Visibility in Event-Based Systems

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

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

vorgelegt von

Dipl.-Inform. Ludger Fiege
aus Remagen

Referenten:	Prof. Alejandro P. Buchmann, Ph. D.
	Prof. Dr. Mira Mezini
Datum der Einreichung:	27. Juli 2004
Datum der mündlichen Prüfung:	22. April 2005

Darmstadt 2005, Darmstädter Dissertationen D17

For Biggi,
and Franzi, Flori, and Johanna

Preface

Acknowlegements

Many persons have contributed to this thesis and I want to thank them for their support. First of all, I have to thank my advisor Prof. Alejandro Buchmann for his demanding questions, his continuous support, and his patience. Without his support this work would not have been possible. My co-advisor Prof. Mira Mezini managed to point me to the engineering issues, which considerably strengthened the statements made in this thesis.

Current and past colleagues at database and distributed systems group at TU Darmstadt provided a wonderful working environment. Especially Mariano Cilia, Felix C. Gärtner, Gero Mühl, and Andreas Zeidler endured many discussions that helped me forming my ideas, and without Gero the whole Rebeca project wouldn't have started at all.

Last but not least, I have to thank my family for all their patience and support. And it is always good to get a reminder once in a while where real life takes place.

Publications

Parts of this thesis are based on previous publications. Excerpts of Chapter 2 on the formal specification of event-based systems are published as joint work with Gero Mühl and Felix C. Gärtner [113, 114, 115]. The discussion about content-based filters and routing in the REBECA notifications service is based on joint work with Gero Mühl [214, 216]. The basis for Chapter 3 was published together with Mira Mezini, Gero Mühl, Felix C. Gärtner, and Alejandro P. Buchmann [110, 111, 115]. Some of the routing and security issues are published in [117]. The disussion of scalability in Chapter 5 is motivated by joint work with Gero Mühl, Felix C. Gärtner, and Alejandro P. Buchmann [217].

A number of additional publications have been published during the time this thesis was prepared [45, 75, 105, 107, 108, 109, 215, 272, 290, 291, 305], and workshop proceedings were edited that are related to this thesis [19] and to the discussion in Chapter 2 [46, 116].

Abstract

In modern IT systems, growing interconnectivity and continuous change demand a loose coupling of the participating components and services to facilitate system evolution. The paradigm of event-based computing and publish/subscribe communication provide the *flexibility* and *scalability* that is required in many application domains. Examples include mobile systems, monitoring, reactive system construction, and application integration.

In *event-based systems* producers and consumers of data interact indirectly. Producers publish notifications about events they have observed. Consumers announce their interest in certain kinds of notifications by issuing subscriptions. An intermediate notification service conveys notifications to those consumers having a matching subscription. Hence, the data flow in event-based systems is determined implicitly by set of *all* notifications and subscriptions.

This decoupling of producers and consumers is one of the main advantages of this paradigm, but in this generic form it also implies some downsides. There is no way to differentiate application-level components or to tailor the functionality of publish/subscribe services. The implied assumption is that such systems are homogeneous. However, open systems are heterogeneous and will not only require the middleware to accommodate to different data formats but also to combine different notification techniques. A one-size-fits-all implementation of the notification service is obviously not possible.

Unfortunately, current research and available products are primarily focusing on scalability issues in terms of communication efficiency and system size, whereas problems of system engineering and management are often neglected. Efficient mechanisms to structure event-based systems are a missing prerequisite for supporting engineering, management, and heterogeneity. Like in the early stages of programming language evolution, current event-based systems are typically characterized by a flat design space, with no structure and global variables only. While producers and consumers are designed and implemented as usual, there is no support for the role of an *administrator*, which is responsible for orchestrating application components and middleware services.

This thesis identifies the visibility of components and notifications as the underlying objective that must be achieved for any form of interaction control. *Scoping* is introduced as tool for administrators to control visibility and thus communication. A scope bundles and identifies a group of producers and

consumers. Notifications published in one scope are, at first, invisible in other scopes. Scopes encapsulate the composed producers and consumers and hide the internal structure. They localize the relationships between components outside of the components themselves.

In order to communicate with the remaining system input and output interfaces are assigned to scopes. They determine which internal notifications are forwarded to the outside, and vice versa. Scopes thus act as a single component to the outside system. Furthermore, they can be recursively arranged as members of other, higher-level scopes. Scopes and components can be arranged according to a variety of objectives, and at different levels of abstraction. On one hand, scopes represent common characteristics of the grouped components, like proximity to a designated location based on geographical coordinates or IP network topology. On the other hand, scopes delimit components of other groups that belong to different administrative domains, use other data models for notifications, or do not have the required security credentials. In the general case, system engineers and administrators decide how the system is logically and physically subdivided—scoping is the tool offered to them by this thesis.

Once the structure of an event-based systems is identified, scoped notification delivery can be customized. *Transmission policies* are associated with scopes to refine the rules where matching notifications are delivered, e.g., to forward a notification only to one out of a set of similar consumers in a scope. *Notification mappings* transform between different data formats used for internal and external notifications. Together with scope interfaces this allows for a controlled integration of independently developed and deployed pub/sub applications.

Different aspects of controlling visibility of events are tackled in a large number of existing products and research contributions. However, so far an approach orthogonal to the other aspects of publish/subscribe was missing. Scopes provide the means by which system administrators and application developers can configure an event-based system. Scopes offer an abstraction to reify structure and to bind organization and control of routing algorithms, heterogeneity support, and administrative tasks to the application structure. By providing a well-defined point of interconnection, scopes may not only delimit the distribution of notifications within a given service implementation but also act as bridges between different implementations. They delimit application functionality and contexts, controlling side effects and interaction. This is of particular importance as platforms of the future must be configurable not only at deployment time but also once an application is in operation.

This thesis investigates the scoping concept and its extensions in three steps. Based on a specification of publish/subscribe semantics, scoping is formally defined using a language adapted from temporal logic. Second, a variety of implementation strategies are compared that differ in the amount of control exerted on communication and the type of underlying communication mechanisms, ranging from point-to-point messaging to multicast, from remote procedure calls to database storage. Finally, a prototype implementation as part of the REBECA distributed notification service is described.

Zusammenfassung

Ereignisbasierte Softwarearchitekturen sind wegen ihrer inhärenten Vorteile zu einem wesentlichen Merkmal großer verteilter Systeme geworden. Die lose Kopplung der beteiligten Komponenten erlaubt es, autonome, heterogene Systemteile leicht zu integrieren und die Entwicklungsfähigkeit und Skalierbarkeit der entstehenden, komplexen Systeme zu steigern. Es wird zunehmend deutlich, dass traditionelle Client/Server Ansätze, basierend zum Beispiel auf Remote Procedure Calls (RPC), diese Anforderungen nur unzureichend unterstützen. Die ereignisbasierte Verarbeitung verspricht hier sowohl eine höhere Leistungsfähigkeit als auch flexiblere Architekturen, die sich an veränderliche Anforderungen anpassen lassen.

In einem ereignisbasierten System kommunizieren Komponenten indem sie Notifikationen über aufgetretene Ereignisse produzieren und konsumieren. Als Ereignis wird dabei die Änderung eines im Computer implementierten Modells bezeichnet. Die betroffene Komponente (Produzent) publiziert daraufhin eine oder mehrere Notifikationen, die das Ereignis beschreiben. Die Art der Informationen, die von einem Produzenten veröffentlicht wird, beschreibt eine vorher verbreitete Ankündigung (engl. Advertisement). Die Notifikationen sind nicht an bestimmte Empfänger adressiert, vielmehr werden sie an diejenigen Konsumenten weitergeleitet, die vorher ihr Interesse an dieser Art von Nachricht durch das Veröffentlichen einer entsprechenden Subskription angemeldet haben. Ein Notifikations-Dienst ist für die Übermittlung der Daten verantwortlich. Die lose Kopplung ist darin begründet, dass Produzenten nur Informationen über ihren eigenen Zustand publizieren, d.h. über interne Ereignisse, und keine Kenntnis über die möglichen Konsumenten haben; sie erwarten somit auch keine direkte Reaktion auf die veröffentlichten Informationen.

Das Potenzial ereignisbasierter Architekturen wurde sowohl in der Industrie als auch in der Forschung erkannt. Eine Reihe von Prototypen und Produkten ist verfügbar (SIENA, JMS, TIB Rendezvous, IBM WebSphere MQ, etc.) und entsprechende Dienste sind Teil moderner Komponentenplattformen, wie z.B. dem Corba Component Model (CCM) und Enterprise JavaBeans (EJB). Allerdings lag bisher das Hauptaugenmerk auf der Effizienz der Notifikations-Dienste, wohingegen Entwurf, Programmierung und Administration solcher Systeme wenig betrachtet wurden. Andererseits hat die Forschung im Bereich Software Engineering frühzeitig die Bedeutung von Kapselung (information hiding) und Abstraktion erkannt, die maßgeblich die Entwicklung des strukturierten Programmierens, des Modul-, Klassen-, und Komponentenbegriffs beeinflusst haben. Obwohl diese Ideen in Request/Reply-basierten Systemen umgesetzt worden sind (z.B. Corba) fehlen vergleichbare Strukturierungsmechanismen für ereignisbasierte Systeme. In der Literatur ist kein Modulkonzept oder Komponentenbegriff für ereignisbasierte Systeme eingeführt worden und typische Implementierungstechniken, wie Publish/Subscribe, führen neben den primitiven API-Aufrufen keine weiteren (Programmier-) Abstraktionen ein. Ein geeignetes Modulkonzept kann die Beziehungen zwischen den Komponenten materialisieren. Es stellt somit für den Programmierer und den Administrator ein wertvolles Hilfsmittel dar, um die Interaktion steuern zu können. Ein Modul bündelt eine Menge von Produzenten und Konsumenten und kann, versehen mit eigenen Ein- und Ausgabe-Schnittstellen, als neue, zusammengesetzte, ereignisbasierte Komponente im System teilnehmen.

Sichtbarkeitsbereiche bündeln mehrere Basiskomponenten und beschränken die Sichtbarkeit von Notifikationen zunächst auf die Konsumenten ein, die sich im gleichen Bereich wie der Produzent befinden. Zusammen mit einer eigenen Schnittstellenbeschreibung agieren die Sichtbarkeitsbereiche wiederum als Komponenten, die in andere Bereichen integriert werden können; sie offerieren ein Modulkonzept wie es oben gefordert wurde. Um die Interaktion zwischen und innerhalb der so definierten Sichtbarkeitsbereiche genauer festzulegen, können Transformationen und Verbreitungsregeln definiert werden. Transformationen (notification mappings) verändern die Notifikationen, wenn sie die Grenzen eines Sichtbarkeitsbereiches überqueren. Auf diese Weise können Bereiche mit unterschiedlichen Datenmodellen und Repräsentationen voneinander getrennt werden. Die Verbreitung von Notifikationen kann mit Hilfe von Regeln (transmission policies) beeinflusst werden, die an Bereiche gebunden werden. Sie wählen aus der Menge der Konsumenten, die im betrachten Bereich eine passenden Subskription besitzen, diejenigen aus, die eine Nachricht tatsächlich erhalten sollen. Dies kann man nutzen, um eine 1-aus-n Auslieferung innerhalb eines Sichtbarkeitsbereiches zu implementieren. Zusammengenommen ist der Systementwickler in der Lage, ereignisbasierte Systeme graduell weiter zu entwickeln und die Leistung und Semantik des Notifikations-Dienstes an die Struktur einer Anwendung zu binden und anzupassen.

In dieser Arbeit werden die Anforderungen analysiert, die ereignisbasierte Systeme an den Entwurf stellen, und Sichtbarkeitsbereiche (Scopes) als ein grundlegender Strukturierungsmechanismus eingeführt, der diesen Anforderungen genügt. Neben einer formalen Definition werden eine Reihe von Umsetzungsstrategien diskutiert, die sich in dem Umfang an Kontrolle unterscheiden, den sie auf die Kommunikation ausüben. Eine prototypische Implementierung im Rahmen des Rebeca Projekts demonstriert die Machbarkeit des vorgestellten Konzepts.

Contents

1 Introduction **1**
1.1 Event-Based Systems 2
1.2 Shortcomings of Event-Based Communication 4
1.3 Scoping in Event-Based Systems 5
1.4 Organization 8

2 Event-Based Systems **9**
2.1 Constituents of Event-Based Systems 9
2.1.1 Events and Notifications 10
2.1.2 Producers and Consumers 10
2.1.3 Subscriptions and Filters 11
2.1.4 Event Notification Service 13
2.2 Models of Interaction 13
2.2.1 Request/Reply 14
2.2.2 Anonymous Request/Reply 14
2.2.3 Point-to-Point Messaging 15
2.2.4 Event-Based 15
2.2.5 Comparison 16
2.2.6 Interaction vs. Implementation 17
2.3 Simple Event-Based Systems 18
2.3.1 Formal Background 18
2.3.2 Specification of a Simple Event System 20
2.3.3 Implementation 26
2.3.4 Correctness 27
2.4 The Rebeca Model 28
2.4.1 System Model 28
2.4.2 Architecture 29
2.4.3 Filter-Based Routing 30
2.5 Missing Functionality 31
2.5.1 Application Scenarios 32
2.5.2 Engineering Requirements 34
2.5.3 Existing Support 38

2.6 Discussion 39

3 Scopes **41**

3.1 Visibility 42
3.1.1 Implicit Coordination and Visibility 42
3.1.2 Explicit Control of Visibility 43
3.1.3 The Role of Administrators 43
3.2 Event-Based Systems with Scopes 44
3.2.1 Visibility and Scopes 44
3.2.2 Specification 45
3.2.3 Notification Dissemination 48
3.2.4 Duplicate Notifications 50
3.2.5 Dynamic Scopes 51
3.2.6 Attributes and Abstract Scopes 52
3.2.7 A Correct Implementation 53
3.3 Component Interfaces 56
3.3.1 Scope Interfaces 56
3.3.2 Event-Based Components 58
3.3.3 Example 58
3.4 Notification Mappings 60
3.4.1 Specification 61
3.4.2 A Correct Implementation 65
3.4.3 Example 67
3.5 Transmission Policies 68
3.5.1 Publishing Policy 68
3.5.2 Delivery Policy 70
3.5.3 Traverse Policy 71
3.5.4 Influencing Notification Dissemination 72
3.6 Scoping of Notifications 73
3.6.1 Dependent Notifications 73
3.6.2 Session Scopes 74
3.7 Security 76
3.8 Engineering with Scopes 76
3.8.1 Development Process 77
3.8.2 Component Definition 77
3.8.3 Scope Graph Composition 79
3.8.4 Scope Graph Deployment 80
3.8.5 Management 80
3.8.6 Scope Graph Language 81
3.9 Discussion 90

4 Scope Architectures **93**
4.1 Architectural Choices 94
4.1.1 Communication Medium 94
4.1.2 Scope Distribution 97
4.1.3 Example Architectures 99
4.1.4 Scope Graph Distribution—Types of Architectures 102
4.1.5 Comparing Architectures 105
4.2 Collapsing Scope Graphs 106
4.2.1 Collapsing Filters 106
4.2.2 Filtering Costs in a Pub/Sub Implementation 112
4.2.3 Coping with Graph Updates 113
4.2.4 Filter Aggregation in Databases 115
4.2.5 Evaluation 115
4.3 Scope Address 116
4.3.1 Addressing Scheme 116
4.3.2 The Resulting Overlay Network 117
4.3.3 Evaluation 118
4.4 Scopes as Event Brokers 119
4.4.1 One Scope, One Broker 120
4.4.2 Distributed Scopes 120
4.4.3 Collocating Broker Scopes 121
4.4.4 Evaluation 122
4.5 Integrate Scoping and Routing 122
4.5.1 Scopes as Overlays 122
4.5.2 Enhancing Routing Tables 124
4.5.3 Setting Up Routing Tables 125
4.5.4 Scoped Routing 127
4.5.5 Crossing Scopes 130
4.5.6 Transmission Policies 132
4.5.7 Scope Multicast 133
4.5.8 Evaluation 133
4.6 Combining Different Architectures 134
4.6.1 Architectures and Scope Graphs 134
4.6.2 Bridging Architectures 135
4.6.3 Integration with other Notification Services 136
4.7 Discussion 137

5 Rebeca—An Implementation of Scopes **139**
5.1 Software Building Blocks 140
5.1.1 Events, Notifications, Messages 140
5.1.2 Subscriptions and Filters 142
5.1.3 Pub/Sub API 143
5.1.4 Broker Network 144
5.1.5 Broker Implementation 145
5.1.6 Channels 147

5.2 Scopes . . . 148
5.2.1 Scope Interface . . . 148
5.2.2 Implementation: Broker Scope . . . 149
5.2.3 Management Interfaces . . . 152
5.2.4 Implementation: Integrated Routing . . . 153

6 Related Work 157
6.1 Distributed Systems . . . 159
6.1.1 Middleware . . . 159
6.1.2 Communication Paradigms . . . 160
6.1.3 Other Notions of Communication . . . 163
6.1.4 Application Scenarios . . . 163
6.2 Notification Services . . . 164
6.2.1 Corba Notification Service . . . 165
6.2.2 Java Message Service . . . 166
6.2.3 Commercial Systems . . . 166
6.2.4 Research Prototypes . . . 168
6.2.5 Other Related Work . . . 170
6.3 Rule-Based Systems . . . 171
6.4 Data Management . . . 171
6.4.1 Active Database Management Systems . . . 171
6.4.2 Reactive Functionality . . . 172
6.4.3 Heterogeneity and Data Integration . . . 173
6.5 Coordination Models . . . 173
6.6 Software Engineering . . . 175
6.6.1 Software Architecture . . . 175
6.6.2 Software Integration . . . 176
6.6.3 Component Models . . . 177
6.6.4 Programming Loosely Coupled Systems . . . 178
6.6.5 Aspects and Reflection . . . 179
6.7 GUI Design . . . 179

7 Conclusions and Future Work 181

Bibliography 187

List of Figures

2.1 Event-based system artifacts: interaction versus implementation . 11
2.2 Taxonomy of cooperation models 14
2.3 Interface operations of a simple event system 19
2.4 Black box view of an event system 23
2.5 A possible implementation view of a simple event system 27
2.6 The router network of Rebeca 29
2.7 Call graphs of applications: bipartite single and multi source, and a general group 32
2.8 An example stock trading application 35

3.1 A meta model of scopes 45
3.2 An exemplary scope graph 46
3.3 Outgoing and incoming notifications 49
3.4 Two ways of generating duplicates 50
3.5 A possible implementation of a scoped event system 54
3.6 Different scope interfaces 57
3.7 The graph of the stock application 59
3.8 Interfaces of the components in the example application 60
3.9 Recursive definition of the relation $(n_1, X) \rightsquigarrow (n_2, Y)$ 62
3.10 Transformation of mappings into components 65
3.11 Architecture of scoped event system with mappings 66
3.12 Three important transmission policies in scope graphs 69
3.13 Scope definition accuracy 90

4.1 Design dimensions of scope architectures 94
4.2 An implicit implementation shifts visibility control into application components. 98
4.3 Scope architectures combine scope graph implementation and communication media. 100
4.4 Steps of scoped notification delivery 102
4.5 Types of architectures, their characteristics, and examples 104
4.6 Comparison of scope architectures 107
4.7 A collapsed sample graph and explicit destinations 108

4.8 Compute list of scopes in which a notification is visible 110
4.9 Collapsing with visibility roots 110
4.10 Notifications are tagged with the ID of *their* scope. 117
4.11 An exemplary scope graph 123
4.12 Scopes as overlays within the broker topology 123
4.13 A flat routing table for broker B_1 124
4.14 Enhanced routing tables of B_1 incorporating scopes 125
4.15 Scope lookup tables 126
4.16 overall routing algorithm 129
4.17 The naïve matching algorithm with mappings 129
4.18 Inter-scope forwarding 130
4.19 Duplicate scopes to separate QoS requirements 135

5.1 Events, notifications, and messages 140
5.2 Message structure 141
5.3 Subscriptions and filters 142
5.4 Components and scopes 144
5.5 A demo publisher 144
5.6 Rebeca brokers 145
5.7 Rebeca routing classes 146
5.8 An event broker configuration file 147
5.9 Structure of a ScopeJoin notification 150
5.10 Adding a management aspect 152
5.11 EventBroker management interface 153
5.12 Scope management interface 153
5.13 Integrated routing scope classes 154

List of Symbols

The following conventions are used for the font shapes of notations: Italic letters A, B, S denote arbitrary artifacts that are described in their context, like components of the system or arbitrary sets. Second, the caligraphic font shape $\mathcal{C}, \mathcal{N}, \mathcal{F}$ signifies abstract sets that are usually not stored in a computer. And finally, a sans serif style $\mathsf{A}, \mathsf{B}, \mathsf{G}$ is used for finite sets that may be explicitly stored somewhere.

Symbol	Explanation	Page
$P(A)$	power set of a set A, i.e., the set of all subsets of A	
$\mathcal{N}$	set of all notifications	21
n	a notification[1] in $\mathcal{N}$	21
$\mathcal{N}^*$	extended set of notifications $\mathcal{N}^* = \mathcal{N} \cup \epsilon$ with $\epsilon \notin \mathcal{N}$ being the empty notification $\mathcal{N}^*$ *and* $\mathcal{N}$ *are still used inconsistently, e.g., in chap. 2*	63
$\mathcal{F}$	set of all eligible filters $\mathcal{F} = \{f \| f : \mathcal{N} \to \mathcal{N}\}$	22
F, F_1, F_2, F'	filters in $\mathcal{F}$	22
$\mathcal{M}$	set of all notification processing functions/ mappings $\mathcal{M} \supset \mathcal{F}$	63
$\mathcal{C}$	set of all simple components	22
x, y, z, c_1, c_2	denote simple components in $\mathcal{C}$	45
$\mathcal{S}$	set of all scopes	45
$S, S_2, S_3, \ldots, T, U$	denote scopes in $\mathcal{S}$	46
$\mathcal{K}$	set of all components, $\mathcal{K} = \mathcal{S} \uplus \mathcal{C}$ with $\mathcal{S} \cap \mathcal{C} = \emptyset$	45
A, B, C, D, X, Y, Z $C, C_1, C_2, \ldots$	denote arbitrary components	22

[1]As usual, n may also denote the number of elements of a set, e.g., $\{c_1, \ldots, c_n\}$, but only when its use is unambiguous.

Symbol	Explanation	Page
$\mathsf{G} = (\mathsf{C}, \mathsf{E})$	scope graph G consisting of set of components/ scopes $\mathsf{C} \subseteq \mathcal{C}$ and edges E denoting the component $\rightarrow$ scope relationship	46
$C \lhd S$ $\quad C \overset{\star}{\lhd} S$ $S \rhd C$ $\quad S \overset{\star}{\rhd} C$	subscope relationship $\lhd$, i.e., $(C, S) \in E$, and its transitive closure $\lhd^*$	46
I_C	the base interface of a component C, where $I_C = ({}^iI_C, {}^oI_C)$ with ${}^iI_C \subset \mathcal{F}$ and ${}^oI_C \subset \mathcal{F}$	63
$I_{C\|S}$	selective interface of a component C w.r.t. a superscope S	57
$I^S \quad I^S_C$	interface the scope S imposes on its members, on C specifically	58
$\hat{I}^S_C \quad \hat{I}^S_C$	effective interface $\hat{I}^S_C = I^S_C \circ I_{C\|S}$	58
${}^iF_C \quad {}^oF_C$	sets of filters describing the input and output configuration of the interface I_C	56
RT, RT_B	filter-based routing table in general and at a specific broker B	30
RT^S_B, RT^S	member routing table for intra-scope routing	124
ST_B	inter-scope routing table	125

Chapter 1

Introduction

The speed at which business is conducted increases. Customer service is important, and mergers as well as joint ventures require flexibility and adaptability of business infrastructures. With the reduction of coordination and communication costs, organizational structures are changed more easily and frequently. So even after the end of the hype about a New Economy, the trend towards more volatile business structures has neither ceased nor lost its importance [198].[1] Services and data are integrated in ever new constellations so that application architectures are getting more volatile. The transition to loosely integrated distributed systems requires IT infrastructures that facilitate both *scalability* and *system evolution*.

Consequently, the development of today's computer systems is mainly influenced by the effects of networking. Increasing connectivity and size of the networked systems give rise to a number of issues. A basic requirement is the availability of scalable communication mechanisms, which are crucial for building and maintaining these systems. The mechanisms not only have to support large numbers of components, but also face complex application environments that are dynamic and subject to unexpected and recurrent change.

A second important aspect of today's systems is the automation of data processing. While systems were traditionally designed to respond to interactive user requests, the aim today is to provide increasingly autonomous data processing to improve functionality and utility. Instead of having human operators mediate between applications, e.g., to replenish an inventory by manually reordering goods, directly connected applications are able to initiate replenishment automatically. In this example low supplies initiate activity. In general, for a computation to be automated, it must be provided with the data necessary to check for such conditions. Applications are driven by information available in the system, they are *data-* or *information-driven*.

[1]To foster processes and applications that cross traditional modules of enterprise systems, SAP, the major enterprise resource planning (ERP) company, has recently identified an "adaptive business" strategy to be the key to competitive advantage [36].

The variability of dynamic networked environments and the automation of data exchange shifts the focus when dealing with the delivery of data and services, moving from a stationary world to one that is in a state of flux. Traditionally, data and services have been viewed as being stationary in a collection of objects or databases, with inquiries directed at them in a request/reply mode of interaction. This concept has led to client/server application architectures that emphasize explicit delegation of functionality, where system components access remote functionality to accomplish their own goal. Remote procedure calls (RPC) and derivative techniques are classic examples [40, 220, 288]; even the incipient Web services mainly rely on sending requests with the Simple Object Access Protocol (SOAP) [270]. These techniques deliberately draw from a successful history of engineering experience, their principles are well understood, and they have been an appropriate choice for many well-defined problems. In the context of dynamic networked systems, however, request/reply has serious restrictions.

The direct and often synchronous communication between clients and servers enforces a tight coupling of the communicating parties and impairs scalability [125]. Clients poll remote data sources, and they have to trade resource usage for data accuracy, especially in chains of dependent servers. Unnecessary requests due to short polling intervals waste resources, whereas long intervals increase update latency. In addition, request/reply restricts system evolution. The control flow is encoded in application components, which makes it accessible to engineers but also mixes the actual configuration of the system with the application logic of individual components. Consequently, the capability to orchestrate the whole system is limited by the means available to adapt application components at runtime. And finally, delegating functionality inevitably implies a functional dependency on the called service, and on its presence.

The obvious need for asynchronous and decoupled operation led to various extensions of existing middleware. For instance, CORBA and Java 2 Enterprise Edition (J2EE) were extended with asynchronous invocation methods and notification services [226, 263, 264, 284], and similar features are available in Microsoft's COM+ and in the language model of the new .Net platform [187, 247], too. And database research, software engineering, and coordination theory corroborate the advantages of loosely coupled interaction as well [74, 137, 237, 276].

1.1 Event-Based Systems

Instead of stepwise amending the conventional request/reply mode of interaction, the paradigm of *event-based computing* takes a contrasting approach and inherently decouples system components. In an event-based mode of interaction[2] components communicate by generating and receiving *event notifications*. An *event* is any occurrence of a happening of interest, i.e., a state change in some

[2]A more formal definition and the distinction between events, publish/subscribe, and messaging is given in Chap. 2.

component. The affected component issues a *notification* that describes the observed event. An *event notification service* mediates between the components of an event-based system and conveys notifications from *producers* to those *consumers* that have registered their interest with a previously issued *subscription.*

The power of the event-based architectural style, as it is also called [57], is that neither the published notifications nor the subscriptions are directed towards specific components. The notification service decouples the components so that producers are unaware of any consumers and consumers rely only on the information published, but not on the producers. Event-based components are not designed to work with specific other components, which facilitates the separation of communication from computation. The event-based style carries the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale [28, 275].

In view of the above arguments, the use of events is clearly superior to request/reply approaches in many information-driven scenarios [124]. In fact, many improvements of tightly coupled communication converge to an asynchronous approach. For instance, caching data in network nodes [254], callback handling (observer pattern [129]), asynchronous remote invocations [264] introduce some form of indirection to decouple interaction from computation. The loose coupling makes applications easier to adapt and integrate, and it allows a specialized mediator, the notification service, to take care of scalability.

As a consequence, the potential of the event-based style has been recognized both in academia and industry. The event-based architectural style is becoming an essential part of distributed systems' design and many applications and their underlying infrastructures have incorporated event-based communication mechanisms. *Information busses* are the basis of many systems [23, 233] and a number of event notification services were developed (for example [62, 83, 138, 273, 279, 293]) as well as integrated into modern component platforms such as Corba Component Model (CCM) [225] and Enterprise JavaBeans (EJB) [280].

The range of application domains is broad. Often, applications use event-based communication to improve scalability or to exploit adaptability. The classic example certainly is stock monitoring and financial data processing. The timely and efficient dissemination of stock quotes to many consumers is a prerequisite in these systems. Information dissemination in general is the apparent application domain of notification services, which includes control systems, real-time control systems, monitoring applications, etc. These are examples of unidirectional data distribution scenarios, which focus on few producers and many subscribers. Many environments are characterized by their variability and need to facilitate change. Enterprise application integration (EAI) is an early adopter of events and information busses [165], which also holds for workflow management and collaborations in inter-organizational enterprise applications [143, 192]. Furthermore, in inherently dynamic environments, like sensor networks [10, 202], pervasive computing and mobile systems [52, 301], the infrastructure always has to cope with reconfigurations, making the event-based style pertinent here, too [84, 107, 161].

However, as soon as the complexity of a system increases and goes beyond the simple information dissemination domain a number of problems must be solved for the event-based style to be used as general purpose computing paradigm.

1.2 Shortcomings of Event-Based Communication

The great majority of existing approaches do not address the need for effective programming and management abstractions. Design, engineering, and administration of event-based systems are mostly disregarded. The indirection of event-based communication results in the desirable loose coupling and scalability but also makes system management more difficult. The indirection detaches aspects of interaction from producers and consumers without making them accessible elsewhere, so far. Knowledge about the system structure is removed from application components, and many problems bound to this structural information arise in event-based systems for which solutions are known in traditional request/ reply systems. The most evident problems are:[3]

1. *Effects and side effects.* The system's overall functionality is defined only implicitly as a result of the interaction between all involved components. The effects of (not) publishing a notification are not known a priori and adding or deleting producers may affect any part of the system. Without any means to control the interaction any change inevitably requires analyzing the whole set of participants to predict its (side) effects [26].

2. *Design and implementation.* Event-based applications cannot readily draw on abstractions or tools to hide engineering complexity, limiting applicability of the event-based style in scenarios beyond simple, unidirectional dissemination.

3. *Management.* Management of the system infrastructure is hardly possible since there is no notion of applications; instead, components interact solely on the availability of information and their interrelationship is not apparent. There is no structure to which application-dependent management tasks can be bound, i.e., access control and security policies, quality of service (QoS) customization, incremental and partial updates of code and configuration, etc.

4. *Security.* Security is accepted to be an urgent open issue in event-based systems [32, 300]. Most proven methods do not apply because of the missing structural knowledge.

These points are neither independent nor complete. Nevertheless, they clearly show that even if scalability in terms of communication efficiency is achieved,

[3]A more detailed discussion follows in Section 2.5.

the scalability in terms of the complexity to engineer and manage the increasing number of participants is not. Said problems erode the very benefit of loose coupling and increases the inherent complexity of designing, understanding, and administering event-based systems. The broad applicability of the event-based style is reduced as long as there is no support for these issues.

Some of the above problems are addressed in existing work. For instance, a number of systems distinguish administrative domains in which specific communication protocols and data formats are used, with bridges governing domain-crossing traffic, e.g., the READY notification service [146], CORBA domains [249], and even IP Multicast [209]. Unfortunately, these approaches consider only simple structures in the infrastructure itself, and thus can be seen as extensions of classic network management. They do not offer explicit support for designing and implementing event-based applications. Of course, one could always resort to implementing the necessary support manually or by more generic techniques (e.g., federations of distribution channels [229] or mediators [276]), but an implementation as part of the infrastructure/middleware is obviously better.

In a more general context, software engineering research early identified information hiding and abstraction [240] as basic principles that have influenced the development of structured programming, modules, classes, and components, all of which provide mechanisms to structure software systems. While being an integral part of request/reply-based distributed systems, e.g., CORBA [224], comparable hierarchical structuring mechanisms are missing in event-based systems. Indeed, event-based systems are generally characterized by a ‘flat’ design space with no apparent structure.

The event-based style removes structural knowledge and control from application components without offering an adequate replacement. In traditional request/reply systems, solutions to all the aforementioned problems are bound to system and software structure. What is missing in event-based systems is a way to represent the structure of both the underlying infrastructure and the applications. Hence, this thesis addresses these issues by creating structural abstractions that comply with the event-based style.

1.3 Scoping in Event-Based Systems

Structure allows to group components and their interaction, in event-based systems as well as in other kinds of systems. However, the loose coupling forbids application components to identify and handle structuring themselves. They should not need to be aware of any other specific components. It is the notification service that plays a mediator’s role and operates on a different level of implementation. It knows the identities of the participating components and is therefore in a position to structure them. The service essentially determines to which components a published notification is delivered. Refining this decision is the main aspect of structuring and controlling event-based systems, and it is also the essence of the previously mentioned problems. Consequently, the *visibility* *of notifications* is the fundamental issue that any solution must address.

This thesis is about visibility of notifications. The notion of *scoping* in event-based systems is introduced to describe visibility constraints. Scopes bundle sets of producers and consumers and limit the visibility of notifications to the components within the original scope. While still using an event-based style within a scope, the interaction of this group of components with the outside is no longer implicit. Scopes can be recursively members of higher level scopes and they can selectively republish internal notifications and forward external notifications to its members. In this way, scopes act as producers and consumers of notifications and they are able to explicitly control inter-scope traffic. Transmission policies can be applied to adapt notification forwarding both within a scope and between scopes, allowing notification delivery semantics to be tailored to application needs in restricted parts of the system. Furthermore, notification mappings at scope boundaries transform between heterogeneous representations of events.

The concept of scopes serves several purposes. First, it is a design tool to describe system layout and configuration. Second, if treated as first class construct in implementation, it is a fundamental approach to control the actual operation of an event-based system. And finally, scoping provides a basis for the integration of different implementation techniques.

As a design tool, scopes offer an abstraction of event-based computation and reify the structure of an event-based system. Without degrading the loose coupling of individual components they modularize complex systems and facilitate the composition of lower level components into more complex ones. The established structure is orthogonal to subscriptions and filter languages.

