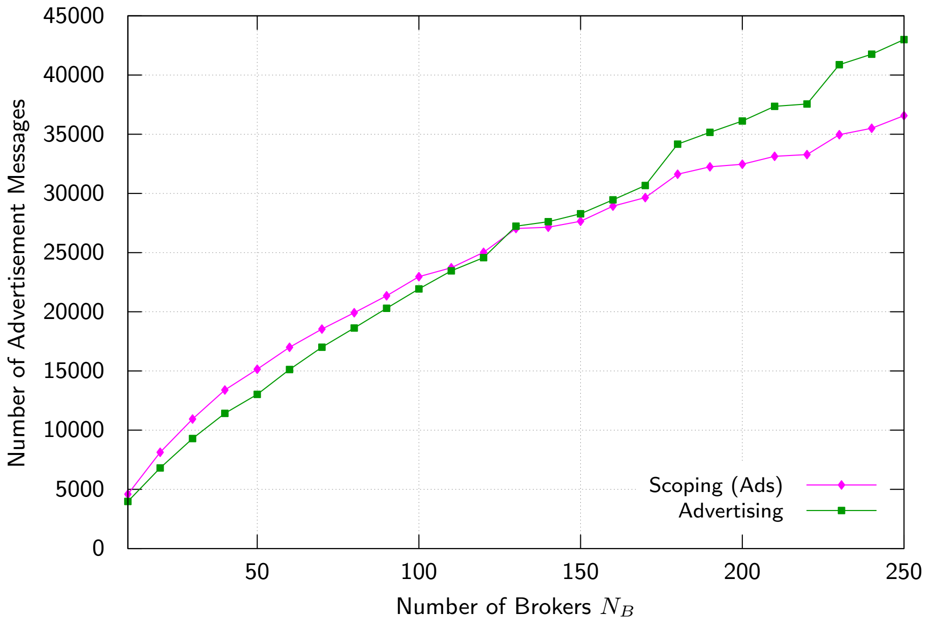


Figure 6.4: Advertisement forwarding overhead versus network size.

Leveraging advertisements significantly reduces the overhead caused by forwarding subscriptions. However, there is a price to pay: brokers must first exchange advertisements among each other, which is quite costly, too. Figure 6.4 shows the overall number of forwarded advertisements as a function of the network size for pure advertising as well as scoping combined with advertisements. As can be seen from the graphs, the resulting overhead is comparable to the cost of forwarding subscriptions as depicted previously in Fig 6.3. In fact, this is not surprising at all. Due to the symmetric experiment setup, there are as many publishers as subscribers expected to be in the system. On the one hand, with pure advertising, the forwarding of an advertisement is not restricted and, thus, causes the same overhead as disseminating a subscription in the whole broker network. Hence, pure filtering and pure advertising induce similar costs for routing subscriptions and advertisements, respectively. On the other hand, scope borders limit the visibility of subscriptions and advertisements alike and restrict their forwarding to the corresponding scope overlay. Therefore, with scoping, the plots for subscriptions and advertisements are similar, too.

Considering the combined forwarding cost of subscriptions and advertisements, at first glance, the usage of advertisements is not beneficial. Although advertisements significantly reduce the number of forwarded subscriptions, the cost of disseminating advertisements clearly outweighs the obtained savings. At second glance, however, the usage of advertisements makes sense for two reasons. First,

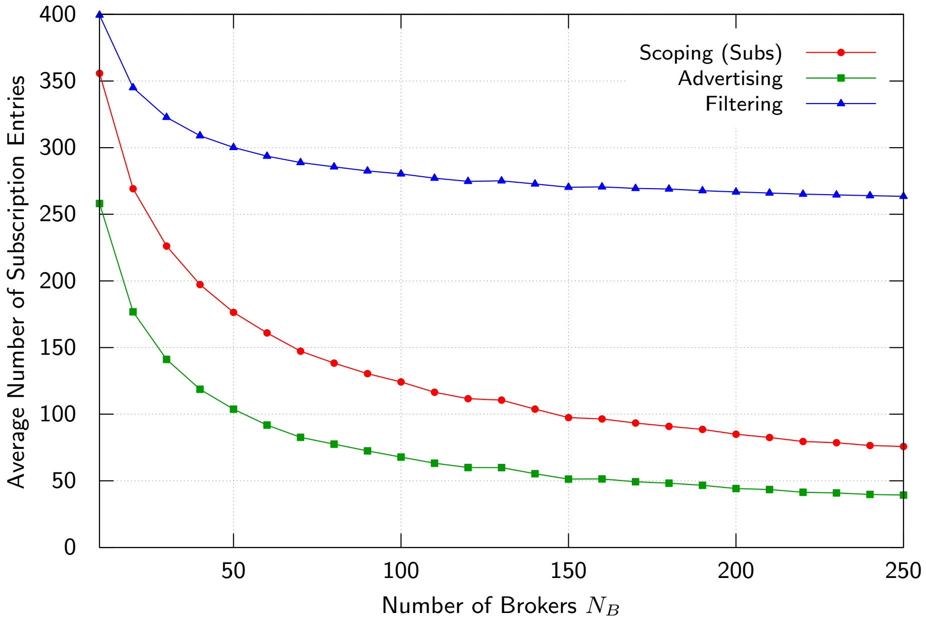


Figure 6.5: Average number of subscription entries versus network size.

in setups with more subscribers than publishers, the routing cost for advertisements are lower. In Sect. 6.4, we evaluate such asymmetric and non-uniform setups in more detail. Second, besides reducing the number of forwarded subscriptions, advertisements also limit the number of subscription entries in the brokers' routing tables and, thus, speed up the matching of notifications. As the number of notifications is usually assumed to be much higher than the number of subscriptions and advertisements together, many publish/subscribe systems are willing to trade off this forwarding overhead for matching speed in order to increase the overall system performance.

Figure 6.5 plots the average number of subscription entries of the brokers' routing tables as a function of the network size for pure filtering, scoping (subscriptions), and advertising. With growing system size, the average number of routing entries decreases asymptotically in all routing variants differing in how fast they approach their lower bounds. Unsurprisingly, pure filtering requires the most subscription entries as every subscription is disseminated in the whole publish/subscribe network. Identity-based routing suppresses the transmission of a subscription, if an identical one has already been forwarded. Thus, for each notification type subscribed, there is at least one corresponding entry in every broker's routing table. Hence, the average number of routing entries is always greater or equal to the number of subscribed notification types. Consequently, the latter is the lower bound the pure filtering plot is approaching. With advertising and scoping, the transmission of a subscription is also suppressed if it leaves the path towards a corresponding publisher or the corresponding scope overlay, respectively. Thus, the subscription itself is only disseminated in a small and limited region of the broker network. In the case of advertising, this region comprises all brokers hosting corresponding publishers or subscribers as well as the brokers lying on the network paths between them. In case of scoping, this region contains all brokers hosting a component of the corresponding scope or one of its subscopes and, likewise, all brokers in between. Compared to the whole network, the region is small in both cases. The number of brokers between any pair of nodes in the network is limited by the network's diameter. Assuming that the overlay tree is not degenerated and the number of clients remains unchanged, the region with corresponding routing entries only grows in the order of $O(\log N_B)$ when increasing the network size N_B. Therefore, the average number of routing entries eventually approaches 0 for very large networks.

Determining the limiting value of the curves, however, is only of minor importance as comparable results are only achieved in setups with more brokers than clients which are quite unrealistic. Instead, the course of the curves up to a network size of 100 brokers is more meaningful. Up to this point, the advertising and the scoping plot decrease strongly although there are still 10 times more clients expected to be in the system than brokers. As advertising is more restrictive than scoping regarding the forwarding of subscriptions, it leads to the lowest number of subscription entries in the system. Nevertheless, scoping is still competitive and clearly reduces the number of filters stored in the subscription tables compared to pure filtering.

Although the usage of advertisements remains the most effective way to restrict and optimize the dissemination of subscriptions in the broker network, it is not necessarily the most efficient at the same time. In fact, advertising requires the management of additional routing tables that store the publishers' announcements which types of notifications they are potentially going to publish. As scopes restrict the forwarding of subscriptions and advertisements alike, scoping can also be used to reduce the sizes of the brokers' advertisement tables. Figure 6.6 shows this effect by comparing the average number of stored advertisement entries when increasing the network size. Due to the symmetric experiment setup regarding publishers and subscribers, measured results are similar to the subscription table sizes as presented in Fig. 6.5. Without scopes, pure advertising leads to the same number of advertisement entries as pure filtering causes subscription entries because, in both cases, advertisements and subscriptions are disseminated in the whole broker network. Likewise, when combined with scopes that limit the forwarding of both advertisements and subscriptions, the routing table sizes are considerably reduced by the same ratio, too. Please note that, for the sake of brevity, we omit plotting the advertisement overhead and the advertisement table sizes in subsequent experiments that use similar distributions and symmetric setups for publishers and subscribers. With Figs. 6.3 and 6.4 as well as Figs. 6.5 and 6.6 as examples, it becomes clear how measured results for subscriptions can be transferred and applied for advertisements, too.

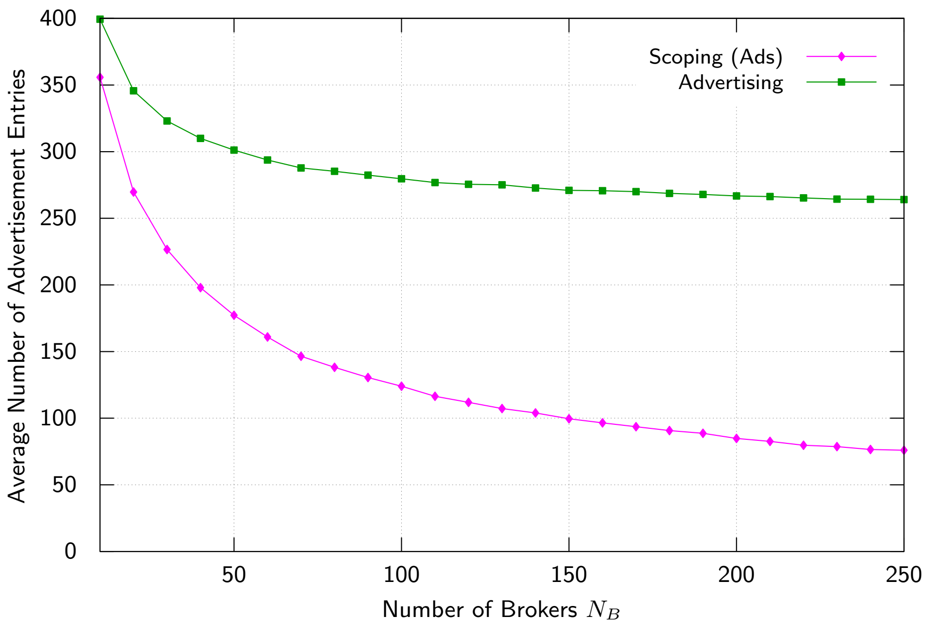


Figure 6.6: Average number of advertisement entries versus network size.

6.3.2 Clients

To thoroughly evaluate the scalability of a system, it is also necessary to analyze its behavior under heavy load. In the following experiment, we, therefore, successively add new clients to the publish/subscribe network while keeping the number of brokers constant. In fact, this is implemented by increasing the birth rate for publishers and subscribers in every run of the experiment. Thereby, each new client randomly chooses a dedicated event type of which it subsequently produces or consumes corresponding notifications throughout its lifetime. This way, it is ensured that the *system load* increases proportionally with the number of clients connected to the system. To analyze the performance, we measure the subscription forwarding overhead as well as the average number of subscriptions in the brokers' routing tables. The advertisement overhead and corresponding table sizes are omitted for symmetry reasons. Experiment parameters and further details are given by the following description.

Experiment 2 (increasing system load). *To evaluate the scalability, the system load is increased by successively incrementing the expected number of publishers and subscribers* $N_P = N_S = 10, \ldots, 2500$ *that are randomly connected to the* $N_B = 100$ *brokers of the network. To achieve this, the birth rate for new publishers and subscribers is varied* $\lambda_P = \lambda_S = 0, \ldots, 10$ *accordingly while keeping their*

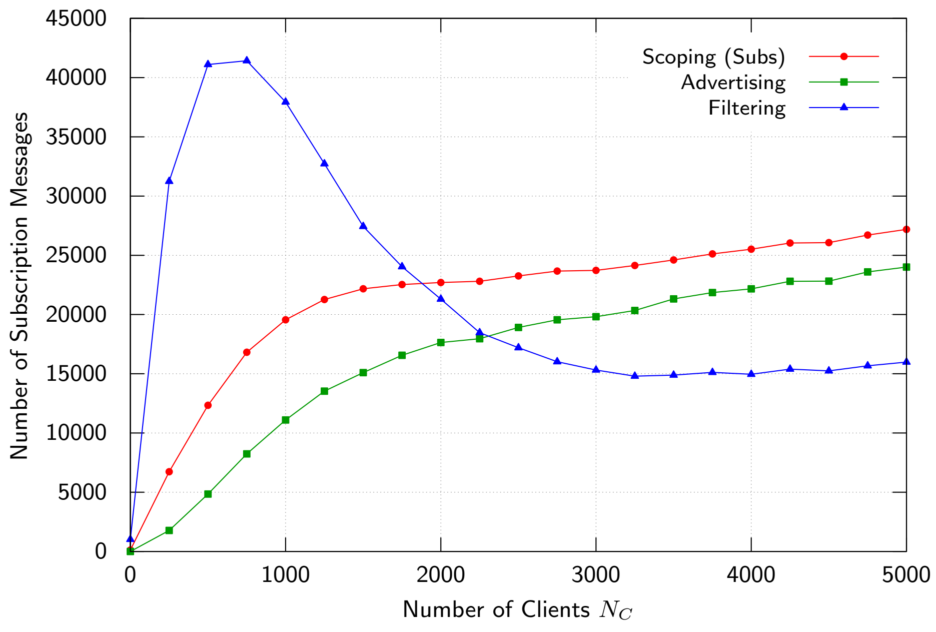


Figure 6.7: Subscription forwarding overhead versus number of clients.

expected lifetime of $\tau_P = \tau_S = 250s$ constant. There are $M_T = M_S = 250$ event types and corresponding scopes in the system that are organized in a hierarchy with $l = 3$ levels and a branching order of $b = 2$. Each publisher and subscriber randomly chooses an event type and joins the scope responsible for its type. The publication rate of notifications is set to $\lambda_n = 5$. The system performance is determined by measuring the overall number of forwarded subscriptions and the average size of the corresponding routing tables.

The results of the experiment are plotted in Figs. 6.7 and 6.8 for different routing configurations and extensions. Basically, we use identity-based routing (filtering) that we combine with advertisements (advertising) or scopes (scoping), respectively. Figure 6.7 shows the subscription forwarding overhead for these routing variants as a function of the expected number of clients in the system. In case of filtering, the shape of the plotted graph is particularly striking. First, the subscription overhead sharply increases up to a peak of over 41000 messages being caused by about 500 to 750 clients in the system. Subsequently, it starts declining although further subscribers are added to the system and even drops down to about 15000 messages. Finally, it remains almost constant at that low level which is considerably less overhead than the 24000 and 27000 subscription messages caused by advertising and scoping in the same load situation, respectively. In fact, this is all the more remarkable since advertisements and scopes have actually been introduced to make the system more efficient. Obviously, however, it does not pay off in this case. As pure filtering does not restrict the dissemination of subscriptions within the broker network, a growing number of subscribers quickly saturates the brokers' routing tables. If all neighboring brokers already possess a routing entry for a particular notification type, an identical subscription for the same type does not need to be forwarded to them anymore and is, therefore, suppressed by the identity-based routing algorithm. Consequently, the subscription forwarding overhead decreases although or rather because the number of subscribers grows further and, thus, makes a corresponding routing entry more likely to be already present in the brokers' tables.

Advertising as well as scoping also profit from saturated routing tables although the effect is less pronounced. Resulting savings are not strong enough to actually reduce the subscription overhead as it is in case of filtering. Nevertheless, the increase is clearly slowed down when there are about 1500 or more clients in the system. Regarding the forwarding of subscriptions, scopes are usually less restrictive than advertisements. While scoping allows the free distribution of subscriptions within their corresponding scope overlays, advertising only forwards subscriptions towards respective publishers. Thus, the subscription overhead of scoping initially grows faster compared to advertising because subscription messages have to be sent to more brokers in the system. With a growing number of subscribers, however, it is later much more likely when applying scopes that a broker's routing table already contains a corresponding entry making it unnecessary to forward a subscription any further. Thus, the growth of the subscription overhead declines stronger than in case of advertisements making both curves asymptotically approach each other. Nevertheless, regarding subscription overhead, both are clearly worse than a pure filtering approach.

When focusing on subscription overhead, at first glance, filtering seems to be very efficient in situations where a large number of publishers and subscribers induce a high system load. On a closer look, however, this is not the case. Figure 6.8 plots the average size of a broker's routing table while adding more clients to the broker network in order to increase the system load. The more filters are stored in a broker's subscription table, the longer it takes to match and forward a notification at each broker reducing the system's performance. As can be clearly seen, pure filtering causes by far the most routing entries. Moreover, the initial growth of the subscription table size is the strongest for filtering followed by scoping and advertising. Basically, this is not surprising as the average number of routing entries depends on the capability of the routing strategy to restrict the distribution of an issued subscription to the minimal number of necessary brokers. Although not as good as advertising, which marks the lower bound, scoping still reduces the routing table sizes significantly when compared to pure filtering. Due to the table's saturation effect the growth declines later in all three cases. However, a slow linear increase remains as for each new subscriber at least one additional entry has to be added to the routing table of the hosting broker even if subscription forwarding can be suppressed completely.

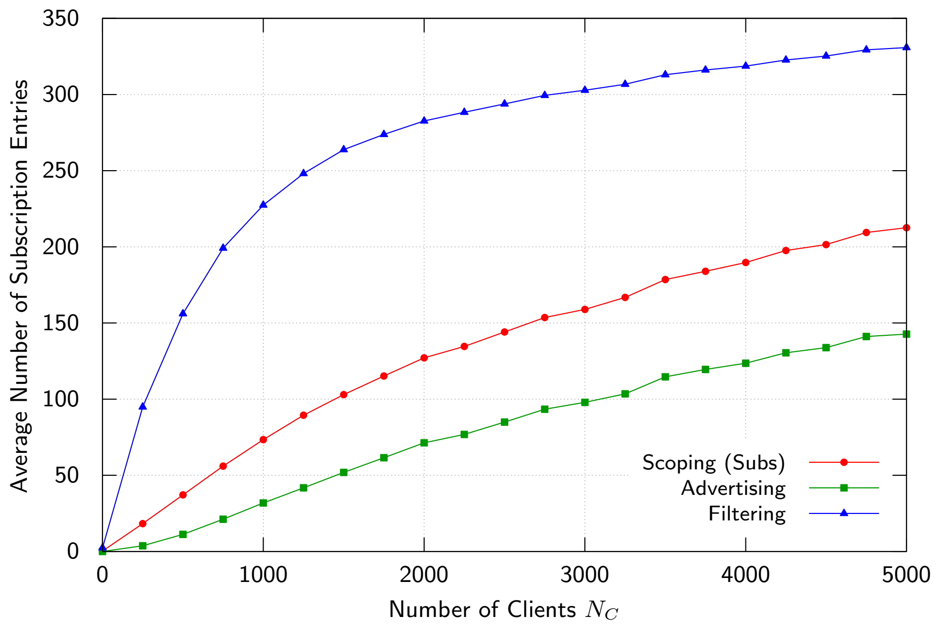


Figure 6.8: Average number of subscription entries versus number of clients.

6.4 Distributions

Although symmetric simulation setups are sufficient to demonstrate the general scalability of scoping as done in Sect. 6.3, many real applications and scenarios differ substantially in one or more aspects. Often, for example, just a few notification types predominate the event traffic produced and consumed, while publishers outnumber subscribers or vice versa. In fact, realistic setups usually exhibit a certain level of *asymmetry* and *inequality*. In order to show that scoping is applicable in such scenarios as well and can even benefit from inherent inequalities, we evaluate various distributions of clients and notification types in the following experiments. In particular, we vary the ratio between publishers and subscribers and also examine local concentrations of clients and notification types in hot spot regions.

6.4.1 Publisher/Subscriber Ratio

Many application scenarios are inherently unbalanced in the sense that publishers outnumber subscribers or vice versa. In data gathering applications, for example, there are usually many sensors periodically publishing their readings while only a few clients correlate and analyze the gathered data. Contrarily, in information dissemination applications such as stock quote tickers, there are usually just a few components that publish new information, while the majority of clients is interested in receiving data updates. Several publish/subscribe routing strategies and extensions are able to take advantage of these unbalances and asymmetries to a certain degree. Advertisements, for example, are known to work best if there are just a few notification sources in the system [142]. In this case, it is quite cheap to invest in a small number of additional advertisement messages and profit from huge savings achieved by forwarding subscriptions towards these publishers only. Regarding scopes, however, we do not expect much savings. As the dissemination of subscriptions and advertisements is restricted by scope borders, the forwarding cost essentially depend on the size of the corresponding scope overlay while the *ratio* between publishers and subscribers is of minor importance.

Evaluating scenarios with a varying number of publishers and subscribers is, nevertheless, very interesting. Primarily, it provides insight in which specific configurations scoping is beneficial compared to filtering and advertising which are more sensitive to the publisher/subscriber ratio. In the following experiment, we, thus, change the average number of publishers and subscribers in the system by varying their birth rates λ_P and λ_S, respectively. Therefore, we set

$$\begin{aligned}\lambda_P &= (1-\varphi)\cdot\lambda_C \quad \text{and}\\ \lambda_S &= \varphi\cdot\lambda_C,\end{aligned}$$

where λ_C denotes the overall birth rate of clients. This way, we are able to easily specify arbitrary publisher/subscriber ratios by varying $\varphi = 0.0,\ldots,1.0$. Further details are given in the experiment description below.

Experiment 3 (publisher/subscriber ratio). *The ratio between publishers and subscribers in the system is adapted by varying their respective birth rates $\lambda_P = (1-\varphi)\cdot\lambda_C$ and $\lambda_S = \varphi\cdot\lambda_C$ with $\varphi = 0.0,\ldots,1.0$, $\lambda_C = 8$, and $\tau_P = \tau_S = 250s$ while keeping the expected overall number of clients $N_C + N_P + N_S = 2000$ constant. There are $N_B = 100$ brokers, $M_T = 250$ event types, and $M_S = 250$ corresponding scopes in the system of which the latter are organized in a hierarchy with $l = 3$ levels and a branching order of $b = 2$. Each publisher and subscriber randomly connects to a broker, chooses an event type, and joins the responsible scope. The publication rate of notifications is set to $\lambda_n = 5$. The system performance is determined by measuring the overall number of forwarded subscriptions and advertisements as well as the average number of stored subscriptions in the brokers' routing tables.*

The experiment is repeated using different routing strategies and extensions. In particular, pure identity-based routing is used as default strategy (filtering), which is subsequently extended by advertisements (advertising) and scopes (scoping subscriptions) as well as their combination (scoping advertisements). The results are graphed in Figs. 6.9 and 6.10.

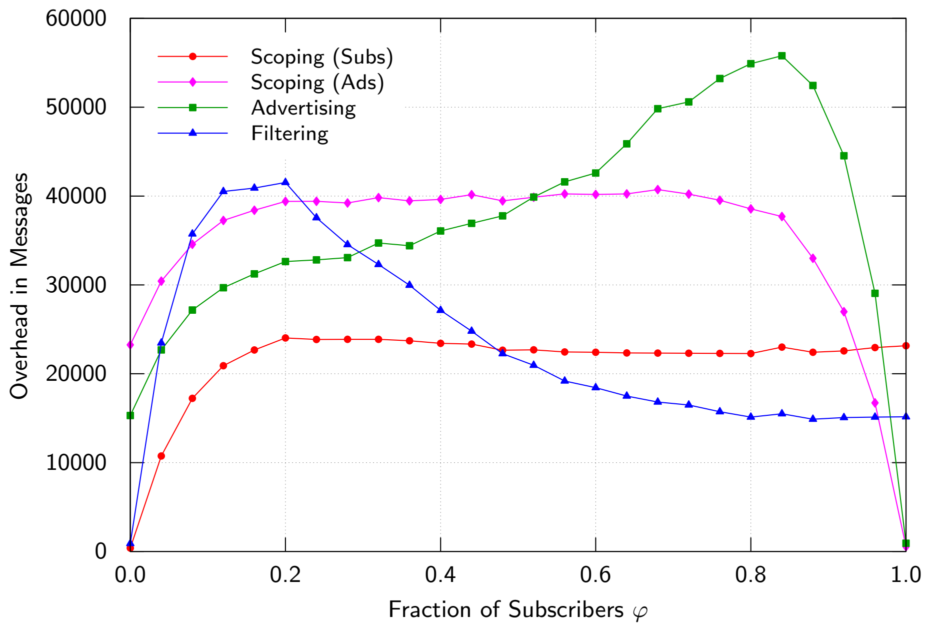


Figure 6.9: Advertisement and subscription forwarding overhead for a varying publisher/subscriber ratio.

Figure 6.9 shows the combined forwarding overhead for subscriptions and advertisements as a function of the fraction φ of subscribers in the system. With increasing φ, the number of subscribers in the system grows while the number of publishers declines to the same extent. When adding the first subscribers to the system, the overhead for forwarding subscriptions also rises inevitably. For pure filtering, it quickly reaches a flat peak of over 40000 messages at $0.15 \leq \varphi \leq 0.20$. Thereafter, it starts to decline as a result of saturated routing tables that make forwarding new subscriptions more and more superfluous. This saturation effect has already been studied in more detail in Sect. 6.3.2 when discussing the results of experimennt 2. Because of its positive influence on the forwarding overhead, pure filtering even becomes the most efficient routing strategy in terms of forwarded messages for $0.50 \leq \varphi \leq 0.95$. By combining scopes with filtering (scoping subscriptions) in order to restrict the dissemination of subscriptions, the observed initial increase in the forwarding overhead can also be dampened. It is even bound at a level of about 25000 messages on which it remains nearly constant because, with scopes, the influence of the saturation effect is also limited. Nevertheless, scoping subscriptions is the most efficient routing strategy for $0.00 \leq \varphi \leq 0.50$.

Considering the overall forwarding overhead, the results confirm that the usage of advertisements alone (advertising) or combined with scopes (scoping adver-

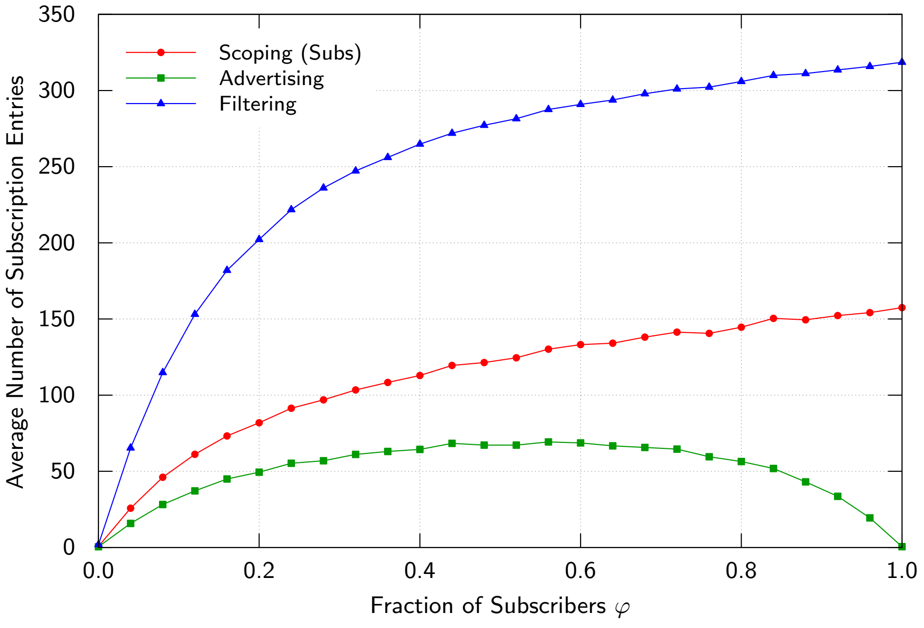


Figure 6.10: Average number of subscription entries for a varying publisher/subscriber ratio.

tisement) really pays off when there are just a few publishers in the system, i.e., for $0.95 \leq \varphi \leq 1.00$. Surprisingly, pure advertising also performs better than pure filtering for $0.05 \leq \varphi \leq 0.30$ when the broad majority of clients are publishers. Here, advertising primarily benefits from saturated advertisement tables while the subscription forwarding overhead for filtering reaches its peak. In fact, both curves are quite similar when mirrored. Following the advertising curve from right to left, the number of publishers grows and increases the advertisement forwarding overhead until the saturation effect reduces the costs again. As the overall costs also include the subscription forwarding overhead, the curve is usually higher than the one for pure filtering except for the cases described above. When advertising is combined with scopes (scoping advertisements), both the maximum cost as well as the savings based on the saturation effect are reduced. Thus, the overhead remains nearly constant over a wide range of the experiment. The costs only decrease at the edges of the curve as there are either just a few publishers or only some subscribers in the system.

Regarding the average number of subscriptions stored in the brokers' routing tables, Fig. 6.10 shows a more familiar situation that we have already seen in previous experiments. For advertising, the advertisement forwarding overhead pays off leading to the lowest number of subscription entries followed by scoping and filtering. With a growing fraction of subscribers in the system, the number of subscriptions and, hence, subscription entries in the routing tables increases for all routing strategies. Depending on how good the forwarding of subscriptions can be restricted, this increase of stored routing entries turns out to be correspondingly slower. Moreover, for advertising, the number of subscription entries starts to decline when publishers outnumber subscribers. If there are no publishers left in the system, it even drops to zero because without any advertising publisher no subscription is forwarded and stored at all.

6.4.2 Hot Spots

In previous experiments, we usually assumed a uniform distribution of clients and notification types in the broker network. In many publish/subscribe systems, however, this is often not the case. Instead, many phenomena studied and measured in physical and social sciences vary over huge ranges that, sometimes, span many orders of magnitude. In [152], Newman lists a few well known examples such as the distribution of city populations, the frequency of words used in human languages, the annual incomes of households, the number of scientific papers and paper citations grouped by author and paper, respectively, as well as the sizes of computer files and the number of page hits generated by web sites. All these phenomena follow power laws and are, thus, characterized by large inequalities and disparities.

A discrete random variable X is said to follow a power law if the probability p_k of obtaining the value $k = 1, 2, \ldots$ obeys

$$p_k = Ck^{-\alpha},$$

for some positive constant C and α. C is determined by the normalization condition requiring that all probabilities p_k sum up to 1. Thus, we get

$$1 = \sum_{k=1}^{\infty} p_k = C \sum_{k=1}^{\infty} k^{-\alpha} = C\zeta(\alpha) \quad \text{and} \quad C = \frac{1}{\zeta(\alpha)},$$

where $\zeta(\alpha)$ is the Riemann ζ-function. Therefore, the resulting probability distribution

$$p_k = \frac{k^{-\alpha}}{\zeta(\alpha)}$$

is also called *zeta distribution*. Often, however, we just have a finite population of n elements whose probabilities of occurrence follow a power law. In this case, we find after normalization that

$$p_{k,n} = \frac{k^{-\alpha}}{\sum_{i=1}^{n} i^{-\alpha}},$$

which is called *Zipf distribution* named after George K. Zipf who coined the empirical law in linguistics that the frequency or probability of a word is inversely

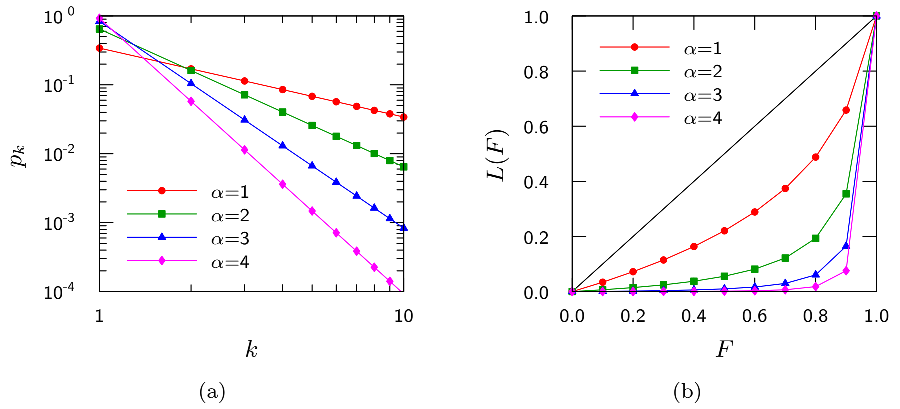


Figure 6.11: Zipf distribution and Lorenz curve: (a) Zipf distributions for different shape constants α and (b) corresponding Lorenz curves.

proportional to its rank k. Obviously, for infinite n, the Zipf distribution is equivalent to the zeta distribution which is why, in literature, both terms are often used interchangeably.