From an implementation point of view, scopes make system structure accessible to engineers. They localize the relationships between components so that interaction control can be implemented at well-defined points, but outside of the components themselves. Scopes play a mediator role, controlling communication without reducing loose coupling [276]. As a bundling unit and module, scopes are the appropriate location to refine and customize notification delivery. In delimited parts of an application, syntax of the distributed notifications and even the semantics of delivery can be varied, while any modifications are encapsulated and do not interfere with the remaining system. Furthermore, scopes provide a module construct as an abstraction for handling heterogeneity as well as for integrating security policies.

Finally, scopes can also serve as an implementation framework for notification services. The delimitation of communication opens the implementation of the service itself and makes it possible to integrate in one system different techniques of network communication, filtering and routing, data representation and security. Scope boundaries localize the functionality to connect specific scopes and their implementations. System engineers do not have to deal with one-size-fits-all approaches, nor do they have to manually bridge independent notification service instances. They delimit application functionality and contexts, controlling side effects and interaction. And they facilitate customization from application-level APIs to network implementation. This is of particular importance as platforms of the future must be configurable not only at deployment time but also once an application is in operation.

The different issues addressed by the notion of scoping are neither raised nor investigated for the first time. Scoping is well-known in programming languages and software engineering [240], and some aspects of visibility have even been studied in the context of distributed and event-based systems, e.g., [146, 209, 229]. However, the full implications of treating visibility control as a first-class concept was not realized before.

In this thesis, the scoping concept is investigated in three ways. A formalization of notification services is given and extended to define visibility and scoping. This specification is complemented with a detailed comparison of different architectures of scope implementation. While all adhere to the same visibility specification, engineers are provided with a wide range of implementation approaches that fundamentally influence system functionality in terms of efficiency, extensibility, and achievable quality of service guarantees. And finally, a prototype implementation in the context of the Rebeca notification service (**R**ebeca **E**vent **B**ased **E**lectronic **C**ommerce **A**rchitecture [112]) illustrates the practical use of scopes.

In terms of the deficiencies discussed in the previous section the scoping concept offers a solution as follows:

1. First and foremost scopes are designed to limit and control the effects of publishing a notification. This directly addresses point 1 of Sect. 1.2, effects and side effects.

2. With respect to designing and implementing event-based applications, scopes provide engineers with a module construct that facilitates information hiding, reuse and integration of existent parts. Scopes are a means to create *new* event-based components in the flavor of Szyperski [285], i.e., they create bundles of components that can be reused and deployed by others.

3. With respect to implementing and managing the event-based infrastructure, scopes structure both infrastructure *and* applications, and thus enable application-aware management. This allows for the provision of differentiated and tailored services, for partial system reconfigurations, and for computational reflection [195]. Scopes also contribute to 'traditional' efficiency aspects by explicitly reducing the amount of notifications sent through the system, making the routing decision simpler.

4. With their ability to delimit groups of components and to detach their internal communication from the outside, scopes propose a way to approach the lack of practical security mechanism for event-based systems.

1.4 Organization

The structure of the thesis is as follows. Chapter 2 gives a detailed discussion of event-based systems in general, their characteristics, and the essential differences to other communication schemes. A formalization of event-based systems is presented that models commonly used approaches, and which is used to sketch a provably correct implementation. Furthermore, the architecture of the distributed event notification service REBECA is described and, based on this, a number of example applications are analyzed to investigate the problems of missing visibility control in event-based systems.

Chapter 3 introduces the scope model. It extends the formal framework specified in Chapter 2 and presents an implementation sketch. The formal scope model includes scope interfaces, which offer a fine-grained control of notification distribution, notification mappings, which are introduced as a way to cope with diverging data and filter models in heterogeneous environments, and transmission policies, which are used to tailor delivery semantics. The chapter closes with an outlook on advanced aspects like grouping of notifications instead of component, sessions, and security issues.

Chapter 4 discusses a variety of different approaches to implement a scoped event-based system and analyzes a number of them in detail: collapsed filters suitable for implementation in database systems and on top of existing publish/ subscribe services, scope group addressing that fits well on top of multicast communication techniques, and an integrated routing approach that combines advanced content-based routing available in REBECA with scoping constraints.

The prototypical scope implementation that extends the REBECA notification service is described in Chapter 5.

Chapter 6 discusses related work. While only a few papers were published that directly address the problem of controlling visibility in event-based systems, a wide range of techniques is available that deal with individual aspects of the problem domain. Diverse areas of computer science are inspected, including distributed systems (related notification services), software engineering (software architecture and implicit invocation), coordination models, and others.

Finally, Chapter 7 summarizes the main results and discusses issues raised by this thesis and possible next steps of future work.

Chapter 2

Event-Based Systems

Contents

2.1	**Constituents of Event-Based Systems**	**9**
2.2	**Models of Interaction**	**13**
2.3	**Simple Event-Based Systems**	**18**
2.4	**The Rebeca Model**	**28**
2.5	**Missing Functionality**	**31**
2.6	**Discussion**	**39**

The notion of event-based systems is used in many areas of computer science. Unfortunately, there are varying, ambiguous definitions of terminology and characteristics and thus of usage scenarios. To distinguish their characteristics four basic interaction models are analyzed in Section 2.2: request/reply, anonymous request/reply, callbacks, and event-based interaction. These models help clarify the important distinction between event-based communication, publish/subscribe, and messaging. Furthermore, a formalization of event-based semantics using linear temporal logic and a correct implementation is presented in Section 2.3. The formalization serves as a simple tool to reason more precisely about notification services, which is mostly lacking in literature, and it is extended later to formally define the visibility of notifications. Section 2.4 describes Rebeca, a distributed notification service that conforms to the previous specification. Its basic functionality is comparable to other prototypes like Siena and JEDI. Finally, several example applications are analyzed to investigate the problem of unlimited visibility of events and the missing support in existing notification services.

2.1 Constituents of Event-Based Systems

An *event-based system* consists of the following constituents (see Figure 2.1): events and notifications as means of communications, producers and consumers as interacting components, subscriptions signifying a consumer's interest in certain notifications, and the event notification service responsible for conveying notifications between producers and consumers.

2.1.1 Events and Notifications

Any happening of interest that can be observed from within a computer is considered an *event*. This may be a physical event such as the appearance of a person detected by sensors, a timer event that indicates progression of real time, or generally an arbitrary detectable state change in a computer system. Since event detection is out of the scope of this thesis, the third kind of events is assumed here.

A *notification* is a datum that reifies an event, i.e., it contains data describing the event. A notification is created by the observer of the event and may just indicate the plain occurrence, but often may carry additional information describing the circumstances of the event. For instance, in the active badge system of Bacon et al. [18] events are raised when persons wearing a badge approach a sensor, and the published notification carries the detected ID of the badge and the time of observation. In general, different notifications can be created that describe the same underlying event, but from multiple viewpoints. This may be done due to application or security reasons, or simply because notifications are encoded in different data models. The most common data models are name-value pairs [62], objects [18, 89, 101], and semi-structured data [14, 214], i.e., XML.

On the lowest level considered here, notifications are conveyed via *messages*, which are data containers on the network level transmitting data between the endpoints of the underlying communication mechanism. The distinction between events, notifications, and messages is used to clearly separate the underlying communication technique from the mode of interaction, cf. Sect. 2.1.4.

2.1.2 Producers and Consumers

The software components of an event-based system act as producers and/or consumers of notifications. *Producers* are components that publish notifications. A producer's implementation is 'self-focused' in the sense that it observes only its own state. The decision to publish a state change is made by the component's internal computation, and is a core part of its function. What changes are published, and how this decision is configured/programmed into the producer, is an issue of past and ongoing research in areas like debugging [29] and monitoring [182], reflection [174] and aspect-oriented programming (AOP) [76]. Published notifications are not addressed to any specific (set of) receivers, they are rather forwarded to the event notification service for further distribution. Producers are unaware of any other components and they do not anticipate any reaction on the receiver side; this is detailed in Sect. 2.2.

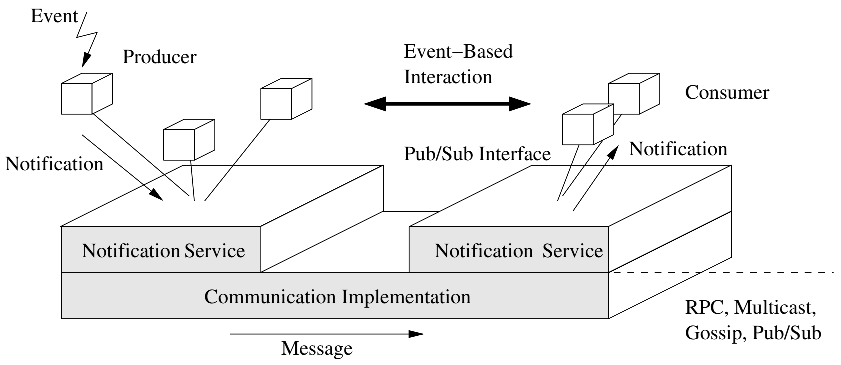


Figure 2.1: Event-based system artifacts: interaction versus implementation

Consumers react to notifications delivered to them by the notification service. They, too, are unaware of their specific communication peers. Not knowing the actual producers of notifications, consumers issue subscriptions to describe the kinds of notifications they are interested in; different classes of subscriptions are depicted in the next subsection. If a component is both consumer and producer, it reacts to both incoming notifications and observed internal state changes, and the resulting computation may lead to newly published notifications.

2.1.3 Subscriptions and Filters

A *subscription* describes a set of notifications a consumer is interested in. Consumers register their interest in receiving certain kinds of notifications by submitting subscriptions to the notification service. The service evaluates the subscriptions on behalf of the consumer and delivers those notifications that match one of the consumer's subscriptions. Subscriptions act as filters, which are basically boolean-valued functions that test a single notification and return either *true* or *false*. Indeed, filters are a common way to implement subscriptions, although, in general, subscriptions may comprise more than only a filter function. They can additionally include (meta-)data to govern notification selection beyond a per-notification level; for example, security credentials for accessing certain classes of notifications [32] or timing information to get past notifications [75]. Subscriptions can be seen as input interfaces of consumers, describing the data they are prepared to process.

Advertisements are issued by producers to declare the notifications they are willing to send. They also describe sets of notifications and may be of the same form as subscriptions. From a network level point of view, advertisements help improving routing decisions, because the notification service knows which notifications can be expected from where. From a software engineering viewpoint, advertisements comprise a component's output interface.

The expressiveness of subscriptions in terms of filtering capabilities depends on the filter language employed. In distributed notification services, essentially four *filter models* are distinguished: channels, subjects, types, and content-based. *Channels* are the simplest form of identifying sets of notifications. In this model, producers select a named channel into which a notification is published. Consumers, on the other hand, select a channel and they will get all notifications published therein. An example of this approach is the CORBA Event Service [227]; the CORBA Notification Service also relies on channels but additionally offers filters on notification content.

Subject-based addressing uses string matching for notification selection [233]. Publishers annotate each notification with a subject string that denotes a rooted path in a tree of subjects. For example, a stock exchange application publishes new quotations of *FooBar Ltd.* under the subject `/Exchange/Europe/London/Technology/FooBar`, classifying it to be traded in London and to belong to the technology sector of the stock market. Consumers subscribe for `/Exchange/Europe/London/Technology/`$*$ to get all technology quotations. It is implementation dependent whether `/Exchange/Europe/London/`$*$ already includes notifications of sub-subjects or not. In principle, arbitrary pattern matching can be executed on subjects.

The simplicity of this approach has deficiencies that limits its applicability. The requirement to use a single path in a tree to classify a notification severely constrains the expressiveness of this model. The subject hierarchy is a tree—multiple super-subjects are not allowed—and it classifies only from a single point of view. Alternative classifications, e.g., `/Exchange/Europe/Technology/London`, are only possible if different subtrees permute the order of subjects. This leads to repeated publications and an exponential growth of tree size if several alternative viewpoints shall be reflected.[1]

Type-based selection uses path expressions and sub-type inclusion tests to select otherwise opaque notifications [28, 101]. With multiple inheritance, the subject tree is extended to type lattices that allow for different rooted paths to the same node. Often, type checking is complemented with content-based filters to improve selectivity [244].

Content-based filtering is the most general scheme of notification selection [59, 212]. Filters are evaluated on the whole content of notifications, where the data model of the notifications and the applied predicates determine the expressiveness of the filters. Available solutions range from template matching [83], simple comparisons [62] or extensible filter expressions [214] on name-value pairs, to XPath expressions on XML [14] and arbitrary programs and mobile code [97].

Concept-based publish/subscribe is orthogonal to the above approaches and is proposed by Cilia et al. [73]. It employs semantic mappings between data and filter models to transform subscriptions from one model to another.

[1]Similarly, from a software engineering point of view such hierarchies have been criticized as restrictive and impeding integration and evolution [153].

2.1.4 Event Notification Service

The event notification service, or notification service for short, is the mediator in event-based systems that decouples producers from consumers. It alone is responsible for conveying notifications and it must deliver every published notification to all consumers having registered matching subscriptions. It implements a publish/subscribe interface, providing *adv*, *pub*, *sub*, *unsub*, and *notify* calls; the last being an output function called on a registered consumer to deliver a notification. The notification service gets notifications from producers via the *pub* call, and they must match the advertisements issued with the *adv* call. The service tests notifications against subscriptions it got from consumers via the *sub* call, and delivers them to those consumers having a matching subscription with *notify*. In essence, it separates communication responsibility from components in the sense that the mediating service is responsible for subscription evaluation on behalf of the consumers and for delivering notifications on behalf of the producers.

From the perspective of application components, the notification service is a black box. Its function does not depend on it being distributed. However, its non-functional attributes, like efficiency and availability, are influenced by the architecture and the communication techniques used to distribute notifications.

In addition to the notification service, event-based systems often contain further event handling capabilities, such as event and notification type repositories, descriptions of available data and filter models, and other 'meta data,' as well as programming language bindings beyond service invocations. To reflect the broader functionality the collection of notification service plus any additional event handling is termed *event system*.

2.2 Models of Interaction

From a technical point of view, an event notification service just provides publish/subscribe functionality, which *may* be used for transporting notifications, but also for sending requests to groups of servers. The essence of event-based systems is not found in the API or the techniques used for transmitting notifications. Event-based interaction is mainly a characteristic of the components, and not of the underlying communication technique [45, 215].

In order to provide a fundamental and simple characterization, four *interaction models* are distinguished by the way interdependencies between components are established. The four models are differentiated by two attributes (see Fig. 2.2). The first attribute, *initiator*, describes whether consumer or provider initiates the interaction, where the former depends on data or functionality provided by the latter. The second attribute, *addressing*, distinguishes whether the addressee of the interaction is known or unknown, i.e., whether the peer component is directly or indirectly addressed.

The resulting four interaction models are independent of any underlying implementation technique. Any interaction between a set of components can be

		Initiator	
		Consumer	Provider
Addressee	Direct	*Request/Reply*	*Pt-to-Pt Messaging*
	Indirect	*Anonymous Request/Reply*	*Event-Based*

Figure 2.2: Taxonomy of cooperation models

classified according to these models. Even though interaction may show more nuances in practice, the models are complete in the sense that they essentially cover all major paradigms.

Furthermore, the interaction models characterize the inner structure of components, because the models determine how dependencies between the components are established. From an engineering point of view, this helps to identify constraints and requirements posed by a given component on its usage scenarios and on the underlying infrastructure. Architectural mismatches are disclosed early, and would otherwise have to be tackled by an integrating implementation, which impedes system evolution and scalability sooner or later [130].

2.2.1 Request/Reply

The most widely used interaction model is request/reply. Any kind of remote procedure call or client/server interaction belongs to this class. The initiator is the consumer (i.e., client) that requests data and/or functionality from the provider (i.e., server), and it expects data to be delivered back or relies on a specific task to be done. The provider is directly addressed, its identity is known, and the caller is able to incorporate information about the callee into his own state and processing, resulting in a tight coupling of the cooperating entities. Replies are mandatory in this model.

2.2.2 Anonymous Request/Reply

The anonymous request/reply model also uses request/reply as basic action, but without specifying the provider that should process the request. Instead, requests are delivered to an arbitrary, possibly dynamically determined set of providers. The consumer does not know the identity of the recipient(s) a priori, yet it expects at least one reply—one request may result in an unknown number of replies.

This model is eligible when redundant providers are available or when the appropriate provider may be different for each request. For instance, load balancing selects a provider either arbitrarily or based on the content of the request; cf. the IP Anycast mechanism [241] tries to route a packet to the nearest member of a group of destinations without resolving the IP address in advance. Similarly, component models and containers decouple component instances and allow for runtime binding of references, cf. JavaBeans [278] and the *Dependency Injection Pattern* [121, 200]. However, this often only means providers are resolved just before the call, making the identity known to the caller and potentially leading to tight coupling as in classic request/reply.

This cooperation model is besides the event-based model the second model that is directly implemented by pub/sub services, which often confuses these two models. Anonymity of providers adds more flexibility to the request/reply model, but dependencies on externally provided data or functionality persists.

2.2.3 Point-to-Point Messaging

Point-to-point messaging comes in two flavors: solicited and unsolicited messages. The latter case is like spam, sent directly towards addressees without requests. Solicited messages in this model correspond to the well-known callback mode of interaction.

In the callback model, which is employed in the well-known *observer* design pattern [129], consumers register at a specific, known provider their interest to be notified whenever some condition becomes true. The provider repeatedly evaluates the condition and if necessary calls the registered component back. The provider is responsible for administering its callback list of registered consumers. If multiple callback providers are of interest, a consumer must register separately for all of them. The identity of the components is known and must be managed on both sides, leading to a tight coupling with no coordination medium in between.

On the other hand, knowing the identities of consumers, callback processing can be customized so that only subsets of consumers are notified in an application-dependent way. However, it would be each component's responsibility to apply callback handlers that implement current application needs, which is an issue of integration rather than of component implementation. In any case, a sophisticated implementation of callback handlers leads to the event-based approach, described next.

2.2.4 Event-Based

The event-based interaction model has characteristics inverse to the request/reply model. The initiator of communication is the provider of data, that is, the producer of notifications. Notifications are not addressed to any specific set of recipients, as was described earlier. A consumer can receive notifications from many providers, because subscriptions are in general neither directed nor limited to a particular producer. If a notification matches a subscription, it is delivered to the registered consumer. Providers are not aware of the consumers. In contrast to the callback model, providers are relieved from the task of interpreting and administering registrations, i.e., subscriptions.

The essential characteristic of this model is that producers do not know any consumers. They send information about their own state only, precluding any assumptions on consumer functionality. A component 'knows' how to react to incoming notifications and it publishes changes to its own state, but it must not publish a notification with the intention of triggering other activity. A component's implementation is 'self-focused' in that the knowledge encoded in the program, and used by the programmer, is limited to the component's own task. This approach completely separates the internals of different parts of an application.

Of course, the overall functionality of the system still depends on the proper interaction of all the components, but this is no longer a matter of individual components. It is rather the composition of components and their interaction that determine the functionality. But event-based interaction withdraws the control of interaction from the participating components and the necessary coordination has to be handled externally. So, in addition to the role of specifying and implementing individual components, the orchestration of an event-based system demands extra support. Currently, no such support is available.

2.2.5 Comparison

The complexity of a decomposed system is characterized by the degree of dependence between its components. Software reliability analysis formally corroborates a result that is informally apparent: If a component relies on other components to accomplish its own goal, its correctness is degraded by failures of others [2, 201]. Conversely, the correctness of individual components is not affected if they process available data only, which is exactly the case in event-based systems. The event-based style clearly separates computation from communication and offers the potential of easily evolvable systems. On the other hand, engineering complexity is considerably affected by the quality of the abstractions and tools available for coordinating the components of a system.

The dichotomy of request/reply and event-based interaction is marked by the simplicity of the former and the flexibility of the latter. Request/reply is easy to handle, implement, and understand, and consequently well established. It corresponds to the imperative nature of common programming languages and component models. Some of its shortcomings are alleviated by a long list of supplementary techniques such as caching, asynchronous request/reply, container-controlled operation, dependency injection, etc., that are used to enhance scalability and system evolution.

However, if interaction becomes less coupled, it gets more indirect. And this raises the question if not the use of events would be a more appropriate solution. In fact, without being formally corroborated, it appears that request/reply and event-based interaction form a duality in the sense that for most problems there exist solutions based on either model. Classic request/reply examples can be rebuild using events. Event-based interaction typically relies on a reversed software architecture, reversing activity and data flows, but the same function can be implemented in both paradigms. The involved tradeoff is between scalability and flexibility, on the one hand, and simplicity on the other. System engineers have to decide whether they opt for a simple implementation or for an extensible one. And one goal of this thesis is to make choosing the extensible solution less costly, and thus eligible for more scenarios.

2.2.6 Interaction vs. Implementation

The mode of interaction influences the design of components and is difficult to change. It is a prerequisite of good design to choose an interaction model that matches the function a component has to accomplish. Otherwise, architectural mismatches would inevitably impede system composition and evolution [130]. For this reason, this basic but principal distinction of interaction models helps system designers to identify the core structure of components, and it avoids mixing interaction and implementation issues [113].

Unfortunately, the mode of interaction is often confused with the choice of implementation techniques currently available. In particular, event-based interaction is often equated with using general publish/subscribe services. While being obvious candidates for implementing notification dissemination, they are not the only ones; other techniques may as well be employed, like point-to-point messaging, IP multicast, Linda tuple space engines, or even classical remote procedure calls. For instance, if a system engineer *knows* that a set of event-based components interact only within a small group, nothing speaks against using RPC. In fact, if the communication happens to be sensitive to eavesdropping, RPC even becomes the most appropriate choice. Note that producers still publish notifications as before, only the underlying implementation is considered here. Conversely, a pub/sub service can also be used to implement anonymous request/reply interaction.

Generally, there is no best implementation technique for a certain interaction model. The technique must be chosen in view of the deployment environment, the demanded quality of service, and the overall need for flexibility and scalability. Event-based interaction facilitates the distinction of interaction and implementation due to its separation of computation from communication. And while traditional pub/sub services focus on unidirectional delivery (see Sect. 2.5), many different techniques can be exploited in building event-based systems.

The preceding description of event-based interaction basically refines the one given in literature, e.g., [57, 134, 237]. The discussion makes it now possible to unambiguously define the involved terminology. The system outline given in Fig. 2.1 on page 11 spans several levels of abstraction. On the lowest level, *messages* are *sent* and *received*. Arbitrary asynchronous messaging techniques can be used, be it connectionless point-to-point network protocols, IP multicast mechanisms, or publish/subscribe implementations.

On the next level, the publish/subscribe interface is implemented. It is used to *publish* data that is *delivered* to *subscribers*. As part of its implementation, messages containing the data are sent and received. From a technical point of view, the pub/sub interface implements both anonymous request/reply and event-based interaction.[2]

On the highest level, where event-based interaction finally takes place, *producers* publish notifications that are delivered to *consumers*. Only this level is of concern when assessing the characteristics of event-based interaction and its effect on system engineering.

2.3 Simple Event-Based Systems

A considerable amount of work on event-based systems and notification services exists, and many concrete systems have been designed and implemented. Unfortunately, understanding and comparing these systems is very difficult because of different and informal semantics. This section contains a formalism that helps to specify the semantics of an event-based system unambiguously. In Section 2.3.2 this formalism is used to specify a simple event system which captures the requirements considered mandatory for the basic level of service. Section 2.3.3 sketches an implementation of a simple event system and presents arguments for the correctness according to the specification given. This specification is extended in later sections to construct complex scoped event systems in an incremental fashion.

2.3.1 Formal Background

In the literature on program verification, there exist well-developed foundations of methods to specify and validate concurrent systems. The aim of the proposed formalisms is to precisely describe the behavior of a system as a "black box", i.e., without referring to its internal (implementation) issues. Usually, the proposed formalisms model an interactive system as a state machine which moves from one state to another by means of an action. Formally this corresponds to the definition of a *labeled transition system.* The black box view entails defining the behavior of such a system in terms of the states and actions it exhibits at its *visible interface.* In the literature this is termed *observation semantics* and there are many different possibilities of defining observation semantics for concurrent systems. A simple example is *trace semantics*, which amounts to defining an observation simply as a sequence of actions that are visible at the system interface. Intuitively, system evolution can be written as a sequence of transitions [44]

$$s_1 \overset{a_1}{\rightarrow} s_2 \overset{a_2}{\rightarrow} s_3 \ldots$$

which denotes that starting from state s_1 the system reaches state s_2 by executing action a_1, etc. Note that trace semantics can also be used to describe the behavior of concurrent systems. The global state space of a set of concurrent processes, for example, is defined by the cross product of the state space of

[2]Although all arguments made in this thesis explicitly target event-based systems, they are equally applicable to any general publish/subscribe scenario.

the individual processes. The evolution of the system can then be viewed as a sequence of global states that occur by interleaving the individual process traces.

One might argue that defining a trace as a total order is unrealistic in a distributed system because it is not possible or desirable to enforce total ordering of operations. Indeed, it is possible to give specifications which are not (efficiently) implementable because of the lack of a notion of global time in distributed systems. However, the specification of the following Def. 2.5 is implementable because it imposes ordering relations only on operations which intentionally should be causally related in any sensible implementation.

Trace semantics is used to specify the behavior of systems by abstracting from states and reasoning only about the sequence of operations at the interface of the system. Given a set A of possible interface actions, a *trace* $\sigma = a_1, a_2, \ldots$ is a sequence of elements of A. A *specification* then is a set of such traces, namely all traces which are allowed to be generated by a system. Equivalently, a specification can be given as a predicate on traces. Temporal logic [248] is utilized to express such predicates. The formal language is built from simple predicates, the quantifiers $\forall$, $\exists$, the logical operators $\vee$, $\wedge$, $\Rightarrow$, $\neg$, and the "temporal" operators $\Box$ ("always"), $\Diamond$ ("eventually"), and $\bigcirc$ ("next"). For a given action $a \in A$, the formula a is true for every trace which starts with a. The formula $\neg a$ is true for every trace which does not start with action a. The other logical operators and quantifiers are defined in the obvious analogous way.

The semantics of the temporal operators are defined as follows: Let Ψ be an arbitrary formula and $\sigma = a_1, a_2, \ldots$ be a trace. Then

- $\Diamond\Psi$ is true for trace σ iff there exists an $i > 0$ such that Ψ is true for the trace $a_i, a_{i+1}, a_{i+2}, \ldots$,
- $\Box\Psi$ is true for trace σ iff for all $i > 0$, Ψ is true for the trace $a_i, a_{i+1}, a_{i+2}, \ldots$, and
- $\bigcirc\Psi$ is true for trace σ iff Ψ is true for $a_2, a_3, \ldots$

Note that the temporal operators have higher precedence than the logical operators.

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

Figure 2.3: Interface operations of a simple event system

To better understand temporal formulas, a few examples are now given using the interface operations of a simple event system listed in Table 2.3.1. In order to capture which client is affected by an operation, the operations include a reference to the respective client. For example, $sub(X, F)$ means that client X subscribes to filter F.

Intuitively, $\Diamond\Psi$ means that Ψ will hold eventually, i.e., there exists a point in the trace at which Ψ holds. For example,

$$\Diamond notify(X, n)$$

specifies all traces in which client X eventually is notified about n. On the other hand, $\Box\Psi$ means that Ψ always holds, i.e., for all "future" points in the trace Ψ holds. For example,

$$\Box\neg unsub(X, F)$$

specifies all traces in which X never unsubscribes to F. Finally, $\bigcirc\Psi$ means that Ψ holds in the next step, i.e., for the trace starting with a_2. For example,

$$\Box\big[notify(Y, n) \;\Rightarrow\; \bigcirc\Box\neg notify(Y, n)\big]$$

specifies all traces in which, if Y is notified about n, Y is never notified about n again.

Given an arbitrary labeled transition system with a set of initial states, the system *satisfies* a given temporal formula iff every observable behavior of the system is a trace specified by the temporal formula. This means that the set of traces generated by the system must be a subset of the set of traces specified by the formula. This notion of satisfaction means that the system *implements* the specification and is sometimes called *refinement* [1] or *process preorder* [33]. Of course, in order to correctly implement a specification, a system usually has to execute internal (unobservable) actions different from the interface actions. To model this, some formalisms define an internal action τ and allow for any finite number of internal actions in between two interface actions. This is sometimes called *weak equivalence* [33] or *stuttering equivalence* [2, 180]. Inference rules and other proof techniques can then be used to formally derive the satisfaction relation.

In the following, the thesis will be very precise when defining the specification of event systems. The proofs that the algorithms implement the specification will follow the standard mathematical textbook style, avoiding a fully formal derivation of the correctness of the given algorithms to not obscure the main contributions, which lie more in the design and specification areas than in verification.

2.3.2 Specification of a Simple Event System

Basic Definitions

The *data model* of an event system describes structure and syntax of published notifications. Every notification must adhere to this model, and it also is the basis for the filter model described below. A number of different data models exist in theory and practice, ranging from unstructured text or bit strings [177], tuples of strings [83], name-value pairs [62], to hierarchical object models [18, 101, 212] and semi-structured data [14, 21, 214]. A homogeneous system is assumed at first, where exactly one data model is used.[3]

Definition 2.1 *A* data model *is a set $\mathcal{D}$ of complex data items with select functions $\sigma_i : \mathcal{D} \to \mathcal{D}_i$ selecting a constituent value. A notification $n \in \mathcal{N}$ represents such a data item, signified by $\sigma(n) \in \mathcal{D}$. A* name-value pair data model *is based on a set of attributes $\mathcal{A} = \{A_1, A_2, \ldots\}$ with associated value domains $\mathcal{D}_{A_i}$:*

$$\mathcal{D}^{NV} = \{\ (a_i, v_i)_i \mid a_i \in \mathcal{A} \ \ \textit{and} \ \ v_i \in \mathcal{D}_{a_i}\ \}$$

where select functions are defined such that

$$\sigma(n) = (a_i, v_i)_i$$
$$\sigma_A(n) = \begin{cases} v_i & \textit{if } (A, v_i) \in \sigma(n) \\ \epsilon & \textit{otherwise} \end{cases}$$

where ϵ is a special empty value never used in any notification.[4] *For short $n.A = v_i$ is written.*

A data model $\mathcal{D}$ and the corresponding set of notifications $\mathcal{N}$ can be used interchangeably in most cases except for representational differences. The definition of $\mathcal{D}^{NV}$ conforms to the widely used name-value pairs model. Notifications in this data model are described by tuples like (“type”, “NewEmployee”, “Name”, “Wile E. Coyote”, “Age”, “42”). A data model that contains tuples instead of records uses values without attribute names and is defined by $\mathcal{A} = \{1, 2, 3, \ldots\}$, some $\mathcal{D}_i$, and $\sigma_i(n) = v_i$. The Linda [140] coordination system uses this model, for instance. A SIENA-like model is defined by allowing strings as attributes and a predefined set of value domains, like string, integers, floats and time, as they are commonly used in data definition languages or programming languages. Note that the name-value pair data model is not limited to a predefined set of value domains. Even hierarchical data can be modeled if attributes denote access paths in structured data types, e.g., by using a dotted notation for attributes (n.customer.name) [214]. The step towards object-oriented data models is then straightforward.

Before defining filters, predicates on notifications are introduced. A predicate is evaluated on a notification and gives a value from the Boolean domain $\mathbb{B} = \{\textit{true}, \textit{false}\}$.

Definition 2.2 *A* predicate *$P \in \mathcal{P}$ is specified by a triple (P^σ, Op, o), containing a select function $P^\sigma : \mathcal{D} \to \mathcal{D}_i$, a predicate operation $Op : \mathcal{D}_i \times \mathcal{D}_P \to \mathbb{B} =$*

[3]Aspects of heterogeneity are investigated later in Section 3.4.

[4]The symbol ϵ is borrowed from formal languages where it denotes the string with no symbols.

$\{true, false\}$, and operand o of some predicate specific domain $\mathcal{D}_P$, such that

$$P : \mathcal{N} \rightarrow \mathbb{B}$$
$$P(n) = \begin{cases} Op(P^\sigma(n), o) & \text{if } P^\sigma(n) \neq \epsilon \\ false & otherwise \end{cases}$$

The select function P^σ must be defined on the underlying data model and selects the data on which the predicate is applied. For the actual test operation Op any boolean-valued function is eligible that is defined on the values selected by P^σ, although in practice the complexity of the applied computation has to be restricted to keep this operation tractable. The constant predicate-specific operand o supplied to each evaluation parameterizes the general purpose implementation in Op. In the name-value pair model, P is specified by a triple (A, Op, o) with a select function σ_A as defined above. An example predicate is $P =$ (“type”, string-comparison, “weather”) that tests for weather notifications, or $P =$ (“location”, within-rectangle, (0,0, 10,10)) to test a location for being within a rectangle given by the corner points $(0, 0)$ and $(10, 10)$.

Filters are often defined as Boolean-valued functions. In the following, however, filters are defined to map notifications back onto the set of notifications. The same expressiveness is achieved if filters result in either the unchanged notification itself or a designated *empty* notification $\epsilon \in \mathcal{N}$, signifying acceptance or rejection by the filter, respectively. This facilitates filter composition, $(F_1 \circ F_2)(n) = F_1(F_2(n))$ and later extensions of their functionality, cf. Section 3.3. The set of all notifications that match a filter F is denoted by $N(F)$.

Definition 2.3 *A* filter model *$\mathcal{F}$ is a set of functions defined on a given data model which maps a notification onto itself or onto the empty notification ϵ. Every* filter *$F \in \mathcal{F}$ is described by a conjunction of predicates $F^P = \{P_i\}_i$:*

$$\mathcal{F} = \{F \mid F : \mathcal{N} \rightarrow \mathcal{N}\}$$
$$F(n) = \begin{cases} n & \text{if } \forall P_i \in F^P : \; P_i(n) = true \\ \epsilon & otherwise \end{cases}$$

The above predicate example may be extended to filter for hot weather news from Berlin: $F^P = \big\{$ (“type”, string-comparison, “weather”), (“city”, string-comparison, “Berlin”), (“temp”, integer-greater, 30) $\big\}$.

The last definition concerns the software artifacts of which event-based applications are composed.

Definition 2.4 *A* component *$C \in \mathcal{C}$ is any software artifact that is part of the event-based system and capable of publishing or consuming notifications; it acts as a client of the notification service.*

Interface Operations

Formally, a *simple event system* is viewed as a black box with an interface (see Figure 2.4). The possible interface operations are listed in Table 2.3.1. All these operations are instantaneous and take parameters from different domains: the set of all clients $\mathcal{C}$, the set of all notifications $\mathcal{N}$, and the set of all filters $\mathcal{F}$. In addition to the preceding definitions, two further assumptions are made: First, notifications are unique, i.e., each notification $n \in \mathcal{N}$ is published at most once. Second, every filter is associated with a unique identifier in order to enable the event system to distinguish subscriptions.