Given a Zipfian probability distribution, the constant α determines its shape. When plotting the distribution in a diagram with logarithmic horizontal and vertical axes as done in Fig. 6.11 (a), the power law makes it follow a straight line with α determining the slope. The larger α, the steeper the curve and, thus, the bigger the disparities grow resulting from the distribution. To visualize them, the corresponding Lorenz curves are shown in Fig. 6.11 (b). The *Lorenz curve* is a cumulative distribution function defined in the unity square of the first quadrant depicting relative concentrations. Therefore, the graph shows the cumulative proportion of ordered elements mapped onto the cumulative proportion of their size or probability, respectively. If all elements have the same size or probability, the Lorenz curve is the diagonal of the unity square, which marks the line of perfect equality. If there is any inequality in the distribution, the Lorenz curve drops below the diagonal equidistribution line while the deviation depicts the resulting disparity. Its extent is measured by the *Gini coefficient*, which is defined as the ratio of the area lying between the equidistribution line and the Lorenz curve over the total triangular area under the equidistribution line. Thus, the Gini coefficient ranges from 0 indicating a perfectly equal distribution to 1 in case of absolute inequality. However, it is mathematically equivalent, but often more convenient, to alternatively calculate the Gini coefficient as the difference between 1 and twice the area under the Lorenz curve.

For a discrete Zipf distribution as defined above, the Lorenz curve is given by the piecewise linear function $L(F)$ that connects the points (F_i, L_i) for $i = 0, \dots, n$, where

$$F_i = \frac{i}{n} \quad \text{and} \quad L_i = \sum_{k=0}^{i-1} p_{n-k,n} = \frac{\sum_{k=0}^{i-1}(n-k)^{-\alpha}}{\sum_{k=0}^{n} k^{-\alpha}}.$$

Please note that the elements are considered from last to first to cumulate their probabilities in increasing order. The area under the Lorenz curve is then calculated by easily summing up the trapezoids under the line segments of the curve. Thus, for the Gini coefficient G, we get

$$G = 1 - \frac{2}{n}\sum_{i=1}^{n}\frac{L_{i-1}+L_i}{2} = 1 - \frac{2}{n}\left(\frac{L_0}{2} + \sum_{i=1}^{n-1} L_i + \frac{L_n}{2}\right)$$

and immediately find with $L_0 = 0$ and $L_n = 1$ that

$$G = 1 - \frac{2}{n}\left(\sum_{i=1}^{n} L_i - \frac{1}{2}\right) = \frac{n+1}{n} - \frac{2}{n}\sum_{i=1}^{n} L_i.$$

By substituting L_i, we further get

$$G = \frac{n+1}{n} - \frac{2}{n}\sum_{i=1}^{n}\sum_{k=0}^{i-1} p_{n-k,n} = \frac{n+1}{n} - \frac{2}{n}\sum_{k=1}^{n} k \cdot p_{k,n},$$

which we finally simplify to

$$G = \frac{n+1}{n} - \frac{2\sum_{k=1}^{n} k^{1-\alpha}}{n\sum_{k=1}^{n} k^{-\alpha}}.$$

Based on the derived formula, we are able to calculate the Gini coefficient for a given Zipfian distribution in order to measure its disparity. But conversely, the equation also allows us to find a corresponding Zipf distribution that exhibits a predetermined degree of inequality. Although there is no compact closed-form expression, we may still solve the equation numerically for α and, thus, determine the distribution's shape for a given population and a specified Gini coefficient.

Leveraging Zipf distributions in the publish/subscribe simulations, we are able to evaluate more realistic network setups and notification workloads where clients are preferentially hosted by particular brokers and prefer to publish or subscribe certain notification types. By calculating the distribution's shape based on a preset Gini coefficient, we can, moreover, fine-tune how strong clients focus on preferred brokers and notification types and, thereby, create hot spot regions within the network. This way, it is possible to gradually increase the distribution's disparity and, thus, to turn a perfectly equal distribution of clients and notification types slowly into a completely unequal distribution consisting of a single *hot spot* where one broker hosts all clients that only produce or consume the same notifications. Further experiment details and parameters are given in the following description.

Experiment 4 (hot spots). *To evaluate the effect of local concentrations of clients and notification types within the publish/subscribe network, a Zipf distribution is used to assign clients to brokers and notification types to clients. The distribution's degree of disparity is gradually increased by varying the expected Gini coefficient* $G = 0.0, \ldots, 1.0$ *in order to form hot spot regions. The publish/subscribe network consists of* $N_B = 100$ *brokers that host an overall number of* $N_P = N_S = 1000$ *publishers and subscribers on average. The birth rate of new publishers and subscribers is set to* $\lambda_P = \lambda_S = 4$*, while their expected lifetime is set to* $\tau_P = \tau_S = 250s$*. There are* $M_T = M_S = 250$ *event types and corresponding scopes that are organized in a hierarchy with* $l = 3$ *levels and a branching order of* $b = 2$*. Based on the Zipf distribution above, each client is assigned a hosting broker and a dedicated notification type to subsequently produce or consume, respectively. The publication rate of notifications is set to* $\lambda_n = 5$*. The system performance is evaluated by measuring the overall number of forwarded subscriptions and, if any, advertisements as well as the average size of the corresponding routing tables.*

The experiment is conducted for different routing strategies and their extensions. In particular, identity-based routing is used as default routing algorithm (filtering), which is extended by advertisements (advertising) and scopes (scoping subscriptions) as well as their combination (scoping advertisements). Measured results regarding the subscription and advertisement overhead as well as the routing table sizes are presented in Figs. 6.12 and 6.13, respectively.

Figure 6.12 shows the cumulated overall overhead for forwarding subscriptions and, if applied, advertisements. It is graphed as function of the expected Gini coefficient that characterizes the distribution's degree of inequality when assigning clients to brokers and notification types to publishers and subscribers. For small Gini values $0.0 \leq G < 0.2$, we still have a roughly uniform and equal distribution where minor concentrations of clients and notification types do not have considerable impact or effect. Because there are an equal number of publishers and subscribers in the system, the usage of advertisements does not pay off making both strategies applying them clearly more expensive. Likewise, the usage of scopes does not have any advantages in this situation. The overhead is nearly the same independent of whether filtering or advertising is complemented with scopes or applied in their pure variants, respectively.

With an increasing Gini coefficient $0.2 \leq G < 0.6$, however, diverging trends occur. With the formation of first hot spots within the network, the overhead caused by pure filtering and advertising is growing. This is because the saturation effect for the less frequent notification types is lost. Hence, new subscriptions and advertisements for these types are forwarded to every broker in the network again as there are no identical, already existing routing entries left that would, otherwise, make their transmission unnecessary. With scoping, however, the dissemination of subscriptions and advertisements is effectively restricted to those brokers that host corresponding components responsible for the respective notification types. Thereby, scoping particularly profits from an unequal

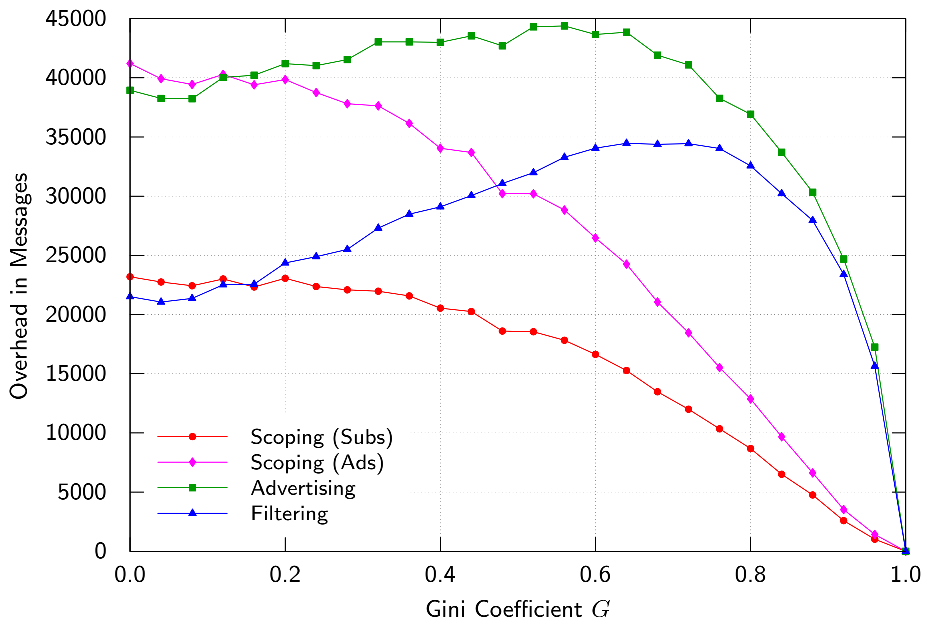


Figure 6.12: Advertisement and subscription forwarding overhead for an increasingly unequal Zipf distribution of clients and notification types.

client distribution and local client concentrations where the majority of publishers and subscribers just focus on a few hot spot brokers. As consequence, the subscription and advertisement overhead decreases notably.

For a Gini coefficient $0.6 \leq G < 1.0$, the overhead caused by pure filtering and pure advertising, respectively, reaches a flat peak and starts decreasing afterwards. For a growing number of notification types, it is becoming extremely unlikely to get published or subscribed at all. Hence, for these types, there are no advertisements and subscriptions to be forwarded which reduces the costs independent of the applied routing strategy. In fact, in case of all four routing variants, the overhead eventually drops to zero when $G = 1.0$. Please note that this theoretical limit of the distribution's disparity is, in practice, implemented by assigning all clients to a single broker and making them all publish and subscribe the same notification type. Thus, after the routing entries for this type have once been established within the broker network, they suppress the forwarding of all subsequently issued subscriptions and advertisements. Moreover, as there are always enough clients in the system producing and consuming corresponding notifications, these routing entries do not need to be updated or revoked anymore. As a consequence, no advertisement or subscription message is counted as overhead during the measurements although publishers and subscribers are actively using the system meanwhile.

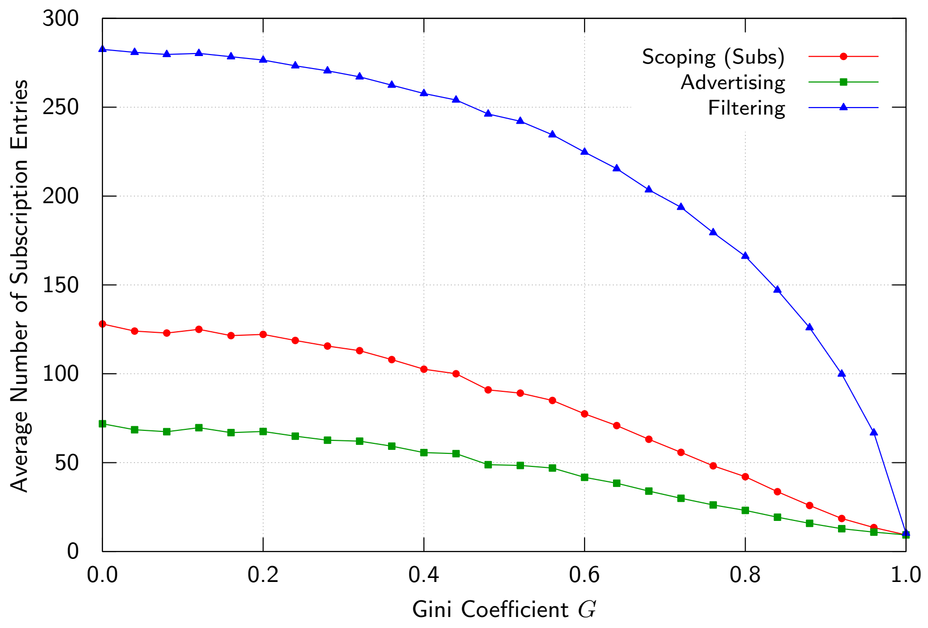


Figure 6.13: Average number of subscription entries for an increasingly unequal Zipf distribution of clients and notification types.

Besides the forwarding overhead, the effects of an unequal distribution of clients and notification types are also reflected by the broker's routing tables. Figure 6.13 plots the average number of stored subscriptions against the Gini coefficient measuring the distribution's disparity. There are no considerable changes in the routing tables for small Gini values $0.0 \leq G < 0.2$. Hence, minor inequalities do not have any concrete impacts yet. Instead, we obtain quite familiar results. Since scopes and advertisements effectively limit the forwarding of subscriptions, their application leads to significantly fewer routing entries as compared to pure filtering. Since advertisements, furthermore, only allow the forwarding of subscriptions towards corresponding publishers, the number of entries is again halved as compared to pure scoping.

For Gini values $0.2 \leq G < 1$, the first hot spots emerge and grow until a single broker and notification type dominate the system. Regarding stored routing entries, an increasing disparity has a positive effect that reduces their number independent of the routing strategy applied. As scopes and advertisements prevent subscriptions to be unnecessarily disseminated to uninvolved brokers, the average number of routing entries decreases linearly with clients being hosted by a few brokers only. For pure filtering, however, the number of routing entries starts to decrease considerably stronger when a growing fraction of notification types is not subscribed anymore. Please note that even for a Gini coefficient

$G = 1.0$, the average number of routing entries does not drop to zero. Independent of the routing algorithm, each subscriber causes at least one local subscription entry within the routing tables of its hosting broker.

6.5 Overhead

As previous experiments and simulations show, the usage of scopes improves the system's performance in many situations. This is primarily achieved by limiting the dissemination of subscriptions and advertisements at scope boundaries and, thereby, preventing them to be unnecessarily forwarded into network regions without matching publishers or subscribers. However, scoping comes at a cost. The price to pay is additional *overhead* caused by the distribution of scope information and the management of scope memberships within the broker network. Components need to join corresponding scopes first in order to communicate with each other afterwards. In particular, this causes additional system messages to be send to apply for a particular scope, requires extra routing entries pointing towards applicants and members of that scope, and introduces an initial delay until the membership is confirmed or denied. To better understand the costs of scoping, we further evaluate the system's overhead in the following experiments. Therefore, we determine the percentage of the system overhead that is directly related to scoping as well as the parameters it depends on. Subsequently, we further analyze the costs of scoping by varying these parameters.

6.5.1 Competitiveness

Scoping causes overhead. Information about each scope needs to be advertised within the publish/subscribe network in order to let brokers and clients know about its existence. Thereafter, clients need to subscribe every scope they want to join while, in return, they are notified about approved or denied scope memberships. Hence, there are many system messages exchanged even before the first event notification is published or received by any client component. In fact, these messages constitute a major part of the system's overhead. In previous experiments and simulations, however, we ignored this part in order to better analyze the effects of scoping on regular subscriptions and advertisements issued by clients. On the one side, this allowed us to easily compare obtained results qualitatively as well as quantitatively to alternative routing strategies and their variants. On the other side, we have not yet answered the question whether scoping is *competitive* at all.

In order to fulfill advanced tasks and purposes, distributed event-driven applications often have to react to multiple different notification types and sources while producing a variety of events themselves. Hence, their components usually maintain several subscriptions and advertisements for the event notifications they consume or produce, respectively. One part of the system's overhead results from distributing these subscriptions and advertisements within the broker network. The more subscriptions and advertisements are issued by the components, the higher the overhead grows. With scoping, the other part of the overhead originates from the system messages required to manage the scope memberships of the components. The more scopes a component joins, the higher this part of the overhead grows. Thus, the purpose of the following experiment is to determine the proportion of the overall costs for each part and to find out when which part outweighs the other. Experiment details and parameter settings are given in the description below.

Experiment 5 (scope overhead). *The share of scoping of the overall cost is measured while increasing the regular forwarding overhead. Therefore, the advertisement and subscription volume* $V = 0, \ldots, 20$ *of the components is varied, i.e., the number of non-identical advertisements and subscriptions issued by a publisher or subscriber for a particular notification type, respectively. The experiment is conducted in a network of* $N_B = 100$ *brokers hosting an overall number of* $N_P = N_S = 1000$ *publishers and clients on average. The birth rate of new publishers and subscribers is set to* $\lambda_P = \lambda_S = 4$ *while their expected lifetime is* $\tau_P = \tau_S = 250s$. *There are* $M_T = M_S = 250$ *event types and corresponding scopes in the system that are organized in a hierarchy with* $l = 3$ *levels and a branching order of* $b = 2$. *Each client randomly chooses a dedicated notification type and hosting broker. The publication rate of notifications is set to* $\lambda_n = 5$. *The system overhead is evaluated by measuring the number of forwarded advertisement, subscriptions, and scope messages as well as the average number of corresponding routing entries.*

In the experiment, identity-based routing combined with scopes is applied as default routing strategy. The experiment is repeated with and without advertisements being exchanged between brokers. Without forwarding the advertisements issued by publishers, client subscriptions are disseminated in the whole broker network unless they are stopped at a scope border. With advertisements, subscriptions are only forwarded towards corresponding publishers saving messages. Instead, advertisements are distributed in the whole broker network unless stopped by a scope border. The generated system overhead is shown in Figs. 6.14 and 6.15 broken down by message types and routing entries, respectively.

Figure 6.14 plots the relative forwarding costs of scoping against the regular advertisement and subscription volume per client component. Obviously, if clients do not publish or subscribe any notification, the overhead is solely made up by scope messages since the components still join their scopes. With an increasing number of advertisements and subscriptions issued by the clients, the regular filter forwarding overhead grows linearly while the costs of scoping remain constant. Hence, the relative share of scoping is first dropping rapidly to asymptotically slow down later. Nevertheless, the costs are significant and cannot be neglected when dimensioning a publish/subscribe system. In fact, if each publisher and subscriber just issues a single advertisement or subscription,

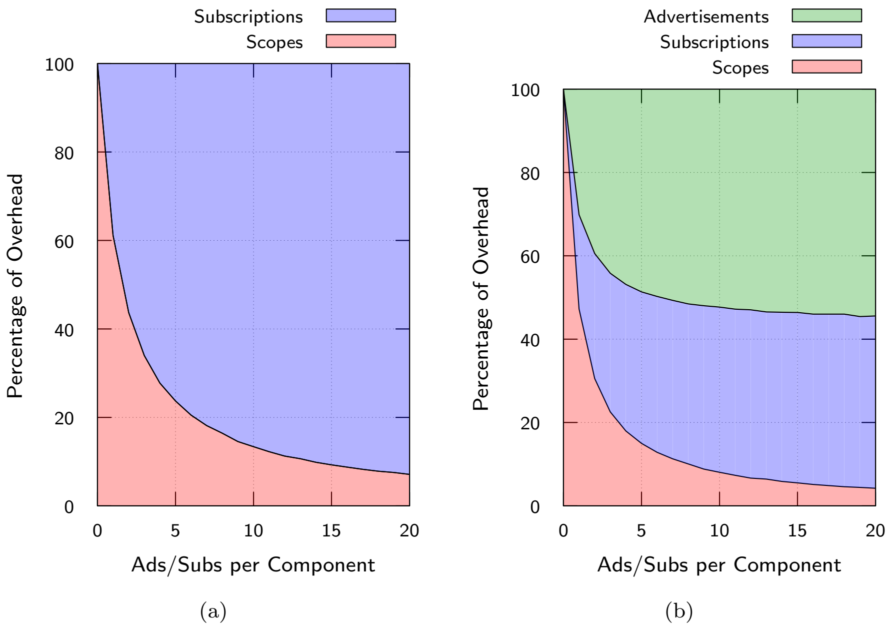
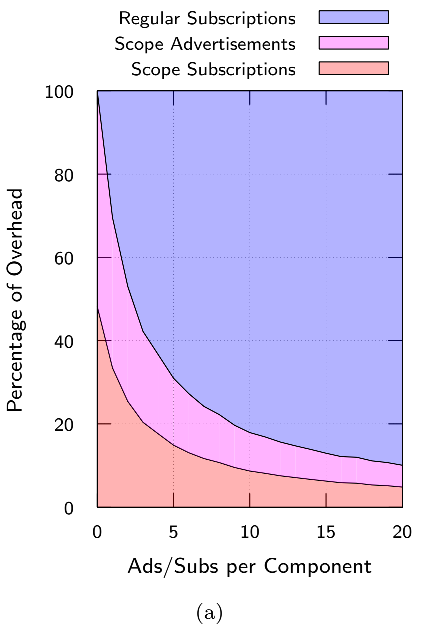


Figure 6.14: Relative share of the overhead per message type for an increasing advertisement/subscription volume: (a) scoping; (b) scoping with advertisements.

respectively, the relative costs of scoping amount to nearly 50% of the total message overhead as graphed in Fig. 6.14 (b). If advertisements are omitted, the relative costs of scoping are even more than 60% as shown in Fig. 6.14 (a). When compared to other routing strategies, scoping thus has to save the same amount of conventional forwarding overhead first in order to be competitive.

At first glance, scoping looks to be rather expensive. But actually, it is more efficient than it appears because of two reasons. First, the cost ratio of scoping improves notably when clients issue more subscriptions and advertisements. For $V = 10$, for example, it drops to 13% and 8% of the overall overhead as depicted in Figs. 6.14 (a) and 6.14 (b), respectively. Second, there are fewer scope messages forwarded than expected. Joining a single scope may require up to three messages: a scope advertisement announcing the existence of the scope, a scope subscription requesting the membership, and a scope approval or denial as reply. Especially since the majority of components in the experiment have to become members of one or two parent scopes first before they can join the right subscope for their event type, scope messages should clearly predominate the system's overhead. But obviously, this is not the case. Because we allow brokers to also admit clients to scopes that they have already joined, brokers can simply inherit their scope memberships to hosted components without the need to for-



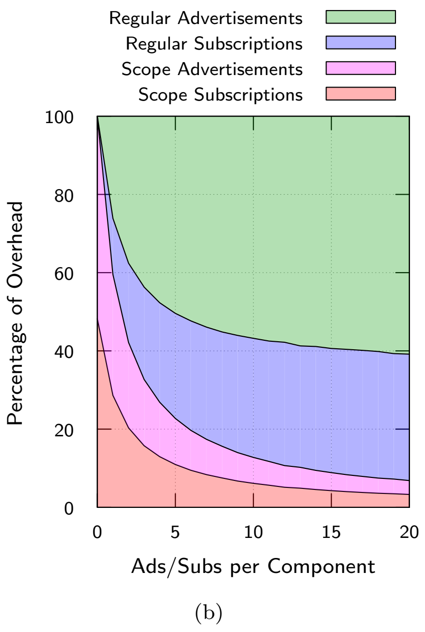


Figure 6.15: Relative share of occupied routing entries per filter type for an increasing advertisement/subscription volume: (a) scoping; (b) scoping with advertisements.

ward any scope message. In fact, this saves a lot of overhead depending on the probability of which the hosting broker already possesses the scope membership that the client requests.

Besides the message forwarding overhead, scoping also occupies routing entries to store scope advertisements and scope subscriptions in the brokers' routing tables. Their relative share is plotted in Fig. 6.15 while the volume of regular advertisements and subscriptions issued by clients is increased. Thus, the absolute number of routing entries required by regular advertisements and subscriptions is growing linearly while the number of scope entries remains constant. Consequently, their relative share drops asymptotically towards zero. Nevertheless, their number cannot be neglected especially in case of small regular advertisement and subscription volumes. For example, for $V = 1$ as usually set in previous simulations, scope advertisements and subscriptions together occupy 70% of the brokers' routing tables as depicted in Fig. 6.15 (a). With regular advertisements, only 60% of the routing entries are occupied as shown in Fig. 6.15 (b). Please note that scoping always requires scope advertisements to be forwarded and processed in both cases.

Although scope subscriptions may represent a major part of the routing entries and are also stored in the same routing table as regular subscriptions, it does not necessarily mean that they slow down the matching of notifications to same extent. Usually, the applied matching algorithm has a much bigger influence on the system's performance. For example, if matching is based on decision trees, a single additional test can early exclude all scope subscriptions. If a predicate counting algorithm is used, the integration of scope subscriptions boils down to the evaluation of one additional attribute for identifying each scope. In fact, many more advanced matching algorithms scale efficiently with an increasing number of scope subscriptions.

6.6 Discussion

Scoping allows to limit and control the visibility of event notifications within publish/subscribe networks and, thereby, provides effective means to modularly build and extend event-based systems. In fact, scoping particularly pays off when systems and applications grow in size and complexity. Hence, the concrete implementation of scopes used within the publish/subscribe system has to be scalable as well. In this chapter, we evaluated scoping as implemented in our publish/subscribe middleware REBECA. REBECA's original processing, matching, and routing logic was executed in a simulated network environment. This allowed us to flexibly set up large-scale broker networks with arbitrary client distributions while obtaining accurate measurements regarding the number of subscriptions, advertisements, and scope messages forwarded and stored in the brokers' routing tables. As part of the evaluation, we compared the results for scoping with those obtained for other routing strategies and extensions.

Regarding the size of the brokers' routing tables, scoping significantly *reduces* the number of stored subscriptions compared to a pure filtering strategy that, in general, does not limit the forwarding of subscriptions. A lower number of routing entries usually speeds up the matching of notifications which has a major impact on the system's overall performance. However, if advertisements are used, even better results are achieved at the expense of additional overhead for disseminating and storing them. This is independent of whether they are combined with scoping. Regarding the message overhead, scoping is also able to reduce the number of both forwarded subscriptions and advertisements except in case of saturated routing tables. Since scoping reduces the number of routing entries, it cannot profit from suppressed messages at the same time that are only saved when routing tables are well-filled.

Simulation results prove that our implementation of scopes does *scale* with both an increasing network size and a growing number of clients. In contrast to the usage of advertisements, scoping is quite insensitive to a varying ratio between publishers and subscribers within the system and is, thus, well suited for deployments where this ratio is not known at design time or may dynamically change at runtime. If publishers and subscribers of corresponding notification types are grouped together and concentrate at hot spots within the broker network, scoping becomes highly effective as it prevents subscriptions and advertisements to be unnecessarily forwarded to remote network regions. As real publish/subscribe systems are often characterized by unequal distributions of clients and notification types, hot spots are likely to occur especially in case of applications that exhibit a certain degree of locality between publishers and subscribers.

Nevertheless, scoping also causes a significant amount of *overhead* that is primarily made of scope advertisements as well as scope subscriptions which have to be forwarded and stored in the brokers' routing tables, too. Thereby, the overhead primarily depends on the number of scopes a component joins and not on the number of regular notification types advertised or subscribed. Hence, the more regular advertisements or subscriptions are issued by the components, the more competitive scoping becomes as its share on the overall costs decreases. The organization of the scope hierarchy only indirectly affects the costs by changing the probability with which the hosting broker is already a scope member. In this case, it is possible to simple inherit the membership to its clients without transmitting any additional message. However, when weighing up costs and benefits of scoping, the advantages clearly predominate. Please remember that the main goal and purpose of scoping is to structure and organize publish/subscribe systems and event-driven applications, so that these can be engineered in a modular fashion. Hence, we cannot expect scoping to come for free. Instead, if we are also able to reduce the overhead and save costs in certain situations, it is an extra bonus on top.

Chapter 7

Conclusions

Modularity is the key concept for engineering publish/subscribe systems and event-driven applications. To improve modularity, we designed a composable publish/subscribe architecture, leveraged scopes as structuring means, derived supporting programming abstractions, and finally evaluated the scope concept. In the following, we first summarize our work highlighting the main contributions and results. We critically review these with regard to our engineering goals and their applicability to event-based infrastructures. Finally, we discuss further question raised in this thesis and give an outlook on future research and the immediate next steps to address these open issues.

7.1 Summary

In this thesis, we have set out to engineer publish/subscribe systems and event-driven applications. Starting from an introduction of event-based infrastructures and a discussion of basic concepts and principles of publish/subscribe communication, we analyzed existing notification services and middleware standards with regard to applied communication mechanisms and provided engineering means. On the one side, these systems offered a wide range of technical functions and features substantially differing in terms of supported network topologies and protocols, applied matching algorithms, routing strategies, and their optimizations as well as the kind of notification selection and its expressiveness. On the other side, however, the systems provided no or only very limited support for system engineering, organization, and management. In fact, industry and research primarily concentrated on technical aspects such as communication efficiency, system performance, or scalability issues and, in the majority of cases, neglected engineering challenges to ease the development and master the increasing complexity of event-based infrastructures. This was particularly reflected by the lack of adequate module concepts and sufficient structural abstractions. In order to facilitate the engineering of event-based systems, we therefore addressed *modularity* as the key concept and tackled it on the level of the middleware design, the publish/subscribe infrastructure, and the event-driven applications.

Regarding the middleware design, we proposed a *composable architecture* for publish/subscribe brokers based on functional modularity and centered around features and their composition. A feature represents a particular functional aspect such as a specific network protocol, a certain routing strategy, or an applied matching algorithm, while the overall broker functionality is composed by selecting and combining these features. This way, publish/subscribe infrastructures can be flexibly tailored to actual requirements and environment conditions. To enable the free composition of features, we designed brokers as containers for pluggable components each of which encapsulates and implements a specific feature that can dynamically be added and removed at runtime. Hence, a broker only manages the components plugged in as well as the message streams between them while the real publish/subscribe logic is provided by the plugins. We defined interfaces by which the plugins intervene the broker's message handling and intercept those event messages that are relevant for their features. Plugins are, thus, able to modify and alter passing event messages as well as to remove old and create new ones in order to realize their functionality. On the client side, we also applied the idea of features and feature composition. Likewise, pluggable components are transparently inserted into the broker connection and, thereby, allow to flexibly add new functions as well as to leverage plugged broker features. The presented architecture formed the basis of the redesign and reimplementation of our publish/subscribe middleware Rebeca. With Rebeca, we demonstrated that mandatory functions, essential publish/subscribe logic, and advanced extensions such as advertising or scoping can easily be implemented as pluggable broker components and that these can flexibly be combined with optional feature plugins which address system management and security or render the network adaptive and fault-tolerant.

On the level of the publish/subscribe infrastructure, we proposed *scoping* as a modular structuring concept to better organize publish/subscribe systems and event-driven applications. For this purpose, scopes bundle related components and limit the visibility of event notifications, subscriptions, and advertisements. This way, scopes create a private environment for interaction that avoids unintended side-effects. Based on concepts of set theory, we formally defined the semantics of scopes and allowed systems to be hierarchically decomposed in subsystems and subscopes. For a flexible system decomposition, different criteria and aspects may be applied while components can be members of multiple scopes, too. To facilitate the engineering of event-based systems with scopes, we also defined scope interfaces, scope attributes, and scope mappings. Scope interfaces help to control component interactions between different subsystems since they precisely specify those event notifications that are allowed to cross scope boundaries. Scope attributes annotate scopes and contained components with configuration and context data in order to ease their orchestration. Attributes are also inherited to subscopes, components, and published notifications by default, but to customize, refine, or transform annotated information, they can be overridden and mapped as needed, respectively. We preferred a close integration of scoping into the publish/subscribe routing layer based on scope overlays because of two reasons. First, we exploited the scope structure to optimize the routing of event notifications, subscriptions, and advertisements by preventing their unnecessary forwarding at scope borders. Second, we mapped management functions for joining and leaving or for creating and destroying scopes to regular publish/subscribe operations allowing us to reuse and profit from existing program logic and data structures. Leveraging REBECA's composable architecture, we implemented scoping as pluggable broker feature that is compatible with all major routing algorithms and extensions.

For event-driven applications, we derived *programming abstractions* that ease content-based publish/subscribe communication and support the orchestration of components and their notification flows. Based on an analysis of pitfalls for publish/subscribe applications and possible remedies, we strongly advocated for a strict separation of the business logic from context and configuration data in order to increase the reusability and extensibility of application components. For this purpose, we introduced event ports as a novel interface that exposes a set of previously annotated component fields directly to publish/subscribe communication. In particular, any data update on an out-port field caused by a component's change in state is automatically published as a new event notification, while data updates received from subscribed notifications are applied on corresponding in-port fields. Components are, thus, orchestrated by subscribing their in-ports to the notifications they have to process. Taking a data flow approach, we further eased this process by enabling application developers to merely connect a component's out-port to one or more in-ports and vice versa. This way, the application logic encapsulated within component methods does not need to contain any publish/subscribe primitives as well as any other middleware-specific code anymore. To process incoming data updates on event ports, we allowed component methods to be conveniently annotated as event handlers. The developed execution model determines which handlers are called in which order and how obtained results are published while active components that possess own threads of execution or dynamically modify issued subscriptions are also supported. To better organize event-driven applications, their components can flexibly be grouped into scopes and subscopes that are flexibly created on demand while established notification flows based on connections between event ports remain in place. In particular, we offered developers the choice as well as the means to either annotate components with own scope definitions and manage their members individually or to adapt and customize provided default scopes and components as needed. To support our programming abstractions within a publish/subscribe middleware, we extended REBECA's component plugin to an advanced component container implementing additional functions and services that are required, for example, to inspect component annotations, to call responsible event handlers and publish their results, or to dynamically set up scope and component instances on demand.