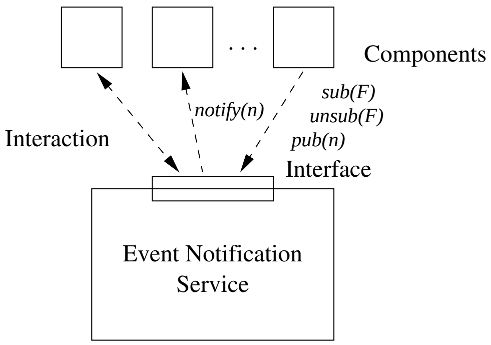


Figure 2.4: Black box view of an event system

Specification Variables

In the formalization a set of *specification variables* is assumed to be present. Specification variables are fictitious devices which keep track of the internal history of the system within a specification and simplify the temporal formulas; they are not necessarily part of any implementation. Two sets of specification variables are assumed at the system interface for every client $X \in \mathcal{C}$:

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

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

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

The behavior of the event system is specified by giving a set of temporal formulas like the examples introduced in Section 2.3.1. Of course, it is also possible to refer to the specification variables. For example,

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

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

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

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

Simple Event Systems

In the following, a specification of simple event systems is presented that relies on the trace-based semantics introduced above [114, 115].[5] Informally, it conforms to the following requirements

(a) only notifications that match one of its active subscriptions should be delivered to a client,

(b) each notification should be delivered at most once to a client,

(c) all notifications matching one of its active subscriptions should be delivered to a client.

Definition 2.5 (simple event system) *A* simple event system *ES is a system that exhibits only traces satisfying the following requirements:*

- *(Safety)*

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

- *(Liveness)*

$$\begin{aligned}\Box\Big[\mathit{sub}(Y,F) \;\Rightarrow\; & \big[\Diamond\Box\big(\mathit{pub}(X,n) \wedge n \in N(F) \;\Rightarrow\; \Diamond\mathit{notify}(Y,n)\big)\big]\\ & \vee \big[\Diamond\mathit{unsub}(Y,F)\big]\Big]\end{aligned}$$

[5]The specification is extended in [213] to include advertisements and self-stabilization in the case of failures.

The specification consists of a safety and a liveness condition [179]. A *safety condition* demands that "something irremediably bad" will never happen, while a *liveness condition* requires that "something good" will eventually happen.[6] It has been shown that all properties on traces can be expressed as the intersection of safety and liveness conditions [12, 135, 136]. Here, the safety condition states that a notification should never be delivered to a consumer more than once, that a delivered notification must have been published by a client in the past, and that a notification should only be delivered to a client if it matches one of the client's active subscriptions; entailing requirements (a) and (b) from the beginning of this section. The liveness condition is probably most difficult to understand. It describes precisely under which conditions a notification must be delivered. The condition can be rephrased as follows: if a client Y subscribes to F, then there exists a future point in time where the publishing of a notification n matching F will lead to a delivery of n to Y. This can only be circumvented by Y unsubscribing to F. The liveness condition can be regarded as a precise formulation of requirement (c). Note that no delivery order is imposed on notifications (like causal order) as it is a highly implementation-dependent and application-specific issue, and hence is left out of consideration when defining the semantics of simple event systems.

As examples consider the following traces where F is a filter and n_i are notifications matching F while n' is a notification not matching F:

$$\begin{aligned}
\sigma_1 &= \mathit{sub}(Y,F), \mathit{pub}(X,n_1), \mathit{notify}(Y,n') \\
\sigma_2 &= \mathit{pub}(X,n_1), \mathit{sub}(Y,F), \mathit{unsub}(Y,F), \mathit{notify}(Y,n_1) \\
\sigma_3 &= \mathit{sub}(Y,F), \mathit{pub}(X,n_1), \mathit{pub}(X,n_2), \mathit{pub}(X,n_3), \ldots
\end{aligned}$$

Traces σ_1 and σ_2 violate the safety requirement because a notification is delivered to Y that does not match an active subscription. In trace σ_3 component Y subscribes to F and X starts to publish a continuous sequence of notifications matching F. Since there is no *notify* in σ_3 it perfectly satisfies safety. However, it violates the liveness requirement (to satisfy liveness, there must be a point in the trace after the subscription where either Y unsubscribes to F or Y begins to receive notifications).

Intuitively, the liveness requirement states that any *finite* processing delay of a subscription is acceptable. By abstracting away from real time a concise and unambiguous characterization system behavior is obtained. For example, if a component has subscribed to a filter F and later unsubscribes to it, the system does not have to notify the component about *any* notifications that match F and are published in the meantime; it may nevertheless do so. Liveness requires delivery if the component continuously remains subscribed to F. Because the system cannot tell the future, it must at some point start to deliver notifications until the component unsubscribes to F.

Furthermore, the definition of liveness does not rely on the initially discussed global order of states and transitions. It does not relate subscribing and pub-

[6]For a formal definition of safety and liveness refer to Broy and Olderog [44].

lishing operations to each other, because they are causally independent and no semantics is implied here. As an advantage future extensions can build on this definition to introduce real-time requirements that prevent old notifications from being delivered to new subscriptions, or caching strategies that allow for a defined history of notifications to be delivered to newly issued subscriptions.

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

2.3.3 Implementation

The following paragraphs show how to implement the specification of a simple event system. For brevity, only a very simple implementation is shown, which still demonstrates feasibility while other approaches can be found elsewhere to improve the scalability of the implementation [62, 104, 244]; in particular companion work on routing in the Rebeca notification service investigates a correct and efficient implementation of this specification [213, 217]. Since later extensions (scoped event systems) utilize any instance of simple event systems, it is possible to exchange the implementation with a more efficient one as long as the interface specification is maintained.

The implementation is based on a system model where a set of asynchronous processes communicate over point-to-point message passing channels. The channels are assumed to be reliable, i.e., no messages are lost or altered and no spurious messages are delivered, and incoming data is served in a fair manner. The communication topology of processes is assumed to be acyclic and connected, i.e., a spanning tree (Figure 2.5). In practice, acyclic connected topologies can be established manually or through spanning tree construction algorithms.

Within an event system, processes are *event brokers*. To invoke the interface operations of the event system, every client invokes a local library function to insert a notification into the system. This library is the *local event broker* of C.

Data Structures

Every local event broker holds two data structures:

1. a table S of active subscriptions, and

2. a table D of previously delivered events.

Both are initially empty, and D can be omitted if the communication graph is acyclic and reliable.

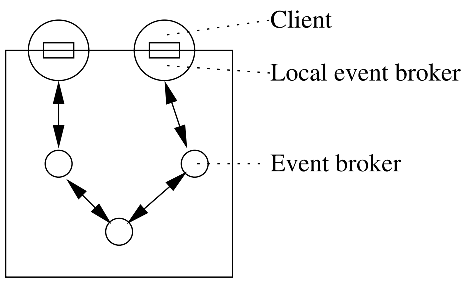


Figure 2.5: A possible implementation view of a simple event system

Algorithm

If a client X invokes $\mathit{sub}(X, F)$, the local event broker of X adds F to S. Conversely, if $\mathit{unsub}(X, F)$ is invoked, F is removed from S. Notifications are processed within the system by a technique called *flooding*. An invocation of $\mathit{pub}(X, n)$ causes sending a message containing n to the neighbors of the local event broker in the network. If any (non-local) event broker receives such a message, it forwards it to all neighbors except the one the message was received from. A local event broker (say of client Y) receiving such a message checks if there exists a filter F in S_Y such that n matches F. If so, it checks whether n is already present in D_Y. If one of these checks fails, it discards n. Otherwise n is added to D_Y and delivered to the client via a call to $\mathit{notify}(Y, n)$.

2.3.4 Correctness

It must be shown that the algorithm from the previous section satisfies the requirements given in Definition 2.5, i.e., the safety and liveness condition. In the proof, the implementation variable S essentially plays the role of the specification variable S_Y.

Proof of Safety

Assuming that the implementation invokes $\mathit{notify}(Y, n)$ at component Y, it must be shown that this implies that the three conjuncts of the implication in the safety condition hold.

From the algorithm and the use of D follows that n will not be delivered again. This proves the first conjunct: $\bigcirc\Box\neg \mathit{notify}(Y, n)$.

Also from the algorithm and the use of S, invocation of $\mathit{notify}(Y, n)$ implies that there exists an F in S_Y such that $n \in N(F)$. This proves the third conjunct: $\exists F.\, F \in S_Y \wedge n \in N(F)$.

It remains to be shown that n was previously published by some component: $\exists X.\, n \in P_X$. Invocation of $\mathit{notify}(Y, n)$ implies the receipt of a message containing n at the local event broker of Y. Because of the reliable channel assumption, this message must have been sent by some neighbor. Because of the forwarding algorithm of the event brokers, the acyclic topology, and the reliable channel assumption there must exist a local event broker of some component X from which n originated. From the algorithm, this implies that X previously published n. This proves the second conjunct and concludes the proof.

Proof of Liveness

Assume a component invokes $sub(Y, F)$ and never unsubscribes to F. Liveness requires that there is a time after which every notification which is published and matches F is eventually delivered to Y. In this case this is rather easy to show since subscriptions become active immediately: Let n be a notification matching F and consider a component X that invokes $pub(X, n)$ immediately after Y subscribed to F. From the algorithm follows that invocation of $pub(X, n)$ leads to the sending of a message containing n to all neighbors of the local event broker. From the forwarding algorithm of the event brokers, the acyclic topology, and the reliable channel assumption follows that the message is eventually received at every local event broker, including that of Y. Since Y has not unsubscribed to F and n matches F, the algorithm invokes $notify(Y, n)$, concluding the proof.

2.4 A Distributed Notification Service— The Rebeca Model

This section describes the system model and the basic characteristics of the Rebeca notification service [112]. It implements the publish/subscribe interface described in Sect. 2.1 and conforms to the preceding definition of simple event systems. Its basic architecture is a representative example of a distributed notification service, which is comparable to that of other services like Siena, JEDI, etc. Rebeca is different from other services with regard to its support for different routing algorithms and data and filter models [213, 217], and the visibility control extensions presented in this thesis. Rebeca serves two roles in this thesis: its system model is the basis for investigating visibility issues, and second, the available implementation acts as testbed to build a prototype of scoping in distributed event-based systems.

2.4.1 System Model

The model assumed in Rebeca and this thesis is a process model in which computational activity is represented by the concurrent execution of processes [181]. Processes interact by passing messages via links between them. A link connects a pair of processes and forwards messages asynchronously so that there is a delay between sending a message and receiving it. Links are assumed to exhibit no failures and to obey FIFO (first-in-first-out) ordering of messages. This means that no messages are lost or corrupted due to link failures and that messages are received in the same order they were sent. Although being impractical in general, it is a reasonable assumption in the present context, because it simplifies the discussion and helps concentrating on the functional aspects of visibility in event-based systems. In fact, initial solutions for both problems exist elsewhere and may be used later to extend the model, e.g., [80, 213].

More concretely, the considered distributed system consists of a set of physical nodes interconnected by a communication network and each node runs one or more processes. Communication links are point-to-point connections in this network, and their failure model is easily matched by TCP/IP connections, for instance. This is the basic model that is broadly applicable, and which nevertheless is open for implementation-dependent options, like using Multicast, to improve communication performance (cf. Section 4.5.7).

2.4.2 Architecture

The system constituents are illustrated in Figure 2.6 and both the application components and the notification service itself are implemented by the aforementioned processes. Each component is executed by a separate process, which is linked to a process of the notification service. The service is accessed as a black box that is conceptually centralized, but distributed across several processes and nodes to split the load and exploit locality in notification delivery.

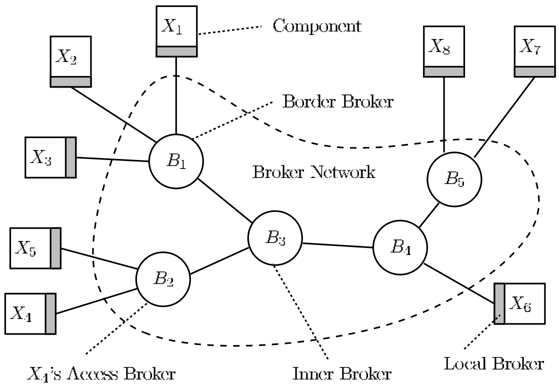


Figure 2.6: The router network of Rebeca

The notification service forms an overlay network in the underlying system. An *overlay network* is a virtual network of processes that communicate by means of a second underlying (physical) network, employing routing strategies different from the underlying ones. Here, the overlay consists of event brokers that run as processes on some of the physical nodes. The communication topology of the overlay is described by an acyclic graph. Edges are process links and as such mapped to point-to-point connections in the underlying network, namely, TCP/IP connections. The acyclic graph used is comparable to the single spanning tree approach of multicast algorithms [91]. Obviously, the single tree is a bottleneck of the system, but, again, it is an adequate model in this context, and extensions exploiting redundancy are available to tackle problems of scalability and single points of failure [80, 244, 291].

Three types of brokers are distinguished: local, border, and inner brokers. *Local brokers* are access points to the middleware. They are typically part of the communication library loaded into application components; they are not represented in the graph, but only used for implementation issues. A local broker is connected to one border broker. *Border brokers* form the boundary of the distributed communication middleware and maintain connections to local brokers, i.e., the clients of the service. *Inner brokers* are connected to other inner or border brokers and do not maintain any connections to clients. Local brokers implement the publish/subscribe interface of the notification service and initially put the first message containing a newly published notification into the network. Border and inner brokers forward the messages to neighbor brokers according to filter-based routing tables and respective routing strategies. At the end the messages are sent to the local brokers of the consumers and from there the notifications are delivered to the application components.

2.4.3 Filter-Based Routing

The function of distributed notification routing is rather simple: just match all notifications with all subscriptions and deliver where appropriate. In a centralized implementation the problem is reduced to efficient matching algorithms [216, 303]. This approach, however, does not only concentrate all computational efforts but also becomes a bottleneck of communication bandwidth. Hence, Rebeca distributes matching on multiple brokers.

Filter-based routing depends on routing tables (RT), which are maintained in the brokers and contain filter-link pairs. Each pair indicates in which direction matching notifications have to be forwarded. The entries are updated by sending new or canceled subscriptions through the broker network. New subscriptions add (F, L) entries with L denoting the link from which they were received, and unsubscriptions delete the respective entries. Every incoming notification is tested against the routing table entries to determine the set of links with matching filters, omitting the originating link to prevent loops. In a second step the notification is forwarded to the respective neighbor brokers. If the incoming notifications of each link are routed sequentially, end-to-end sender FIFO characteristics hold.

Different flavors of filter-based routing exist, which differ in their strategy to update the routing tables. *Simple routing* assumes that each broker has global knowledge about all active subscriptions. It minimizes the amount of notification traffic, but the routing tables may grow excessively. Moreover, every (un)subscription has to be processed by every broker resulting in a high filter forwarding overhead if subscriptions change frequently. In large-scale systems more advanced routing algorithms must be applied to exploit commonalities among subscriptions in order to reduce routing table sizes [217]. REBECA includes three of them [213]. *Identity-based routing* avoids forwarding of subscriptions that match identical sets of notifications. *Covering-based routing* [62] avoids forwarding of those subscriptions that only accept a subset of notifications matched by a previously forwarded subscription. Note that this implies that it might be necessary to forward some of the covered subscriptions along with an unsubscription if a subscription is canceled. *Merging-based routing* [216] can be implemented on top of covering and goes even further. In this case, each broker can merge existing routing entries into a broader subscription, i.e., the broker creates a new cover for the merged routing entries that replaces the old ones. Only the resulting merged filter has to be forwarded to neighbor brokers, where it covers and replaces existent base filters. Merging can be done either in a perfect or imperfect way. Perfectly merged filters only accept notifications that are accepted by at least one of its base filters, whereas imperfectly merged filters accept notifications besides their base filters. Imperfect routing table entries increase network traffic but allow for lazy updates, hiding frequent reconfigurations in covered parts of the network.

Advertisements are an additional mechanism to optimize subscription forwarding. Subscriptions need only be forwarded into those subnets of the overlay network where a producer has issued an overlapping advertisement, i.e., where matching notifications can be produced at all. If a new advertisement is issued, overlapping subscriptions are forwarded appropriately. Similarly, if an advertisement is revoked, it is forwarded, and remote subscriptions that can no longer be serviced are dropped. Advertisements can be combined with all routing algorithms discussed above.

2.5 Missing Functionality

The previous section illustrated the operation of a distributed notification service similar to the ones considered in other current research. This subsection points out shortcomings of these services that make them difficult to maintain, let alone control, and that impede their use in complex application scenarios. The deficiencies sketched in the introduction are analyzed with the help of example scenarios, and a set of engineering requirements are inferred which must be supported by event systems. Two main problems are identified. The first is that event-based systems do basically not imply other requirements for designing and engineering than those already known from engineering request/reply systems. The second observation is that while supporting abstractions are available for the latter they are missing for event-based systems.

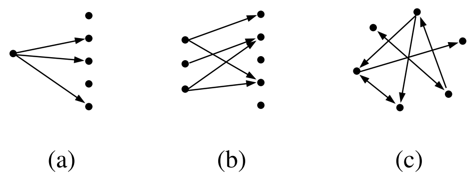


Figure 2.7: Call graphs of applications: bipartite single and multi source, and a general group

2.5.1 Application Scenarios

A taxonomy of application scenarios is created according to the complexity of interaction between application components, which is described by the call graph of who is sending notifications to whom: one-to-many, many-to-many, and repeated, ‘stateful’ communication.

Information Dissemination

The most simple and obvious application scenario of event-based communication is information dissemination and push services. It is typically characterized by a single, well-defined information source publishing notifications towards consumers (one-to-many communication). Applications are oblivious to the actual set of receivers and typically require high scalability. The call graph is bipartite, cf. Fig. 2.7(a), which means it consists of two distinct sets of components and communication takes place only between, not within the sets. Example applications are:

- *monitoring* of stock prices, sensor data, real-time control systems, process execution, etc. [143, 175, 203],
- *push services* in electronic commerce, news feeds like weather forecasts and sports [66, 93], and
- *content delivery networks* [9, 261].

This is the classic application domain of event-based systems, and also of network level multicast [253]. However, even in this simple scenario issues arise that are not covered by typical event services. The weather information may contain temperatures in Fahrenheit whereas consumers expect degree centigrade. Stock quotations may be published using an established financial markup language like FIXML [223] to facilitate interoperability, whereas internal operations of producers and consumers stick to more efficient binary representations. The heterogeneity of data models and the limited support thereof often demands manual adaptations before connecting components to information busses. Furthermore, security in event-based systems is a critical open issue. Who is allowed to view sensor data that monitors a person's presence or health? Access to real-time stock quotations may be restricted requiring subscriptions with additional fees.

Groups of Producers

In Figure 2.7(b) a slightly more complex scenario is depicted that includes multiple producers publishing similar notifications. This raises new problems if it is necessary to distinguish the sources, especially when systems evolve from type (a) to (b). Consider

- *multiple* stock markets or auction platforms publishing similar information [42, 110],
- *multiple* application-specific beacons or sensors that are deployed somewhere in the infrastructure [18].

When a system implementing one stock market is connected to another market, measures must be taken to prevent unintended effects on existing consumers. It must be possible to restrict communication to one market so that components do not react incorrectly to external events. The necessary distinction of markets is often achieved by simply having producers annotate notifications with a name or an ID (of the market for example). Here, producers encode in the notification the context of an event, e.g., the market it originated from. Consumers can be put into a specific context if they test for this information in their subscriptions.

This is a straightforward approach, of course, but it draws context knowledge into application components that pertains to the interaction and not to the component's implementation. Moreover, this context specification does not only counteract the characteristics of event-based systems, but it is unnecessary *within* the respective context. Consider the second example where presence awareness sensors inform about people/objects moving within a building. The notifications include an ID of the object tracked and a room number. If events from multiple buildings are integrated in a facility management application, an identifier of the 'source building' must be included in the notifications. The integration of existing applications thus would influence their internal configuration.

Therefore, application components should not be forced to deal with their execution context. They would have to consider all possible contexts, which inhibits run-time evolution and is neither desirable nor needed.

Complex Interaction

The third class considered comprises complex applications that have arbitrary call graphs and include bidirectional communication, see Figure 2.7(c). Examples are:

- chat groups, multi-player games, or computer supported cooperative work (CSCW) tend to cluster interacting groups of components [97, 128];

- *virtual marketplaces* exhibit complex interactions where sequences of published notifications are interrelated, e.g., auctions [42, 110];
- wireless sensor networks [10, 161] convey data from sources to sinks and process and filter data within the network

Apart from the last example, such scenarios are seldom considered in the context of event-based systems because their requirements exceed pure scalability considerations. They point out that the principle of locality is important in event-based systems, too. Clusters of closely interacting participants can be identified as part of larger applications; the call graphs are more dense within these clusters than towards the outside. And within such groups often more stringent requirements are placed on communication quality. For instance, a chat application exchanging user input via notifications will certainly gain from ordering guarantees for notification delivery, e.g., atomic broadcast providing each participant with the same perceived order of inputs. In general, intra-cluster communication may require dedicated services whereas interaction with the remaining system gets by with the basic functionality of notification dissemination.

The virtual marketplaces illustrate the need to group notifications. Producers and consumers do not know each other but must establish a conversation[7] by relating notifications that belong to the same auction. Again, a simple workaround is directly found by inserting identifiers in notifications, and the same counter-arguments as above still apply. Identifiers may be viable in this simple case, but in more general terms the context of notifications must be distinguished to relate bids to auctions, reactions to actions, and events to transactions.

2.5.2 Engineering Requirements

The above discussion exemplifies the problems already sketched in the introduction: effects and side effects, design, implementation, and engineering, management and security issues. From these problem domains four requirements for the engineering of event-based systems are inferred: bundling of components, support for heterogeneity, flexible customization, and support for activities.

Illustrative Example

A stock trading application will be used as an illustrative example. It shall not, of course, describe a perfect architecture for stock trading. The example illustrates most of the aforementioned problems and helps underline the requirements of engineering event-based systems and the features of the scope model presented in the next chapter.

The following components of a stock market can be identified (see Fig. 2.8):

[7]repeated, possibly bidirectional communication

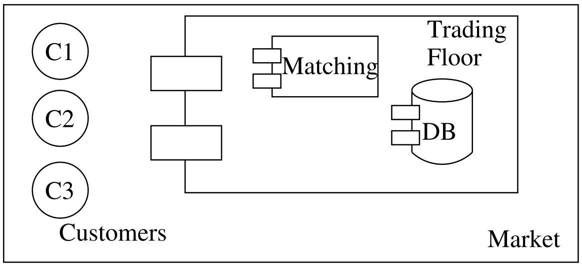


Figure 2.8: An example stock trading application

- Customers monitor quotations and issue orders to buy or sell shares.
- A central matching engine implements the matching algorithm and generates quotations.
- A database logs the generated data to ensure consistency and persistence, and to audit the operation.

Nearly all parts of a stock trading application are inherently event based. The dissemination of stock quotations from the central trading floor (or its computerized equivalent) to the market participants is an accepted and plausible example of applying event notification services. Database and matching engine are composed into the virtual trading floor, a component which consumes orders and publishes notifications carrying share prices of successfully executed trades.

Bundling of Related Components

Locality, encapsulation, and the composition of existing components into higher-level units are well-known concepts for mastering complexity and for supporting evolution [240]. These concepts are used in request/reply systems, but they are equally important here. The grouping of components that share some commonality or achieve a common goal is a prerequisite for reasoning about effects and side effects, and it is the basis for addressing both engineering and management issues.

Bundling is both a syntactical and a semantical abstraction. From the syntactic point of view such a bundle limits the distribution of notifications produced within; it identifies notification delivery localities. The bundling mechanism should be orthogonal to any subscription mechanism so that grouping is independent from component implementation and it should not influence the subscriptions issued by them. This is important to draw locality not only based on the described interests of consumers but also on other criteria, such as organizational and geographical constraints of a company or some other application-specific semantics.

From the semantical point of view, bundles of components must be components themselves with own semantics. The bundles should not only delimit visibility, but also publish notifications themselves as result of notifications produced within the bundle, indicating important state changes of the bundle as a whole. Similarly, they should consume notifications from the outside by further propagating them to their internal locality. This opens the possibility to recursively bundle component compositions into higher-level components and to hierarchically structure an event-based system.

Consider the running example. The virtual trading floor in the stock trading application is the first candidate of a component bundle. One can imagine a 'verbose' matching engine producing detailed notifications about the progress of the matching algorithm, of which the majority is only relevant for logging purposes (e.g., for auditing system operations) and only a few are relevant for customers. Hence, it makes sense to constrain the visibility of most of the notifications to the DB component and to allow only a few of them to pass the boundary of the trading floor bundle.

The next reasonable structuring step would be to bundle the trading floor and a set of customers (i.e., the participants in the market described in Fig. 2.8) into a higher-level syntactical and semantical market component. In this way multiple trading floors could be supported without having customers receive duplicate and inconsistent notifications. Such duplication cannot be avoided in a flat design space where all components in the system are visible to each other. The absence of market bundles would require to encode knowledge about the market structure into the subscriptions of individual components, which impedes reuse and system evolution (cf. Section 2.1.3).

Supporting Sessions and Activities

The engineering of complex systems benefits not only from bundling related components according to application structure but also from grouping notifications into sessions. Be it because notifications originate from the same source or because they belong to a set of cooperating components, sometimes it is necessary to distinguish sessions of dependent interactions to identify conversational state. This is especially important in event-based systems, where the identity of peers is unknown and communication is a priori stateless. That is, without any additional information consecutive notifications cannot be related to each other. The pub/sub paradigm does not offer any intrinsic means to identify conversational state other than introducing IDs manually.

An example for sessions is a stockbroker who listens to a specific share traded on two stock markets. Obviously, notifications distributed in one market must, generally, be invisible in the other. However, the stockbroker should be able to observe and distinguish both. In abstract terms, individual components should be able to participate in multiple sessions to delimit them from each other and to support session state. However, taking up the discussion about IDs in Section 2.5.1, it is generally undesired to have components do session handling on their own. From an engineering point of view, it complicates their implementation.[8] But more importantly, it reduces the loose coupling of publish/subscribe by explicitly tangling notifications and interaction control. Using IDs is an ad-hoc approach to distinguish groups of producers, whereas the underlying problem was not yet analyzed closely.

Furthermore, activities made up of bundles of notifications can be modeled as well-defined structures as described for bundles of application components above. Activities structure the interaction in the system and in themselves are components with well-defined semantics. Drawing on localities of distribution, they may help determine when 'internal' notifications are to be made visible to the outside. This will help to prevent side-effects, to build structured, hierarchical sessions, and to customize and orchestrate them. An analogy to the activity concept in the world of request/reply-based systems would be a simplified version of the notion of transactions [145].

Mastering Heterogeneity

A single uniform event notification service with uniform syntax and semantics is hardly able to cope with the diverging requirements of large distributed systems, which typically operate in heterogenous environments [74]. As pointed out in the examples of Sect. 2.5.1, an event service that, e.g., relies on a global naming scheme is not scalable and complicates system integration. Furthermore, syntax and semantics of notifications are likely to vary and there are inevitably different data models in use, which can be induced by hardware-dependent issues (like bounded message size) or by middleware- or application-layer differences. While heterogeneity is a well known problem in other areas of computer science, it only recently starts gaining attention in the context of notification services [74, 147].

From the observations above an apparent conclusion is that bundling of related components should not only encapsulate functionality but also delimit common syntax and semantics. This requires mechanisms to support adapting data that cross boundaries of component bundles by mapping content and representation. To motivate the requirement consider again the running example. For efficiency reasons it is reasonable to distinguish between low-volume external representations in XML versus more optimized internal representations. The matching and database components may use a binary representation while stock quotations are published using an established financial markup language like FIXML [223] to facilitate interoperability. Hence, transformation between the external XML representation and the internal binary representation would be needed for notifications crossing the border of a trading floor composite.

[8]Enterprise Java Beans introduce session beans as a remedy to this problem in the request/reply approach.

Flexible Configuration and Customization

Similar to the diverse requirements regarding data representation in heterogeneous environments, a static definition of notification transmission semantics is not adequate either. The service must be adaptable, it must be configured to meet applications needs. As pointed out in Sect. 2.5.1, subsets of closely interacting participants often rely on communication guarantees different from those of basic notification dissemination. This includes ordering or real-time guarantees that refine the specification of simple event-based system given in Section 2.3. But also application-specific needs may demand to deviate from this basic specification. For example, instead of the default 'broadcast' of notifications to all eligible consumers with matching subscriptions, only a specific subset of them may be selected due to an application-specific policy. An 1-of-n policy realizes load balancing within a bundle of components, and outside of the components themselves.

In the stock application, the matching engine might be replicated to distribute processing load over multiple instances using a delivery policy that routes orders to instances dedicated to the respective share. Furthermore, if the structure of the bundles is not static, security policies must control who is allowed to join. The trading floor component could be compromised if everyone is allowed to join and issue notifications influencing the matching engine. On a lower level of adaptation the implementation of the trading floor will use broadcast mechanisms of a local area network, whereas the dissemination of price information on the Internet has to use other techniques.

In general, the ability to adapt and program bundles of components tackles the design, implementation and engineering problems stated above. The whole event service is subject to customization with respect to these bundles: API, syntax and semantics of subscriptions and notifications, security policies, and implementation techniques of notification dissemination must be tailored to fit the needs of evolving complex systems.

2.5.3 Existing Support

The bundling of components is the basic requirement presented in the previous paragraphs, and it complies with the fact that information hiding and abstraction have long been identified as a fundamental principle in software engineering [240]. In request/reply-based distributed systems, like the CORBA platform [224], solutions exists for all of the outlined requirements. Object-oriented programming and decomposition, heterogeneity by standardized interconnection protocols (e.g., CORBA-IIOP, SOAP [299] based on XML), bundling of activities with the help of transactions [35, 228], and security services, e.g., Kerberos [222], provide the appropriate support.

However, comparable hierarchical structuring mechanisms are missing in event-based systems. The missing knowledge about communicating peers leads to the desired separation of communication from computations. But issues of component interaction are thus drawn out of the application components themselves, and any adequate support for the mentioned requirements must respect and facilitate the external control of interaction. Unfortunately, existing services recognize and address these issues only partially.

A first approach to achieve these goals would be to build on existing features of notification services. For example, one could make use of content-based filtering mechanisms [62, 212] to decompose and delimit sets of components and notifications from each other. Subscriptions of individual components have to be adapted to encode additional constraints on the decomposed structure. This approach of modifying application components counteracts the stated separation. Knowledge about the application structure is put into the components, contradicting the idea of components being loosely coupled and self-focused. Furthermore, the structure is not explicitly enforced by the system and all components are eligible receivers if they have subscribed accordingly: compromised filters may evade security measures and reflection, i.e., investigation and change [195], is restricted. Subject-based addressing is too limited to implement any sensible structuring in addition to existing subscriptions, because different points of view are not supported. Event channels like in the CORBA Notification Service combine structuring and notification selection to some extent. However, individual components still have to select channels manually.

The next chapter introduces a different approach and introduces a scoping concept that addresses the underlying problem of controlling notification visibility and serves as a tool of both application design and event system implementation.

2.6 Discussion

This chapter has presented a thorough description of event-based systems. First, an abstract definition of interaction models is presented to clarify the fundamental differences between request/reply and event-based interaction. Although event notification is often seen as a communication paradigm, it is shown that the use of events is mainly a characteristic of the software and its inherent structure than of the communication techniques applied to transport notifications. Together with a description of the constituents of event-based systems, several ambiguously used terms have been clarified to show the differences between events, notifications, and messages, and between event-based communication, publish/subscribe, and messaging. This is partly based on work first published in Mühl et al. [215] and Fiege et al. [114, 115] and has been refined here. It facilitates the discussion about visibility and its implementation in the next chapters.

Furthermore, a formal specification of simple event-based systems is given, using temporal logic to define the semantics of interface operations of a black box notification service. The specification is complemented with a ‘correct’ if naïve implementation to illustrate its feasibility. The specification serves as a basis for reasoning about notification dissemination semantics and it is extended in the next chapter to formally specify the semantics of scoped event-based systems. This chapter has also presented an overview of Rebeca [112], a distributed notification service whose basic architecture and function is comparable to other services. It provides the system model used and extended in this thesis.

The concluding discussion in the previous section has investigated the deficiencies of event-based communication. Currently, missing control of side effects as well as the missing support for design, implementation, and management of event-based systems limits their applicability. A number of engineering requirements are postulated, which, when met, considerably extend the applicability of event-based systems. The analysis suggests that most of the issues are centered around the visibility of notifications.

Chapter 3

Scopes

Contents

3.1 Visibility **42**
3.2 Event-Based Systems with Scopes **44**
3.3 Component Interfaces **56**
3.4 Notification Mappings **60**
3.5 Transmission Policies **68**
3.6 Scoping of Notifications **73**
3.7 Security **76**
3.8 Engineering with Scopes **76**
3.9 Discussion **90**

This chapter introduces scopes for event-based systems. The first section analyzes the notion of visibility in event-based systems and relates it to the requirements defined in Section 2.5.2. The scoping concept is defined in Section 3.2, including a formal specification of scoped event-based systems that refines the specification of simple systems given in the previous chapter. Scopes re-introduce control on communication, which was drawn out of the components in event-based interaction, without impairing the benefits of loose coupling. The concept is extended in sections 3.3 and 3.4 to include interfaces and mappings; the former further refines visibility control, the latter generalizes interfaces to transform notifications at scope boundaries, coping with heterogeneous data models. While communication within scopes is by default like in traditional pub/sub systems, the transmission policies presented in Section 3.5 adapt the semantics of notification dissemination within scopes. Session scopes (Section 3.6) introduce dynamic scopes. They apply ideas of transaction processing to provide for modeling activities in event-based systems. Security and publish/subscribe was a problem before, because of the anonymity and loose coupling of interaction. The reified structure of scoped systems now enables the integration of security measures without impeding the event-based style (Section 3.7). Finally, scopes are a module concept and they promise to open new ways of engineering event-based systems. A development process and an SQL-like scope language is presented in Section 3.8. The chapter concludes with a summary and a discussion of the contributions (Section 3.9).

3.1 Visibility

The visibility of transmitted data is of little concern in request/reply systems where destinations are explicitly addressed. In event-based systems, however, the visibility of notifications complements subscription techniques for it determines which subscriptions have to be evaluated at all. Surprisingly, visibility was rarely considered so far.

3.1.1 Implicit Coordination and Visibility

The problems of current event-based systems, which are described in the previous chapter, stem from the loss of control of interaction. This control has been relinquished deliberately in favor of the loose coupling. It is withdrawn from the components, replacing explicit addressing with the matching of notifications to subscriptions. The explicit control of interaction given in request/reply approaches is replaced by the implicit interaction in event-based systems.