Finally, we presented a detailed *evaluation* of the proposed module concepts primarily focusing on system performance and the effects of scoping. For this purpose, we leveraged REBECA's composable architecture by adapting and extending a number of feature plugins in order to allow brokers to be executed in a real network deployment as well as in a simulation environment. In particular, the PEERSIM simulator enabled us to conveniently set up large-scale networks, measure performance parameters such as routing table sizes and message overhead, and study the effects of different distributions of publishers, subscribers, and event notifications. To better interpret the obtained results, we qualitatively and quantitatively compared the measurements with simulations of related routing strategies and extensions. This way, we showed that scoping scales with network size in terms of brokers and clients while significantly reducing the number of stored routing entries and the message overhead. On the one hand, the savings are particularly strong in cases of unequal distributions of clients when these form network regions with hot spots for certain notification types. On the other hand, the performance of scoping is, in contrast to other routing extensions such as advertising, largely independent of the ratio between publishers and subscribers in the system. Furthermore, we measured the overhead for managing scopes and its proportion on the overall system costs while thoroughly analyzing relevant parameters and their influence. In general, it is not guaranteed that the scope overhead is always outweighed by the number of advertisements, subscriptions, and event messages that are prevented from being unnecessarily forwarded at scope borders. But nevertheless, we showed that there are many scenarios, setups, and situations in which scoping can be leveraged to both effectively organize event-based infrastructures and to reduce the overall costs at the same time.

7.2 Goals Reviewed

The central focus of the thesis lied on engineering publish/subscribe systems and event-driven applications. Within our work, we primarily aimed at improving and simplifying the development process for this kind of systems while taking the inherent properties and characteristics of event-based infrastructures into account. In particular, the loose coupling between components as well as the flexibility of the publish/subscribe communication hampered the application of conventional software engineering concepts. Moreover, the lack of sufficient structural abstractions made it difficult to design event-based systems in a modular fashion, ensure the reusability and extensibility of their components, and to keep their interactions comprehensible and manageable. In the following, we take the time to reflect how the major contributions of the thesis support and help in achieving the engineering goals we initially defined.

Modularity and composability. We considered modularity and composability as key concepts of a methodical engineering approach that enables developers to build and implement complex systems and applications out of subsystems and prefabricated components which themselves may recursively be made up of even simpler building blocks. On the level of the middleware design, we proposed a *composable architecture* for publish/subscribe brokers primarily focusing on functional modularity. We split a broker's processing and management logic into a fine-grained set of features each responsible only for a specific functional aspect and, subsequently, allowed middleware developers to freely compose and combine them as needed. Thus, from a functional point of view, brokers are highly adaptable and customizable making them versatilely deployable in a wide range of application scenarios. From a structural point of view, the proposed architecture leverages well-known object-oriented concepts to implement each feature as a pluggable component. Therefore, we specified class and component interfaces and gave development guidelines in order to ensure that feature plugins seamlessly fit when added to a broker.

Contrarily, on the application level, we first created the structural abstractions to facilitate a modular development of distributed event-driven applications. With scopes for publish/subscribe systems, we enabled application developers to bundle related components to modules and artifacts while providing them a protected environment for interaction. This way, for example, multiple instances of the same application can be embedded in private scopes and, subsequently, executed on the same publish/subscribe network without interfering with each other. Scopes, thus, implement the modularity that enables further compositions of application artifacts.

Reusability and extensibility. Modularity is an essential, albeit not sufficient, precondition for implementing reusable and extensible components. Hence, a modular and composable architecture for publish/subscribe networks does not automatically imply and guarantee extensible brokers with reusable plugins. In the course of this thesis, nevertheless, REBECA brokers showed to be easily modifiable and adaptable. In particular, we added support for scoping, extended the component plugin to a feature-rich component container, and reused existing middleware logic in our simulations evaluating the system performance. With these kind of test cases, we then substantiated the previous claims about the middleware's reusability and extensibility.

On the application level, we analyzed common pitfalls when developing publish/subscribe components and found out that often business logic is mixed up with configuration and context data which significantly limits the component's reusability and extensibility. *Event ports* addressed this problem by providing a clear interface separating the business logic which processes the data received on the event port from the configuration that determines to which notifications the event port is subscribed. While the business logic is encapsulated in the component's methods and event handlers, the configuration is either annotated to the component, contained in separated configuration classes, or stored in external files. Likewise, when using scopes to bundle components to new application artifacts, we also considered the needs and requirements of both application developers and system administrators. On the one side, we derived programming abstractions for developers to group components and connect their event ports in order to organize applications, direct the event flows between components, and ensure application's functionality. On the other side, we also provided means for administrators to subsequently adapt and fine-tune components and scopes at deployment, for instance, by defining and setting new component or scope attributes that carry additional context information or configuration data. In general, by keeping functional aspects apart from configuration issues, we improved the conditions for writing reusable and extensible software artifacts.

Comprehensibility and manageability. Publish/subscribe systems and event-driven applications are primarily characterized by the indirect form of communication and the resulting loose coupling of components. As a consequence, it is usually difficult to predict all effects and side-effects of a published event notification. It even becomes much worse when systems and applications grow in size and complexity. With *scopes*, we therefore provided structuring means to limit and control the visibility of event notifications. This way, we enabled developers to create a protected environment where application components can interact in isolation. Hence, unintended side-effects are mainly avoided making the application and system behavior significantly easier to predict and comprehend. Furthermore, with event ports, we introduced a novel publish/subscribe interface that not only separates the component's business logic from configuration data, but also clearly identifies the component fields and attributes that are going to be communicated as event notifications. Moreover, by virtually connecting out-ports and in-ports of different components, we allowed developers to conveniently specify event flows in a more direct form which eases component orchestration. As all event connections are eventually mapped onto publish/subscribe primitives, no additional dependencies are introduced that may constrain the application's extensibility and further development.

Regarding system management, scoping also plays a central role. With scope interfaces, system administrators got a way to determine and constrain which event notifications are allowed to enter or leave a particular scope and, thus, are disseminated in which part of the network to which components. By allowing scopes to be recursively divided into subscopes with own interfaces, we also took fine-grained security concerns into account. Likewise, it is possible to map organizational structures and responsibilities onto the scope hierarchy. In particular, we made sure that components can be members of different scopes at the same time supporting a flexible system organization according to multiple criteria that goes far beyond a simple functional decomposition. At the same time, the automatic assignment and grouping of application components to scopes and subscopes based on predefined properties and tagged attributes notably simplified the administration. This way, scopes not only support a modular development of event-driven applications, they are also well suited as an effective management tool for publish/subscribe infrastructures.

7.3 Outlook

Increasingly often, modern computing systems have to respond to a variety of events and situations in a timely fashion, for instance, to autonomously adapt their configuration to changing environment conditions or to actively assist and support their users. We, therefore, expect that such event-driven systems and applications will take over a central role in future infrastructures. In this thesis, we focused on engineering means to master and shape their development even when systems continue to grow in size and complexity. In the course of our work, we touched several open issues that raise new questions whose detailed discussion, however, was beyond the scope of the thesis. Nevertheless, they provide directions for future work and offer interesting new research opportunities. In the following, we give an overview about these open issues while discussing challenges and sketching first solution ideas.

Formal analysis of interacting features. We proposed a middleware architecture for publish/subscribe brokers that is based on the concept of features and their composition. We leveraged the idea of feature composition to tailor brokers to actual requirements and conditions by selecting and composing necessary features and omitting superfluous ones. This works perfectly if features are orthogonal and independent to each other, it is manageable if a feature depends on the functions of other features and these dependencies can be resolved, but it usually becomes intricate and complex if features directly or indirectly interact with each other. Please note that the latter may cause positive as well as negative effects. In the positive case, new properties and additional features may arise from the interaction making the composition result more than the sum of its parts. In the negative case, however, the individual features and functions may hamper and countervail each other when combined. For instance, we experienced oscillations when composing two features that autonomously optimized the network topology and the routing configuration, respectively, but were part of the same control loop [171]. Since such feature interactions are usually not obvious, developers can profit from a formal approach for composition. Based on a concise specification of the system and the individual features, a *formal analysis* may proof a certain set of features to be freely composable while the interactions of a different feature set are shown to have unintended side effects. In particular, the more we depend on certain features of our computing infrastructure, the more confidence we need to have in their correct behavior.

Optimized architectures. When designing our middleware architecture, we primarily focused on the concept of feature composition. Facilitating the free composition of features, we avoided to make further assumptions about the network environment and the application scenario of the middleware. On the one side, we thus gained a generic architecture to flexibly build customized publish/subscribe brokers suited for a broad range of application areas. However, if we restrict the middleware to a specific application domain, we can further adapt and optimize the architecture based on stronger assumptions and more knowledge about its purpose while taking additional requirements into account. In resource-constrained wireless sensor networks (WSNs), for example, the limited power, computation, and communication capabilities of the sensor nodes need to be considered. Here, it is probably beneficial to trade off the high degree of granularity with which independent features can be composed against more *cross-feature* and *cross-layer* optimizations that are centered around a common set of shared data structures, e.g., the routing tables. In [224] and [198], we presented a stack of composed self-organizing and self-stabilizing algorithms that may serve as a starting point for an optimized publish/subscribe architecture for WSNs. In contrast to WSNs where resources are scarce, the abundance of computing power and connectivity also presents many challenges. In the face of the ongoing trend towards multi- and many-core processor architectures, for instance, it is still an open question how to efficiently leverage the high degree of parallelism for publish/subscribe communication. Resulting architectural implications are, thus, difficult to assess today.

Security. Scoping is a key element for engineering publish/subscribe systems and event-driven applications in a modular fashion. Scopes bundle related components to new application artifacts and, thereby, restrict the dissemination of published notifications by default to members of the same scope only. We limited the visibility of events as part of the encapsulation process in order to hide internal communication details from the outside and vice versa. This way, we avoided to have internal events causing unintended side-effects elsewhere in the system. From a security point of view, however, limiting the visibility of events clearly restricts the access to data and, thus, may be used to protect the confidentiality of information. Hence, in literature [76], scoping has already been proposed for security purposes. In fact, scopes are well suited as *security domains* specifying and enforcing requirements and policies for a set of related components. In order to leverage scopes for protecting applications and data, several open issues need to be tackled. First, because of the inherent loose coupling of components, an analysis of the security requirements and needs is significantly more complex in event-based systems. Components, notifications, and even notification attributes need to be considered. Hence, scope interfaces must also support the specification of security policies at the same level of granularity. Second, the right security mechanisms such as encryption and authentication need to be chosen and applied in a way that complies to the event-based communication style. This also includes many general and conceptual questions, for instance, to which degree it is possible to perform content-based routing within a publish/subscribe network that contains trustworthy as well as untrustworthy brokers. Third, implementation details and management issues need to be discussed. In particular, the secure dissemination of scope information within the broker network as well as the management of encryption keys within a loosely coupled publish/subscribe infrastructure are challenging subjects of research.

Quality of Service. Quality of Service (QoS) is a broad, interesting, and active research topic in event-based systems [14]. Similar to the security constraints as discussed above, scopes are well suited as a place to specify and annotate QoS requirements and policies for a set of related components and their event notifications. In particular, QoS aspects include, but are not limited to, latency and bandwidth, message priorities, notification delivery and ordering guarantees, or even support for transactions. There is no general approach that covers all of these at once, instead each aspect poses specific challenges on the conceptual and the implementation level of a publish/subscribe middleware. In this context, *self-organizing* and *self-optimizing* systems based on an adaptive and reconfigurable broker network take a special position. By reorganizing their publish/subscribe infrastructure, they continually seek to ensure a constant QoS level for applications or even try to improve it while decreasing system costs at the same time. In [103] and [199], for example, we proposed self-optimizing algorithms that adapt the broker topology as well as the routing configuration and, thereby, decrease the average notification latency and the system's overall bandwidth requirements, respectively.

Event correlation and composition. In this thesis, we assumed event-based infrastructures to primarily deal with the mere transport of event notifications. With *Complex Event Processing* (CEP), there is, beyond the efficient dissemination of event notifications, another active application domain of publish/subscribe systems that is concerned with the correlation of events and the detection of spatio-temporal patterns therein. If such an event pattern is detected, a new higher-level composite event is generated that informs about the pattern's occurrence, may contain aggregated data from constituent event notifications, and itself becomes subject to further correlation and composition. The detection of event patterns is often performed by a central correlation engine, although distributed approaches promise to better exploit the locality of events and, thus, can safe a considerable amount of routing and matching overhead. Against this background, the engineering means, tools, and concepts presented in this thesis may ease a direct integration into the publish/subscribe middleware. For instance, the composable architecture enables the implementation of the correlation logic as a pluggable broker feature while scopes help to narrow down the set of potential candidate events for correlation and event ports may be used to conveniently annotate the event patterns and composite events an application component has to process. In [202] and [203], we have already sketched first ideas of such an adaptive and integrated approach to distributed event composition.

Performance modeling and analysis. With publish/subscribe networks and event-driven applications becoming an integral part of future computing infrastructures, it gets the more important not only to guarantee their correctness, but also to ensure their performance. For the latter, it is particularly essential to make reliable *performance predictions* for the system already at design time.

This way, potential bottlenecks can be identified as early as possible while the broker network is correctly sized for the expected load and traffic. In [200], we presented a stochastic performance model that is well suited for the capacity planning of publish/subscribe systems. The model takes many details such as the applied routing algorithm, the overlay network structure, and the physical links in the underlay network into account. Nevertheless, it still has to be extended to also consider the effects of scoping or different matching algorithms that leverage the increasing parallelism of modern processor architectures. In order to allow accurate quantitative estimations, the model also needs further calibration and fine-tuning by comparing results with extensive simulations as well as measurements of real publish/subscribe network deployments. Based on precise predictions, the performance of event-based infrastructures can be reliably evaluated before the real system is actually build. This provides a significant added value and represents another step towards a comprehensive engineering and development methodology for event-based systems.

Bibliography

[1] M. Adler, Z. Ge, J. Kurose, D. Towsley, and S. Zabele. “Channelization Problem In Large Scale Data Dissemination”. In: *Proceedings of the 9th International Conference on Network Protocols (ICNP ’01)*. Riverside, CA, USA, Nov. 2001, pp. 100–109.

[2] M. Altinel and M. J. Franklin. “Efficient Filtering of XML Documents for Selective Dissemination of Information”. In: *Proceedings of the 26th International Conference on Very Large Data Bases*. Ed. by A. El Abbadi, M. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang. Kairo, Ägypten: Morgan Kaufmann, Sept. 2000, pp. 53–64.

[3] S. Apel and C. Kästner. “An Overview of Feature-Oriented Software Development”. In: *Journal of Object Technology* 8.5 (July 2009), pp. 49–84.

[4] V. Arslan, P. Nienaltowski, and K. Arnout. “Event Library: An Object-Oriented Library for Event-Driven Design”. In: *Proceedings of the Joint Modular Languages Conference (JMLC ’03)*. Ed. by L. Böszörményi and P. Schojer. Vol. 2789. Lecture Notes in Computer Science. Klagenfurt, Austria: Springer, Aug. 2003, pp. 174–183.

[5] E. A. Ashcroft and W. W. Wadge. “Lucid, a Nonprocedural Language with Iteration”. In: *Communications of the ACM* 20.7 (July 1977), pp. 519–526.

[6] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. “Revised Report on the Algorithmic Language ALGOL 60”. In: *Communications of the ACM* 6 (1 Jan. 1963). Ed. by P. Naur, pp. 1–17.

[7] J. Bacon, D. M. Eyers, K. Moody, and L. I. W. Pesonen. “Securing Publish/Subscribe for Multi-domain Systems”. In: *Middleware*. Ed. by G. Alonso. Vol. 3790. Lecture Notes in Computer Science. Grenoble, France: Springer, Nov. 2005, pp. 1–20.

[8] J. Bacon, D. M. Eyers, J. Singh, and P. R. Pietzuch. “Access Control in Publish/Subscribe Systems”. In: *Proceedings of the Second International Conference on Distributed Event-Based Systems (DEBS ’08)*. Ed. by R. Baldoni. Rome, Italy: ACM Press, July 2008, pp. 23–34.

[9] J. Bacon, L. Fiege, R. Guerraoui, H.-A. Jacobsen, and G. Mühl, eds. *Proceedings of the First International Workshop on Distributed Event-Based Systems (DEBS 2002)*. Part of the Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2002). Vienna, Austria: IEEE Computer Society, July 2002.

[10] J. Bacon, K. Moody, and W. Yao. “Access Control and Trust in the Use of Widely Distributed Services”. In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*. Ed. by R. Guerraoui. Vol. 2218. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, Oct. 2001, pp. 295–310.

[11] J. Bacon, K. Moody, and W. Yao. “A Model of OASIS Role-Based Access Control and Its Support for Active Security”. In: *ACM Transactions on Information and System Security* 5.4 (4 Nov. 2002), pp. 492–540.

[12] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity.* Vol. 1. Cambridge, MA, USA: MIT Press, 1999.

[13] T. Ballardie, P. Francis, and J. Crowcroft. “Core Based Trees (CBT)”. In: *Proceedings of the ACM SIGCOMM ’93 Conference on Communications Architectures, Protocols and Applications (SIGCOMM ’93)*. San Francisco, CA, USA: ACM Press, Sept. 1993, pp. 85–95.

[14] S. Behnel, L. Fiege, and G. Mühl. “On Quality-of-Service and Publish/Subscribe”. In: *Workshop Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCSW 2006)*. Lisbon, Portugal: IEEE Computer Society, July 2006, p. 20.

[15] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. “Role-Based Access Control for Publish/Subscribe Middleware Architectures”. In: *Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS 2003)*. Ed. by H.-A. Jacobsen. San Diego, CA, USA: ACM Press, June 2003, pp. 1–8.

[16] P. A. Bernstein. “Middleware: A Model for Distributed System Services”. In: *Communications of the ACM* 39.2 (1996), pp. 86–98.

[17] T. Biggerstaff and C. Richter. “Reusability Framework, Assessment, and Directions”. In: *IEEE Software* 4.2 (Mar. 1987), pp. 41–49.

[18] S. Bittner and A. Hinze. “Dimension-Based Subscription Pruning for Publish/Subscribe Systems”. In: *Workshop Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCSW 2006)*. Lisbon, Portugal: IEEE Computer Society, July 2006.

[19] S. Bittner and A. Hinze. “Pruning Subscriptions in Distributed Publish/ Subscribe Systems”. In: *Proceedings of the 29th Australasian Computer Science Conference (ACSC 2006)*. Ed. by V. Estivill-Castro and G. Dobbie. Hobart, Tasmania, Australia: Australian Computer Society, 2006, pp. 197–206.

[20] G. W. Bond, E. Cheung, K. H. Purdy, P. Zave, and J. C. Ramming. “An Open Architecture for Next-Generation Telecommunication Services”. In: *ACM Transactions on Internet Technology* 4.1 (Feb. 2004), pp. 83–123.

[21] G. Booch. *Object-Oriented Analysis and Design with Applications*. 2nd. Redwood City, CA, USA: Benjamin Cummings Publishing, 1994.

[22] C. Bornhövd, M. Cilia, C. Liebig, and A. Buchmann. “An Infrastructure for Meta-Auctions”. In: *Proceedings of the 2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*. Ed. by P. S. Yu. Milpitas, CA, USA: IEEE Computer Society, June 2000, pp. 21–30.

[23] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. “The Feature Interaction Problem in Telecommunications Systems”. In: *Proceedings of the 7th International Conference on Software Engineering for Telecommunication Switching Systems (SETSS ’89)*. Bournemouth, UK: IEEE, July 1989, pp. 59–62.

[24] T. Bu and D. Towsley. “On Distinguishing between Internet Power Law Topology Generators”. In: *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM ’02)*. Vol. 2. New York, NY, USA: IEEE, June 2002, pp. 638–647.

[25] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. “Feature Interaction: A Critical Review and Considered Forecast”. In: *Computer Networks* 41.1 (Jan. 2003), pp. 115–141.

[26] N. Carriero and D. Gelernter. “Linda in Context”. In: *Communications of the ACM* 32.4 (Apr. 1989), pp. 444–458.

[27] A. Carzaniga, D. R. Rosenblum, and A. L. Wolf. “Challenges for Distributed Event Services: Scalability vs. Expressiveness”. In: *Proceedings of the ICSE 99 Workshop on Engineering Distributed Objects (EDO 1999)*. Ed. by W. Emmerich and V. Gruhn. Los Angeles, CA, USA: University College London, May 1999, pp. 72–77.

[28] A. Carzaniga and A. L. Wolf. “Forwarding in a Content-Based Network”. In: *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Ed. by A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall. Karlsruhe, Germany: ACM Press, Aug. 2003, pp. 163–174.

[29] A. Carzaniga. “Architectures for an Event Notification Service Scalable to Wide-area Networks”. PhD Thesis. Milan, Italy: Politecnico di Milano, 1998.

[30] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. “Issues in Supporting Event-based Architectural Styles”. In: *Proceedings of the 3rd International Workshop on Software Architecture (ISAW ’98)*. Orlando, FL, United States: ACM, 1998, pp. 17–20.

[31] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. “Design and Evaluation of a Wide-Area Event Notification Service”. In: *ACM Transactions on Computer Systems* 19.3 (Aug. 2001), pp. 332–383.

[32] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. “A Routing Scheme for Content-Based Networking”. In: *Proceedings of the 23rd IEEE International Conference on Computer Communication (INFOCOM ’04)*. Vol. 2. Hong Kong, China: IEEE, Mar. 2004, pp. 918–928.

[33] A. Carzaniga and A. L. Wolf. “Forwarding in a Content-Based Network”. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM ’03)*. Karlsruhe, Germany: ACM, 2003, pp. 163–174.

[34] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. “SCRIBE: A large-scale and decentralised application-level multicast infrastructure”. In: *IEEE Journal on Selected Areas in Communication* 20.8 (2002), pp. 100–110.

[35] C.-Y. Chan, P. A. Felber, M. Garofalakis, and R. Rastogi. “Efficient filtering of XML documents with XPath expressions”. In: *The VLDB Journal* 11.4 (Nov. 2002), 354–379.

[36] X. Chang. “Network Simulations with OPNET”. In: *Proceedings of the 1999 Winter Simulation Conference (WSC ’99)*. Phoenix, AZ, USA, Dec. 1999, pp. 307–314.

[37] X. Chen, Y. Chen, and F. Rao. “An Efficient Spatial Publish/Subscribe System for Intelligent Location-Based Services”. In: *Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS 2003)*. Ed. by H.-A. Jacobsen. San Diego, CA, USA: ACM Press, June 2003, pp. 1–6.

[38] A. Cheung and H.-A. Jacobsen. “Dynamic Load Balancing in Distributed Content-Based Publish/Subscribe”. In: *Middleware 2006*. Ed. by M. van Steen and M. Henning. Vol. 4290. Lecture Notes in Computer Science. Springer, 2006, pp. 141–161.

[39] Y.-H. Chu, S. Rao, and H. Zhang. “A Case for End System Multicast”. In: *Proceedings of the ACM 2000 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’00)*. Santa Clara, CA, USA: ACM Press, June 2000, pp. 1–12.

[40] M. Cilia, M. Antollini, C. Bornhövd, and A. Buchmann. “Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach”. In: *Proceedings of the Third International Workshop on Distributed Event-Based Systems (DEBS 2004)*. Ed. by A. Carzaniga and P. Fenkam. Edinburgh, Scotland, UK: IET, May 2004, pp. 26–31.

[41] M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. P. Buchmann. "Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware". In: *Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS 2003)*. Ed. by H.-A. Jacobsen. San Diego, CA, USA: ACM Press, June 2003, pp. 1–8.

[42] M. Cilia. "An Active Functionality Service for Open Distributed Heterogeneous Environments". Ph.D. Thesis. Darmstadt, Germany: Technische Universität Darmstadt, Aug. 2002.

[43] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Boston, MA, USA: Addison-Wesley, 2002.

[44] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. Network Working Group, May 2008.

[45] G. Cugola and J. E. M. de Cote. "On Introducing Location Awareness in Publish-Subscribe Middleware". In: *Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS 2005)*. Ed. by J. Dingel and R. Storm. Part of the Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW 2005). Columbus, OH, USA: IEEE Computer Society, June 2005, pp. 377–382.

[46] G. Cugola, E. Di Nitto, and A. Fuggetta. "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS". In: *IEEE Transactions on Software Engineering* 27.9 (Sept. 2001), pp. 827–850.

[47] G. Cugola, A. Margara, and M. Migliavacca. "Context-Aware Publish-Subscribe: Model, Implementation, and Evaluation". In: *Proceedings of the IEEE Symposium on Computers and Communications (ISCC 2009)*. Sousse, Tunisia: IEEE, July 2009, pp. 875–881.

[48] G. Cugola and G. P. Picco. "REDS: A Reconfigurable Dispatching System". In: *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM '06)*. Ed. by E. Wohlstadter and C. Zhang. Portland, OR, USA: ACM Press, Nov. 2006, pp. 9–16.

[49] J. B. Dabney and T. L. Harman. *Mastering SIMULINK*. Prentice Hall, Nov. 2003.

[50] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. London, UK: Academic Press Ltd., 1972.

[51] J. B. Dennis and D. Misunas. "A Preliminary Architecture for a Basic Data Flow Processor". In: *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA 1975)*. Ed. by W. K. King and O. N. Garcia. Housten, TX, USA: ACM Press, Jan. 1975, pp. 126–132.

[52] R. Devine. “Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm”. In: *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO ’93)*. Ed. by D. B. Lomet. Vol. 730. Lecture Notes in Computer Science. Chicago, IL, USA: Springer, Oct. 1993, pp. 101–114.

[53] A. K. Dey. “Understanding and Using Context”. In: *Personal and Ubiquitous Computing* 5 (1 Feb. 2001), pp. 4–7.

[54] K. R. Dittrich, S. Gatziu, and A. Geppert. “The Active Database Management System Manifesto: A Rulebase of ADBMS Features”. In: *Proceedings of the 2nd International Workshop on Rules in Database Systems (RIDS ’95)*. Ed. by T. Sellis. Vol. 985. Lecture Notes in Computer Science. Glyfada, Athens, Greece: Springer, Sept. 1995, pp. 3–20.

[55] G. Eisenhauer, K. Schwan, and F. E. Bustamante. “Publish–Subscribe for High-Performance Computing”. In: *IEEE Internet Computing* 10.1 (2006), pp. 40–47.

[56] P. T. Eugster. “Type-based Publish/Subscribe”. Ph.D. Thesis. Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne, Dec. 2001.

[57] P. T. Eugster. “Type-Based Publish/Subscribe: Concepts and Experiences”. In: *ACM Transactions on Programming Languages and Systems* 29.1 (Jan. 2007).

[58] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. “The Many Faces of Publish/Subscribe”. In: *ACM Computing Surveys* 35.2 (June 2003), pp. 114–131.

[59] P. T. Eugster, B. Garbinato, and A. Holzer. “Location-based Publish/Subscribe”. In: *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications (NCA 2005)*. Cambridge, MA, USA: IEEE Computer Society, July 2005, pp. 279–282.

[60] P. T. Eugster and R. Guerraoui. “Content-Based Publish/Subscribe with Structural Reflection”. In: *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*. USENIX Association. San Antonio, TX, USA, Jan. 2001, pp. 131–146.

[61] P. T. Eugster and R. Guerraoui. “Distributed Programming with Typed Events”. In: *IEEE Software* 21.2 (Mar. 2004), pp. 56–64.

[62] P. T. Eugster, R. Guerraoui, and C. H. Damm. “On Objects and Events”. In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’01)*. Tampa Bay, FL, USA: ACM, 2001, pp. 254–269.

[63] P. T. Eugster, R. Guerraoui, and J. Sventek. “Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction”. In: *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*. Ed. by E. Bertino. Vol. 1850. Sophia Antipolis and Cannes, France: Springer, June 2000, pp. 252–276.

[64] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01).* Santa Barbara, CA, USA: ACM, 2001, pp. 115–126.

[65] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. "A Report on the Sisal Language Project". In: *Journal of Parallel and Distributed Computing* 10.4 (1990), pp. 349–366.

[66] J. Ferber. "Computational Reflection in Class-based Object-Oriented Languages". In: *Proceedings of the 4th International Conference on Object-Oriented Programming Systems, Languages and Applications.* New Orleans, LA, USA: ACM, 1989, pp. 317–326.

[67] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. "The PADRES Distributed Publish/Subscribe System". In: *Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI '05).* Ed. by S. Reiff-Marganiec and M. Ryan. Leicester, UK: IOS Press, June 2005, pp. 12–30.

[68] L. Fiege. "Visibility in Event-Based Systems". Ph.D. Thesis. Darmstadt, Germany: Technische Universität Darmstadt, Apr. 2005.

[69] L. Fiege, M. Cilia, G. Mühl, and A. Buchmann. "Publish/Subscribe Grows Up: Support for Management, Visibility Control, and Heterogeneity". In: *IEEE Internet Computing* 10.1 (Jan. 2006), pp. 48–55.

[70] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. "Supporting Mobility in Content-Based Publish/Subscribe Middleware". In: *Proceedings of the 2003 ACM/IFIP/USENIX International Middleware Conference (Middleware '03).* Ed. by M. Endler and D. C. Schmidt. Vol. 2672. Lecture Notes in Computer Science. Rio de Janeiro, Brazil: Springer, June 2003, pp. 103–122.

[71] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. "Engineering Event-based Systems with Scopes". In: *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002).* Ed. by B. Magnusson. Vol. 2374. Lecture Notes in Computer Science. Málaga, Spain: Springer, June 2002, pp. 309–333.

[72] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. "Visibility as Central Abstraction in Event-based Systems". In: *Concrete Communication Abstractions of the Next 701 Distributed Object Systems (ECOOP 2002 Workshop).* Ed. by A. Beugnard, S. Sadou, L. Duchien, and E. Jul. Vol. 2548. Lecture Notes in Computer Science. Málaga, Spain: Springer, June 2002.

[73] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. "Komponenten in ereignisbasierten Systemen". In: *Thema Forschung* 4.1 (2003). In German., pp. 108–114.

[74] 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).* Madrid, Spain: ACM Press, Mar. 2002, pp. 385–392.

[75] L. Fiege, G. Mühl, and F. C. Gärtner. “Modular event-based systems”. In: *The Knowledge Engineering Review* 17.4 (Dec. 2002), pp. 359–388.

[76] L. Fiege, A. Zeidler, A. P. Buchmann, R. Kilian-Kehr, and G. Mühl. “Security Aspects in Publish/Subscribe Systems”. In: *Proceedings of the 3rd International Workshop on Distributed Event-Based Systems (DEBS ’04).* Edinburgh, Scotland, UK: IEE The Institution of Electrical Engineers, May 2004, pp. 44–49.

[77] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. Network Working Group, June 1999.

[78] C. L. Forgy. “Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem”. In: *Artificial Intelligence* 19.1 (1982), pp. 17–37.

[79] D. Frey and G.-C. Roman. “Context-Aware Publish Subscribe in Mobile Ad Hoc Networks”. In: *Proceedings of the 9th International Conference on Coordination Models and Languages.* Ed. by A. L. Murphy and J. Vitek. Vol. 4467. Lecture Notes in Computer Science. Paphos, Cyprus: Springer, June 2007, pp. 37–55.

[80] E. Friedman-Hill. *Jess in Action: Rule-Based Systems in Java.* Greenwich, CT, USA: Manning Publications, June 2003.

[81] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston, MA, USA: Addison-Wesley, 1995.

[82] N. H. Gehani, H. V. Jagadish, and O. Shmueli. “Composite Event Specification in Active Databases: Model & Implementation”. In: *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB ’92).* Ed. by L.-Y. Yuan. Vancouver, Canada: Morgan Kaufmann, Aug. 1992, pp. 327–338.

[83] D. Gelernter. “Generative Communication in Linda”. In: *ACM Transactions on Programming Languages and Systems* 7.1 (Jan. 1985), pp. 80–112.

[84] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* 2nd ed. Addison-Wesley, June 2000.

[85] D. Graff, H. Parzyjegla, J. Richling, and M. Werner. “Verteilte aktive Objekte für verteilte mobile Systeme”. In: *Tagungsband zum 7. GI/ITG KuVS-Fachgespräch Ortsbezogene Anwendungen und Dienste.* Ed. by A. Küpper and J. Roth. Berlin, Germany: Logos Verlag, Sept. 2010, pp. 55–62.

[86] D. Graff, M. Werner, H. Parzyjegla, J. Richling, and G. Mühl. “An Object-Oriented and Context-Aware Approach for Distributed Mobile Applications”. In: *Workshop Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS 2010 Workshops).* Ed. by M. Beigl and F. J. Cazorla-Almeida. Hannover, Germany: VDE Verlag, Feb. 2010, pp. 191–200.

[87] R. Guerraoui, ed. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001).* Vol. 2218. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, Oct. 2001.

[88] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification.* 2003.

[89] M. Henning. “The Rise and Fall of CORBA”. In: *Queue* 4.5 (June 2006), pp. 28–34.