The implicit interaction is characterized by an indirection of communication. Producers make notifications available and consumers select with the help of subscriptions. This indirection gives room for a concept complementary to the notification selection done by consumers. The *visibility* of a notification limits the set of consumers that may pick this notification. If a notification is not visible to a consumer, its subscriptions need not to be tested at all. Notifications and subscriptions are unaltered, and matching takes place as before but under the constraints of visibility limitations. Clearly, visibility influences the interaction of components, it can even be seen as a means to govern implicit coordination.

The implicit coordination[1] of the components offers the desired loose coupling but makes the overall functionality an *implicit* result of *all* the participating components. However, extracting control from application components must not necessarily mean to have it nowhere. In fact, the requirements posed in Section 2.5.2 demand some form of control on event-based communication. Visibility may offer such a control of notification dissemination.

The implications are twofold. First, visibility is an important factor of implicit coordination, and second, it promises to be an important abstraction in event-based systems. While subscriptions are related to the function of individual consumers, visibility governs the interaction in the system. Hence, the

[1]Explicit and implicit coordination are also termed objective and subjective coordination in coordination theory [257].

visibility of notifications is essential for the overall function of an event-based system.

3.1.2 Explicit Control of Visibility

The key to exploiting visibility is to regard it as first class citizen. While existing work has addressed some facets of visibility, it was never taken as a fundamental concept in event-based systems. Nevertheless, it will prove to be the basis for both controlling and extending dissemination functionality.

Explicit visibility control constrains the areas where loose coupling and implicit coordination is applied. It makes bundles of implicitly interacting components explicit, and these bundles reify the structure of applications. They serve as a tool for designing and programming event-based systems, because once the interaction is localized at well-defined points additional mechanisms can be applied to control the interaction within and between definite parts of the system.[2]

But how is visibility actually represented in an event-based system? Where is it exposed? Any form of re-integrating control into the components counteracts the event-based paradigm. Whenever notifications are annotated to reach a specific set of consumers, external dependencies are encoded in application components, which defeats the benefits of the event paradigm. Visibility of notifications is not a matter of producers for it concerns interaction and communication, but not the computation within the component. Thus, the necessary control must be exerted outside of the components themselves.

3.1.3 The Role of Administrators

When designing and engineering event-based systems, only the *roles* of producers and of consumers were considered so far. They represent the tasks of designing and programming individual application components. The self-focus of event-based components is mirrored in these roles. They concentrate on internal computation alone and disregard interaction. Due to the implicit coordination, responsibility for the overall functionality is not assigned to any specific role. It is delegated to producers and consumers, but with no adequate support. The preceding discussion corroborates that an additional role in the system to handle visibility is needed.

The obvious implication is to introduce the role of an *administrator* which is responsible for orchestrating components in an event-based system. An administrator may be human, but it can also be comprised of programs and rules that maintain some system properties (cf. autonomnic computing).

The main objective of this role is to support component assembling and the management of their interrelationships. This role is employed to associate visibility control with a distinguished role different from producers and consumers. It is similar to those identified in component-based development or in reference

[2]Technically, this is the essence of the scope concept presented in the following.

architectures of open systems [162]. In terms of coordination theory, administrators are a means of objective coordination providing an exogenous extension of event-based interaction [34], which separates the shaping of interaction from, and generally makes it invisible to, the computation in the base entities.

Effective means to control visibility in event-based systems are necessary to support the administrator's role, and with respect to the requirements given in Section 2.5.2, such a control is a prerequisite to solve the underlying problems of current event systems. The demanded bundling of related components is directly addressed by the visibility of notifications. Heterogeneity issues can only be solved if communication is intercepted and converted, which requires a limited visibility in the first place. The same holds for the customization and configuration of the event service itself. With limited visibility the interaction within certain system parts may receive a dedicated service tailored to its needs, whereas interaction with the outside is handled differently, like the case of heterogeneous data models. Similar arguments hold for supporting sessions of interrelated notifications, the details of which are postponed to Section 3.6.

Unfortunately, current work disregards this important role and does not provide any appropriate support. The scoping concept presented in the next section, however, describes visibility in event-based systems and offers the explicit control needed by administrators.

3.2 Event-Based Systems with Scopes

This section formally introduces the notion of scoping in event-based systems. It extends the specification of the simple event system presented in Section 2.3.2 and is the basis for further extensions and reasoning about scoping functionality.

3.2.1 Visibility and Scopes

The notion of scoping in event-based systems is introduced to realize the visibility of notifications. A *scope* bundles a set of producers and consumers and limits the visibility of notifications to the enclosed components. The event-based style of matching notifications and subscriptions is still used within the scope, whereas the interaction of this bundle with the outside is no longer implicit, it is prohibited at first. The notion of scopes serves two purposes. The term is used to describe the visibility of notifications and to name the entity that defines visibility.[3]

Scopes have interfaces to regulate the exchange of notification with the remaining system. Scopes forward external notifications to its members and republish internal ones to the outside if they match the output and input interfaces of the scope. In addition, scopes can recursively be members of higher level scopes and in this way offer a powerful structuring mechanism. Scopes thus act as

[3]In fact, in most cases this thesis refers to the entity, which implies the scope of notifications in the former meaning.

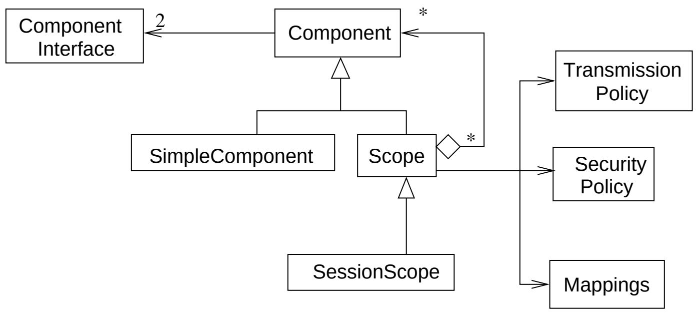


Figure 3.1: A meta model of scopes

components in an event-based system. They publish and consume notifications and can be deemed equivalent to the simple base components considered so far. So, the system consists of simple components and of complex components that bundle other simple or complex components.

The concept of scopes as illustrated in Figure 3.1 includes further features that will be described in the course of this chapter. *Transmission policies* can be applied between scopes and within a scope to adapt notification forwarding, allowing for tailoring notification delivery semantics to application needs in a restricted part of the system. Furthermore, *event mappings* at scope boundaries generalize scope interfaces and are capable of transforming between different data models of notifications. Security policies are a straightforward way to control the access to the scoping structure. Session scopes are included in this figure as example of other, special types of scopes that use specific combinations of the other scope features; they are described in Section 3.6.

3.2.2 Specification

The notion of components is extended to distinguish simple and complex components. The set of all simple components $\mathcal{C}$ includes any possible software entity that accesses the notification service API. The set of all complex components $\mathcal{S}$ describes all possible scopes. The set of all components $\mathcal{K}$ is defined to be the union of the disjoint sets of simple components $\mathcal{C}$ and complex components $\mathcal{S}$, $\mathcal{K} = \mathcal{C} \uplus \mathcal{S}$.

A scope bundles a set of components, and a component can be a member of multiple scopes. To denote the relationship between components and scopes, a graph of scopes is defined.

Definition 3.1 (scope graph) *Let $\mathcal{K} = \mathcal{C} \uplus \mathcal{S}$ be the set of all simple and complex components. A* scope graph *is an acyclic directed graph $\mathsf{G} = (\mathsf{C}, \mathsf{E})$. The graph consist of a set of components $\mathsf{C} \subseteq \mathcal{K}$ as nodes and a relation $\mathsf{E} \subset \mathcal{K} \times \mathcal{K}$ as edges between the nodes so that $(C_1, C_2) \in \mathsf{E} \Rightarrow C_2 \in \mathcal{S}$.*

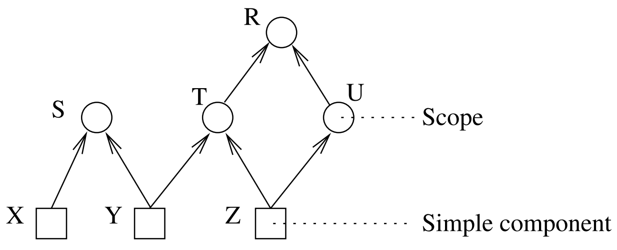


Figure 3.2: An exemplary scope graph

A scope graph denotes the scope-component relationship. An edge (C, S) from node C to node S indicates that C is a component of *scope* S.[4] The stated property $(C_1, C_2) \in \mathsf{E} \Rightarrow C_2 \in \mathcal{S}$ ensures that a simple component cannot be a superscope of any node in G. C is a subscope if $C \in \mathcal{S}$. Conversely, the scope of a component C is any S such that $(C, S) \in \mathsf{E}$. S is also called superscope of C to emphasize the relationship between S and C, e.g. in cases where C is a scope itself. In Figure 3.2, X is a component of S, Y is a component of both S and T, and T is a component/subscope of R and superscope of Y and Z.

The edges of the scope graph describe a partial order $\leq$ on C, where $C_1 \leq C_2$ iff $(C_1, C_2) \in \mathsf{E} \vee C_1 = C_2$. Avoiding the reflexivity of $\leq$, the scope-component relation is described by $\lhd$, where $C_1 \lhd C_2 \Leftrightarrow (C_1, C_2) \in \mathsf{E}$. The transitive closure of $\lhd$ is denoted by $\overset{\star}{\lhd}$; $\rhd$ and $\overset{\star}{\rhd}$are defined accordingly. In the example of Figure 3.2, $Y \lhd T$ and $Y \overset{\star}{\lhd} R$ hold. According to the partial order, the simple components are the minimal elements and those scopes having no superscopes are the maximal elements of C. Additionally, the following terms are borrowed from graph theory. T is a *parent* of Y, and Y is a child of T. Y is a sibling of Z, and vice versa, i.e., they have the same parent.

Based on these definitions, visibility can be defined formally. In the first instance, the visibility of components is defined, which implies a visibility of notifications.[5] Informally, component X is visible to Y iff X and Y "share" a common superscope.

Definition 3.2 (visibility of components) *The* visibility of components *is a reflexive, symmetric relation v over $\mathcal{K}$, also written as $v(X, Y)$, and is recursively defined as:*

[4]Edges could have been defined in the inverse direction to emphasize that components do not need to know their scopes and how they are aggregated. However, the presented notation follows the one originally published in Fiege et al. [114].

[5]The more general visibility of individual notifications is discussed in sections 3.3 and 3.6 together with scope interfaces and session scopes.

$$\begin{aligned} v(X,Y) \Leftrightarrow\ & X = Y \\ & \vee\, v(Y,X) \\ & \vee\, v(X',Y) \ \textit{with}\ X' \rhd X \\ \Leftrightarrow\ & \exists Z.\, X \overset{\star}{\lhd} Z \ \wedge\ Y \overset{\star}{\lhd} Z \end{aligned}$$

In the graph of Figure 3.2, for example, $v(X,Y)$ and $v(Y,U)$ hold, but not $v(X,Z)$.

Using this visibility, the specification of simple event-based systems given in Definition 2.5 of Section 2.3 can be refined. For presentation purposes, the specification is at first restricted to static scopes, i.e., the scope hierarchy and membership cannot change once the first notification has been published. This restriction is relaxed later.

Definition 3.3 (scoped event system) *A* scoped event system $ES^{\mathcal{S}}$ *is a system that exhibits only traces satisfying the following requirements:*

- *(Safety)*

$$\Box\Big[\textit{notify}(Y,n) \ \Rightarrow\ \big[\bigcirc\Box\neg\textit{notify}(Y,n)\big] \\ \wedge\ \big[\exists X.\, n \in P_X \ \wedge\ v(X,Y)\big] \\ \wedge\ \big[\exists F \in S_Y.\, n \in N(F)\big]\Big]$$

- *(Liveness)*

$$\Box\Big[\textit{sub}(Y,F) \Rightarrow \\ \Big(\Diamond\big[\Box v(X,Y) \ \Rightarrow\ \Box\big(\textit{pub}(X,n) \wedge n \in N(F) \ \Rightarrow\ \Diamond\textit{notify}(Y,n)\big)\big]\Big) \\ \vee\ \Big(\Diamond\textit{unsub}(Y,F)\Big)\Big]$$

Definition 3.3 differs only slightly from Definition 2.5 on page 24. The safety requirement contains an additional conjunct $v(X,Y)$. This means that in addition to the previous conditions, the producer and the subscriber must also be visible to each other when a notification is delivered. The liveness requirement has an additional precondition $\Box v(X,Y)$ that can be understood in the following way: If component Y subscribes to F, then there is a future point in the trace such that if X remains visible to Y every publishing of a matching notification will lead to its delivery at Y. The *always* operator requires the scope graph to be static.

Note that Definition 3.3 is a generalization of Definition 2.5. A simple event system can be viewed as a system in which all components belong to the same "global" scope. This implies a "global visibility," i.e., $v(X,Y)$ holds for all pairs of components (X,Y) and can be replaced by the logical value *true* in the formulas of Definition 3.3, resulting in Definition 2.5.

3.2.3 Notification Dissemination

According to the previous definition a published notification is delivered to all visible consumers that have a matching subscription. In order to clarify the impact of the scoping structure and the dissemination of notifications through the scope graph, the visibility of notifications is analyzed in the following.

The visibility of a notification n to a component C determines C's ability to deliver this notification at all, and is denoted by $\overset{n}{\leadsto} C$. Visibility is a test that precedes any subscription matching. Subscriptions decide in a second step whether to deliver a visible notification or not. The visibility of notifications in the scope graph is directly related to the visibility of components, of course. The visibility of a notification n, which is published by X, to a specific component Y is denoted by $X \overset{n}{\leadsto} Y$, where

$$pub(X, n) \wedge v(X, Y) \Rightarrow X \overset{n}{\leadsto} Y.$$

A published notification is made visible in the scopes the producer belongs to. $Y \overset{n_1}{\leadsto} S$ in Figure 3.3(a), or simply $\overset{n_1}{\leadsto} S$ to denote the visibility alone if the specific producer is not important. This rule is applied recursively to make notifications visible in all further superscopes; $Y \overset{n_1}{\leadsto} T$ and $Y \overset{n_1}{\leadsto} T'$. On the other hand, if a notification is visible within a scope S, $\overset{n}{\leadsto} S$, it is visible to all its children. Recursively applying this rule yields in Figure 3.3(b) $X \overset{n}{\leadsto} T \Rightarrow X \overset{n}{\leadsto} S \Rightarrow X \overset{n}{\leadsto} Y$. Note that edge direction indicates scope membership but notifications can travel in both directions. In summary, notification dissemination is governed by two rules, a publishing policy PP and a delivery policy DP:

$$\textbf{PP}: \quad X \overset{n}{\leadsto} S \wedge X \lhd S \lhd T \Rightarrow X \overset{n}{\leadsto} T \tag{3.1}$$

$$\textbf{DP}: \quad \overset{n}{\leadsto} T \wedge S \lhd T \Rightarrow \overset{n}{\leadsto} S \tag{3.2}$$

Consider Figure 3.3. A notification n_1 published by Y is forwarded to S and to all children of S, and from S to T and T' and to all of their children, i.e., to all siblings of S. n_1 is an internal notification of S, T, and T', which means it is visible to their children. $X \overset{n_2}{\leadsto} S$ is at first an external notification to S and is made internal by the delivery policy of Eq. 3.2. A notification forwarded in the direction of an edge, e.g., $(S, T) \in E$, is an *outgoing notification* with respect to S; it leaves the scope of S. Conversely, a notification that travels against an edge is an *incoming notification*, e.g., from T to X in Fig. 3.3(a) or from T to S in Fig. 3.3(b); in the latter case n_2 is external to S.

The semantics of notification dissemination is that incoming notifications are forwarded to all children of a scope, and outgoing notifications are forwarded to superscopes and to all siblings. Note that incoming notifications are not forwarded to superscopes; n_2 is not visible to T' in Figure 3.3 as X is not visible to T'. This default transmission of notification dissemination is the consistent extension of the semantics of simple event systems. The intuitive meaning of scope membership corresponds to this definition. That is, i) siblings are eligible consumers as they are in the same scope, ii) being a subscope also denotes

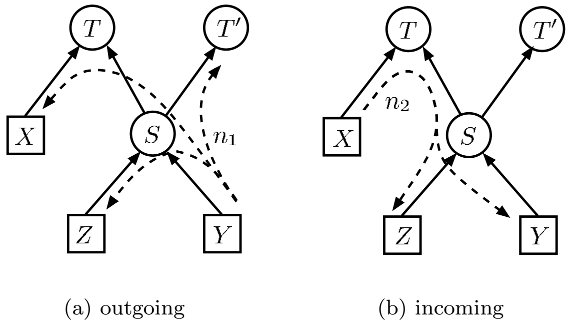


Figure 3.3: Outgoing and incoming notifications

a part-of relationship, which makes it obvious that internal notifications are also forwarded to superscopes, and iii) external notifications are made visible to members of complex components.

Visibility is a set inclusion test so far, which disregards the way a notification becomes visible. In practice, however, the paths of dissemination in the scope graph are of great importance for any analysis of system behavior.

Definition 3.4 *A* delivery path *p between two components X and Y is a sequence of components $p = (C_i) = (X, C_2, \ldots, C_{n-1}, Y)$ for which holds:*

1. *p is an undirected path in the graph of scopes,*
2. *p obeys the visibility v in that $v(C_i, C_j)$ holds for all $1 \leq i < j \leq n$.*

Delivery paths are not directed, which means that either $(C_i, C_{i+1}) \in \mathsf{E}$ or $(C_{i+1}, C_i) \in \mathsf{E}$. The dissemination in the scope graph is described by the following

Lemma 3.1 *Every delivery path $p = (C_1, \ldots, C_n)$ can be subdivided into two, possibly empty, parts: an upward path $(C_1, \ldots, C_j)$ where $(C_i, C_{i+1})_{i<j} \in E$, i.e., $C_i \lhd C_{i+1}$, and a downward path $(C_j, \ldots, C_n)$ where $(C_{i+1}, C_i)_{i \geq j} \in E$.*

Proof: Show that p turns at most once. A delivery path $p = (C_1, \ldots, C_n)$ connects two components C_1 and C_n that are visible, $v(C_1, C_n)$. If $C_1 \overset{\star}{\lhd} C_n$, the downward path is empty and C_n is reached by forwarding notifications to superscopes according to equation 3.1. If $C_1 \overset{\star}{\rhd} C_n$, the upward path is empty and C_n is reached by propagating visible notifications to children according to equation 3.2. Otherwise, the path turns at least once and two cases can be distinguished: p starts with an upward or a downward edge.
Assume p starts with a downward edge, $C_1 \rhd C_2$. Select d such that $1 \leq d \leq n$ and $C_i \rhd C_{i+1}$ for all $i \leq d$. If $d \neq n$, the downward path is $(C_1, \ldots, C_d)$ and $C_d \lhd C_{d+1}$. However, equation 3.1 allows this upward delivery only if the

notifications originated in C_d. This is not the case and by contradiction the downward path ends at $C_d = C_n$.
Assume p starts with an upward edge, $C_1 \lhd C_2$. In the same way p starts with an upward path of length $u \leq n$ such that $C_i \lhd C_{i+1}$ for all $i \leq u$. If $u \neq n$, $C_u \rhd C_{u+1}$. However, the path $p' = (C_u, \ldots, C_n)$ starts with a downward edge and from the preceding arguments follow that p' consists only of downward edges.
If p starts downwards, $C_1 \overset{\star}{\rhd} C_n$. If p starts upwards, either $C_1 \overset{\star}{\lhd} C_n$ or the path turns once downwards at a C_j, proving the lemma. □

3.2.4 Duplicate Notifications

Between any two nodes of the directed acyclic scope graph there may exist zero, one, or more different delivery paths—the scope graph is not a tree (Fig. 3.4). This may lead to duplicate notifications in certain implementations. The specification of scoped event systems does not consider delivery paths but demands notifications to be delivered at most once. So, concrete systems may violate the specification. However, there are two reasons for not eliminating duplicates in the scope model itself. First, duplicates generation and handling is highly implementation dependent. And second, in some applications delivery along different paths leads to different semantics of notifications so that they are not really duplicates.

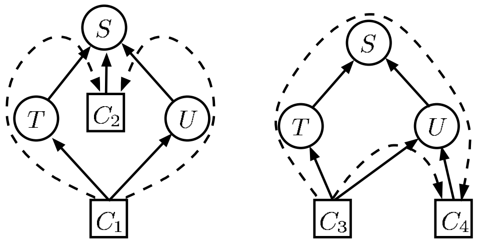


Figure 3.4: Two ways of generating duplicates

The utilized implementation of scoping determines whether the conceptual replication really results in duplicate deliveries. A broad range of possible implementations of scoping exist[6] and in some of them different delivery paths have no effect. For example, an explicit, externally available scope graph data structure can be used in a centralized implementation to infer all destinations before delivery is commenced. Furthermore, available countermeasures for duplicate detection are also highly dependent on the underlying implementation technique.

From an application point of view, there are several reasons for not eliminating duplicates in the scoped event system itself. First of all, in some applications

[6]Please refer to Section 4.1 for an overview.

notification processing is idempotent so that duplicate delivery does not influence the function of an application. On the other hand, if duplicates are not wanted, it is often easier to handle the elimination in the application layer, or at least as an additional layer on top of simple notification dissemination. In fact, the scope boundaries themselves offer a platform to install such logic.

The most interesting point, however, is that on application level different delivery paths may connote different notification semantics. Consider the left example of Fig. 3.4 where two different delivery paths connect C_1 and C_2, and assume that $C_1 \overset{n}{\rightsquigarrow} C_2$ results in two notifications n' and n'' being forwarded by T and U, respectively. Are the two notifications really equal? Are these notifications really duplicates if they originate, at least from the consumers point of view, from different components T and U? Within S, these two notifications were published from different producers in the first place. The base event notified with n' may have a different meaning in the context of T than the event notified with n'' in U. Scope interfaces and mappings presented in the next section will enable administrators to control notification forwarding in a finer way.

In summary, there is no generic solution to handle duplicate notifications in a scoped event-based system. The many available choices of possible implementation techniques offers all sorts of corresponding duplicate handling capabilities, which are too divergent to be included in the general scope model. Note that duplicate notifications are forbidden in the specification of simple event systems but are possible in scoped systems. Different delivery paths conceptually deliver different notifications, even if triggered by the same base event.

3.2.5 Dynamic Scopes

The above definition assumed a static scope hierarchy to provide a basic definition that can be adapted and refined based on further requirements. In the case of dynamic scopes, four additional operations have to be offered. $\mathit{cscope}(S)$ and $\mathit{dscope}(S)$ to create and destroy a scope S, $\mathit{jscope}(X, S)$ and $\mathit{lscope}(X, S)$ are used to join X to scope S or leave it, respectively. These operations are typically available to the administrator role only, for individual components do not necessarily need to know about their scope membership.

A system with static scopes can then be simulated by having the administrator set up the scope hierarchy with the appropriate operations before clients start to publish and subscribe. However, dynamic scopes are not directly covered by the above specification. A changing scope graph may conflict with the safety condition, which is ambiguous in dynamic asynchronous system models. A notification n is only allowed to be delivered to Y if the producer X is visible to Y. But because delivery cannot be instantaneous, X may leave the scope in which n was published before it is delivered, and so $v(X, Y)$ may hold at time of publication but not on delivery, rendering the specification ambiguous. The specification does not cover systems that allow traces of the form

$$\sigma_4 = \mathit{pub}(X, n), \ldots, \mathit{lscope(X,S)}, \ldots, \mathit{notify}(Y, n),$$

where scope graph reconfigurations and notification publication and delivery are mixed.

Several approaches to this problem exist. First of all, the assumed system model may require delivery to be instantaneous so that notification dissemination and scope reconfiguration cannot interleave. Any form of centralized implementation is able to achieve this guarantee. A second approach is to allow producers to leave a scope only if all their published notifications have been delivered, preventing the interleaving in σ_4 so that the resulting traces are equivalent to the static case with respect to the safety condition. In effect, this results in a type of synchronization similar to that of a global transaction: scope joins and scope leaves must be reliably acknowledged by all other brokers before the action is performed. Obviously, this type of dynamic scope semantics is unfavorable since it incurs a high synchronization overhead. However, scope reconfigurations may be so infrequent in practice that this is tolerable for medium size systems. At least these semantics have the advantage that the safety part of Definition 3.3 can be used in the simple unmodified form. Interestingly, this restriction resembles an object-oriented programming approach where new subclasses and new methods are readily added, but modifying the inheritance hierarchy is complicated.

A different approach would be to not hide scope graph changes but to explicitly consider them in the specification. For the safety condition the visibility restriction $v(X, Y)$ would have to reflect time delays in notification delivery. On the other hand, the liveness part of Definition 3.3 does not consider dynamic scopes at all. By including $\Box v(X, Y)$ in its precondition, only static graphs can fulfill liveness in the current definition. This specification is intentionally restricted because it is intended to specify only basic functionality. It currently covers a broad range of system models, and it can be refined (safety) and extended (liveness) to incorporate dynamic scopes in more specific system models. So, currently the following trace complies to the specification:

$$\begin{aligned}\sigma_5 = \mathit{sub}(Y, F), \mathit{jscope}(X, s), \mathit{jscope}(Y, s), \mathit{pub}(X, n_1), \mathit{lscope}(Y, s), \ldots,\\ \mathit{jscope}(Y, s), \mathit{pub}(X, n_i), \mathit{lscope}(Y, s), \ldots\end{aligned}$$

In σ_5 components X and Y start off in the same scope and X publishes an "infinite" sequence of notifications n_i. However, since Y leaves the scope again after every publish operation, there is no point in time from which on X and Y remain in the same scope. Therefore, delivery is not required and σ_5 satisfies the liveness requirement. Of course, without knowing future traces a notification service has to try to deliver any pending notifications.

So, dynamic changes of a scope graph can be supported if changes and publications are serialized, or the safety condition has to be relaxed to cover only durations in which the visibility of producer and consumer remain unchanged.

3.2.6 Attributes and Abstract Scopes

The layout of a scope graph carries information on system structure. Annotations of scopes allow the administrator to associate further information on system operation, which will be done in the next subsections. Or annotations are simply used to add application-specific data into the structure. Technically, the notion of *scope attributes* is introduced. Attributes associate data to a specific scope according to a simple name-value pair model.

For example, a scope S is named and stores its time of creation in two attributes:

$$S.name = \text{"ItsMe"} \qquad S.creation = \text{"2004-12-20 12:22"}$$

How attributes are set and used is described in Section 3.8.

Attributes may carry information about system configuration and management. Chapter 4, for example, introduces alternative implementation approaches, and attributes can store such annotations that refine the model expressed in the scope graph. However, these kinds of information are typically valid for more than one component of the graph. An obvious way to assign this information to a group of components is to use a scope, which bundles the components in question, just as a container carrying configuration data. This scope would be a special type of scope, termed abstract scope.

Abstract scopes group components, but there is no communication within. They are created for descriptive purposes and not to control communication of their members. They are used for system management (cf. Section 3.8).

3.2.7 A Correct Implementation

The following presents a possible implementation of Definition 3.3 as a proof of concept. The implementation uses a simple event system as specified in Sect. 2.3.2 as basic transport mechanism. This modular approach underlines the system's structure and shows the possibility of implementing the specification. But as before, it does not concentrate on efficiency issues, and any available notification service satisfying the simple event system specification can be used instead.

The architecture of the implementation is sketched in Figure 3.5. The interface operations of the scoped event system are local library calls which are mapped to appropriate messages of the underlying simple event system. Again, this part of the client process is the *local event broker* of the client. Conceptually, for every client an additional process at the interface of the simple event system is generated, the client's *proxy*. Practically, the proxy will be part of the local event broker. Note that the clients' proxies are the only components accessing the underlying simple service; no complex components are instantiated in this implementation scenario.

Although dynamic scoping is not considered in the specification, the presented algorithm includes dynamic scopes in the style of Section 3.2.5. To simplify the implementation, changes to the scope graph $\mathsf{G} = (\mathsf{C}, \mathsf{E})$ are restricted: only components with no incoming edges may join or leave scopes. This restriction prevents individual brokers from having to store G completely.

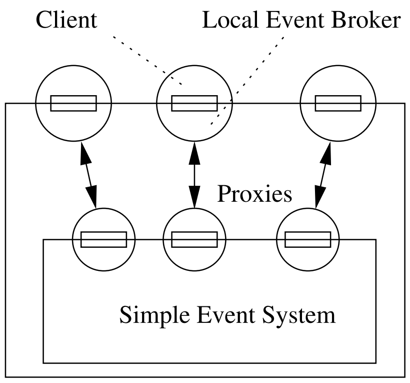


Figure 3.5: A possible implementation of a scoped event system

As noted above the scope graph describes a transitive partial order $\leq$ on C with $X \leq X' \Leftrightarrow (X, X') \in \mathsf{E}$. The maximal elements of C have no outgoing edges, i.e., they have no superscopes. These elements are termed *visibility roots*, as the recursive definition of $v(X, Y)$ is terminated by common superscopes. The maximal elements that are visible from a component are used to determine visibility of notifications.

Data Structures

For every client X, its proxy Prox_X holds a list V_X of its visibility roots. In a system with static scopes, V_X is initialized to the set of its visibility roots in the given scope graph. With dynamic scopes where changes are limited to the addition of new leaves—nodes with no incoming edges—V_X is set at the time of addition. In both cases, it remains constant and is not changed until the whole systems stops or X is deleted.

Algorithm

If a client invokes $\mathit{pub}(X, n)$, a message (pub, X, n) is sent to the client's proxy. At the interface of the simple event system, the proxy then invokes $\mathit{pub}(\mathit{Prox}_X, (n, R))$, where R is set to the constant value V_X.

Calls to $\mathit{sub}(X, F)$ and $\mathit{unsub}(X, F)$ are sent in a similar way to Prox_X. Using F, the proxy derives a filter $\tilde{F}$ that matches all notifications $\tilde{n} = (n, R)$ for which n matches F, and subsequently calls $\mathit{sub}(\mathit{Prox}_X, \tilde{F})$.

Whenever the simple event system notifies the proxy of Y about a notification $\tilde{n} = (n, R)$, the proxy checks whether $V_Y \cap R \neq \emptyset$. If the test succeeds, a message is sent to the local broker of Y to invoke $\mathit{notify}(Y, n)$. Otherwise the notification is discarded.

Correctness

In order to show that Definition 3.3 is satisfied, the presented implementation must obey the visibility $v(X, Y)$ of the safety condition and the additional pre-condition $\Box v(X, Y)$ of the liveness condition. The remaining part is satisfied by using the simple event system which satisfies Definition 2.5.

Lemma 3.2 *For every pair of clients X and Y and for the set of visibility roots V_X and V_Y stored at the proxies, the following holds:*

$$v(X, Y) \Leftrightarrow V_X \cap V_Y \neq \emptyset$$

PROOF: We need to show two implications. The first implication ($\Rightarrow$) is proved by induction over the "visibility" path from X to Y. The second implication ($\Leftarrow$) is shown as follows: If $V_X \cap V_Y \neq \emptyset$, there exists a maximal element Z of $\leq$ such that $X \leq Z$ and $Y \leq Z$. By the definition of $\leq$ this implies $v(X, Y)$. $\Box$

Now, the correctness of the sketched implementation can be proved in terms of the safety and liveness conditions of scoped event systems.

Proof of Safety. Assume that $\mathit{notify}(Y, n)$ is invoked at client Y. It must be shown that this implies validity of the three conjuncts of the implication in the safety property of Def. 3.3.

The first conjunct follows directly from the safety property of the simple event system.

To prove the second and the third conjunct, assume that the local broker issues $\mathit{notify}(Y, n)$ at client Y. This means that (a) the proxy of Y has previously received a notification $\tilde{n} = (n, R)$ and that (b) the test $V_Y \cap R \neq \emptyset$ succeeded.

From (a) and the safety property of the simple event system follows that $\tilde{n}$ was previously published by some proxy Prox_X. From Lemma 3.2 and (b) follows that $v(X, Y)$ holds. This proves the second conjunct.

From (a) and the safety property of the simple event system follows that $\tilde{n}$ matches some transformed filter $\tilde{F}$ of Prox_Y. This together with the algorithm proves the third conjunct. This concludes the proof of the safety property.

Proof of Liveness. Assume a client Y invokes $\mathit{sub}(Y, F)$ and never unsubscribes to F. From the algorithm it is implied that an "equivalent" subscription $\tilde{F}$ is issued into the simple event system. Since scope reconfigurations are restricted to occur at leaves, the values of V_X and V_Y of existent components are constant. From Lemma 3.2 this implies that $v(X, Y)$ is always true for all clients X and Y for which $V_X \cap V_Y \neq \emptyset$.

From the liveness property of the simple event system and the algorithm follows that there is a point in time after which every published notification $\tilde{n} = (n, R)$ that matches $\tilde{F}$ is delivered to every client proxy. So assume that after this point in time some client X publishes a notification n matching F. From the algorithm we have that $\tilde{n} = (n, V_X)$ is published within the simple event system. Its liveness property gives us that $\tilde{n}$ is eventually delivered at the client proxy of Y. From the algorithm and because $v(X, Y)$ holds, the test $V_X \cap V_Y \neq \emptyset$ will succeed and Y will eventually be notified of n.

3.3 Component Interfaces

So far, visibility is an only two-level hierarchy induced by the top-most superscopes, the visibility roots of the graph G. Any two components are either able to see all of their published notifications or non at all. In order to overcome this problem and to improve the structuring abilities, visibility is refined by assigning input and output interfaces to scopes.

Input and *output interfaces* for simple components are subscriptions and advertisements, respectively. Both include filters that describe the set of notifications allowed to cross a component's boundary. As defined on page 22, a notification n is either mapped on itself or to ϵ, indicating that n is either matched or blocked. In the following, similar filter sets are associated with scopes to make interfaces a feature of all components.[7]

3.3.1 Scope Interfaces

Scope input and output interfaces describe the set of notifications that are allowed to cross the scope boundary. Only those notifications that match one of the scope's output filters are forwarded up into its superscopes as outgoing notifications, and only those matching at least one of its input filters are treated as incoming notifications that are forwarded to scope members. Filters of scope interfaces are expressed in the same filter model used for subscriptions and advertisements of simple consumers and producers.

The base interface I_C of a component C contains two sets of filters, ${}^iF_C$ and ${}^oF_C$, representing the input and output interfaces of the currently active subscriptions and advertisements of the component. This base interface is associated with every component of the event-based system with the known function of letting notifications pass if they match one of the filters in ${}^iF_C$ for incoming notifications or ${}^oF_C$ for outgoing notifications.