[90] R. Hickey. “The Clojure Programming Language”. In: *Proceedings of the 2008 Symposium on Dynamic Languages (DLS ’08).* Paphos, Cyprus: ACM, 2008, p. 1.

[91] D. D. Hils. “Visual Languages and Computing Survey: Data Flow Visual Programming Languages”. In: *Journal of Visual Languages and Computing* 3.1 (1992), pp. 69–101.

[92] A. Hinze and A. P. Buchmann, eds. *Principles and Applications of Distributed Event-Based Systems.* IGI Global, June 2010.

[93] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston, MA, USA: Addison-Wesley, 2003.

[94] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. “The Evolution of Lua”. In: *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages.* San Diego, California: ACM, 2007, pp. 1–26.

[95] V. Issarny, M. Caporuscio, and N. Georgantas. “A Perspective on the Future of Middleware-based Software Engineering”. In: *Future of Software Engineering (FOSE ’07).* Minneapolis, MN, USA: IEEE Computer Society, 2007, pp. 244–258.

[96] M. Jackson and P. Zave. “Distributed Feature Composition: A Virtual Architecture for Telecommunications Services”. In: *IEEE Transactions on Software Engineering* 24.10 (Oct. 1998), pp. 831–847.

[97] H.-A. Jacobsen, ed. *Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS 2003).* San Diego, CA, USA: ACM Press, June 2003.

[98] H.-A. Jacobsen, A. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. “The PADRES Publish/Subscribe System”. In: *Principles and Applications of Distributed Event-Based Systems.* Ed. by A. Hinze and A. P. Buchmann. IGI Global, June 2010. Chap. 8, pp. 164–205.

[99] H.-A. Jacobsen, G. Mühl, and M. A. Jaeger, eds. *Proceedings of the Inaugural International Conference on Distributed Event-Based Systems.* Toronto, ON, Canada: ACM Press, June 2007.

[100] M. A. Jaeger. "Self-Managing Publish/Subscribe Systems". Ph.D. Thesis. Berlin, Germany: Technische Universität Berlin, Dec. 2007.

[101] M. A. Jaeger, G. Mühl, M. Werner, and H. Parzyjegla. "Reconfiguring Self-Stabilizing Publish/Subscribe Systems". In: *Proceedings of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2006).* Ed. by R. State, S. van der Meer, D. O'Sullivan, and T. Pfeifer. Vol. 4269. Lecture Notes in Computer Science. Dublin, Ireland: Springer, Oct. 2006, pp. 233–238.

[102] M. A. Jaeger, G. Mühl, M. Werner, H. Parzyjegla, and H.-U. Heiss. "Algorithms for Reconfiguring Self-Stabilizing Publish/Subscribe Systems". In: *Proceedings of the 8th International Workshop held at Shanghai Jiao Tong University: Autonomous Systems – Self-Organization, Management, and Control.* Ed. by B. Mahr and S. Huanye. Shanghai, China: Springer, Oct. 2008, pp. 135–147.

[103] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann. "Self-Organizing Broker Topologies for Publish/Subscribe Systems". In: *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC 2007).* Ed. by Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo. Seoul, Korea: ACM Press, Mar. 2007, pp. 543–550.

[104] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. "Advances in Dataflow Programming Languages". In: *ACM Computing Surveys* 36.1 (Mar. 2004), pp. 1–34.

[105] G. Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Information Processing '74: Proceedings of the IFIP Congress.* Ed. by J. L. Rosenfeld. Stockholm, Sweden: North-Holland, Aug. 1974, pp. 471–475.

[106] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA): Feasibility Study.* Tech. rep. CMU/SEI-90-TR-21. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[107] A. M. Keller, R. Jensen, and S. Agarwal. "Persistence Software: Bridging Object-Oriented Programming and Relational Databases". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93).* Washington, DC, USA: ACM, 1993, pp. 523–528.

[108] J. S. Kilby. "Invention of the Integrated Circuit". In: *IEEE Transactions on Electron Devices* 23.7 (July 1976), pp. 648–654.

[109] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action.* Greenwich, CT, USA: Manning Publications, Jan. 2007.

[110] H. Kopetz, G. Grünsteidl, and J Reisinger. "Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System". In: *Proceedings of the IFIP Working Conference on Dependable Computing for Critical Applications.* Vol. 4. Dependable Computing and Fault-Tolerant Systems. Santa Barbara, CA, USA, Aug. 1989, pp. 411–429.

[111] H. Kopetz and R. Obermaisser. "Temporal Composability". In: *Computing Control Engineering Journal* 13.4 (Aug. 2002), pp. 156–162.

[112] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Dordrecht, The Netherlands: Kluwer Academic Publishers, 1997.

[113] H. Kopetz and G. Bauer. "The Time-Triggered Architecture". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 112–126.

[114] H. Kopetz and W. Ochsenreiter. "Clock Synchronization in Distributed Real-Time Systems". In: *IEEE Transactions on Computers* C-36.8 (Aug. 1987), pp. 933–940.

[115] G. E. Krasner and S. T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". In: *Journal of Object Oriented Programming* 1.3 (Aug. 1988), pp. 26–49.

[116] A. M. Law. *Simulation Modeling and Analysis.* 4th ed. Mcgraw Hill Higher Education, 2006.

[117] M. Lehman. "Programs, Life Cycles, and Laws of Software Evolution". In: *Proceedings of the IEEE* 68.9 (Sept. 1980), pp. 1060–1076.

[118] M. Lehman. "Laws of Software Evolution Revisited". In: *Proceedings of the 5th European Workshop on Software Process Technology (EWSPT '96).* Ed. by C. Montangero. Vol. 1149. Lecture Notes in Computer Science. Nancy, France: Springer, Oct. 1996, pp. 108–124.

[119] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff. "A Brief History of the Internet". In: *ACM SIGCOMM Computer Communication Review* 39.5 (Oct. 2009), pp. 22–31.

[120] G. Li, A. Cheung, S. Hou, S. Hu, V. Muthusamy, R. Sherafat, A. Wun, H.-A. Jacobsen, and S. Manovski. "Historic Data Access in Publish/Subscribe". In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems (DEBS '07).* Toronto, Ontario, Canada: ACM, 2007, pp. 80–84.

[121] G. Li, V. Muthusamy, and H.-A. Jacobsen. "Adaptive content-based routing in general overlay topologies". In: *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware.* Leuven, Belgium: Springer-Verlag New York, Inc., 2008, pp. 1–21.

[122] G. Li, V. Muthusamy, and H.-A. Jacobsen. "A Distributed Service-Oriented Architecture for Business Process Execution". In: *ACM Transactions on the Web* 4.1 (Jan. 2010), 2:1–2:33.

[123] G. Li, V. Muthusamy, and H.-A. Jacobsen. "A Distributed Service-Oriented Architecture for Business Process Execution". In: *ACM Transactions on the Web* 4.1 (Jan. 2010), pp. 1–33.

[124] M. A. Linton, J. M. Vlissides, and P. R. Calder. *Composing User Interfaces with InterViews*. Tech. rep. CSL-TR-88-369. Stanford, CA, USA: Stanford University, Nov. 1988.

[125] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley, May 2002.

[126] M. Lutz. *Programming Python*. 4th ed. Sebastopol, CA, USA: O'Reilly, Jan. 2011.

[127] P. Maes. "Concepts and Experiments in Computational Reflection". In: *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications*. Orlando, FL, USA: ACM, 1987, pp. 147–155.

[128] J. L. Martins and S. Duarte. "Routing Algorithms for Content-Based Publish/Subscribe Systems". In: *IEEE Communications Surveys and Tutorials* 12.1 (2010), pp. 39–58.

[129] D. McCarthy and U. Dayal. "The Architecture of an Active Database Management System". In: ed. by J. Clifford, B. Lindsay, and D. Maier. Portland, OR, USA: ACM Press, 1989, pp. 215–224.

[130] A. Medina, A. Lakhina, I. Matta, and J. Byers. "BRITE: An Approach to Universal Topology Generation". In: *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*. Cincinnati, OH, USA, Aug. 2001, pp. 346–353.

[131] A. Medina, I. Matta, and J. Byers. "On the Origin of Power Laws in Internet Topologies". In: *ACM SIGCOMM Computer Communication Review* 30.2 (Apr. 2000), pp. 18–28.

[132] R. Meier and V. Cahill. "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks". In: *Proceedings of the First International Workshop on Distributed Event-Based Systems (DEBS 2002)*. Ed. by J. Bacon, L. Fiege, R. Guerraoui, H.-A. Jacobsen, and G. Mühl. Part of the Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2002). Vienna, Austria: IEEE Computer Society, July 2002, pp. 639–644.

[133] R. Meier and V. Cahill. "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications". In: *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2003)*. Ed. by J.-B. Stefani, I. Demeure, and D. Hagimont. Vol. 2893. Lecture Notes in Computer Science. Paris, France: Springer, Nov. 2003, pp. 285–296.

[134] J. Mersel. “Program Interrupt on the Univac Scientific Computer”. In: *Proceedings of the Western Joint Computer Conference (WJCC ’56)*. San Francisco, CA, USA: ACM, Feb. 1956, pp. 52–53.

[135] B. Meyer. *Eiffel: The Language*. Prentice Hall, Oct. 1991.

[136] B. Meyer. “The Power of Abstraction, Reuse, and Simplicity: An Object-Oriented Library for Event-Driven Design”. In: *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*. Ed. by O. Owe, S. Krogdahl, and T. Lyche. Vol. 2635. Springer, 2004, pp. 236–271.

[137] Z. Miklós. “Towards an Access Control Mechanism for Wide-area Publish/Subscribe Systems”. In: *Proceedings of the First International Workshop on Distributed Event-Based Systems (DEBS 2002)*. Ed. by J. Bacon, L. Fiege, R. Guerraoui, H.-A. Jacobsen, and G. Mühl. Part of the Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2002). Vienna, Austria: IEEE Computer Society, July 2002, pp. 516–521.

[138] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Vol. 1. Boston, MA, USA: Addison-Wesley, Nov. 2012.

[139] A. Montresor and M. Jelasity. “PeerSim: A Scalable P2P Simulator”. In: *Proceedings of the 9th IEEE International Conference on Peer-to-Peer Computing (P2P ’09)*. Seattle, WA, USA: IEEE, Sept. 2009, pp. 99–100.

[140] L. Moreau. “A Syntactic Theory of Dynamic Binding”. In: *Higher-Order and Symbolic Computation* 11.3 (3 Sept. 1998), pp. 233–279.

[141] J. P. Morrison. *Flow-Based Programming: A New Approach to Application Development*. 2nd ed. CreateSpace, May 2010.

[142] G. Mühl, L. Fiege, F. C. Gärtner, and A. P. Buchmann. “Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems”. In: *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS ’02)*. Ed. by A. Boukerche, S. Das, and S. Majumdar. Fort Worth, TX, USA: IEEE Computer Society, Oct. 2002, pp. 167–176.

[143] G. Mühl. “Generic Constraints for Content-Based Publish/Subscribe Systems”. In: *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS 2001)*. Ed. by C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella. Vol. 2172. Lecture Notes in Computer Science. Trento, Italy: Springer, Sept. 2001, pp. 211–225.

[144] G. Mühl. “Large-Scale Content-Based Publish/Subscribe Systems”. Ph.D. Thesis. Darmstadt, Germany: Technische Universität Darmstadt, Sept. 2002.

[145] G. Mühl and L. Fiege. “Supporting Covering and Merging in Content-Based Publish/Subscribe Systems: Beyond Name/Value Pairs”. In: *IEEE Distributed Systems Online (DSOnline)* 2.7 (2001).

[146] G. Mühl, L. Fiege, and A. P. Buchmann. “Filter Similarities in Content-Based Publish/Subscribe Systems”. In: *Proceedings of the 15th International Conference on Architecture of Computing Systems (ARCS 2002).* Ed. by H. Schmeck, T. Ungerer, and L. C. Wolf. Vol. 2299. Lecture Notes in Computer Science. Karlsruhe, Germany: Springer, Apr. 2002, pp. 224–238.

[147] G. Mühl, L. Fiege, and P. R. Pietzuch. *Distributed Event-Based Systems.* Berlin/Heidelberg, Germany: Springer, Aug. 2006.

[148] G. Mühl, A. Schröter, H. Parzyjegla, S. Kounev, and J. Richling. “Stochastic Analysis of Hierarchical Publish/Subscribe Systems”. In: *Proceedings of the 15th European Conference on Parallel Processing (Euro-Par 2009).* Ed. by H. Sips, D. Epema, and H.-X. Lin. Vol. 5704. Lecture Notes in Computer Science. Delft, The Netherlands: Springer, Aug. 2009, pp. 97–109.

[149] G. Mühl, A. Ulbrich, K. Herrmann, and T. Weis. “Disseminating Information to Mobile Clients Using Publish-Subscribe”. In: *IEEE Internet Computing* 8.3 (May 2004), pp. 46–53.

[150] G. Mühl, M. Werner, M. A. Jaeger, K. Herrmann, and H. Parzyjegla. “On the Definitions of Self-Managing and Self-Organizing Systems”. In: *Workshop Proceedings of the 15th GI/ITG Conference on Communication in Distributed Systems (KiVS 2007 Workshops).* Ed. by T. Braun, G. Carle, and B. Stiller. Bern, Switzerland: VDE Verlag, Mar. 2007, pp. 291–301.

[151] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. “Multiple Dispatch in Practice”. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications.* Nashville, TN, USA: ACM, 2008, pp. 563–582.

[152] M. Newman. “Power laws, Pareto distributions and Zipf’s law”. In: *Contemporary Physics* 46.5 (Sept. 2005), pp. 323–351.

[153] S. Oaks. *Java Security.* 2nd ed. O’Reilly, May 2001.

[154] Object Management Group, Inc. (OMG). *Persistent State Service Specification, Version 2.0.* Needham, MA, USA, Sept. 2002.

[155] Object Management Group, Inc. (OMG). *Security Service Specification, Version 1.8.* Needham, MA, USA, Mar. 2002.

[156] Object Management Group, Inc. (OMG). *Transaction Service Specification, Version 1.4.* Needham, MA, USA, Sept. 2003.

[157] Object Management Group, Inc. (OMG). *Event Service Specification, Version 1.2.* Needham, MA, USA, Oct. 2004.

[158] Object Management Group, Inc. (OMG). *Notification Service Specification, Version 1.1.* Needham, MA, USA, Oct. 2004.

[159] Object Management Group, Inc. (OMG). *Data Distribution Service for Real-time Systems (DDS), Version 1.2.* Needham, MA, USA, Jan. 2007.

[160] Object Management Group, Inc. (OMG). *Common Object Request Broker Architecture (CORBA) Specification, Version 3.2*. Needham, MA, USA, Nov. 2011.

[161] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. “The Information Bus: An Architecture for Extensible Distributed Systems”. In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP ’93)*. Ed. by A. P. Black and B. Liskov. Asheville, NC, USA: ACM Press, Dec. 1993, pp. 58–68.

[162] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. “Exploiting IP Multicast in Content-Based Publish-Subscribe Systems”. In: *Proceedings of the Middleware 2000 IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*. Ed. by J. S. Sventek and G. Coulson. Vol. 1795. Lecture Notes in Computer Science. New York, NY, USA: Springer, Apr. 2000, pp. 185–207.

[163] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language, Version 2.0*. Billerica, MA, USA, Apr. 2007.

[164] Organization for the Advancement of Structured Information Standards (OASIS). *Advanced Message Queuing Protocol (AMQP), Version 1.0*. Billerica, MA, USA, Oct. 2012.

[165] D. L. Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules”. In: *Communications of the ACM* 15 (12 Dec. 1972), pp. 1053–1058.

[166] H. Parzyjegla, D. Graff, A. Schröter, J. Richling, and G. Mühl. “Design and Implementation of the Rebeca Publish/Subscribe Middleware”. In: *From Active Data Management to Event-Based Systems and More*. Ed. by K. Sachs, I. Petrov, and P. Guerrero. Vol. 6462. Lecture Notes in Computer Science. Berlin/Heidelberg, Germany: Springer, Nov. 2010, pp. 124–140.

[167] H. Parzyjegla, M. A. Jaeger, G. Mühl, and T. Weis. “A Model-driven Approach to the Development of Autonomous Control Applications”. In: *Proceedings of the 1st Workshop on Model-driven Software Adaptation (M-ADAPT 2007) at ECOOP 2007*. Ed. by G. Blair, N. Bencomo, R. France, and M. Cebulla. Vol. 2007-10. Berlin, Germany: Technische Universität Berlin, July 2007, pp. 25–27.

[168] H. Parzyjegla, M. A. Jaeger, G. Mühl, and T. Weis. “Model-driven Development and Adaptation of Autonomous Control Applications”. In: *IEEE Distributed Systems Online* 9.11 (Nov. 2008), pp. 1–9.