Formally, the interfaces are bound to edges of the scope graph. Depending on the conceptual placement of filters with respect to the starting or ending node of an edge, two refinements and the resulting combination of filters are distinguished: selective, imposed, and effective interfaces (see Figure 3.6). While the next paragraphs discusses the different forms of interfaces, the formal definition of a scoped event system with interfaces is given in subsection 3.4.1.

[7]The relationship between scopes and simple components is shown in the UML class diagram in Fig. 3.1 on page 45.

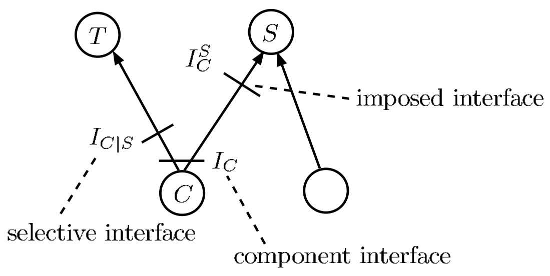


Figure 3.6: Different scope interfaces

Selective Interfaces

According to the preceding definition a component has an interface independent of its scopes; it does not distinguish between superscopes. This conforms to the intended loose coupling of event-based interaction. However, the administrator knows the configuration of scopes and as part of this role it is possible to distinguish superscopes.

A *selective interface* $I_{C|T}$ control the communication between a component C and a specific superscope T. It functions in the same way I_C does, but governs communication only between C and T. It is applied in addition to the base component interface. In Figure 3.6, for instance, some of the notifications published by C are forwarded to S but not to T. If, in a type-based scheme, $I_{C|T}$ contains an output filter that accepts notifications of type A but not B, and if C happens to publish notifications n_A and n_B of type A and B, n_A would be visible in T but n_B not. Communication with S is not affected by $I_{C|T}$.

So, notification forwarding depends on the destination scope. A component may now exhibit different interfaces towards different superscopes. From an engineering point of view, this offers a fine control of interaction, which is especially important when composing existing subsystems. Furthermore, the functionality of the selective interfaces may be used to mitigate problems of duplicate notifications by blocking certain delivery paths. On the other hand, the administrator must be aware of possible effects of discriminating interfaces. If the distinguished superscopes share a common visibility root two different delivery paths may exist that preclude duplicate notifications but break causal order of messages. Consider S and T in Fig. 3.6 having a common superscope Z, then a short path exists connecting C and T directly, and a longer one crossing S and Z to reach T. A first notification n_1, which is blocked by $I_{C|T}$, may reach T after a second notification n_2 that matches $I_{C|T}$. Although the specification of simple event systems does not assume a specific ordering, many concrete systems provide a sender FIFO ordering that would be broken in this way.

Imposed Interfaces

A converse refinement of interface definition is to install filters at the "other" end of the scope graph edge. An *imposed interface* I^S is specified within a scope and wraps all of its members with an extra interface. It allows only those notifications that match the imposed interface to be exchanged within this scope, dedicating the scope to a specific kind of data. This interface does not influence the communication of the affected component in other scopes. Furthermore, interfaces can also be imposed on individual components. I^S_C in Figure 3.6 restricts the interaction of C with S, without affecting the other children in S. If I^S_C contains an output filter that accepts notifications of type B but rejects A, the above mentioned notification n_B published by C would be forwarded into S, but n_A is rejected by the imposed interface. Note that notifications of type A may published by other members of S, which are not affected by I^S_C.

Imposed interfaces are a means to control communication within a scope. Especially when an administrator integrates existing preconfigured components, not all of their provided interfaces are of interest within the new scope, or on the other hand, not all of the scope's internal traffic shall be visible to all components. As such, imposed interfaces are a security mechanism, too. They enforce predefined filters on scope members and thus control what is published and consumed within the scope. For instance, depending on security credentials different interfaces may be imposed on newly connected scope members.

Effective Interfaces

The *effective interface* of a component concatenates the previously introduced base interface with the selective and imposed interfaces. It is given with respect to a specific outgoing edge of the component and describes the set of notifications that are effectively allowed to cross the respective edge of the scope graph. A notification matches the effective interface $\hat{I}^S_C$ of a component $C \lhd S$ iff it matches I_C *and* $I_{C|S}$ *and* I^S_C *and* I^S.

3.3.2 Event-Based Components

Scopes are a composition mechanism that facilitates creating new, more complex event-based components, showing essential characteristics of component frameworks in the flavor of Szyperski [285]. They encode the interactions between components and act themselves as components on a higher level of abstraction. The composed function is provided through a defined interface, thus facilitating the reuse of the bundle while abstracting from its internal configuration. Scopes are distributed event-based components (see Section 3.8).

3.3.3 Example

The example stock trading application introduced in Section 2.5.2 is expanded to illustrate the use of scopes (see Figure 3.7). There are two main scopes, `M1`

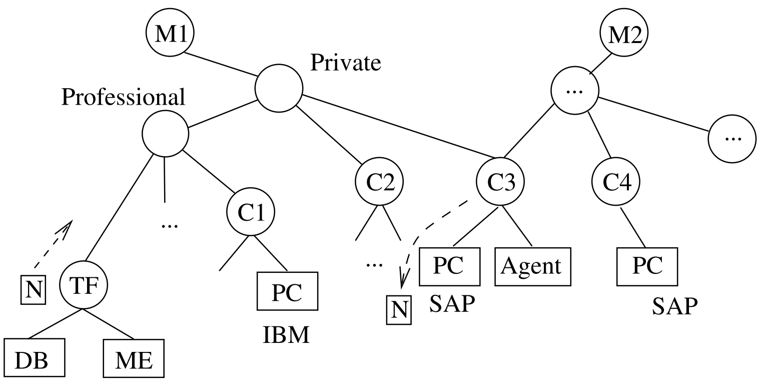


Figure 3.7: The graph of the stock application

and M2, denoting two different stock markets. Within each market customers are grouped into sub-scopes distinguishing private and professional customers. Each customer is permanently represented by one of the scopes C1, C2, etc., which remain connected in the graph of scopes even if customers are not personally logged in. They group a customer PCs, cellular phones, or agents running on a remote server. An example 'agent' would be a limit watcher which continuously monitors a share's price and issues a notification when a specific share deviates from the overall market performance. Such agents can be installed within a customer's scope without changing existing components—one of the obvious benefits of event-based systems—and without affecting other parts of the system, which is the prime attribute of scoping.

For the sake of simplicity, interest for at most one share is indicated below the rectangles representing the customers' PCs. The figure illustrates the scenario when the trading floor TF participates in the stock market M1 and issues a notification concerning SAP quotes. Although both consumers C3 and C4 have subscribed for notifications on SAP quotes, this notification will only reach C3, because C4 is not visible from the trading floor and C1 has subscribed to a different share. On the other hand, consumer C3 listens to both markets and may receive 'duplicate' SAP quotes (the implied problems are discussed in sections 3.2.4 and 3.6).

To illustrate how scope interfaces help in structuring event-based applications, let us consider the interfaces of the components in our running example as summarized in Fig. 3.8.

Customers send out notifications of type *Order* which contain a share identification, the number to be sold or bought, and potential price limits. The trading floor TF listens to these orders, issues acceptance notifications, and sends out *Quotes*, informing about successfully executed orders. The trading floor itself is

[8]Delayed forwarding is discussed in Sec. 3.5.

Component	Description	Input	Output
`M1,M2`	The Stock Markets	–	–
`Private`	scope of all private customers	–	Trade
`Prof.`	scope of all professionals	Order	Accept, Quote(delayed)[8]
`C1,C2,...`	Customer representation	Accept	Order
`TF`	Trading Floor	Order	Accept, Quote
`ME`	Matching engine	Order	Accept, Quote, OrderBook
`DB`	The logging database	Order, Quote	

Figure 3.8: Interfaces of the components in the example application

composed of the matching engine `ME` and the database `DB`. While the database only logs all *Orders* and *Quotes*, the matching engine receives orders and issues *Quotes* of current prices. It maintains a list of open orders and executes the matching algorithm that leads to acceptance notifications (*Accept*) of matched orders. Additionally, the matching engine publishes an orderbook summary with prices and volumes of the 10 best bid and ask orders. The summary is only visible within the trading floor, because the interface of `TF` prohibits further distribution. Based on this data, additional services may be integrated into the trading floor, like market makers ensuring that there is always at least one buy and one sell order open.

3.4 Notification Mappings

So far, uniform data and filter models were assumed, which prescribe syntax and semantics of notifications and filters throughout the whole system. In large systems, however, characteristics and demands of applications are likely to diverge and homogeneous models will not fit the needs, as pointed out in the discussion of the engineering requirements in Section 2.5.2. If all components are forced to agree on the same data and filter model, system integration and efficiency is impeded drastically.

The diverging requirements will best be met with tailored data and filter models—an idea which is obvious but hardly considered in the context of event systems. Different system parts will use different representations and semantics of events. With an appropriate support, one part of an application can exchange binary encoded notification while still being able to communicate with other parts of the system via serialized Java objects or XML encoded notifications. Efficiency considerations result in differentiating low-volume external representations in XML from more efficient, optimized internal representations.

An obvious implication of decomposing applications is that bundling of related components should not only encapsulate functionality but also delimit common syntax and semantics. Constraining the visibility of notifications is the basis for dealing with heterogeneity issues. Consequently, *notification mappings* are introduced as extensions of scope interfaces. They transform notifications at scope boundaries to map between internal and external representations, without interfering with internal notifications.

Scopes are an appropriate place to localize such transformations because bundled components are likely to agree on a common data and filter model, whereas the interaction with the remaining system is decoupled by the scope boundary. Notification mappings clearly address the heterogeneity requirements stated in Section 2.5.2 and facilitate construction and maintenance of large event-based systems.

3.4.1 Specification

Notification mappings transform notification from one data model to another. Mappings, however, do not primarily block notifications but transform them. *Notification mappings* are defined as binary, asymmetric relations on the set $\mathcal{N}$ of notifications. They are associated with scope graph edges, like scope interfaces, and two mappings $\nearrow_e$ and $\searrow_e$ are attached to every edge $e = (C, S) \in \mathsf{E}$. Let n_1 and n_2 be two notifications. For any edge e and its associated relation $\nearrow_e$, the mapping $n_1 \nearrow_e n_2$ means that when "traveling" upwards along the edge (i.e., in direction of the superscope) n_1 is transformed into n_2. The relation $\searrow_e$ is defined analogously for the reverse direction. Note, in order to support heterogeneous data models the relations map between two sets of notifications used in C and S, respectively, i.e., $\nearrow_e \subset \mathcal{N}_C \times \mathcal{N}_S$, but it is implicitly assumed that $\mathcal{N}$ contains the different models for simplicity.

Now, the general visibility of notifications can be defined using these relations.

Definition 3.5 *The* visibility of notifications *in a scope graph* $\mathsf{G} = (\mathsf{C}, \mathsf{E})$ *is defined by the relation* $\rightsquigarrow$ *on* $\mathcal{N} \times \mathcal{K}$*, where*

$$(n_1, X) \rightsquigarrow (n_2, Y) \quad \textit{or shorter} \quad X \overset{n_1}{\rightsquigarrow}_{n_2} Y$$

means that n_1 *visible to* X *is also visible to* Y*:*

$$\begin{aligned}
(n_1, X) \rightsquigarrow (n_2, Y) \Leftrightarrow \\
&\big(X = Y \wedge n_1 = n_2\big) \\
&\vee \big(\exists e = (X, X') \in \mathsf{E}.\, \exists n' \neq \epsilon. \quad n_1 \nearrow_e n' \\
&\qquad\qquad \wedge \big[(n', X') \rightsquigarrow (n_2, Y)\big]\big) \\
&\vee \big(\exists e = (Y, Y') \in \mathsf{E}.\, \exists n' \neq \epsilon. \quad n' \searrow_e n_2 \\
&\qquad\qquad \wedge \big[(n_1, X) \rightsquigarrow (n', Y')\big]\big)
\end{aligned}$$

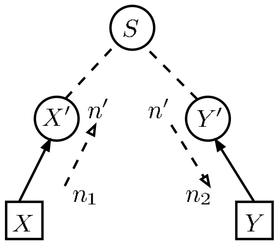


Figure 3.9: Recursive definition of the relation $(n_1, X) \rightsquigarrow (n_2, Y)$

The recursive definition of $(n_1, X) \rightsquigarrow (n_2, Y)$ is illustrated by Figure 3.9. Intuitively, notification n_1 "flows" from X to Y and, after potentially being transformed several times, it is received as notification n_2. The path on which n_1 flows to n_2 is the same as for the visibility relation defined in Section 3.2, i.e., it can be characterized by a path from X up to a common superscope and then down to Y. But in addition the notification is subject to any mappings assigned to the relevant edges.

The semantics of scoped event systems with mappings are derived from those of scoped event systems by the refined visibility definition. With like arguments the graph of scopes and the relations $\nearrow$ and $\searrow$ are assumed to be static in the sense that a component's mappings are not allowed to change until all of its published notifications are delivered; otherwise the visibility clause may corrupt the safety condition in the specification.

Definition 3.6 (scoped event system with mappings) *A* scoped event system with mappings $ES^{\mathcal{M}}$ *is a system that exhibits only traces satisfying the following requirements:*

- *(Safety)*

$$\Box\Big[\mathit{notify}(Y, n') \;\Rightarrow\; \big[\bigcirc\Box\neg\mathit{notify}(Y, n')\big] \\ \wedge \big[\exists n.\, \exists X.\, n \in P_X \;\wedge\; \big((n, X) \rightsquigarrow (n', Y)\big)\big] \\ \wedge \big[\exists F \in S_Y.\, n' \in N(F)\big]\Big]$$

- *(Liveness)*

$$\Box\Big[\mathit{sub}(Y, F) \Rightarrow \\ \Big(\Diamond\big[\Box\big((n, X) \rightsquigarrow (n', Y)\big) \;\Rightarrow \\ \Box\big(\mathit{pub}(X, n) \wedge n' \in N(F) \;\Rightarrow\; \Diamond\mathit{notify}(Y, n')\big)\big]\Big) \\ \vee \Big(\Diamond\mathit{unsub}(Y, F)\Big)\Big]$$

The difference between this definition and that of scoped event systems (Def. 3.3) is that the term $v(X, Y)$ is replaced by the term $(n, X) \rightsquigarrow (n', Y)$ and that the published notification n is not necessarily equal to the delivered n'. This formulation extends the system to not only obey the visibility of components but the visibility of individual notifications. The delivered notification n' is the result of repetitive applications of the mappings $\nearrow$ and $\searrow$ along the path implicitly defined by $\rightsquigarrow$. The present definition is even a generalization of the scoped delivery. This is because a scoped event system can be regarded as one with event mappings where all mappings are the identity relation, i.e., they do not change anything along the delivery paths. In such a system, $v(X, Y)$ is implied by the existence of a notification n such that $(n, X) \rightsquigarrow (n, Y)$.

Interfaces as Mappings

Notification mappings are a generalization of and subsume scope interfaces. The relation $\nearrow$ might be undefined for an outgoing notification n_1 so that there is no n_2 such that $n_1 \nearrow n_2$. This blocks the notification just as a non-matching filter does. In order to seamlessly extend scope interfaces, $\nearrow$ and $\searrow$ are constrained to always map to some notification, with the empty notification ϵ as default.

Definition 3.7 (notification mappings) *A* notification mapping *is given by a function in* $\mathcal{M} = \{m \mid m : \mathcal{N} \to \mathcal{N}\}$*.*

$$n_1 \nearrow n_2 \Rightarrow \exists m \in \mathcal{M}.\, m(n_1) = n_2$$

Whenever a notification is mapped to ϵ it is considered to be blocked so that filters are but special mappings: $\mathcal{F} = \{f \in \mathcal{M} \mid f(n) = n \vee f(n) = \epsilon\} \subset \mathcal{M}$. With this definition, a uniform way of filtering and transforming notifications is accomplished so that, conceptually, interfaces and mappings can be concatenated at scope boundaries, e.g., $F_1 \circ F_2 \circ M_1 \in \mathcal{M}$.

Next, interfaces and their concatenation are defined more formally to define $\nearrow$ and $\searrow$ as concatenated interfaces and mappings.

Definition 3.8 (interface) *An* interface *I consist of an input mapping ${}^iI$ and an output mapping ${}^oI$: $I = ({}^iI, {}^oI) \in \mathcal{M} \times \mathcal{M}$. The base interface I_C of a component C represents the sets of open subscriptions and advertisements of C:*

$$\begin{aligned} I_C &= ({}^iI_C, {}^oI_C) \in \mathcal{M} \times \mathcal{M} \\ &\triangleq ({}^iF_C, {}^oF_C) = \big\{\{F_1, F_2, \dots, F_k\}, \{F'_1, F'_2, \dots, F'_l\}\big\} \in P(\mathcal{F}) \times P(\mathcal{F}) \end{aligned}$$

where ${}^iI_C$ and ${}^oI_C$ are defined as

$${}^iI_C(n) = \begin{cases} n & \exists F \in {}^iF_C.\, F(n) = n \\ \epsilon & \textit{otherwise} \end{cases}$$

$${}^oI_C(n) = \begin{cases} n & \exists F \in {}^oF_C.\, F(n) = n \\ \epsilon & \textit{otherwise} \end{cases}$$

Selective interfaces $I_{C|S}$, and imposed interfaces I^S and I^S_C are defined likewise.

According to this definition an interface can transform notifications for the seamless concatenation of filters and mappings.

Definition 3.9 (concatenation of interfaces) *Two interfaces I_1 and I_2 are concatenated by*

$$I_1 \circ I_2 = ({}^iI_1 \circ {}^iI_2, {}^oI_2 \circ {}^oI_1).$$

Note that the resulting interface evaluates the composed input and output interfaces in inverse order. This is not necessary if only filters are considered, but by incorporating mappings the sequences are not commutative any longer. The effective interface between two components $C \lhd S$ describes the notifications transmitted along this edge in the scope graph and combines the aforementioned interfaces *and* notification mappings assigned to this edge, extending the informal description given in 3.3.1.

Definition 3.10 (effective interface) *The* effective interface $\hat{I}_C^S$ *between two components $C \lhd S$ is given by concatenating base interface, selective interface, mapping, and imposed interface:*

$$\hat{I}_C^S = I_C \circ I_{C|S} \circ M_C^S \circ I_C^S \circ I^S$$

Finally, the interfaces between two components $C \lhd S$ are correlated to the mapping relations $\nearrow$ and $\searrow$ as follows:

$$\begin{aligned} n_1 \searrow n_2 &\Leftrightarrow (I_C \circ I_{C|S} \circ {}^iM_C^S \circ {}^iI_C^S \circ {}^iI^S)(n_1) = n_2 \\ &\Leftrightarrow {}^i\hat{I}_C^S(n_1) = n_2 \\ n_1 \nearrow n_2 &\Leftrightarrow ({}^oI^S \circ {}^oI_C^S \circ {}^oM_C^S \circ {}^oI_{C|S} \circ {}^oI_C)(n_1) = n_2 \\ &\Leftrightarrow {}^o\hat{I}_C^S(n_1) = n_2 \end{aligned}$$

The rules of notification forwarding in the scope graph given by the publishing and delivery policies in equations 3.1 and 3.2 on page 48 can be refined corresponding to the above discussion:

$$\textbf{PP}: \quad X \overset{n_1}{\leadsto} S \wedge X \lhd S \lhd T \wedge {}^o\hat{I}_S^T(n_1) = n_2 \Rightarrow X \overset{n_1}{\leadsto}_{n_2} T \qquad (3.3)$$

$$\textbf{DP}: \quad \overset{n_1}{\leadsto} T \wedge S \lhd T \wedge {}^i\hat{I}_S^T(n_1) = n_2 \Rightarrow \overset{n_2}{\leadsto} S \qquad (3.4)$$

Despite the integration of interfaces and mappings, the scope overview in Figure 3.1 on page 45 still distinguishes interfaces and mappings to underline their different intentions, and also because their implementations are apt to diverge.

Some Further Comments

The already mentioned issue of duplicate notifications has to be reconsidered here. A notification is duplicated if it travels along different paths from producer

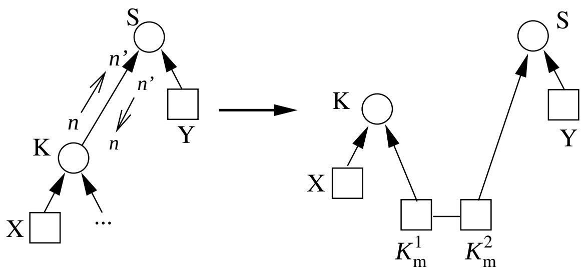


Figure 3.10: Transformation of mappings into components

to consumer, but it may now be subjected to different mappings so that different versions of the same original notification are created. The specification cannot rule out this case since it is highly application-dependent whether this is an unwanted situation or not. The mappings may help handling alternative delivery paths as they can annotate passing notifications, e.g., to include information about the delivery path in the notification.

Trying to offer a sophisticated concept of heterogeneity support in event-based systems is beyond the scope of this thesis, and thus notification mappings are presented as a starting point for including appropriate enhancements. The mappings underline the extensibility of the scoping concept and open it to integrate existing works in the area of syntactical and semantical transformations that are applicable here [41, 71, 183]. Furthermore, the current if implicit assumption that notifications are mapped one-to-one is used for simplicity only. Scope boundaries may turn out as the appropriate place to implement more sophisticated event composition [189, 304].

3.4.2 A Correct Implementation

The following presents an implementation sketch of the scoped event system with mappings. The implementation of a scoped event system with mappings $ES^{\mathcal{M}}$ is based on a scoped system $ES^{\mathcal{S}}$ and a transformation of the graph of scopes G that essentially follows the idea of adding activity to edges. Figure 3.10 sketches the transformation that creates G' by replacing every edge (K, S) that does not apply the identity mappings $n \nearrow n$ and $n \searrow n$ for two extra mapping components K_m^1 and K_m^2. *Two* mapping components are taken to constrain the visibility of the transformed notifications to the appropriate scopes. If only *one* K_m would be inserted, additional measures had to be taken to distinguish the superscopes.

Figure 3.11 describes the architecture of the implementation for the example system in Figure 3.10. A component X connected to $ES^{\mathcal{M}}$ is also directly connected to an underlying scoped event system $ES^{\mathcal{S}}$. Calls to $pub(X, n)$ of $ES^{\mathcal{M}}$ are forwarded to $ES^{\mathcal{S}}$ without changes, and vice versa, calls to $notify(X, n)$ of $ES^{\mathcal{S}}$ are forwarded to $ES^{\mathcal{M}}$.

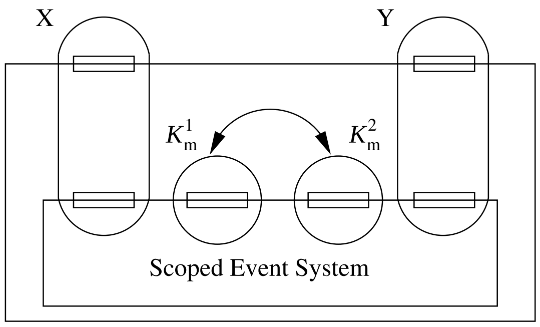


Figure 3.11: Architecture of scoped event system with mappings

In general, if a scope K is to be joined to a superscope S by calling *jscope*(K, S), two mapping components K^1_m and K^2_m are created that communicate directly via a point-to-point connection. K^1_m joins K, subscribes to all notifications published in K, transforms and forwards them to its peer. Furthermore, subscriptions in K have to be transformed before they are forwarded. The implementation relies on externally supplied functions that map notifications and filters/subscriptions between the internal and external representations in K and S, respectively. K^2_m joins S and republishes all notifications it gets from its peer K^1_m. It subscribes in S according to the subscriptions forwarded by K^1_m, transforms any notifications received out of S, again with externally supplied functions, and forwards them to K^1_m which republishes them into K.

Correctness

The algorithm from the previous section has to satisfy the requirements given in Definition 3.6 of $ES^{\mathcal{M}}$, i.e., safety and liveness conditions. The correctness proof largely depends on the correctness of the underlying scoped event system $ES^{\mathcal{S}}$. The next lemma relates the graph transformation to the structure of delivery paths.

Lemma 3.3 *If $(n, X) \rightsquigarrow (n', Y)$ holds, then in the implementation of $ES^{\mathcal{M}}$ exists a sequence $\rho = C_1, C_2, \ldots, C_m$ of components for which holds:*

1. *$C_1 = X$ and $C_m = Y$,*
2. *for all $1 < i < m$ holds that C_i is a mapping component, and*
3. *for all $1 \leq i \leq m-1$ holds that C_i and C_{i+1} either share a communication link or reside in the same scope of $ES^{\mathcal{S}}$.*

Proof: Assume $(n, X) \rightsquigarrow (n', Y)$ holds. From the definition of $\rightsquigarrow$ follows that there exists a delivery path $\tau = (X, S_1, S_2, \ldots, S_l, Y)$ in the scope graph G. Since

visibility is recursively defined by having common superscopes, all S_i must be scopes.
The construction method of building G' from G implies that every consecutive pair of scopes (S_i, S_{i+1}) in τ where mappings are applied is enhanced with two mapping components K_i^1 and K_i^2 which are joined by a direct communication link. The mapping components K_i^2 and K_{i+1}^1 of neighboring edges reside in the same scope S_{i+1} or are visible to each other. The projection of τ to mapping components (and X and Y) results in a sequence $X, K_1^1, K_1^2, K_2^1, K_2^2, K_3^1, \ldots, K_l^2,$ Y, which is the witness for the sequence ρ of the lemma. □

Proof of Safety. Assume that Y is a simple component and that *notify*(Y, n') of $ES^{\mathcal{M}}$ is called. It must be shown that the three conjuncts of the implication in the safety property of Def. 3.6 hold.

From the algorithm description follows that *notify*(Y, n') of $ES^{\mathcal{S}}$ was called before, implying that n' is notified at most once and that n' matches an active subscription of Y. This proves the first and the third conjunct.

The second conjunct is proved by a backward induction on the path guaranteed by Lemma 3.3. The fact that Y is notified about n' implies that there is a component Z that has published n' which resides in the same scope. If this Z is not a mapping component, Z plays the role of X in the formula, $n' = n$, and the second conjunct follows immediately (this is the base case of the induction). The step case of the induction is as follows: Assume that a component Z'' along the path has published some notification n'' which from backward notification mappings resulted from n'. Then there exists a component Z''' which is either in the same scope or connected by a communication link to Z''. In the first case, the step follows from the properties of $ES^{\mathcal{S}}$ and in the second case from the algorithm. This implies that $n \in P_X$ and that $\big((n, X) \rightsquigarrow (n', Y)\big)$, giving the second conjunct.

Proof of Liveness. The liveness property is proved by forward induction on the path guaranteed by Lemma 3.3 in a similar way as in the proof of the safety property. Assume that Y subscribes to F and never unsubscribes. Then assume that after subscribing, $(n, X) \rightsquigarrow (n', Y)$ begins to hold indefinitely. Then Lemma 3.3 guarantees a path between any publisher X of a relevant notification n and Y. A similar way of reasoning as in the safety proof implies that n is forwarded and transformed along the path resulting in n' which Y is eventually notified about.

3.4.3 Example

Returning to the stock exchange example, mappings can be exploited to convert between different currencies.[9] Quotations are typically given in a local currency which need to be transformed at the boundary of the local scope in order to

[9]At least from a technical point of view, disregarding varying exchange rates.

achieve comparability. As another example for the usefulness of mappings consider XML languages like FIXML [223] that standardize financial data exchange. These languages are used to connect external partners, but they are typically too expensive for internal representations due to efficiency reasons. Also, most likely, different representations of events will be used inside the consumers, within the market, and within the trading floor, e.g., Java objects, XML financial data, and EBCDIC mainframe text fields. Notification mappings are installed at the consumers and at the trading floor to map between serialized Java objects and their XML representation and between XML and EBCDIC, respectively.

3.5 Transmission Policies

The discussion of engineering requirements in Section 2.5.2 argued not only for the heterogeneity of data models but also emphasized the necessity to adapt notification delivery semantics. The ability to accommodate diverging application needs improves the utilizability of the event service. It helps to provide tailored and efficient implementations, and it avoids a one-size-fits-all approach, which is not appropriate for a communication substrate targeted at evolving networked systems.

The next paragraphs distinguish *transmission policies* to describe how notifications are forwarded in the scope graph. Transmission policies are a way to influence notification dissemination beyond filtering on notifications. While filters operate independently on independent notifications, i.e., they are stateless, transmission policies may have their own state and they exploit additional information not available in filters and interfaces. They refine the visibility definition both within a scope and with respect to its superscopes. Changing it affects the functionality of the overall system in a fundamental way. However, once delimited by scope boundaries, such modifications are the means that allow administrators to customize the interaction within and the composed functionality of specific scopes.

Conceptually, notification forwarding at a node in the scope graph first determines a set of eligible next-hop destinations according to the effective interfaces and then applies the policies to refine this set before transmission. Default policies implement the known semantics of notification delivery, and by explicitly binding them to individual scopes in the specification of event systems, they are subjected to modification on a per scope basis. This gives the administrator a tool to not only compose but to program scopes. Three different policies are involved in notification transmission: publishing, delivery, and traverse policies.

3.5.1 Publishing Policy

A *publishing policy* is associated with a component and controls into which superscopes an outgoing notification is forwarded. In Figure 3.12, a publishing policy at S can prevent a notification $Y \overset{n_1}{\leadsto} S$ from being forwarded to T, even

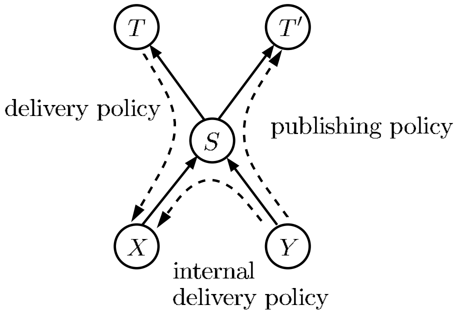


Figure 3.12: Three important transmission policies in scope graphs

if the notification conforms to the effective output interface ${}^{o}\hat{I}_S^T$. Out of the set of eligible superscopes the publishing policy selects the subset to which a notification is actually forwarded. One might reject the idea of manually selecting the scopes into which data is published as contradicting the event-based paradigm. However, the same arguments as for selective interfaces apply here, too. The selection is part of the administrator's role and not interwoven with application functionality in simple components. It can be seen as an additional way to control interaction of components outside of the components themselves.

In general, a publishing policy of a component C is a mapping of notifications to a subset of its scopes:

$$pp_C : \mathcal{N} \to P(\mathcal{S})$$

The mapping relation $\nearrow$, which determines the visibility of notifications, can be extended to respect publishing policies. For an edge $e = (S, T)$ of G let

$$n_1 \nearrow_e n_2 \Leftrightarrow {}^{o}\hat{I}_S^T(n_1) = n_2 \wedge T \in pp_S(n_1)$$

The general rule of forwarding outgoing notifications in the scope graph is implied as follows. Assume Y made a notification n_1 visible in its scope S, $Y \overset{n_1}{\rightsquigarrow} S$, and S is a subscope of T, $S \lhd T$, then the notification shall be visible in T if n_1 matches the effective output interface between S and T and the publishing policy does not object to T. That is,

$$\mathbf{PP}: \quad \underbrace{Y \overset{n_1}{\rightsquigarrow} S \wedge Y \lhd S \lhd T}_{\text{component visibility}} \wedge \underbrace{{}^{o}\hat{I}_S^T(n_1) = n_2}_{\text{interfaces mappings}} \wedge \underbrace{T \in pp_S(n_1)}_{\text{publishing policy}} \Rightarrow S \overset{n_1}{\rightsquigarrow}_{n_2} T \qquad (3.5)$$

This definition of PP refines the previous one of scoped delivery with mappings given in Equation 3.3. It can be reduced to the former definition by setting $pp_S(n_1) = \mathcal{S}$, which always validates $T \in pp_S(n_1)$ and makes equations 3.5 and 3.3 equivalent. Note that the equation also implies $Y \overset{n_1}{\rightsquigarrow}_{n_2} T$.

A publishing policy might be used to check for attributes not available in filters and interfaces. Since it is implemented as part of the administrator role it possibly has access to the scope graph layout and associated meta data. If the availability of security credentials can be checked by the policy, a scope may thus mandate that *its* notifications are only delivered if a certain privilege level is held by the destination scope. But this simple definition leaves room for any form of implementation. In the stock exchange example a market was divided into a professional and a private market (see page 58). The former gets undelayed stock quotations and is modeled as a subscope of the private market. A publishing policy at the boundary between these two scopes may be used to delay each notification for a certain amount of time. Such implementation-specific issues are not excluded by the above definition.

3.5.2 Delivery Policy

A *delivery policy* is associated with a scope and guides notifications that are to be delivered to scope members. They may either be published in a superscope or by some other constituent component. The delivery policy determines to which members of the scope a notification is forwarded. In Figure 3.12, a delivery policy at S might direct a notification $T \overset{n}{\leadsto} S$ to X, prohibiting the delivery to Y even if the notification conforms to the effective input interface ${}^{i}\hat{I}^{S}_{Y}$. Out of the set of eligible children the delivery policy selects a subset to which the notification is actually forwarded.