[169] H. Parzyjegla, G. Mühl, and M. A. Jaeger. “Reconfiguring Publish/Subscribe Overlay Topologies”. In: *Workshop Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCSW 2006)*. Lisbon, Portugal: IEEE Computer Society, July 2006, p. 29.

[170] H. Parzyjegla, A. Schröter, A. Busse, D. Graff, A. Schepeljanski, J. Richling, M. Werner, and G. Mühl. "REBECA – Eine autonome Publish/Subscribe Middleware". In: *Praxis der Informationsverarbeitung und Kommunikation (PIK)* 34.3 (Aug. 2011), pp. 135–137.

[171] H. Parzyjegla, A. Schröter, D. Graff, A. Busse, A. Schepeljanski, J. Richling, M. Werner, and G. Mühl. "Autonomy Features and Feature Composition in REBECA". In: *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011).* Karlsruhe, Germany, June 2011.

[172] N. W. Paton and O. Díaz. "Active Database Systems". In: *ACM Computing Surveys* 31.1 (Mar. 1999), pp. 63–103.

[173] L. I. W. Pesonen and J. Bacon. "Secure Event Types in Content-Based, Multi-Domain Publish/Subscribe Systems". In: *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM '05).* Lisbon, Portugal: ACM, 2005, pp. 98–105.

[174] L. I. W. Pesonen, D. M. Eyers, and J. Bacon. "A Capability-Based Access Control Architecture for Multi-Domain Publish/Subscribe Systems". In: *Proceedings of the 2006 International Symposium on Applications and the Internet (SAINT '06).* Phoenix, AZ, USA, Jan. 2006, pp. 222–228.

[175] L. I. W. Pesonen, D. M. Eyers, and J. Bacon. "Access Control in Decentralised Publish/Subscribe Systems". In: *Journal of Networks* 2.2 (Apr. 2007), pp. 57–67.

[176] L. I. W. Pesonen, D. M. Eyers, and J. Bacon. "Encryption-Enforced Access Control in Dynamic Multi-Domain Publish/Subscribe Networks". In: *Proceedings of the Inaugural Conference on Distributed Event-Based Systems.* Ed. by H.-A. Jacobsen, G. Mühl, and M. A. Jaeger. Toronto, ON, Canada: ACM Press, June 2007, pp. 104–115.

[177] P. Pietzuch, D. Eyers, S. Kounev, and B. Shand. "Towards a Common API for Publish/Subscribe". In: *Proceedings of the Inaugural Conference on Distributed Event-Based Systems.* Ed. by H.-A. Jacobsen, G. Mühl, and M. A. Jaeger. Toronto, ON, Canada: ACM, June 2007, pp. 152–157.

[178] P. R. Pietzuch. "Hermes: A Scalable Event-Based Middleware". Ph.D. Thesis. Cambridge, UK: Queens' College, University of Cambridge, Feb. 2004.

[179] P. R. Pietzuch and J. Bacon. "Hermes: A Distributed Event-Based Middleware Architecture". In: *Proceedings of the First International Workshop on Distributed Event-Based Systems (DEBS 2002).* Ed. by J. Bacon, L. Fiege, R. Guerraoui, H.-A. Jacobsen, and G. Mühl. Part of the Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2002). Vienna, Austria: IEEE Computer Society, July 2002, pp. 611–618.

[180] P. R. Pietzuch and J. Bacon. “Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware”. In: *Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS 2003)*. Ed. by H.-A. Jacobsen. San Diego, CA, USA: ACM, June 2003, pp. 1–8.

[181] P. Pietzuch and S. Bhola. “Congestion Control in a Reliable Scalable Message-Oriented Middleware”. In: *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference (Middleware ’03)*. Vol. 2672. Lecture Notes in Computer Science. Rio de Janeiro, Brasilien: Springer, June 2003, pp. 202–221.

[182] P. Pietzuch, B. Shand, and J. Bacon. “Composite Event Detection as a Generic Middleware Extension”. In: *IEEE Network Magazine* 18.1 (Jan. 2004), pp. 44–55.

[183] C. Prehofer. “Feature-Oriented Programming: A Fresh Look at Objects”. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*. Ed. by M. Aksit and S. Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Jyväskylä, Finland: Springer, June 1997, pp. 419–443.

[184] C. Prehofer. “Feature-oriented programming: A new way of object composition”. In: *Concurrency and Computation: Practice and Experience* 13.6 (May 2001), pp. 465–501.

[185] J. Richling. “Message Scheduled System – A Composable Architecture for Embedded Real-Time Systems”. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*. Ed. by H. R. Arabnia. Las Vegas, NV, USA: CSREA Press, June 2000.

[186] J. Richling. *Komponierbarkeit eingebetteter Echtzeitsysteme*. Cuvillier Verlag, 2006.

[187] J. Richling and M. Malek. “Message Scheduled System (MSS): A Composable Architecture for Distributed Real-Time Systems”. In: *Workshop Proceedings of the 13th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB Workshop Proceedings 2006)*. Ed. by W. Dulz and W. Schröder-Preikschat. Nürnberg, Germany: VDE Verlag, Mar. 2006.

[188] J. Richling, L. Popova-Zeugmann, and M. Werner. “Verification of Non-functional Properties of a Composable Architecture with Petrinets”. In: *Fundamenta Informaticae* 51.1-2 (2002), pp. 185–200.

[189] J. Richling, M. Werner, and L. Popova-Zeugmann. “A Formally-Proven Composable Architecture for Real-Time Systems”. In: *Proceedings of the Workshop about Architectures for Cooperative Embedded Real-Time Systems (WACERTS) at the 25th IEEE Real-Time Systems Symposium (RTSS ’04)*. Lisbon, Portugal, Dec. 2004, pp. 31–34.

[190] G. F. Riley and T. R. Henderson. "The *ns-3* Network Simulator". In: *Modeling and Tools for Network Simulation.* Ed. by K. Wehrle, M. Günes, and J. Gross. Berlin/Heidelberg, Germany: Springer, 2010, pp. 15–34.

[191] W. Rjaibi, K. R. Dittrich, and D. Jaepel. "Event Matching in Symmetric Subscription Systems". In: *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research.* Ed. by D. A. S. Stewart and J. H. Johnson. Toronto, Ontario, Canada: Pub-IBM, 2002, p. 9.

[192] S. Rosen. "Electronic Computers: A Historical Survey". In: *ACM Computing Surveys* 1.1 (Mar. 1969), pp. 7–36.

[193] D. S. Rosenblum and A. L. Wolf. "A Design Framework for Internet-Scale Event Observation and Notification". In: *Proceedings of the 6th European Software Engineering Conference (ESEC '97).* Zurich, Switzerland: Springer, 1997, pp. 344–360.

[194] A. I. Rowstron and P. Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001).* Ed. by R. Guerraoui. Vol. 2218. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, Oct. 2001, pp. 329–350.

[195] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. "Role-Based Access Control Models". In: *Computer* 29.2 (Feb. 1996), pp. 38–47.

[196] M. Santori. "An instrument that isn't really (Laboratory Virtual Instrument Engineering Workbench)". In: *IEEE Spectrum* 27.8 (Aug. 1990), pp. 36–39.

[197] D. C. Schmidt. "Middleware for Real-time and Embedded Systems". In: *Communications of the ACM* 45.6 (June 2002), pp. 43–48.

[198] J. H. Schönherr, H. Parzyjegla, and G. Mühl. "Clustered Publish/Subscribe in Wireless Actuator and Sensor Networks". In: *Proceedings of the 6th International Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC 2008).* Ed. by S. Terzis. Leuven, Belgium: ACM Press, Dec. 2008, pp. 60–65.

[199] A. Schröter, D. Graff, G. Mühl, J. Richling, and H. Parzyjegla. "Self-optimizing Hybrid Routing in Publish/Subscribe Systems". In: *Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2009).* Ed. by C. Bartolini and L. P. Gaspary. Vol. 5841. Lecture Notes in Computer Science. Venice, Italy: Springer, Oct. 2009, pp. 111–122.

[200] A. Schröter, G. Mühl, S. Kounev, H. Parzyjegla, and J. Richling. "Stochastic Performance Analysis and Capacity Planning of Publish/Subscribe Systems". In: *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS '10)*. Cambridge, UK: ACM Press, July 2010, pp. 258–269.

[201] A. Schröter, G. Mühl, J. Richling, and H. Parzyjegla. "Adaptive Routing in Publish/Subscribe Systems using Hybrid Routing Algorithms". In: *Proceedings of the 7th Workshop on Adaptive and Reflective Middleware (ARM 2008)*. Ed. by F. Taïani and R. Cerqueira. Leuven, Belgium: ACM Press, Dec. 2008, pp. 51–52.

[202] E. Seib, H. Parzyjegla, and G. Mühl. "Distributed Composite Event Detection in Publish/Subscribe Networks – A Case for Self-Organization". In: *Workshop Proceedings of the 17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011 Workshops)*. Ed. by H. Hellbruck, N. Luttenberger, and V. Turau. Vol. 37. Kiel, Germany: European Association of Software Science and Technology (EASST), Mar. 2011, pp. 1–12.

[203] E. Seib, H. Parzyjegla, and G. Mühl. "Adaptive Distributed Composite Event Detection". In: *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware (ARM 2012)*. Ed. by P. Ferreira, L. Veiga, and F. Araújo. Montreal, QC, Canada: ACM Press, Dec. 2012, 2:1–2:6.

[204] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.

[205] J. Singh. "Controlling the Dissemination and Disclosure of Healthcare Events". Ph.D. Thesis. Cambridge, UK: Queens' College, University of Cambridge.

[206] J. Singh and J. Bacon. "Event-Based Data Control in Healthcare". In: *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*. Leuven, Belgium: ACM, Dec. 2008, pp. 84–86.

[207] J. Singh and J. Bacon. "Event-Based Data Dissemination Control in Healthcare". In: *Proceedings of the First International Conference on Electronic Healthcare (eHealth 2008)*. Ed. by D. Weerasinghe. London, UK: Springer, Sept. 2008, pp. 167–174.

[208] J. Singh, D. M. Eyers, and J. Bacon. "Credential Management in Event-Driven Healthcare Systems". In: *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*. Leuven, Belgium: ACM, Dec. 2008, pp. 48–53.

[209] M. Srivatsa and L. Liu. "Secure Event Dissemination in Publish-Subscribe Networks". In: *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*. June 2007, p. 22.

[210] W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. 3rd ed. Boston, MA, USA: Addison-Wesley, 1999.

[211] G. L. Steele. *Common LISP: The Language*. 2nd ed. Bedford, MA, USA: Digital Press.

[212] W. R. Stevens. *TCP/IP Illustrated: The Protocols*. Vol. 1. Boston, MA, USA: Addison-Wesley, 1993.

[213] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*. San Diego, California, United States: ACM, 2001, pp. 149–160.

[214] A. Sulistio, C. S. Yeo, and R. Buyya. "A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools". In: *Software: Practice and Experience* 34.7 (2004), pp. 653–673.

[215] Sun Microsystems, Inc. *Java Message Service, Version 1.1*. Santa Clara, CA, USA, Apr. 2002.

[216] Sun Microsystems, Inc. *Java Management Extensions (JMX) Specification, Version 1.4*. Santa Clara, CA, USA, Nov. 2006.

[217] C. A. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley, 1998.

[218] A. Ulbrich, G. Mühl, T. Weis, and K. Geihs. "Programming Abstractions for Content-Based Publish/Subscribe in Object-Oriented Languages". In: *Proceedings of the 2004 OTM Confederated International Conferences CoopIS, DOA, and ODBASE (OTM 2004)*. Ed. by R. Meersman, Z. Tari, W. van der Aalst, C. Bussler, A. Gal, V. Cahill, S. Vinoski, W. Vogels, T. Catarci, and K. Sycara. Vol. 3291. Lecture Notes in Computer Science. Agia Napa, Cyprus: Springer, Oct. 2004, pp. 1538–1557.

[219] T. Van Cutsem. "Ambient References: Object Designation in Mobile Ad Hoc Networks". Ph.D. Thesis. Brussels, Belgium: Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008.

[220] T. Van Cutsem and W. De Meuter. "Event-Driven Mobile Computing with Objects". In: *Principles and Applications of Distributed Event-Based Systems*. Ed. by A. Hinze and A. P. Buchmann. IGI Global, June 2010. Chap. 14, pp. 324–345.

[221] A. Varga. "OMNeT++". In: *Modeling and Tools for Network Simulation*. Ed. by K. Wehrle, M. Günes, and J. Gross. Berlin/Heidelberg, Germany: Springer, 2010, pp. 35–59.

[222] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. "Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems". In: *Proceedings of the 35th Annual Hawaii International Conference on System Science*. Big Island, HI, USA: IEEE Computer Society, Jan. 2002, pp. 3940–3947.

[223] K. Wehrle, M. Günes, and J. Gross, eds. *Modeling and Tools for Network Simulation.* Berlin/Heidelberg, Germany: Springer, 2010.

[224] T. Weis, H. Parzyjegla, M. A. Jaeger, and G. Mühl. "Self-organizing and Self-stabilizing Role Assignment in Sensor/Actuator Networks". In: *Proceedings of the 2006 OTM Confederated International Conferences CoopIS, DOA, GADA, and ODBASE (OTM 2006), Part II.* Ed. by R. Meersman and Z. Tari. Vol. 4276. Lecture Notes in Computer Science. Montpellier, France: Springer, Oct. 2006, pp. 1807–1824.

[225] M. Werner, M. A. Jaeger, and H. Parzyjegla. "An Application of the $(\max, +)$ Algebra to Information Flow Security". In: *Proceedings of the 7th International Conference on Networking (ICN 2008).* Ed. by J. Bi, T. Gyires, and I. Pozniak-Koszalka. Cancun, Mexico: IEEE Computer Society, Apr. 2008, pp. 262–266.

[226] M. Werner, G. Mühl, H. Parzyjegla, and H.-U. Heiss. "Betriebssystemunterstützung für verteilte Anwendungen in realer Raumzeit". In: *Tagungsband zum 2. GI/ITG KuVS-Fachgespräch Systemsoftware und Energiebewusste Systeme.* Ed. by F. Bellosa. Vol. 2007-20. Interner Bericht, Fakultät für Informatik. Karlsruhe, Germany: Universität Karlsruhe, Oct. 2007, pp. 33–37.

[227] M. E. Whitman and H. J. Mattord. *Principles of Information Security.* 4th ed. Boston, MA, USA: Course Technology, Jan. 2011.

[228] A. Wollrath, R. Riggs, and J. Waldo. "A Distributed Object Model for the Java System". In: *Proceedings of the 2nd Conference on USENIX Conference on Object-Oriented Technologies (COOTS '96).* Toronto, Ontario, Canada: USENIX Association, 1996, p. 17.

[229] *Workshop Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCSW 2006).* Lisbon, Portugal: IEEE Computer Society, July 2006.

[230] X/Open Company Ltd. *Data Management: Structured Query Language (SQL), Version 2, X/Open CAE Specification, Document C449.* Reading, England, UK, Mar. 1996.

[231] T. W. Yan and H. García-Molina. "Index Structures for Selective Dissemination of Information under the Boolean Model". In: *ACM Transactions on Database Systems* 19.2 (June 1994), pp. 332–364.

[232] P. Zave and M. Jackson. "A Component-Based Approach to Telecommunication Software". In: *IEEE Software* 15.5 (Sept. 1998), pp. 70–78.

[233] P. Zave. "Modularity in Distributed Feature Composition". In: *Software Requirements and Design: The Work of Michael Jackson.* Ed. by B. Nuseibeh and P. Zave. Chatham, NJ, USA: Good Friends Publishing Company, 2010, pp. 267–292.

[234] A. Zeidler. "A Distributed Publish/Subscribe Notification Service for Pervasive Environments". Ph.D. Thesis. Darmstadt, Germany: Technische Universität Darmstadt, Nov. 2004.

[235] A. Zeidler and L. Fiege. “Mobility Support with REBECA”. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW ’03)*. Providence, RI, USA: IEEE Computer Society, May 2003, pp. 354–361.

[236] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. Tech. rep. UCB/CSD-01-1141. Berkeley, CA, USA: Computer Science Division (EECS), University of California at Berkeley, Apr. 2001.

Erklärung

Hiermit erkläre ich, dass ich die eingereichte Dissertation mit dem Titel „*Engineering Publish/Subscribe Systems and Event-Driven Applications*“ selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Ich habe bisher noch keinen Promotionsversuch unternommen.

Rostock, 10. Dezember 2012 Helge Parzyjegla