Similar to publishing policies, a delivery policy of a scope S is a mapping of notifications to a subset of components:

$$dp_S : \mathcal{N} \rightarrow P(\mathcal{K})$$

The mapping relation $\searrow$ can be refined so that it obeys scope interfaces and reflects delivery policies on incoming notifications. Consider $e = (X, S)$ as given in Figure 3.12 and a notification visible to S in T, $T \overset{n_1}{\leadsto} S$. The visibility of the notification within S is then determined by

$$n_1 \searrow_e n_2 \Leftrightarrow {}^{i}\hat{I}^{S}_{X}(n_1) = n_2 \wedge X \in dp_S(n_1).$$

Please note that this equivalence does not only guide forwarding of incoming notifications but also of internal notifications published by scope members; in the example, $T \overset{n_1}{\leadsto} S$ and $Y \overset{n'_1}{\leadsto} S$ would go down the same edge $e = (X, S)$. However, since internal and external communication is typically treated differently an additional *internal delivery policy* idp_S is introduced to facilitate this differentiation. The definition of $\searrow_e$ has to distinguish between applying dp_S and idp_S. In the first case, n_1 is an incoming[10] notification that is made visible by a superscope T, i.e., $X \lhd S \lhd T$ and $T \overset{n_1}{\leadsto} S$. In the second case n'_1 is an internal notification that is made visible by a member of S, i.e., a sibling of the

[10]The term "internal" and "incoming" notifications are also discussed on page 49.

considered consumer X, $X \lhd S \rhd Y$ and $Y \overset{n_1'}{\leadsto} S$.

$$n_1 \searrow_e n_2 \Leftrightarrow \begin{cases} {}^i\hat{I}^S_X(n_1) = n_2 \wedge X \in dp_S(n_1) & \text{if } X \lhd S \lhd T \wedge T \overset{n_1}{\leadsto} S \\ {}^i\hat{I}^S_X(n_1) = n_2 \wedge X \in idp_S(n_1) & \text{if } X \lhd S \rhd Y \wedge Y \overset{n_1}{\leadsto} S \end{cases}$$

The rule of downward notification delivery (p. 64) is thus given as follows.

$$\mathbf{DP}: \quad \underbrace{T \overset{n_1}{\leadsto} S \wedge X \lhd S \lhd T}_{\substack{\text{component visibility} \\ \text{incoming notification}}} \wedge \underbrace{{}^i\hat{I}^S_X(n_1) = n_2}_{\substack{\text{interfaces} \\ \text{mappings}}} \wedge \underbrace{X \in dp_S(n_1)}_{\substack{\text{delivery} \\ \text{policy}}} \Rightarrow S \overset{n_1}{\leadsto}_{n_2} X \quad (3.6)$$

$$\mathbf{iDP}: \quad \underbrace{Y \overset{n_1}{\leadsto} S \wedge X \lhd S \rhd Y}_{\substack{\text{component visibility} \\ \text{internal notification}}} \wedge \underbrace{{}^i\hat{I}^S_X(n_1) = n_2}_{\substack{\text{interfaces} \\ \text{mappings}}} \wedge \underbrace{X \in idp_S(n_1)}_{\substack{\text{internal} \\ \text{delivery policy}}} \Rightarrow S \overset{n_1}{\leadsto}_{n_2} X \quad (3.7)$$

Again, from the equations and the definition of $\leadsto$ also follows that $T \overset{n_1}{\leadsto}_{n_2} X$ and $Y \overset{n_1}{\leadsto}_{n_2} X$.

An example of a delivery policy is an 1-of-n delivery where an incoming notification is forwarded to only one out of a group of possible receivers. In this way load balancing characteristics may be implemented in a specific scope. Internal delivery policies are pertinent whenever the data flow within a scope shall be controlled in addition to the established filters. An internal delivery policy is able to arrange multiple consumers into a chain. Consider a sequence of exception handlers, each subscribed to the same type of failure, which it tries to solve, and if not possible it republishes the received notification. An internal delivery policy can forward each published error notification to the next hop in the preconfigured list of consumers/handlers.

3.5.3 Traverse Policy

The last, only informally presented policy is the *traverse policy*, which is associated with a scope S and controls the downward path of incoming notifications in a scope. In contrast to the preceding policies, the traverse policy does not select destinations within a certain scope but selects the scope into which to descend first. It searches at different levels in the scope hierarchy below S for a scope with eligible consumers, and if one is found it will stop searching and refer the notification to the respective scope.

Actually, this policy allows a notification to deviate from a default path through the graph of scopes. In a top-down traverse policy eligible receivers, i.e., simple components with a matching subscription, are searched in the current scope first. If no consumer is found at this stage, the search is continued in the next lower level of scopes if the policy still applies there (same administrative domain). The bottom-up traverse policy starts the search in the deepest subscopes. "Broadcast" is the default policy which does not inhibit descending the scope graph and delivers to all eligible consumers $C \overset{\star}{\lhd} S$ below the current scope S, subject to interfaces and delivery policies, of course.

This kind of dissemination control is apparently inspired by dynamic binding and method lookup in object-oriented class hierarchies. Multiple consumers of the same notification, which are located at different levels in the inheritance/scope hierarchy, can be considered to implement some form of generalized method overriding. While traditional programming languages like C++ and Java use only one, static policy to resolve calls to overridden methods, traverse policies draw ideas from meta-object protocols [170] to determine what kind of method lookup is used. The bottom-up policy resembles a virtual method call in Java in that the implementation of the most derived class is used. Other policies are possible that implement other kinds of method lookups.

3.5.4 Influencing Notification Dissemination

Transmission policies are a means to adapt the event-based dissemination within scopes. They make the interaction in the graph programmable.

To some extent transmission policies bear similarities to meta object protocols (MOP) known in object-oriented programming [170]. Meta object protocols offer the ability to redirect or transform messages sent as method calls, and this control allows to influence object interaction outside of the objects' implementation. Here, notifications are selected, transformed, ordered, or queued, to manipulate the default visibility of notifications and to adapt event-based interaction within the bounds given by the scoping structure to which the policies are associated.

As for the expressiveness and possible implementations of transmission policies note that the above definition is not intended as an algorithmic description. It integrates with the specification of scoped event systems with mappings given in Definition 3.6, and since the specification relies on linear temporal logic, it only describes valid traces of system execution. In particular, any implementation that exhibits such traces conforms to the specification. So, even if the rules PP, DP, and iDP might connote an algorithm for notification forwarding, possible implementations covered by the definition of pp_S, dp_S, and idp_S, and of $\nearrow_e$ and $\searrow_e$, can be Turing-complete. For instance, delaying notification as part of a transmission policy is sanctioned as long as any later delivered notification still adheres to the visibility definition and the safety condition of the specification.

The decision made by a transmission policy is based on additional data not available in filters and interfaces. Various characteristic approaches to decision making can be distinguished. There are policies that essentially implement filters on notifications like component interfaces, but which are able to exploit additional meta data. Notifications carry management information, which is annotated by the event system and stripped off before delivery, and as a tool of the administrator policies might access this data. So, they would be able to differentiate producers, e.g., to check security credentials. Furthermore, transmission policies probably have (limited) knowledge about the current scope graph layout and of a notification's (partial) path through the graph. The Session Scopes presented in the next section utilize this kind of information.

The second, more complex form of transmission policy does not filter any data contained in notifications, but compares all eligible destinations, ranking them to do a top-k selection. The ranking may be random, based on lowest utilization, etc. And finally, when the policy implementation maintains its own state, it might keep a record of the last sent notifications in order to limit the maximal bandwidth towards a consumer by rejecting too frequent notifications. Or it might realize a round robin 1-of-n delivery. With its own state the policy is capable of delaying notifications for a certain amount of time or until a specific condition becomes valid, i.e., a "releasing" event occurs. This opens a venue to bind event composition to scope boundaries, or to implement a form of acknowledged notification forwarding where acknowledgment is given components other than the original producer.[11]

3.6 Scoping of Notifications

The discussion in this chapter has concentrated on scoping as a means to structure applications, implying a scoping of notifications due to component visibility. This section will take a first look into controlling the visibility of notifications directly. It will show how the scoping concept can be applied to structure the dynamic aspects of applications.

It would be illusionary to think that the loose coupling of event-based interaction allows components plugged into a black box system to process any received notification in a meaningful way. This is because a single notification is typically not self-contained and is received out of any context. The context of a notification is the components interacting with the producer and the preceding notifications, which led to its publication—only events of the physical world occur without apparent history in the computer system. So far, scoping helps to describe and define the context of notifications in terms of the components interacting in the system.

3.6.1 Dependent Notifications

In the simple application scenario of uni-directional flows of notifications, stateless communication prevails, cf. Section 2.5. However, in more complex scenarios notifications are interrelated, and it gets more important to make the context of notifications discernible. As pointed out, receiving notifications out of context impairs functionality, and thus distinguishing the context is not only necessary to decide which past notifications are related but also to set any reaction into the "right" context.

Consider three scopes C, S, and T, where scope C participates in both S and T, $C \lhd S$ and $C \lhd T$; for example, C may be a customer of two stock markets. Notifications published in S are not visible in T, and vice versa. However, C acts like a bridge and its reactions are visible in both superscopes. While scoping

[11]Let's call the releasing notifications *commit* and *abort*... Please refer to Section 3.6.

provides notification context to some extent, the reactions published by C are obviously related to the original notifications in S, but possibly out of context in T. The delimitation imposed by scopes is diluted and implications of invisible causes are diffused; an effect that is not always desired.

If the above customer C has locally installed rules to automatically buy shares when prices exceed predefined limits, it would be better to direct orders only to the market that has offered the triggering prices; or converse if arbitrage opportunities shall be exploited. For another example assume C is a security service that consumes, timestamps, signs, and republishes notifications so that applications S and T are able to publish signed notifications. It would be unacceptable to publish the signed data into both scopes.

As far as this discussion is concerned, *dependent notifications* are related to past notifications and should, usually, only be visible to same components as the causing notifications. The implied basic requirement is that relationships must be distinguishable to make the reaction part of the given context.

This problem can be approached in a number of ways. A supposed solution is to replicate the component C so that each instance no longer belongs to multiple superscopes. But this is only possible if the instances do not share a common state. If they do, other forms of communication are necessary between the instances to ensure their consistency, and thus the event-based model and the scoped system model would be abandoned. On the other hand, in large systems the administrators of S and T might be forced to use a component C that is already deployed in a different administrative domain. Finally, the graph transformation would result in a tree, which is a very limited means to structure applications [153].

A different, often used approach to relate notifications is to add identifiers (IDs) to notifications. Producers annotate notifications to indicate their own identity or that of the notifications. However, as pointed out in the discussions in Chapter 1 and Section 2.5, filtering on annotated notification does not solve this problem adequately. Manually modifying notifications and filters contradicts the event-based paradigm and provokes a tighter coupling of components by reestablishing the explicit handling of peers. This virtually negates the benefits of using events and should be avoided if other, more appropriate solutions can be found, like the session scopes presented next.

3.6.2 Session Scopes

There is a clear connection between scopes and dependent notifications. The dependency implies a grouping of notifications and represents their context. For an initial support of dependent notifications *session scopes* are introduced, which utilize and extend the previous notion of scoping. They reify this grouping of notifications and make it possible to delimit the interactions belonging to different contexts, even if they involve the same components.

Session scopes also have to employ identifiers to mark notifications as being part of the respective scope, but the management of these identifiers is transparent to the participating components. The basic idea is as follows. A session scope has an associated session ID, which is annotated on every notification published in the scope. On consumer side, no filters are changed and the processing of delivered notifications takes place as before; however, the processing is wrapped by the event system, which applies session-specific publishing policies to direct triggered notifications back into the originating session scope.

A scope becomes a session scope by assigning a session ID to it. A session scope installs a mapping on each child that tags every published notification with this ID.[12] The semantics of notification forwarding within session scopes is not altered and remains the same as in 'normal' scopes. To discuss delivery and processing of notifications recall the conceptual system model presented in Section 2.4. In each component the local event broker connects the application component to the event system, and it is this local broker that wraps the components' notification processing and handles session ID tags. It maintains a hidden processing context containing the ID, which remains valid during the processing of the tagged notification, that is, during the wrapped $\mathit{notifiy}(X, n)$ call. All notifications published while a valid context is available are again tagged with the ID. Tagged notifications are not delivered into any other session superscopes but the originating one. Non-session superscopes are not affected by this differentiation so that newly created session scopes do not influence the existing scope configurations. Of course, this behavior can only be considered as a reasonable default; it must be possible to adapt the function of session scopes, e.g., to restrict reactions to the originating session scope irrespective of any other superscope.

Applied in the above scenario of a security service C, the two application superscopes S and T would be declared as *session scopes*. On delivery of a notification the previously registered processing function is invoked that computes the signature and republishes the signed result. The tag carried by the notification is maintained as hidden context during the execution and the result is transparently directed to the originating superscope. Note that the API of the event service and the application component are not changed.

To illustrate session scopes in the stock application, the interaction of a customer with two markets can be modeled in the same way as the security service.

Session scopes are a contribution to model interactions in event-based systems. They leverage the presented visibility model and help to separate simultaneous interactions acting on the same components, controlling the diffusion of implications of otherwise invisible events. Technically, they can be generalized to context-dependent scopes, whose interfaces depend on the context of a notification. Currently, different interfaces are assigned with each incident edge of a component in the scope graph, but every notification is checked against the same set of interfaces. With context-dependent interfaces each context would have its own set of interfaces on all edges. Different notifications are then tested against

[12]For simplicity assume a content-based data model where such modifications are feasible, though other non-opaque data models might work, too.

different interfaces, and different sessions could have different cross-session forwarding semantics, etc. This would make the influence and semantics of sessions totally customizable, and might offer a way to deal with event-based interactions that goes beyond existing approaches [190, 286].

Furthermore, an apparent enhancement is to make the context stored in the local event broker accessible to the components. This does not seriously impair loose coupling because the resulting stateful component functionality would be still part of a self-focused, event-based processing—stateful interactions just group several small state changes into a bigger one. Having a context-dependent storage, which may carry additional application data, is comparable to thread-local storage used in multi-threaded applications where the address space is shared but each thread has still a private memory of local state.

To conclude, session scopes are an initial proposal of activities in structured event-based systems. These scopes can be instantiated as first-class representatives of activities, allowing all the investigated features of scoping to be applied, not only to structure the application but also to structure the interaction itself. Once these session scope have output interfaces of their own, once they have the ability to delay notifications until certain conditions are met, they resemble the classical notion of transactions, but mapped to the ideosyncrasies of event-based systems.

3.7 Security

Scopes can be used to localize security policies in event-based systems. Associated with a specific scope, they control which components are allowed to join, which interfaces are imposed on them, and which selective interfaces are applied to outgoing traffic. The decision depends on credentials presented when an edge is established or modified. Furthermore, security requirements may govern the choice of the specific form of scope implementation (regarding its architecture and communication technique, cf. Section 4.1). A more detailed discussion is out of the scope of this thesis, but is given in Fiege et al. [117].

3.8 Engineering with Scopes

Scopes are an engineering abstraction for event-based systems. And to some extent they are comparable to classes and objects in object-oriented design and programming. They can be used to model system entities and their relationship and, on the other hand, they provide the basis for system implementation in form of a specific object/component model.

So far, there was no clear distinction made between using the scope graph as a modeling tool or as means of implementing system structure. In order to reflect the different objectives two types of scope graphs are distinguished. *Descriptive scope graphs* describe a set of components, their relationships and visibility constraints as expressed by the scope features annotated in the graph.

An *instantiated scope graph* describes a running scoped event system, which contains instances of various descriptive scope graphs. The former can be seen as a collection of scope types and classes, while the latter constitutes the runtime environment. Interestingly, both can be combined in one graph. If the descriptive graph is treated as abstract scopes (cf. Section 3.2.6) in a combined graph, instantiated components are members of their respective descriptive scopes. This combination does not affect communication within the instantiated scope graph, but allows for instance grouping and runtime reflection.

In the remaining subsections a development process is described that shows how descriptive scope graphs are created and how they are deployed, i.e., transformed into instantiated scope graphs. A language for specifying and programming scopes and scope graphs is introduced afterwards.

3.8.1 Development Process

The development process for scoped event systems consists of four stages:

1. Component design
   Individual simple components and preconfigured scopes are created and put into repositories for later use. The design at this stage specifies required and provided interfaces and employed scope features. Larger descriptive scope graphs can be built up from these preconfigured components.

2. Scope graph design
   From a selection of existing and newly created components a descriptive scope graph is created. This step concentrates on orchestrating preconfigured components, resolving open interface constraints. No implementation issues are handled.

3. Scope graph deployment
   An existing descriptive scope graph is translated into a running system. Implementation techniques are choosen, integration code to bridge with existing systems is generated, infrastructure code is deployed to selected nodes of the network, etc.

4. System management
   A running system is monitored and adapted at runtime. This is necessary to react to failures, to install new components and to evolve the system where necessary.

3.8.2 Component Definition

From the engineering point of view, a scope can be considered as *a module construct for event-based systems*, being an abstraction and encapsulation unit at the same time. As an abstraction unit, a scope provides the rest of the world with common higher-level input and output interfaces to the bundled subcomponents, eventually mapping these interfaces to the interfaces of the individual constituents. As an encapsulation unit, a scope constrains the visibility of the notifications produced by the included components. It hides the details of the composition implementation. The engineering of single scopes is about building new event-based components.

Generally, programming of scopes has two sides. First, it is about arranging and orchestrating a set of components; this is the structure of the scope. Second, programming is about specifying the dependencies on other components that are not part of the predefined scope. At runtime a certain environment of available producers and consumers might be required, which are essential for the operation of this scope, but not part of its definition; this is the context of the scope.

How are these two tasks accomplished? Three ways for specifying and programming scopes are considered in this thesis: Java API, XSchema, SQL-like language. A Java API is described in Section 5. Obviously, simple components are implemented using this API to the publish/subscribe service. It also supports the definition of scopes in the sense that scope objects with a specific set of features can be created for later deployment. However, the context of a scope is a list of requirements that can only be encoded in the XML or SQL-like language.

In [219] an XSchema is defined that covers descriptions of network layout, broker networks, scope graphs, single scopes, and their dependencies, called coupling points. A coupling point is a description of what other components are needed at deployment. It contains an expression on scope attributes, required interfaces, and the roles eligible components must play. Roles are introduced here as a suggestion to describe functionality on a level more abstract than interfaces. Technically, roles involve only string matching on a well-defined attribute. However, they enable system engineers to distinguish components even if they have identical interfaces. As an example consider two components subscribing for temperature events. One component calculates the average, the other one logs all published temperatures. Both would use the same interface and a role annotation could help distinguish them. Roles are used to name sets of interfaces and/or semantics of interfaces. A meaningful interpretation of the names relies on agreements made outside of the notification service.[13]

The SQL-like language presented in Section 3.8.6 facilitates the definition of scopes and their features, includes coupling points to express dependencies and rules for modifying scopes and their position in the scope graph.

Who is responsible for setting up and maintaining the scope graph? In order to not impair their loose coupling, application components should not be forced to interact with the scope graph. For this reason, they may access the graph structure through the Java API, but typically programming and configuration is done by the administrator, who knows the included components and is able to govern their interaction. Of course, different administrators may be responsible

[13]The use of ontologies like in concept-based pub/sub [71] is an example of such externally provided agreements.

for different scopes. Different administrative domains that build on abstract scopes are separated in [219].

As a result of component design a repository of components, i.e., a descriptive scope graph, is created for later composition in bigger scope graphs and for later deployment.

3.8.3 Scope Graph Composition

This second stage of the development process creates the descriptive scope graph. From a selection of existing and newly created components a graph is designed, typically for a specific application. This task includes the resolution of dependencies on interfaces, attributes, and roles, and the specification of application-specific implementation requirements.

The graph describes the relationship between components and stores predefined configurations on a larger scale than single components. Similar to class hierarchies, the scope graph offers a way to statically describe system structure. The graph is created for a specific application, and so the question is raised, what can be modeled with a scope graph? Since scopes are a generic concept to partition applications and control their interaction, this question asks for a methodology and design guidelines, which is out of the scope of this thesis.

The question by what means an administrator creates this graph is answered, though. Scope graph design must comprise tools and primitives to compose scope graphs from given specifications, to create and configure connections in the graph, and to resolve open dependencies. The Java API can be used to wire specific components or to resolve dependencies by application-specific rules. Existing scope specifications based on the XSchema grammar can be joined, whereby unambiguous dependencies can be resolved with a simple search on the available component definitions. The SQL-like scope language also facilitates this step by altering existing definitions, substituting descriptions of coupling points with lists of concrete components. A graphical tool for creating scope graphs was created that stores definitions as XML [219] and allows for manual composition of the scope graph.

However, it may happen that not all dependencies can be resolved before deployment, especially if the runtime environment consists of instances of different descriptive scope graphs. They must be resolved at deployment time or even at runtime. The scope language offers event-condition-action rules for this purpose.

Finally, the descriptive scope graph can carry annotations that have no immediate meaning in this step, but are interpreted in later on, similar to stereotypes in the Unified Modeling Language (UML, [122]). For example, annotations of required quality of service attributes may govern the following deployment step, hinting at appropriate implementation techniques.[14]

[14]Obviously, the stepwise transformation and deployment of the scope graph resembles the ideas of model-driven development [123].

3.8.4 Scope Graph Deployment

Scope deployment creates or extends an instantiated scope graph, which contains all scopes currently running in the system. This step deploys preconfigured scopes of one or more descriptive scope graphs, it resolves open dependencies, and chooses and parameterizes the implementation techniques for the deployed scopes.

The remaining context dependencies of the descriptive scope graph are resolved at deployment time. Often multiple descriptive graphs are used to describe different applications and subsystems. Their models evolve independently and they only rely on some of the services provided by others. So the deployment step is also an integration step that combines (independently administered) systems at a high level of abstraction.

Some dependencies are not resolved once and for all at deployment. They do not pertain to the static structural layout of the system, but rather depend on the execution of the event-based system. This is described as part of the *management* paragraph below.

An important point, not only in this step but for the scope concept in general, is the fact that the choice of a concrete implementation technique is postponed until now. The implementation of a scope and its communication facilities is determined here based on annotations made in the descriptive scope graph and/ or based on decisions made by the administrator. This approach allows for a model driven implementation, which fits the needs of the application to the services available in the system. Requirements on causal ordering or security considerations can be part of the application model, and the administrator decides how these things are implemented using available group communication protocols and encryption and key management schemes. Consequently, scopes are the appropriate place to customize specific parts of a system, as demanded in Sect. 3.1.2.

3.8.5 Management

Scope graph management comprises tools and primitives to maintain and update the instantiated scope graph. All features of scopes are subject to updates and even the layout of the scope graph can be changed, establishing and destroying edges by joining and leaving scopes. It also convers the manual creation of new scopes, and thus deployment is part of scope graph management. These tasks must be available in the API of the publish/subscribe service.

It gets interesting when considering automatic updates. As mentioned above, scope graph layout can be dynamic depending on the execution of the system. Automatic updates of the graph use the management functions to react to events and conditions observed in the system. The scope language presented below allows ECA rules to be associated with scopes. Each rule react to arbitrary notifications visible to the respective scope and if an optional conditional expression is fulfilled arbitrary management commands are executed. Binding these rules to scopes uses the visibility constraints of the scope graph to apply them only in limited areas of the graph. As for the scoped communication, this controls the execution of rules and reduces the complexity of rule analysis [26].

Such rules can be used to define scopes that automatically include all components conforming to a certain condition. One example are mobile systems, which are an apparant application domain of scoped notification delivery. The geographic vicinity to a reference location groups all components within this area.[15] In fact, whenever location models do not strictly correlate to the topology of the network infrastructure, some form of application-specific scoping is necessary [107]. Another example are the session scopes describe in Section 3.6.2. One way to populate a session scope is to join all consumers of a specific base notification. Using notifications to trigger scope joins follows the idea of extending transactional spheres of control by propagating transactional contexts [145].

3.8.6 Scope Graph Language

In order to support the development process a specification language for scope graphs is defined next. Corresponding to the abstract nature of the scope concept, the language definition is intended to be open for further refinements, which are probably domain dependent.[16] A Backus-Naur form is used to specify the syntax in form of production rules like

```
rule1 ::= ( "A" | rule2 ) [ rule3 ] rule4-commalist
```

Here, rule `rule1` is expanded to either the literal 'A' *or* the result of `rule2`, followed by zero or one expansion of `rule3`, followed by one or more comma separated expansions of `rule4`.

The next paragraphs introduce a grammar for defining scopes, their features and dependencies. It includes the rule '...' at places of possible future extensions.

Component References

In order to identify any specific component, a reference scheme for components must be defined. For the sake of simplicity only symbolic names are considered here.

```
simple-component-name ::= symbolic-name
simple-component-ref ::= simple-component-name

scope-ref ::= symbolic-name | ( "MEMBERS(" scope-ref ")" )

component-ref ::= simple-component-ref | scope-ref
```

[15]Grouping always implies a common context, and scoping thus may contribute to the discussion about context in mobile systems [262].

[16]A more thorough investigation can be found in Mühleisen [219], a refinement for wireless sensor networks is in [271], and a related work on views in peer-to-peer system is [31].

Wherever a scope is referenced by its name, e.g., *scope1*, the scope itself is meant, that is, the node in the scope graph. The *MEMBERS(scope1)* expression is used to refer to the members of the scope, that is, the set of nodes $C_i \lhd scope1$ of the scope graph.

Names are not globally unique, they are scoped. A component is part of some scope and its name is, at first, only valid within its scope. A reference to a scope is always resolved from a specific node in the scope graph. If for a given name no component exists in the current scope, all super scopes are considered recursively. This approach is similar to references to overloaded methods in object-oriented programming languages, where the 'nearest' definition is used up the inheritance hierarchy. Of course, it may happen that a name cannot be resolved or a name is ambiguous. For a concrete system, rules may be established to devise globally unique names.

Scope Definition

The definition of a scope consists of several parts: component selection, interface and attribute definitions, actions, and update rules. Implementation issues are not specified here. Defining a scope makes it part of the descriptive scope graph, deployment is a second step described on page 89.

```
scope-definition ::=
     "DEFINE SCOPE" component-name "AS"
     component-selection-clauses
     scope-feature-clauses

component-selection-clauses ::=
  component-selection-clause [ component-selection-clause ]

component-selection-clause ::=
     [ component-identifier ":" ]
     ( super-selection | member-selection )
     [ "WHERE" boolean-expression ]
     [ ":" selection-property-clause ]

super-selection ::= "SUPERSCOPE"
     selection-qualifier
     "FROM" ( scope-ref-commalist | "*" )

member-selection ::= [ "MEMBER" ]
     selection-qualifier
     "FROM" ( component-ref-commalist | "*" )

scope-feature-clauses ::=
     scope-feature-clause [ scope-feature-clause ]
```

```
scope-feature-clause ::= (
     ( component-identifier ":" selection-property-clause ) |
     interface-clause |
     role-clause |
     set-clause |
     action-clause |
     update-clause )

selection-property-clause ::= "{"
     [ interface-clause ]
     [ action-clause ]
     "}"

boolean-expression ::=
     ( attribute-test | interface-test | role-test | ... )
     [ ( "OR" | "AND" ) boolean-expression ]

attribute-test ::=
     attribute-name
     ( numerical-comparison | string-comparison | ... )

numerical-comparison ::= numerical-operator number
string-comparison ::=
     ( string-comparison-op | string-matching-op ) string
```

Component selection determines superscopes and members of the defined scope if it starts with SUPERSCOPE or MEMBER, respectively. Any selection consists of two steps. First, a base set of components is given after the *FROM* keyword, and the *where* clause selects in a second step those satisfying a boolean expression.

The base set can be given as an enumeration of specific components like in

```
DEFINE SCOPE example AS ALL FROM prod1, prod2, scope1
```

which defines a scope *example* that contains exactly the components *prod1*, *prod2*, and *scope1*. Or specific components and members of other scopes can be mixed.

```
DEFINE SCOPE temp AS
ALL FROM MEMBERS(world), A, B
WHERE has-temp-sensor = 1
```

defines a scope *temp* containing those components of the predefined scope *world* plus *A* and *B* which have an attribute *has-temp-sensor* set to one. The star $*$ is a special scope name available for template definitions. It is later replaced with the superscopes and siblings of the current scope when it is deployed. It denotes all components visible at deployment time.

The *where* clause is a boolean expression on component attributes and acts as filter. The expression tests individually each of the components given in the from clause, no pairwise comparisons of components are done here. If the where clause is omitted, a scope is defined containing exactly the specified list of components.

The same syntax is used for selecting superscopes. The following definition additionally specifies the superscopes *S1, S2* of *temp*.

```
DEFINE SCOPE temp AS
m: ALL FROM MEMBERS(world), A, B
   WHERE has-temp-sensor = 1
s: SUPERSCOPES ALL FROM S1, S2
```

A component identifier is a name that is valid only within the scope definition. It denotes each component included by the selection it is prepended to. The names do not correspond to nodes in the scope graph, they rather identify selections for later references, for example, when updating or refining a scope definition. In the above example, *s* refers to S1 and S2 and *m* refers components selected by the first selection clause.

Selection Qualifier

So far, *all* components matching the where clause are selected for the new scope. However, sometimes a comparison and ranking of eligible components is necessary. For example, the administrator may want to select those that are nearest to a specific location or have the most free computing resources. A *selection qualifier* is part of the selection:

```
selection-qualifier ::=
     ( ALL |
       ( TOP "(" attribute-name "," number ")" ) |
       ( "[" [ number ] ".." [ number ] "]" ) )
```

The default qualifier is *ALL* (as in the previous examples). *TOP* performs a top-k selection of all components satisfying the where clause. It sorts components by the given attribute and chooses the first k of them. A qualifier of the form $[n..m]$ specifies the size of the respective selection. A minimum of n matching components up to a maximum of m are choosen here. Either boundary can be omitted, denoting a cardinality of zero and as many as possible, respectively. Omitting both is like choosing *ALL*.

Many extensions are conceivable at this point. The top selector may take a predicate as argument to evaluate expressions like '*fixed-location - location-attribute*' which sorts according to a distance metric. Other domain-dependent functions may be added in specific implementations.

Interfaces

Component interfaces are defined as part of the scope feature clauses after all selections. Selective and imposed interfaces are specified in selection property clauses, which are either appended to the respective selection clauses or are also given after all selections (see below). The *interface* clause begins with the keyword 'INTERFACES' and then includes a comma separated list of interface specifications. There is no specific filter model preset in the language (cf. Section 3.3) and so a syntax corresponding to the available filter model must be choosen.

```
interface-clause ::= [ "INTERFACES" interface-commalist ]
interface ::= ( "INPUT(" | "OUTPUT(" )
     [ "0" | "1" | channel-interface | topic-interface |
       typebased-interface | content-interface | ... ]
     ")"

channel-interface ::= channel-name-commalist
topic-interface ::= topic-commalist
topic ::= "/" topic-name [ topic ]
typebased-interface ::= notification-type-name-commalist
content-interface ::= boolean-attribute-expression-commalist
```

The two special interfaces '0' and '1' denote filters rejecting and accepting all notifications. The following snippet defines a scope that outputs temperature alarm notifications, but it does not receive any input from its superscopes S1 or S2.

```
DEFINE SCOPE temp AS
ALL FROM MEMBERS(world)
WHERE has-temp-sensor = 1
SUPERSCOPE ALL FROM S1, S2
INTERFACES OUTPUT(AlarmNotification)
```

The next example is an extension that also includes imposed interfaces on the components of *temp* that allow them only to send temperature notifications. All other kinds of input or output traffic of members is prohibited.

```
DEFINE SCOPE temp AS
m: ALL FROM MEMBERS(world)
   WHERE has-temp-sensor = 1
SUPERSCOPE ALL FROM S1, S2
m:{
  INTERFACES OUTPUT(TempNotification), INPUT(0)
}
INTERFACES OUTPUT(AlarmNotification)
```

or alternatively

```
DEFINE SCOPE temp AS
ALL FROM MEMBERS(world)
  WHERE has-temp-sensor = 1 : {
    INTERFACES OUTPUT(TempNotification), INPUT(0)
  }
SUPERSCOPE ALL FROM S1, S2
INTERFACES OUTPUT(AlarmNotification)
```

Note that omitting a component interface is like setting it to '0', whereas omitting a selective or imposed interface is like setting it to '1' (cf. Section 3.3.1).

Coupling Points

Coupling points generalize component selection. Coupling points are queries on available components and their properties. They are half edges in the scope graph that describe dependencies on other components based on properties like component interfaces, roles, or attributes.[17] The dependencies must be resolved at deployment by creating the necessary edges in the scope graph.

A coupling point either provides or demands a specific property. If it demands, the coupling point of matching components must provide the required properties, and vice versa. So far, where clauses request for attributes and interface clauses provide interfaces. What is still needed are means to set attributes, to require interfaces, and to set and require roles. References to the following grammar rules are already part of where clause and scope feature clause.

```
interface-test ::= [ "HAS" ] interface
role-test ::= "IS ROLE(" role-name ")"

role-clause ::= "ROLES" role-name-commalist
role-name ::= symbolic-name

set-clause ::= "SET" set-attribute-commalist
set-attribute ::= attribute-name "="
    ( value | notification-attribute | component-attribute )
```

The *set* clause supports setting scope attributes to constant values as well as to values of notification or components declared in the update clause of the scope (see below).

The next statements defines two scopes *admin* and *company*. The latter includes one instance of the former due to its role definition. It imposes an output interface so that only notifications conforming to the `holidayAnnouncement` type can be passed into *company*. The latter also includes the top ten components, termed *worker*, that either produce or consume other important notifications.[18]

[17]Dependencies on attributes can subsume the other two if a sufficiently rich data model is available.

[18]Actually, two distinct clauses should select producers and consumers to avoid getting only one kind of components.

```
DEFINE SCOPE admin AS
ALL FROM c1, c2, c3
INTERFACES INPUT(something), OUTPUT(else)
ROLES boss

DEFINE SCOPE company AS
b:[1..1] FROM world
  WHERE IS ROLE(boss) INTERFACES OUTPUT(holidayAnnouncement)
worker:TOP(experience,10) FROM world
  WHERE OUTPUT(necessaryInformation) OR INPUT(furtherProcessing)
SET name = "Acme, Inc."
```

Actions

Scopes put components into groups for visibility purposes, but they can also perform actions on notifications and components. Scope features like mappings and transmission policies are functions executed on notifications.

```
action-clauses ::=
     ( map-clause | policy-clause | do-clause )
     [ action-clauses ]

map-clause ::= "MAP" ( "INWARD" | "OUTWARD" )
     ( "{" set-attribute-commalist "}" |
       external-code-ref )

policy-clause ::=
     ( delivery-policy | publication-policy | ... )
     [ policy-clause ]
```

The *map* clause defines a mapping which is either inward or outward, transforming incoming or outgoing notifications, respectively. If only one direction is specified, the other one must be derivable or prohibited by interface. Mappings may be defined within the specification language, but most likely externally provided functionality will be used as implementation. So, the map clause includes a reference to external code, which could be a symbolic name that refers to a repository of the notification service or a URL to an external code repository. For the same reason there is no syntax for defining transmission policies, they are supposed to be externally provided, too.

```
do-clause ::= "DO" command
```

The *do* clause is included as hint for future extensions, which is not used so far. It may provide a way to customize scope functionality or even to apply code to all members of the scope. The latter is sketched in [272] for a scenario of wireless sensor networks: application code is assigned to network nodes based on scoped definitions.

Updates

The update clause defines event-condition-action (ECA) rules to adapt instantiated scopes. Any kind of (application-specific) event visible to the scope can be used in these rules. There are special event types like `pub(F(n))`, which is the publication of a notification n conforming to filter F, `sub(F)`, which is the event of some component subscribing to the filter F, etc.

```
update-clause ::= "UPDATE ON" event
     [ condition ]
     DO action-commalist

event ::=
     ( ( "pub(" |
         "con(" ) notification-identifier ":" interface ")" ) |
       ( "sub(" | "unsub(" | "adv(" | "unadv(" ) interface ")" ) |
       join(C,S) | leave(C,S) | ... )

condition ::= "IF" boolean-attribute-expression

action ::= scope-change | create-clause

create-clause ::=
     "CREATE NOW"
     [ INCLUDE COMPONENT [ component-identifier ] ]
```

The notification identifier is a symbolic name valid within the scope definition. It is bound to the actual notification triggering the action and can be used in other parts, e.g., in the set clause to update scope attributes.

Actions comprise the alter scope statement explained below and creation rules. The create clause is a powerful tool to control the dynamics of scope graphs. It defines rules to automatically create predefined scopes when specific events occur. Because this automatic creation can be combined with join actions, new scopes can be created with the publisher of the triggering notification as first member of the scope. INCLUDE COMPONENT joins the component that triggered the action. This is the producers or the consumer of a notification (consuming a notifiation is considered as an event here), the component changing its interface, etc.

In this way session scopes can be defined. They include the initial publisher, all consumers, and consumers of subsequently produced notifications. The condition of the ECA rule controls the extension of such a dynamic scope—a precondition to implement spheres of control or transaction contexts in event-based systems.

Deploying Scopes

Scope definitions extend the descriptive scope graph of the system. It is like defining a class or type in a programming language, it does not create an instance of the subject. An instance of a scope is created and deployed with the following statement.

```
scope-deployment ::= "DEPLOY SCOPE" scope-ref
     [ component-selection-clauses ]
     [ scope-feature-clauses ]
     architecture-clause

architecture-clause ::=
     ( brokerscope-clause | intergrated-routing-clause | ... )

brokerscope-clause ::= "BROKERSCOPE(" host ")"
```

To deploy a scope, an existing definition and an implementation is necessary. The architecture clause lists scope architectures, which are introduced in Chapter 4. Essentially, it refers to a scope implementation available in the system. It carries implementation-specific parameters, like a host name for a brokerscope implementation.

```
DEFINE SCOPE temp AS
a: ALL FROM *
   WHERE has-temp-sensor = 1

DEPLOY temp
SUPERSCOPE ALL FROM S
a:{ INTERFACES OUTPUT(TempNotification) }
BROKERSCOPE(localhost)
```

This example defines a scope containing all members of S that have temperature sensors. The scope is deployed in an existing scope S using a brokerscope implementation on host *localhost*. It also adds imposed interfaces on selection a permitting only temperature notifications.

Changing Scopes

An ALTER SCOPE statement is introduced to change any part of a scope. It may refer to a definition as well as to an instantiated scope.

```
scope-change ::= "ALTER SCOPE" scope-ref
     [ "ADD" | "DEL" ]
     [ component-selection-clause ]
     [ scope-definition-clauses ]
```

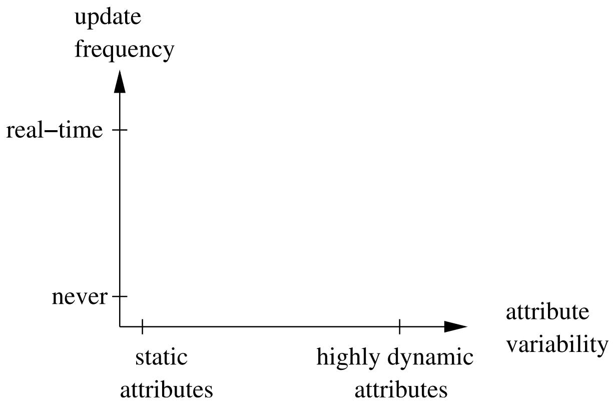


Figure 3.13: Scope definition accuracy

The statement adds new selections or features to an existing scope, or deletes or replaces existing parts of it.

```
ALTER SCOPE temp ADD
ALL FROM c
SUPERSCOPES ALL FROM S
```

The above statement adds a component c to the scope *temp* and joins it to S, i.e., $temp \lhd S$.

Maintenance and Definition Accuracy

The where clauses of component selections are rules that determine to which of the available components edges are established in the scope graph. But when are these rules evaluated? Once at deployment? Every t seconds? Or if attributes deviate by more than 20%? Figure 3.13 sketches alternative views on the accuracy of scope definitions.

The degree of correlation between the rules expressed in the where clauses and the currently established connections in the scope graph is called *scope definition accuracy*. It depends on the variability of attributes and the frequency with which rules are reevaluated.

In this thesis, queries are assumed to be evaluated at deployment time only and that their result is not automatically updated afterwards. This corresponds to the lower left point in Figure 3.13. However, the update clauses in scope definitions allow system engineers to install custom ECA rules to maintain accuracy.

3.9 Discussion

This chapter has presented the scoping concept for event-based systems. Based on an analysis of the deficiencies pointed out in the previous chapter, visibility is introduced as first-class concept. Visibility is the key aspect underlying all the open issues described in Section 2.5. Scoping is an effective means of controlling visibility and makes the interaction between groups of components explicit and manageable. A scope delimits a group's internal from outside traffic, and scope interfaces control any crossing traffic. This delimitation follows the basic engineering principles of information hiding and encapsulation. Scoping is invisible to the composed constituents and it facilitates design and implementation of reusable event-based components.

The graph of scopes reifies application structure, and besides being a design tool, it offers the ability to tailor the actual operation of a system. Notification mappings transform notifications between different representations when they pass scope boundaries, which facilitates system integration in heterogeneous environments. Transmission policies refine semantics of notification dissemination, both within a scope and with respect to the outside. The flexibility and extensibility of the scoping concept is shown with session scopes. They use scopes to group notifications into sessions, identifying the relationship between consecutive, dependent notification. Transmission policies and interfaces are used to distinguish sessions and to make notification processing session-local without modifying the participating components. Scopes offer a components model for event-based systems, and together with the proposed development process and the SQL-like scope language they open new ways of building event-based systems.

The scoping concept is defined in a formal specification. It extends the specification of simple event systems given in the previous chapter. Furthermore, a correct implementation is sketched, which illustrates the feasibility of the concept.

The contributions made in this chapter are the following. For the first time, visibility in event-based systems is investigated as a central concept. Controlling the visibility is made possible with scopes and this chapter has shown that a wide range of functions can be assigned to scopes. System engineers are provided with a tool to design and implement event-based systems. Scopes foster system design by reducing the inherent complexity, and they exert fine-grained control on event-based communication without degrading the loose coupling of components. Futhermore, scopes can be tailored to refine notification delivery semantics and to evolve system functionality within well-defined boundaries. In fact, scopes address each of the initially stated deficiencies. Side effects are obviously controlled by scope interfaces; they are a tool for both design and implementation; the management of event-based systems is made easier because scopes offer a place to localize management tasks; and security issues can be tackled by scope admission tests for new members and by mandating special quality of service requirements for any implementation.

Chapter 4

Scope Architectures

Contents

4.1 Architectural Choices **94**
4.2 Collapsing Scope Graphs **106**
4.3 Scope Address **116**
4.4 Scopes as Event Brokers **119**
4.5 Integrate Scoping and Routing **122**
4.6 Combining Different Architectures **134**
4.7 Discussion **137**

The concept of scopes so far defines visibility constraints, but is independent from any specific implementation. This chapter investigates and classifies principal approaches to implement scopes. They are distinguished by the characteristics of underlying communication media and by the strategies for scope graph distribution. The respective combinations are called *scope architectures* as they determine the layout of the implementation. All these architectures implement the visibility constraints defined by the scope graph, but diverge in their extensibility and their support for other quality of service parameters, like communication reliability and performance.

The first section introduces the architectural dimensions and analyzes their basic implications on scope graph implementation. This characterization is used to compare the detailed descriptions in subsequent sections. Sections 4.2 to 4.4 present a number of different architectures: i) collapse a scope graph and map it to filter mechanisms of the underlying communication medium; ii) instantiate scopes as administrative components and implement graph edges with the underlying filter mechanisms; iii) instantiate scopes as brokers; iv) distribute the aforementioned scope brokers. Section 4.5 integrates the scoping concept with a content-based routing framework as given in Rebeca. While the presentation looks at the respective architectures separately till then, Section 4.6 finally

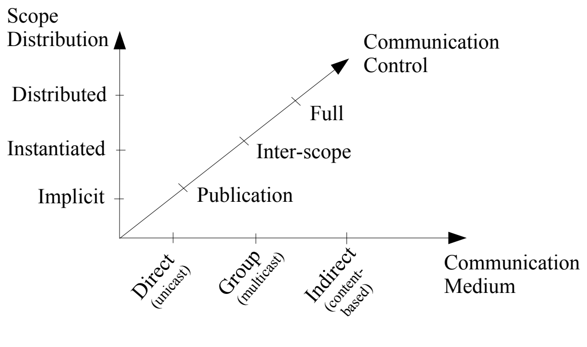


Figure 4.1: Design dimensions of scope architectures

elaborates on combining different scope architectures, before 4.7 concludes the chapter.

4.1 Architectural Choices

The concept of scopes can be implemented to target any of a wide range of diverse requirements. The implementation influences the functionality and quality of service an application can count on. The architectures presented in this chapter cannot be ranked in general; they may fit the needs of an application or not. There is no best architecture.

Two architectural dimensions are distinguished (Figure 4.1): communication medium and scope implementation. The combination of these dimensions give rise to a number of *scope architectures* that determine the principal layout of the scoped event service (cf. Fig. 4.3 on page 100 and Section 4.1.4). The third dimension turns out to classify the architecures' ability to control communication. This section details the architectural choices and defines a metric for comparing the architectures presented in this chapter.

4.1.1 Communication Medium

The notion of a *communication medium* denotes any technology that is used to convey notifications between nodes of the scope graph. The communication medium is the basic building block of scope implementation and determines which scope features are supported directly, which features can be implemented efficiently on top, and which features are hardly achievable at all. Any means of data sharing and transport can act as communication medium, ranging from shared memory and TCP [287] connections to IP-multicast [91] and Peer-to-Peer networks [245, 291]. Moreover, existing publish/subscribe services, database management systems [138], and tuple spaces [140] are also eligible candidates for implementing scope graphs. They offer different quality of service and determine the flexibility and functionality of a scoped event system beyond visibility rules.

Although within a single scope different kinds of traffic might be conveyed on top of different communication media, a single medium per scope is assumed for simplicity here; please refer to Section 4.6 for a general discussion on combining media and scopes.

In the following, communication media are differentiated according to their support for unicast/multicast delivery and their addressing capabilities. These are not orthogonal dimensions, they highlight different technical aspects that affect scope implementation.

Unicast vs. Multicast

The basic distinguishing feature of communication media is whether they forward data point-to-point or point-to-multipoint, i.e., unicast versus multicast. *Unicast media* send data directly to a specific, identified receiver. In order to reach a number of recipients the send operation must be repeated. Examples include TCP, RPC, and messaging systems. Perhaps surprisingly, unicast media are viable implementation techniques for certain classes of event-based systems; they are considered as a medium to implement scopes, the producer's and consumer's view (API) on the notification service remains unchanged. *Multicast media* send data to groups of receivers. Multicast media like shared memory, IP-multicast, existing notification services, and database tables are common implementation techniques that intuitively correspond to the characteristics of notification distribution.

Obviously, multicast media distribute notifications more efficiently than unicast media. On the other hand, multicast limits the ability to distinguish recipients and control the actual set of receivers. Scope features like delivery and security policies, which are meant to re-introduce control, cannot be implemented directly on top of multicast media without additional filtering (cf. *client-side filtering* in Section 4.1.3). Exploiting the knowledge about scope members enables system engineers to shape traffic, implement advanced transmission policies, encrypt data, etc. At the cost of multiple send operations and the need to maintain the current set of scope members, unicast media are more flexible than multicast media. In practice there are applications for both unicast and multicast media and the main issue is a tradeoff between efficiency of data distribution and addressing granularity.

Direct, Group, and Indirect Addressing

Communication media can be further distinguished according to their addressing schemes. While unicast media use *direct addressing*, which identifies an individual receiver uniquely in the network, multicast media can be subdivided into group addressing and indirect addressing. In *group addressing* data is sent to a named group of recipients. The name of the group is specified by the sender and all members of the group get messages sent within. Group membership is handled separately via membership protocols. IP Multicast and group communication protocols [250] are examples of this form of communication.

In *indirect addressing*, the second form of multicasting to a set of receivers, no destinations are specified. Instead of naming groups of receivers, the set of receivers is determined indirectly with the help of information given in messages and by potential receivers. For instance, content-based routing delivers notifications according to consumer-provided filters that test notification content. Another example is proximity group communication [206, 251], where messages are sent only to receivers that are physically close by, i.e., addressees are implicitly determined by location meta data.

Communication Media, Pub/Sub, and Visibility

The choice between unicast and multicast media is mainly a tradeoff between efficiency and control, as described above. But what media are good candidates to implement a publish/subscribe service and do some of them even offer a visibility mechanism comparable to scopes? What are the characteristics of group and indirect addressing that influence the implementation of scopes?

As for the general applicability to implement a pub/sub API, group and indirect addressing is related to the discussion on filter models (channel-, subject-, and content-based filtering) given in Section 2.1.3. Group addressing is like channels in that a name representing a set of receivers is used by the sender to disseminate data. Subject-based addressing is an extension that allows for subgroups [233, 292], which is, to some extent, also supported by IP-Multicast [209].

Group-based multicast media establish visibility constraints in that they encapsulate intra-group traffic. Notifications published within a multicast group, or under a specific subject, are a priori not visible to outside consumers. However, groups classify messages either based on content (all notifications of type A) or based on application structure (all database servers in a company's back-end infrastructure). Furthermore, groups are often not able to reflect the acyclic scope digraph, because they are mostly arranged in trees, as in IP-Multicast and subject-based addressing. Even if one tries to model different viewpoint with the help of subgroups, the exponentially growing number of necessary groups limits practical applicability (see Section 2.1.3).

Scopes, on the other hand, are orthogonal to consumer subscriptions, they handle interfaces (i.e., subscriptions, group names, etc.) and system structure (the organization of scopes in the scope graph) independently. Thus, groups do not directly implement scopes.

Indirect addressing media can avoid many of the problems of group addressing. They are typically more flexible, but less efficient as they do not easily map to hardware supported multicast mechanisms. In the generic form, like in content-based pub/sub, DBMS-based implementations, and tuplespaces, they are able to carry different viewpoints (content vs. structure) simultaneously. Available products/prototypes are able to offer only a few[1] of the features of

[1]For an in-depth discussion on related work see Section 6.

scopes, but they are an ideal basis for their implementation.

4.1.2 Scope Distribution

Considering individual scopes, there are three basic choices of how a scope can be realized: *implicit* with all the control in the local event brokers of members; *instantiated* with an explicit administrative component that represents the scope and is responsible for membership control, transmission policies, and mappings; and finally, the implementation of a single scope can be *distributed* on multiple administrative components residing in different nodes of the network.

Note that similar alternatives exist for the scope graph. Implicit scopes imply an implicit scope graph, administrative components can either be centralized in a single node or run on different nodes of the network (centralized or distributed scope graph), and distributed scopes imply a distributed scope graph.

Implicit Scope Implementation

The first approach is to collocate scoping with application components. The implementation is shifted into the communication library used to connect application components to the notification service, i.e., into the local event brokers in Rebeca terminology (cf. Sect. 2.4). The local event brokers use the addressing and filtering capabilities of the underlying communication medium to implement scope boundaries. The main idea is to annotate notifications to carry scope graph data. Extended subscriptions then exploit these annotations to filter not only on the original consumer's interest, but also on visibility constraints imposed by the scope graph. Consider, for instance, a scope graph with unique scope names, local event brokers that annotate notifications with scope names ($n.scope$ = *'MY-SCOPE'*) and modify each original subscription F to $F' = F \wedge n.scope =$ *'MY-SCOPE'* + *interfaces*.

The extended subscriptions F' must be mapped to the medium's filter capabilites, which is possible if expressive filter models are available like in the Java Message Service or in Rebeca. If this mapping is not possible, client-side filtering must enforce the visibility constraints to guarantee that all requirements of the safety condition of scoped event systems are met, cf. Def. 3.3 on page 47 and see Section 4.2 for details.

For example, consider the members of a scope forming a group that communicates notifications via a group-addressing medium like subject-based publish/subscribe to all scope members. This floods all notifications to all members of this scope, postponing original subscription processing to the client side. If content-based filters are available, processing of both client subscriptions and scope interfaces can be shifted into the medium; the former F' could be supplied to JMS or Rebeca.

In an implicit scope implementation the structure modeled by the scope graph is transformed into a flat implementation, as illustrated in Figure 4.2. Every component is connected to the same medium and conventions must de-

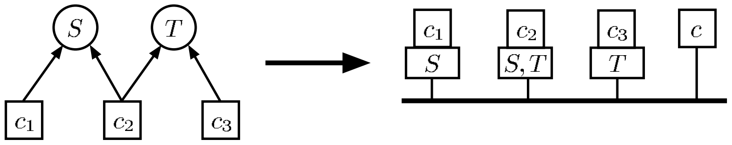


Figure 4.2: An implicit implementation shifts visibility control into application components.

termine how visibility constraints are implemented on top of the addressing mechanisms offered by the medium. In order to meet the safety and liveness conditions, each component must maintain the necessary management information about the layout of the scope graph and the current scope interfaces. So, scoping structure can be transparently implemented in the local event brokers without modifying application code, but scope graph changes require update processing in potentially many of the components.

The problem of shifting scope control into local event brokers is that components not adhering to these conventions may bypass visibility constraints, both as consumer and as producer. Since the scope structure exists only implicitly in the components of the system, no external entity controls and enforces scope boundaries, giving rise to both reliability and security concerns. Consumers might arrange to listen to notifications they are not intended to receive, and even worse, they may send notifications to any component, disrupting correctness in other parts of the system as well. Moreover, more advanced features of scopes, namely transmission policies and mappings, are even harder to implement using an implicit implementation.

Instantiated Scope Implementation

To exert more control on notification dissemination the scope graph must be managed within the notification service infrastructure. A basic approach is to explicitly instantiate administrative components to represent scopes. They are generated and controlled by the notification service itself and contain an implementation of scopes outside of application components.

This scenario is further subdivided into a centralized graph and a centralized scope form. The former implements the whole scope graph in a single node of the distributed system and amounts to a central information hub. This is a widely used approach for implementing unscoped event system, because it simplifies notification routing and access control, but comes at the expense of scalability and diversity support. Examples range from centralized databases [138, 236] (see Section 4.2.4) to content delivery networks [261], which can be seen as logically centralized nodes optimized for one-way delivery efficiency. In the centralized scope form, each scope is represented by one administrative component, but each such component may run on a different node in the network.

Administrative components make the scope structure explicit and accessible to the system engineer, who is now able to customize (parts of) it to the local needs of an application. This approach facilitates configuration and integration of heterogeneous components on a per scope basis as each administrative component may act as bridge between different implementations (different data/filter models, communication medium, etc., see Section 4.6).

In contrast to an implicit solution, instantiated scopes make it easier to control adherence to a specific scope graph and it relieves clients from management tasks.

Distributed Scope Implementation

A single, distributed scope consists of multiple administrative components that together constitute this scope. Each scope member is assigned to one administrative component. The same type of communication medium is still assumed for delivery to scope members, but communication between the administrative components may be based on a different technique. Scalability is obviously improved since multiple administrative components share and subdivide the load to distribute intra-scope notifications; they may even exploit effects of locality when notifications are only forwarded within one administrative component.

For example, consider two groups of application components belonging to the same scope, but located at two different border brokers of the underlying network, e.g., an internet of two LANs connected by a WAN. Instantiating a scope implementation solely in one LAN would diminish the benefits of locality for the other side. But if administrative components are available on both sides, they may draw on a local broadcast medium and connect each other using a point-to-point link.

4.1.3 Example Architectures

Figure 4.3 shows possible architectures that are defined as specific combinations of scope distribution and communication medium. They are sketched in the following and the major approaches are described in sections 4.2 to 4.5.

Static Deployment. The combination of implicit scope implementation and point-to-point communication leads to a *Static Deployment* where every scope member knows its siblings and communicates directly with them. When subscriptions are known to all members, notifications are sent to subscribed consumers only. Otherwise, notifications are sent to all scope members, which evaluate their own filters on any notification published in the scope. Output interfaces towards superscopes and input interfaces of sibling subscopes must be known as well so that cross-scope notifications can be sent to a consumer in the destination scope that in turn relays them within. This scenario is called static deployment since it is an eligible architecture option if the scope graph is static, rather small, and does not change at runtime. System configuration can then be compiled into

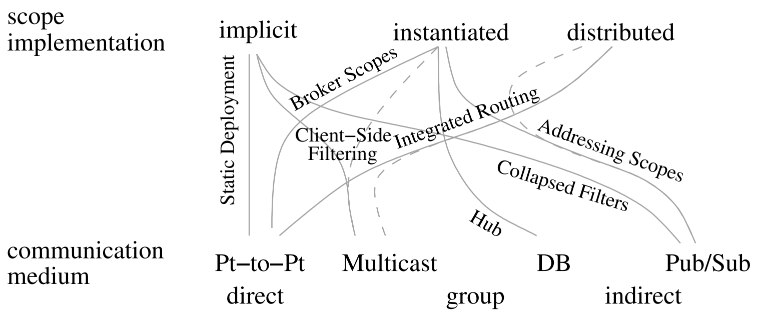


Figure 4.3: Scope architectures combine scope graph implementation and communication media.

the local event brokers without affecting the pub/sub API, as it is for any of the presented architectures. In such a situation even remote procedure calls are a suitable implementation technique to convey notifications. If the system is not static the necessary configuration data in the components must be kept up to date. Examples of this approach are data-driven coordination languages (e.g. Manifold [238, 239]), which connects input/output ports of coordinated entities, and even an implementation using TCP/IP to connect the participants is eligible, particularly if system footprint has to be kept small. Interestingly, the JavaBeans programming model [77, 278] and component-oriented programming in general [186, 285] are related to this approach in that they facilitate the wiring of interfaces and ports.

Another application of static deployment is to wrap a callback-based system with a pub/sub API. That is, undirected subscriptions are resolved and directly registered at corresponding callback handlers that are visible according to the locally stored scope graph. Although somewhat unusual, this might offer a way to draw from existing request/reply or directed messaging systems, when possible, and from their established benefits, for instance, in security and transactional data management.

Client-Side Filtering. The *Client-Side Filtering* architecture also utilizes an implicit scope implementation but is built on a multicast medium that provides group-based addressing, like IP-Multicast. Each scope is assigned a multicast group address and all members of a scope are reached with only one call to the medium. Compared to Static Deployment the required network bandwidth is considerably reduced. However, since there are still no administrative components the visibility constraints defined by the current scope graph must be enforced on producer and/or consumer side. As described in Section 4.1.2, the local event brokers may annotate notifications and must select appropriate destination group addresses on producer side. And on consumer side incoming notifications must be filtered out so that in combination only matching notifications are delivered that comply with the scope graph and satisfy the safety condition of the scoped event systems definition.

A different way of using group-based multicast here is to group according to content instead of structure. In such a scenario multicast groups might be used to group subscriptions, which is the common use of multicast in publish/subscribe systems [81, 235]. Consumers would have to determine the visibility of incoming notifications by evaluating the interfaces of the scope graph as part of their client-side filtering. Thus, in the first approach producers have to know the current scope graph layout to select the correct destination scopes, while in the second approach consumers are in charge of this. The two approaches differ mainly in the selectivity of the grouping and the implied costs of keeping the graph information up-to-date.

Another extension is to instantiate administrative components within scopes that are responsible for relaying incoming and outgoing notifications. In this way the need to store the full scope graph in local event brokers is removed, since these relaying components have to know their adjacent nodes only.

Client-Side Filtering is obviously applicable when scope graphs are rather static and of limited size. For instance, if scope graph changes are just induced by moving simple components the assignment of group addresses to scopes remain unchanged. The moving components have to join the respective groups but the scope graph information need not be updated elsewhere. Scope graph management is thus reduced to group membership management, which is provided by the communication medium. Nevertheless, this architecture is left out of consideration in favor of more flexible solutions.

Collapsed Filters. In the *Collapsed Filters* architecture visibility constraints of the scope graph are merged into subscriptions issued by consumers. This leads to a flat notification service where enhanced subscriptions implement the scope graph implicitly, requiring an expressive subscription like in content-based pub/sub. This architecture is described in Section 4.2.

Central Hub. The 'classic' data management approach of using a central database may also be beneficial in an event scenario. It is an alternative implementation of collapsed filters and is able to offer sophisticated quality of service guarantees in addition to the basic safety and liveness requirements of scoped event systems (cf. Section 4.2.4).

Addressing Scopes. *Addressing Scopes* is an extension of the client-side filtering approach that no longer relies on multicast but on content-based publish/subscribe. Announced subscriptions are prepended with an additional term that filters on the (necessarily) unique names of the scopes the subscriber has currently joined; refer to Section 4.3.

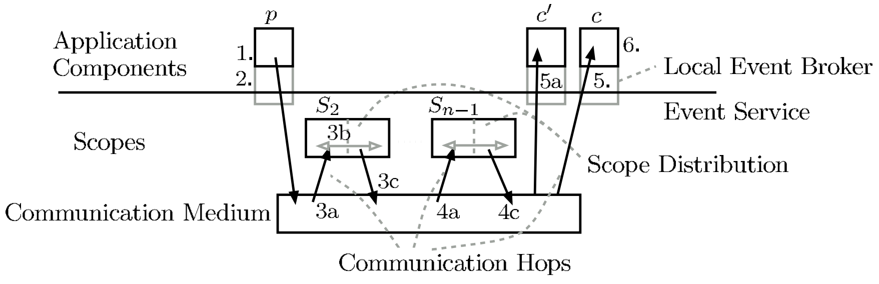


Figure 4.4: Steps of scoped notification delivery

Broker Scopes. *Broker Scopes* are a one-to-one implementation of the scope graph in that each scope is explicitly represented by an event broker of the broker network (cf. Section 2.4). This approach is detailed in Section 4.4.

Integrated Routing. *Integrated Routing* fully integrates scoped notification delivery into the routing infrastructure. The routing tables themselves are extended to reflect visibility constraints of the scope graph. This architecture is described in Section 4.5.

4.1.4 Scope Graph Distribution—Types of Architectures

While the choices described above consider individual scopes only, the following looks at scope graph implementation as a whole. The general processing steps of scoped notification delivery are described, which identify potential places to implement scoping functionality in the system. These steps serve as a basis to compare the preceding example architectures and to classify them in three types of architectures. These types differ in the degree they support scope graph reconfigurations, transmission policies, and, in general, any distribution control beyond scope interfaces.

Figure 4.4 sketches the delivery in a scoped event system. The numbered course shows the forwarding of a notification that moves along an exemplary delivery path $(p, S_2, \ldots, S_{n-1}, c)$ between producer p and consumer c in an arbitrary scope graph.

1. In the first step a notification is published by producer p.

2. The access to the event notification service is provided by the local event broker, which is conceptually part of the application component. The broker may process the notification as part of an implicit scope implementation (cf. static deployment) before it is forwarded by accessing the communication medium.

3a. If scopes are instantiated in administrative components, the notification is delivered to an instance of S_2 of the example delivery path.

3b. If scopes are distributed, the notification is also sent to other instances of this scope if needed.

3c. Delivery in S_2 is completed when the notification is forwarded towards its members and superscopes, accessing the underlying medium for the second time.

4. The previous three steps are repeated for all other scopes.

5. The notification is received by the local event broker of the potential consumer, which may again process and filter the notification before it is delivered to the consumer.

6. Finally, the notification is delivered to the consumer c.

An implementation of scope graphs may stretch across up to three layers: On the lowest layer, the communication medium is parameterized to distinguish scopes or at least administrative components representing scopes. On the middle layer explicit administrative components implement scope features within the notification service. At the highest layer, code is collocated to application components in local event brokers to modify notifications and subscriptions.

The figure illustrates all possible steps although only a subset is relevant for a specific architecture. In an implicit scope implementation no scopes are instantiated within the event service and steps 3 and 4 are omitted. With a centralized scope implementation step 3b is not needed. Whether any processing is done in the local event brokers (steps 2 and 5) depends on the concrete implementation, but it is definitely required in implicit approaches. When group-based multicast is used to address all members of a scope additional client-side filtering is also needed in step 5.

The different choices to partition scope implementation among these steps turn out to be a fundamental characteristic of scope architectures. It determines their ability to adopt scope graph changes and to implement any sophisticated control of communication beyond interfaces. For an assessment it is crucial to compare the amount of control residing within the notification service with the amount shifted into the communication medium and the application components, respectively. For this purpose the number of accesses to the communication medium that are necessary to forward a notification along a delivery path is taken as a measure to distinguish architecture types. These accesses are labeled as communication hops in Figure 4.4, whereas communication between instances of the same distributed scope (step 3b) is not counted as it does not leave the scope's sphere of control. Based on this consideration, three modes of notification forwarding are identified and depicted in Fig. 4.5.

	Publication Control	Inter-Scope Control	Full Control
# Accesses	1	$n-2$	$n-1$
Possible Medium	any	group or indirect	direct
Scope Distribution	implicit	central./ distributed	central./ distributed
Data Flow Control	no explicit control	control inter-scope traffic	control every edge
Examples	Static Deployment, Client-Side Filtering, Collapsed Filters	Addressing Scopes	Broker Scopes, Integrated Routing

Figure 4.5: Types of architectures, their characteristics, and examples

1. Publication control
   All consumers are reached with only one access to the medium. All interfaces and delivery policies bound to the scope graph must therefore be evaluated within the communication medium or as part of the local event brokers of producers and consumers. There is no control within the medium or the publish/subscribe service infrastructure once the message is sent.

   Accessing the communication medium means here that all eligible consumers in the whole system get the notification.

2. Inter-scope control
   In this approach, scopes are represented by administrative components that govern the interfaces towards superscopes and relay incoming and outgoing notifications if they match the respective input and output interfaces. Within scopes, however, lists of members are not maintained and notifications are not directed to specific addressees. A multicast medium is used that may reach all scope members in one step. Since producers do not distinguish any siblings, the consumers' subscriptions must either be completely handled by the communication medium or, if all scope members are indistinctively addressed as a group, consumer-side filtering must be applied.

   Accessing the communication medium means here that a scope and all of its members get the notification. For an arbitrary delivery path, one access to the communication medium is needed for every edge, except for the root scope of the path where sending and receiving components are siblings. This leads to $n - 2$ calls to the communication medium for a path of length n.

3. Full control
   Each scope is represented by an administrative component and notifications are forwarded strictly along the edges in the scope graph, resulting in $n - 1$ accesses to the medium for a delivery path of length n. Each scope is implemented in one or more brokers in the routing network. Delivery is controlled even within a scope.

   This is an one-to-one implementation of the scope graph and accessing the communication medium means here that notifications are sent to the next hop node in the scope graph or only within one scope graph node that resides on multiple network nodes (e.g. integrated routing).

This classification describes what part of the scope graph is offered through the communication medium and the implicit implementation in application components, on the one hand, and what part is implemented in administrative components instantiated in the infrastructure, on the other hand. This distinction determines how the different number of accesses to the communication medium determines the ability of a scope architecture to adapt the current configuration of the system. While explicit administrative components are readily adaptable, it is far more difficult to update infrastructure code in a consistent and transparent way when it resides in local event brokers.

An even more important fact is that the granularity of the control exerted on notification distribution gets inevitably more coarse if fewer accesses to the medium are needed. With fewer accesses more consumers are reached in one step, which implies uniform delivery to larger sets of non-discriminated components. However, any form of refining and controlling dissemination will have to differentiate subsets of these components. And the number of accesses to the medium characterize how much of the structure identified in the scope graph is reflected in the implementation.

4.1.5 Comparing Architectures

Scope architectures can be classified in the architectural dimensions given above. However, further criteria are necessary for comparing and assessing their functionality from an application point of view. The architectures presented in the next sections are compared according to the following criteria:

- Impact on infrastructure and components
  What must be changed to implement scoping?

- Implementation overhead
  What is the overhead implied by a given scope architecture? What are

the communication costs compared to unscoped pub/sub and compared to other scope architectures?

- Reliability
  How do failures of components affect single scopes or overall system correctness?

- Reconfiguration
  What kinds of changes of the scope graph are possible in the running system? What are the costs of scope graph updates? Adaptability and flexibility to change system structure are the main issues here.

- Customization
  While all scope architectures obey the visibility constraints expressed in a scope graph, which of the other features of scopes are supported? What kinds of mappings, transmission policies, security policies, etc. can be established?

The comparison of the scope architectures is summarized in Figure 4.6.

4.2 Collapsing Scope Graphs

This section details the collapsed scope graph type of architecture, which extends a previously unscoped infrastructure to obey visibility constraints without introducing administrative components. A layered approach is taken which implements scopes on top of arbitrary existing communication media. The following concentrates on utilizing a content-based pub/sub service, however, database management systems (DBMS) or broadcast media plus consumer-side filtering are also eligible, as sketched at the end of the section.

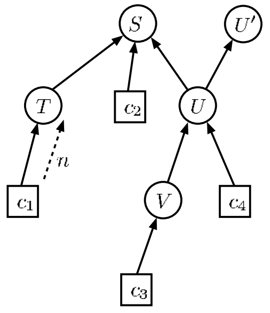
Scopes are not treated as first-class objects in this type of architecture. In terms of Figure 4.4 on page 102, steps 3 and 4 are not present; there is no explicit forwarding in scope components. Producers and consumers are linked to the event system via local event brokers and in principle any consumer is reachable with only one access to the underlying medium. Since no administrative components are assumed in the infrastructure the implementation of the scope graph has to be collocated with local event brokers, i.e., as a layer on top of the existing system. These brokers are provided with additional processing and filtering capabilities to enforce visibility constraints. In this sense, the layout of the scope graph is collapsed in these places.

4.2.1 Collapsing Filters

The layered scoping implementation has to ensure that received notifications are only delivered to consumers if the notifications are visible and conform to a delivery path defined in the scope graph. The extra filtering necessary to

	impact on				ability to	
	infrastr.	components	overhead	reliability	reconfigure	customize
Collapsed filters	+	−	○	−	−	−
DB	+	○	○	+	○	+
Static deploy.	+	−	+	○	−	○
Addressing Scopes	+	○	○	○	+	○
Broker Scopes	○	+	○	+	+	+
Integrated Routing	−	+	+	+	+	+

Figure 4.6: Comparison of scope architectures



(a) Example Graph

$$\begin{aligned}
c_1 &: \quad n.scopes = \{T, S, U, V\} \\
c_2 &: \quad F_2' := F_2 \wedge S \in n.scopes \\
c_3 &: \quad F_3' := F_3 \wedge V \in n.scopes \\
c_4 &: \quad F_4' := F_4 \wedge U \in n.scopes
\end{aligned}$$

(b) Scope Destinations

Figure 4.7: A collapsed sample graph and explicit destinations

implement the visibility constraints of a scope graph is subdivided into two parts:

- Producers, i.e., their local event brokers, may preprocess notifications, add data necessary for scoping/filtering purposes, and in case of direct or group addressing they also have to select the appropriate destination address before publishing the notification.
- Consumers have to subscribe/register for the appropriate destination address they want to listen to and must locally evaluate remaining filters on content and management data that is not encoded in the subscription or the destination address.

The basic question is how the responsibility of selecting valid delivery paths is split between producers and consumers. In general four approaches are conceivable, which are differentiated with the help of the exemplary delivery path (c_1, T, S, U, V, c_3) of Figure 4.7(a).

1. The producer selects all consumers, that is, all complete paths. This is the approach taken by the Static Deployment architecture and amounts to a point-to-point communication between producers and consumers (see p. 99).
2. The producer determines all destination scopes in which a notification is visible. It analyzes all delivery paths except for the last hop to simple components.
3. The producer finds the topmost scopes of all delivery paths. It has to analyze the upward parts of delivery paths while consumers are responsible for testing downward parts. This refines the notion of *visibility roots* introduced in Section 3.2.7.

4. The producer just marks the origin of the notification and leaves testing of valid paths to consumers.

The above distinction is applicable to many different communication media. For instance, addressing destination scopes is obviously an example of group communication and both classic group communication systems as well as subject-based publish/subscribe may be used for implementation. However, the discussion of points 2 and 3 below is restricted to content-based publish/subscribe for it illustrates the solution in the context of this thesis best. For simplicity, all components are assumed to have a local copy of the whole current scope graph for the time being; a detailed analysis is given below.

Scope Destinations

This option requires producers to analyze the scope graph to determine all destination scopes. The routing decision is thus completely drawn into the producers. Figure 4.7(b) illustrates the necessary annotations made in producers and consumers. Every notification takes a scope attribute (*n.scopes*) that carries the list of scope names that are crossed by valid delivery paths. These names must be unique in the set of scopes implemented on the same 'shared' communication medium. Conversely, subscriptions have to be adapted before they are issued into the underlying pub/sub service. They test for the presence of the names of the consumer's scopes.

An algorithm to search for destination scopes is given in Figure 4.8. It follows all potential delivery paths as long as the output and input interfaces on upward and downward paths match the specified notification, respectively. While consumers just have to add a new predicate to all their subscriptions to test whether a notification is supposed to be visible in their scope, computation in producers is expensive.

Handling of transmission policies is included in the algorithm, partially. The publishing policy pp_C is correctly applied in the first part, whereas the delivery policy dp_T in the second part does not distinguish between internal delivery and incoming notifications. idp_T would have to be applied in the first run of the second group of *foreach* loops, where the upward paths stored in *list* start to go downward with an internal delivery in the considered scope T.

Visibility Roots

In the third of the above alternatives producers only check for the upward part of eligible delivery paths and annotate notifications with all reachable superscopes. To accomplish this they have to recursively analyze the output interfaces of all superscopes, which is the first half of the aforementioned algorithm. On the other hand, consumers have to test the interfaces of the downward part of potential delivery paths by extending their subscriptions or by applying filters locally (see Figure 4.9). In combination, the responsibility for filtering is more evenly split: producers specify the upward part of eligible delivery paths in notifications and

```
program $DetermineScopeDestinations$
input: scope graph $\mathsf{G} = (\mathsf{C}, \mathsf{E})$, publisher node $p$, notification $n$
begin
  $list \leftarrow \{(p, n)\}$
  foreach $(C, n) \in list$ do
    foreach $(C, S) \in \mathsf{E}$ do
      if ${}^{o}\hat{I}_C^S(n) = n' \neq \epsilon$ and $S \in pp_C$ then
        $list \leftarrow list \cup \{(S, n')\}$
      endif
    endfor
  endfor
  $list \leftarrow list \setminus \{(p, n)\}$
  foreach $(T, n) \in list$ do
    foreach $(S, T) \in \mathsf{E}$ do
      if ${}^{i}\hat{I}_S^T(n) = n \neq \epsilon$ and $S \in dp_T$ then
        $list \leftarrow list \cup \{(S, n')\}$
      endif
    endfor
  endfor
end
```

Figure 4.8: Compute list of scopes in which a notification is visible

$$
\begin{aligned}
c_1 : &\quad n.scopes = \{T, S\} \\
c_2 : &\quad F_2' := \begin{cases} F & \text{if } S \in n.scopes \\ \epsilon & \text{otherwise} \end{cases} \\
c_3 : &\quad F_3' := \begin{cases} F & \text{if } V \in n.scopes \\ F \circ {}^{i}\hat{I}_V^U & \text{if } U \in n.scopes \\ F \circ {}^{i}\hat{I}_V^U \circ {}^{i}\hat{I}_U^S & \text{if } S \in n.scopes \\ \epsilon & \text{otherwise} \end{cases} \\
c_4 : &\quad F_4' := \begin{cases} F & \text{if } U \in n.scopes \\ F \circ {}^{i}\hat{I}_U^S & \text{if } S \in n.scopes \\ \epsilon & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.9: Collapsing with visibility roots

consumers test whether there is a valid downward path from one of the specified superscopes.

Producers do not even have to specify all of its reachable superscopes explicitly. Only those are needed that characterize the eligible delivery paths and facilitate correct filtering of the downward parts. A scope can be omitted if the notification is visible in its superscope and the effective input interface permits a downward forwarding. Note the difference between routing in a graph and the visibility decision made here. In the former notifications never travel back on an originating link, while in the latter case visibility in a superscope implies visibility in a subscope if the interfaces match. These arguments refine the notion of visibility roots introduced in Section 3.2.7. They were defined to be the topmost scopes visible from a certain node, but so far used without scope interfaces. Generally, a notification has to be delivered if producer and consumer share a common visibility root *and* if the notification is not blocked by interfaces.

When considering a notification published at a specific node, the topmost scopes this notification can reach are visibility roots. The set of all reachable superscopes of a component p publishing n is

$$\sigma_p(n) = \{S \in \mathcal{S} \mid p \overset{\star}{\lhd} S \wedge p \overset{n}{\rightsquigarrow} S\}$$

and the set of visibility roots $V_p(n)$ include

$$V_p(n) \supseteq \{S \in \mathcal{S} \mid \nexists T \in \sigma_p(n).\, S \lhd T \wedge {}^{o}\hat{I}_S^T(n) \neq \epsilon\}$$

Additionally, if a scope S publishes n upward, but prevents its downward forwarding, the respective superscopes are not visibility roots of any delivery path below S. Therefore,

$$V_p(n) = \{S \in \sigma_p(n) \mid \nexists T.\, S \lhd T \wedge {}^{i}\hat{I}_S^T({}^{o}\hat{I}_S^T(n)) = n\} \tag{4.1}$$

A scope S is a visibility root of component p with respect to a notification n if there is no superscope of S, if there is no superscope into which n is forwarding, or if n is forwarded the input interface would prevent downward delivery, in which case testing for downward delivery paths in the consumers would accidentally reject n. Consider the above example graph in Figure 4.7(a). A delivery path (c_3, V, U, c_4) between c_3 and c_4 is described by the root U of the path. If the interface ${}^{o}\hat{I}_U^S$ between U and S matches a published notification n in both directions, S would be the visibility root $V_{c_3}(n)$, which means the filtering of the path (S, U, c_4) is no different from only (U, c_4).

The adaptation of subscriptions is as follows. Every simple component x has to extend its subscriptions to include a filter sequence for every path $(S_1, \ldots, S_m, c)$ from a visibility root S_1 to consumer c. For any such path the necessary filter extension is

$$F_p := \begin{cases} F & \text{if } S_m \in n.scopes \\ F \circ {}^{i}\hat{I}_{S_m}^{S_{m-1}} & \text{if } S_{m-1} \in n.scopes \\ \cdots & \\ F \circ {}^{i}\hat{I}_{S_m}^{S_{m-1}} \circ \ldots \circ {}^{i}\hat{I}_{S_2}^{S_1} & \text{if } S_1 \in n.scopes \end{cases} \tag{4.2}$$

The extended subscription must take into account all possible downward paths, there may be even multiple different paths from the same visibility root. The new collapsed filter F' that is effectively applied is built as follows:

$$F' = \bigvee_p F_p \tag{4.3}$$

Note that the above expression is still subject to considerable simplification as different paths most likely share common subsequences.

Client-Side Filtering

The last of the above options to collapse filtering of delivery paths is to leave all the filtering to the clients. This is the client-side filtering approach mentioned at the beginning of this chapter. Producers tag notification with their own ID/ name and consumer subscriptions have to be extended to filter both the upward and the downward part of delivery paths. Obviously, this approach has only a limited applicability.

4.2.2 Filtering Costs in a Pub/Sub Implementation

The preceding discussion assumed the availability of a content-based notification service like Rebeca so that extended filters can operate on annotated notifications. Especially in the visibility root approach most filter processing is shifted into the middleware. The visibility defined in the scope graph is mapped to the routing and filtering capabilities of the underlying pub/sub service, which has to evaluate interfaces, but may also draw from optimizations like covering and merging.

The subscriptions resulting from the presented algorithms seem to be rather complex and costly to evaluate, but there exist a number of proposals for filter optimization that are applicable here. For instance, Handurukande et al. make filters broader to cover more base filters by omitting single predicates from the boolean filter expression [150]. This could be utilized within the Rebeca routing framework to govern and detect mergers of filters. In this way, filters of adjacent scope members are easily abstracted and a set of hierarchical filters are created, which first test visibility according to scope interfaces and later include tests of individual component subscriptions. Another option is to consider adapting the matching algorithm itself to better accommodate to the prevalent combinations of visibility predicates and application subscriptions. Variations of the well-known counting algorithm [104, 303] would diminish the problem of multiple testing of stacked input filters. The algorithm does not iterate through a list of (complex) filters, it rather determines which atomic predicates match to finally compare the number of matched with the total number of predicates in each filter. Since each different predicate is evaluated only once, the costs of testing input interfaces is drastically reduced in contrast to the naïve approach of iterating through the list of filters.

Another possibility to reduce filter complexity is to mandate certain restrictions that ease filter evaluation and composition: restrict the expressiveness used in scope interface filters (forcing the consumers to do a client-side filtering with the 'remaining' expressiveness); restrict the data model, and thus the filter model to allow for more efficient covering tests and merging, cf. typing and type inclusion tests [298].

Filtering of incoming notifications in consumers is only necessary if the underlying communication medium is not able to implement the above filter expressions. Note that the effective interfaces used in the algorithm of Figure 4.8

simplify the presentation and may even include notification mappings. An implementation on top of an existing pub/sub service likely requires to separate filtering and mapping. While easily possible with destination scopes, visibility roots require the filtering of the downward path to be combined with any mappings on the path, or the consumer-side filtering has to be applied.

The processing of filters in consumers need not be done within the application components itself. Filter processing may as well be moved from the local event brokers to the border brokers of the routing network; at least if the implementation of the pub/sub service is accessible, this is an application- and deployment-specific optimization conceivable for embedded systems and small devices. To reduce the costs of the client-side filtering itself the set of eligible delivery paths can be narrowed by precomputation based on static cover and overlap tests of advertisements, subscriptions, and interfaces. Consider Figure 4.7(a) and c_1 annotating notifications with visibility roots. If the input interface ${}^{i}I_U$ of U prevents any notification published by c_1 and c_2 from passing down into U, the enhanced subscriptions F_p of c_3 and c_4 do not need to take ${}^{i}\hat{I}_U^S$ or any notifications visible in S into account, which may reduce filtering load considerably. Conversely, if an output interface is known to block all notifications of a certain producer, its search for reachable superscopes can be statically confined to a smaller subgraph.

4.2.3 Coping with Graph Updates

The aforementioned analysis of active advertisements and interfaces as well as the creation of collapsed filters itself rely on up-to-date information about the current configuration of the scope graph. Furthermore, for reasons of simplicity components were so far supposed to store the complete current scope graph, which is, of course, not always necessary. The issue of handling updates of the scope graph is closely related to the question: what parts of the graph do simple components have to know at all. In principle, components need to store and observe scope graph updates only along their possible delivery paths. The following updates must be handled.

1. Addition and removal of leaves
   Newly added components join existing scopes and do not influence existing components.

2. Addition and removal of filters of existing edges
   The scope graph structure is not changed, only the effective interfaces assigned to the edges change.

3. Addition and removal of edges
   The actual layout of the scope graph changes arbitrarily.

The amount of information that must be distributed upon graph changes increases with each of the three steps, and it is different in the scope destinations

(SD) and the visibility roots (VR) approaches. The first point, when only the leaves of the scope graph are allowed to change, is the simplest one. In the visibility roots approach existing components are not influenced by these changes at all. And in the scope destinations approach, the input interfaces of newly added scopes must be propagated along all delivery paths from which matching notifications may be received. In this way, the producers in the affected scopes can update their local copy of the scope graph. To find the affected scopes the new input interface is recursively tested against input interfaces of superscopes and thence to output interfaces of the respective subscopes. This is similar to the algorithm of Figure 4.8 that finds the destinations in all delivery paths except that now two interfaces are compared. If they overlap testing continues, but if they accept disjoint sets of notifications the search for affected scopes need not follow the considered edge because notifications are blocked at this point anyway.

The administrator, who creates new scopes must know the complete graph and publishes the new interface with the scope destination $n.scopes$ set to the list of affected scopes. The local event brokers of each producer, on the other hand, subscribe to and transparently process these interface announcements to update their copy of the scope graph. The administrator also has to initialize newly added simple components with that part of the current scope graph configuration that contains the possible delivery paths originating at the new component.

The second kind of graph updates, namely the change of existing scope interfaces, also do not necessarily require components to store the complete scope graph. In contrast to the first point, the distribution of graph changes is not limited by the currently existing interfaces but by the original visibility definition $v(X, Y)$, cf. Section 3.2.2. Updates are confined by the visibility of components and not by the visibility of notifications, which may change when interfaces are adapted. When using visibility roots, producers need to know the output interfaces of superscopes and consumers about input interfaces of superscopes. So, producers and consumers subscribe to the respective change announcements, which need only be distributed downward in the graph. When using destination scopes, producers need to know the output interfaces on upward parts of delivery paths and the input interfaces on downward parts. This can be exploited to approximately halve the number of affected components.

Furthermore, producers may decide to store two copies of the scope graph. One that is currently active and used to determine the destinations of published notifications, and one backup that holds the layout of the visible graph according to the definition of v, which may become relevant when changed interfaces open up new delivery paths that will then be included in the first, active copy of the graph. Of course, it is not imperative to store the backup copy of the visible (sub)graph in every producer. Even if the collapsed scope graph architecture does not instantiate explicit administrative components, an external repository of the current scope graph is conceivable that will serve the local event brokers with the required information.

The third point allows arbitrary changes of the scope graph. However, new edges may ‘open up’ new subgraphs, which alters the visible portion of the scope graph. Consequently, the scope graph cannot be pruned and the components must have access to the whole graph, in principle, but maintaining two different copies of the graph is still applicable, too.

4.2.4 Filter Aggregation in Databases

Database management systems are an alternative communication medium for storing and distributing notifications among a set of consumers. Database tables can be seen as a kind of multicast medium which carries notifications to consumers that query the table, that is, subscribe to notifications sent in the medium. In fact, database technology provides a wide spectrum of functionality [138] that may be exploited to extend the quality of service offered by the event system beyond the definitions given in chapters 2 and 3. On the other hand, there are drawbacks like their maintenance complexity, resource consumption, and acquisition and operation costs.

Collapsing the scope graph by extending subscription filters might well be implemented on top of database tables, blurring the distinction between the collapsed filter and the central hub scope architectures. Similar to the content based pub/sub medium assumed above, a database table can hold all published notifications and subscriptions are merely queries to this table. Moreover, the sequences of input and output interfaces evaluated to determine notification visibility resembles queries on views [148]. Scope S and all of the notifications visible inside can be considered to be collected in a database table of same name. The input interface of scope T and U is the equivalent to a view definition on table S in that it selects a subset of the data available in S; the same holds for V with respect to U. The other way round, a superscope can be thought of defining a view on the base data provided by its members. In this way, producers would publish in the scopes they have joined, which means they write into the respective database tables. Consumers query the tables of their scopes and the views that map notifications from super- and subscopes, respectively.

Database management systems seem to be a promising implementation technique for implementing collapsed scope hierarchies. Especially in environments of static, not necessarily small graphs and where advanced features like transactional processing and auditability are needed a centralizing database approach may be beneficial.

4.2.5 Evaluation

In collapsed scope graphs producers and consumers have to parameterize the underlying communication medium to establish visibility constraints. While being a simple implementation of scopes as a layer on top of an existing infrastructure, it does not provide the full control of visibility at runtime. Notification mappings and delivery polices are not always implementable. Furthermore, graph changes are difficult and costly to deploy, for application components are not easily reconfigurable and changes to the graph have to be consistently distributed to affected components. Collapsing the scope graph leads to a parameterization of simple components and of the communication medium, e.g., by annotating notifications with scope names, which have to be unique, and extending subscriptions of its members to filter on these names. This spreads control of the scope graph configuration, and in the end, visibility constraints are 'only' accomplished cooperatively, if both producers and consumers operate correctly.

The system's functionality in a collapsed graph depends on the correct function of *all* participating components. It renders control of the visibility to the components—library-based implementation in local event brokers or not, it's part of the client-executed code. A corrupted or malevolent component may publish or eavesdrop in any scope, eroding visibility constraints and eluding any security measures. The discussion on combining different scope architectures in Sect. 4.6 leads to a possible solution when gateway components bridge two separated subgraphs and provide a physical encapsulation of visibility constraints.

Comparing the presented scope destination and visibility root approaches, the former requires more processing in the producers, which are exposed to more graph changes, but it keeps the filter extensions necessary in the consumers simple. In the latter case both producers and consumers are expected to know and evaluate paths to visibility roots, which leads to more complex filters that, however, may delegate (parts of) the prerequisite filtering to the underlying medium.

4.3 Scope Address

This section describes the scope address type of architecture, which is still a layered approach on top of an existing publish/subscribe medium. In fact, scopes and all members of scopes are addressed as a group and so any communication medium offering a group-based multicast is eligible.

4.3.1 Addressing Scheme

Within a specific scope, notifications are sent without discriminating scope members—the scope is addressed. However, in contrast to collapsed scope graphs, administrative components are instantiated to handle inter-scope communication. They represent scopes and control the transition of notifications between scopes. In terms of Figure 4.4 on page 102, steps 3 and 4 are only used for inter-scope traffic while components that reside in the same scope will exchange data with only one access to the underlying medium, omitting steps 3 and 4. Furthermore, the reception of the data in steps 5 and 5a is not distinguished, i.e., senders do not control delivery.

Consumers, that is, their local event brokers, have to accomplish two tasks to implement visibility constraints. When joining a certain scope, they need to

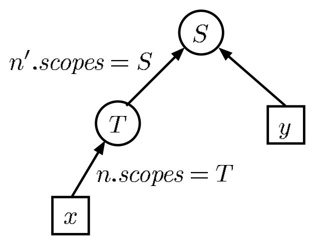


Figure 4.10: Notifications are tagged with the ID of *their* scope.

use the addressing capabilities of the medium to select the traffic visible in the scope. And additionally they are responsible for filtering received notifications so that only those matching the original subscriptions are actually delivered. Just as in the collapsed scope graph with scope destinations, consumers are coerced to filter all visible notifications locally if the medium supports only group addressing. If the medium is capable of indirect addressing, consumers may enhance subscriptions and move filter processing into the medium. For instance, using a content-based publish/subscribe service, the principal idea is to annotate notifications with the ID of the scope into which they are published.[2] Of course, these IDs have to uniquely identify the scope in this medium. In Figure 4.10 component x publishes a notification n by sending it with an additional attribute $n.scopes = T$, and it should be visible in its scope S. This attribute is utilized by consumers to select notifications of their respective scopes in the first place. If component y is about to subscribe to F, this filter is extended to $F' = F \wedge n.scope = T$, which is issued in the underlying pub/sub service.

The administrative component of T implements the scope's interfaces and relays incoming and outgoing notifications if they match the input and output interfaces, respectively. It acts as consumer in S and T. It extends the input and output filters of the interfaces and issues ${}^{i}F'_{T} = {}^{i}F_{T} \wedge n.scopes = S$ and ${}^{o}F'_{T} = {}^{o}F_{T} \wedge n.scopes = T$. In this way, the admin components get external notifications matching the input interfaces and internal notifications matching the output interface. In the example, the admin component of T relays n by publishing a modified n' with $n'.scopes = S$, acting as a regular producer in S and as a bridge between the internal and external distribution 'domains.'

4.3.2 The Resulting Overlay Network

Communication in the scope graph is realized as an overlay network here. It relies on the underlying communication medium, which does not heed any visibility constraints but the addressing/filter specifications given by simple and administrative components. The application level routing of notifications by

[2]Note that this annotation is transparently done in the local event brokers and neither controlled nor seen by producers or consumers.

the administrative components governs the distribution with the abstract scope graph and is unaware of the physical layout of the underlying communication network. The admin components may be instantiated at arbitrary access points to the medium and they thus implement the scope graph as overlay network on the medium.

On the other hand, scope-internal traffic is not explicitly controlled and the scope implementation does not influence routing. It relies on a location transparent addressing scheme offered by the filter model of the underlying dissemination infrastructure. This offers the advantage of easier reconfiguration, but it also makes controlling QoS parameters harder due to the lack of knowledge about the physical structure. Performance and reliability issues may be raised by effects of distribution.

Moreover, the layout of the scope graph is not (necessarily) stored in the admin components. Since all components, simple and administrative, access the same communication medium, no admission to join a scope is necessary; the administrator can just instruct a specific local event broker to join a scope so that it adapts publications, subscriptions, and local filtering respectively. In this case, the ability to implement delivery policies in a certain scope is limited to cooperative filtering done in all scope members, as in collapsed scopes. Even without knowing the children, an explicit scope structure offers a number of advantages. Reconfigurations of the scope graph need not be propagated through the whole graph, because they affect only the admin components of the modified edge or node. Also, any notification mappings can be applied and they are easier to handle in the admin components than in any simple components.

Of course, an implementation could also mandate that components make themselves known to their scopes when joining. This would extend the amount of control that admin components exert on communication, moving towards the scope brokers presented in the next section. The implementation of arbitrary delivery policies would be simplified since notifications, in principle, could be directed to individual consumers, if they create additional filters on their own name. Interestingly, the admission could be used to enforce access control, that is, security policies. Even encrypted intra-scope communication is possible if all internal communication is encrypted with a temporary session key. This key is installed anew at authorized components whenever scope members join or leave. For the distribution of the key a public key infrastructure must be available.

4.3.3 Evaluation

The scope address type of architecture introduces administrative components that localize the implementation of interfaces, publishing policies, and mappings. They offer a finer control of inter-scope communication than the collapsed scopes approach and also avoid most of the complexity and inflexibility inherent to collapsing filters in simple components.

However, scoping is still implemented on a shared multicast medium and the implementation is not aware of the underlying network layout. In fact, intra-scope communication is not directly governed by the admin components and relies on the filtering capabilities of the communication medium. The local event brokers of producers and consumers modify notifications and subscriptions before sending them out. With respect to intra-scope communication scope addressing is similar to collapsed scopes. Internal delivery policies, admission to scopes, and, in general, conformance to the visibility defined in the scope graph is achieved only if producers and consumers operate cooperatively and correct.

Compared to the collapsed scopes, which need only one access to the medium to reach every consumer, the administrative components repetitively access the medium to forward a notification along a delivery path in the scope graph. In situations where some consumers are connected via long delivery paths, this approach apparently induced a considerable communication overhead. But the indirection introduced by the administrative components relieves simple components from maintaining the current graph structure. Especially the last point touches a well-known tradeoff between scalability and expressiveness [59]. In the collapsed scope graph approach lots of extended filters are issued, whereas with scope addresses the filter complexity is limited at the expense of increasing communication bandwidth.

4.4 Scopes as Event Brokers

The broker scope approach is the most general implementation of scopes. It uses administrative components representing scopes, as before, but relies on their forwarding even for intra-scope communication. It directly implements the structure of the scope graph in the sense that publishing within a scope first requires to access the communication medium to send the notification to the representing scope instance, which, in the second step, sends the notification to all its children and, after applying the output filters, to the eligible superscopes. In terms of Figure 4.4 on page 102, all the steps are explicitly implemented. With brokering each notification individually, even the delivery of notifications to separate consumers could be distinguished in steps 5 and 5a. The existence of step 3b depends on the internal implementation of each scope representative, of course.

The characteristics of this approach are the independently operating administrative components that represent each scope and have full knowledge about adjacent subcomponents and superscopes. And, in principle, a point-to-point communication between the nodes is assumed so that arbitrary delivery can be implemented in scopes. In practice, a number of different communication media and schemes for implementing and locating administrative components are possible.

4.4.1 One Scope, One Broker

The simplest form is a one-to-one implementation of the scope graph, which instantiates exactly one administrative component per scope and uses point-to-point media to convey data as defined by the edges of the graph. The point-to-point communication to all children offers the full control of intra-scope traffic. Any constraint bound to the scope graph is easily implemented at this explicit point in the infrastructure: no restrictions of applicable transmission policies, mappings, and security measures are imposed.

From a technical point of view, an implementation with scopes as brokers is similar to the architecture described in Section 2.4, only that a strict tree-like network is no longer mandated. Instead, the undirected form of the directed acyclic scope graph constitutes the overlay network used to convey the data. The original restriction to trees was made to simplify analysis and implementation of general routing protocols, which is a reasonable initial assumption for a research prototype. Here, this restriction is removed. However, the problems inherent to arbitrary graphs are not solved in general, rather scoping and the definition of visibility constrains the possible routing configurations in the graph. The network layout is no longer an infrastructure independent of the application components, the administrator of the system is provided with means to shape its layout and control the distribution of notifications. Routing is the implementation of visibility, and the responsibility of ensuring sensible routing is now partially transfered to the administrator.

A possible drawback of this approach might be its degrading of communication efficiency. To convey data along a given delivery path of length n, $n-1$ accesses to the underlying medium are necessary, which is only one more than in the scope address approach. But if only intra-scope traffic is considered, which may dominate in many systems anyway, the necessary accesses are doubled. However, even if other implementation approaches may be more efficient for certain system configurations, broker scopes provide the most general implementation of scope graphs, and the ones most adaptable to any kind of reconfigurations. So, the alleged inefficiency has to be compared with the indirection of the scope brokers and the enhanced control they introduce thereby.

4.4.2 Distributed Scopes

The above discussion assumed a single administrative component per scope, which is responsible for filtering incoming and outgoing traffic and internal forwarding. With distributed scopes, this task is performed by multiple instances, that is, by distributed administrative components of one scope. Whenever the instances are not independent, they have to communicate with each other and thus implement step 3b of Figure 4.4 on page 102. For the communication between these instances a communication medium can be used that is different from the one conveying data between the scope graph nodes. However, the same arguments regarding addressing capabilities, scalability, and flexibility hold as before.

A number of objectives are achievable with distributed scopes. An obvious improvement is to instantiate multiple administrative components for each scope to prevent single points of failure. The instances may be identical replicas using a primary/backup approach [13] or operating in parallel independent of each other. Alternatively, each of the instances may be responsible for a different subset of the scope's components so that in case of failure only one subset is affected, but not all components of the scope. In these cases, a point-to-point communication within a known set of scope representatives is indicated.

Furthermore, scope distribution facilitates adaptation. For example, if one administrative component is instantiated per superscope, each instance handles the interfaces, mappings, and transmission policies with respect to one superscope. The addition of edges simply requires adding the respective administrative components. And if a multicast medium is used to forward notifications from scope members to all the admin instances, edge configuration does not even influence any other parties in the scope. Another option is to provide specialized services by different scope representatives for certain types of notifications, such as internal delivery policies or encryption for specific notifications. This implementation partially backs off the initially stated assumption that only one communication medium is used per scope. The same result could be achieved if each of the specialized admin components is created as a full scope in the scope graph.

The above examples employ separate administrative components to facilitate the implementation and reconfiguration of a scope graph, but they do not consider distribution with respect to the actual layout of the physical network. A very important aspect of distributed scopes is their ability to bridge between the structure of the application given in the scope graph and the structure of the underlying network. Consider a scope that groups physically dispersed members located in two different subnetworks. With a single administrative component all traffic would be centralized, whereas distribution helps exploiting locality. If an instance of the scope is present in each of the subnetworks, notification forwarding is decoupled and done locally in each network. And the bandwidth necessary between the networks can be reduced once the connected admin components remember the remotely published subscriptions, i.e., they maintain a routing table.

The previous description shows clearly that multiple explicit scope instances constitute a distribution network by itself. When several scopes are distributed, several of these overlay networks coexist. In this situation scoping and routing is mixed, which is investigated in Section 4.5.

4.4.3 Collocating Broker Scopes

A special solution is to collocate all administrative components at one node in the network. Scope-internal traffic still needs two accesses to the underlying medium, but all inter-scope communication is done locally. Although closely related to the central hub approach, cf. Section 4.2.4, the scope graph is explicitly instantiated here, only that inter-scope communication is implemented by interprocess communication (IPC). Separate admin components can still evolve independently, they just happen to be collocated, so to speak, to improve efficiency, auditability, or other global constraints.

4.4.4 Evaluation

Scopes as brokers are the most flexible implementation of the scope graph. They offer all features of the scoping concept and the flexibility to adapt all aspects of the one-to-one realization of the scope graph. Every feature is localized in the infrastructure. Apart from this configuration viewpoint, broker scopes make the infrastructure itself visible and adaptable, for it provides administrators with means to map application structure to infrastructure components, that is, to event brokers.

This scope architecture is possibly not the most efficient implementation of a certain scope graph, but the most generic one. It is not a service of the pub/sub infrastructure, but a way to define and adapt the infrastructure itself, and it will serve as a basis for refining the implementation of subgraphs, as discussed in Section 4.6. However, it is not always acceptable to have such a close correlation between the application structure supposedly encoded in the scope graph and the implied, dependent layout of the network infrastructure.

4.5 Integrate Scoping and Routing

The explicit instantiation of administrative components described in the previous section makes the full range of scope features available to system engineers, i.e., administrators. However, it also determines the layout of the underlying network infrastructure, which is no longer independent of the applications. In contrast, the following integrates scoping into the routing infrastructure. Visibility control becomes an inherent service of the event notification service and is no longer implemented as a layer above the underlying broker network.

4.5.1 Scopes as Overlays

Given a network of brokers and a scope graph, the simple components of a specific scope are in general connected to arbitrary border brokers, irrespective of their scope membership. They are reachable via a subset of the border brokers, and the notification service must ensure that notifications are forwarded to these brokers if they match one of the subscriptions of the respective simple components. Consider the exemplary scope graph and the broker network depicted in figures 4.11 and 4.12. Brokers B_1, B_2, and B_4 are part of scope T, they are *scope brokers*[3] of T. Together with B_3, they are also scope brokers of S. B_2 is

[3]Mind the difference between scope brokers and broker scopes. The former are part of an independent broker network and sustain a specific scope, whereas the latter is a scope

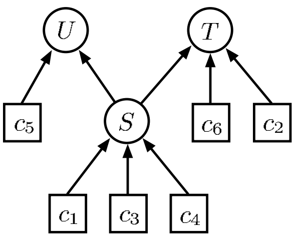


Figure 4.11: An exemplary scope graph

in both cases an intermediate broker that does, currently, not have any directly connected scope members. B_1 and B_5 are scope brokers of U.

The main idea is to rely on any of the existing routing schemes, e.g., those offered by Rebeca (see Section 2.4), as before, but to use it for intra-scope traffic only and for each scope separately. Still, the same broker network is used to route all notifications and a connected subset of brokers routes the traffic internal to a given scope without heeding other scopes. The separate routing for each scope effectively establishes *scope overlays* in the broker network, which are sketched in Figure 4.12. On the other hand, the separation of scope-internal routing necessitates a special handling of inter-scope transitions. In Figure 4.12, B_1 is scope broker of both S and U to bridge between the overlays of the two scopes.

Consequently, two kinds of routing are utilized to integrate scoping into the broker network: intra-scope within a specific scope and inter-scope routing between scopes adjacent in the scope graph. In intra-scope routing each scope

architecture and a different way to implement the scope graph (see Section 4.4).

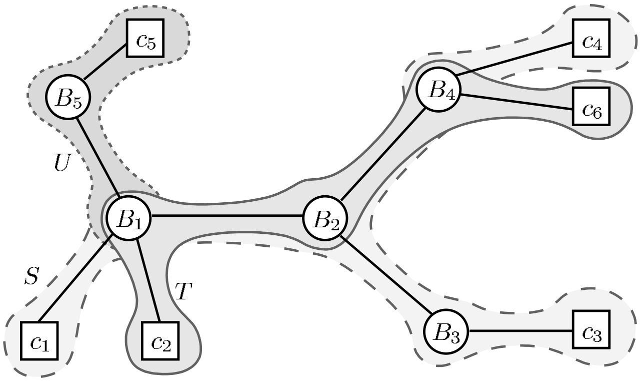


Figure 4.12: Scopes as overlays within the broker topology

Filter	Destination
${}^{i}I_{c_1}$	c_1
${}^{i}I_{c_2}$	c_2
${}^{i}I_{c_3}$	B_2
${}^{i}I_{c_4}$	B_2
${}^{i}I_{c_6}$	B_2
${}^{i}I_{c_5}$	B_5

Figure 4.13: A flat routing table for broker B_1

overlay maintains its own routing tables so that each broker has a routing table per scope it supports. The employed routing scheme maintains the independent routing tables and handles advertisements and notifications as before. Hence, brokers constituting a scope overlay behave like a traditional flat pub/sub service in which no visibility constraints exist. In inter-scope routing brokers must arrange for the transition of notifications between scope overlays according to the scope graph and the assigned interfaces and mappings. The current assumption is that two scopes $S \lhd T$ have to share at least one common scope broker to implement the scope graph edge at this point. In the previous example both B_4 and B_5 support scopes S and T, and both are able to let notifications cross the respective boundaries; the same holds for B_1 and S and U.

4.5.2 Enhancing Routing Tables

The original flat routing tables maintained in each broker contain filter-destination pairs that list issued subscriptions and the next-hop nodes from which they were received, describing the paths to consumers. Figure 4.13 shows the flat routing table RT_{B_1} of broker B_1 of the previous example. The *enhanced routing tables* subdivide these entries and group them in separate scope-specific tables $\mathsf{RT}^S_{B_1}$, $\mathsf{RT}^T_{B_1}$, and $\mathsf{RT}^U_{B_1}$ sketched in Figure 4.14. From the point of view of a specific scope S, both simple and complex components are entries in a scoped routing table $\mathsf{RT}^S_{B_1}$. Although technically equal, entries of subscopes are distinguished from entries of superscopes, which is necessary to correctly implement the visibility of components as described in the next subsection.

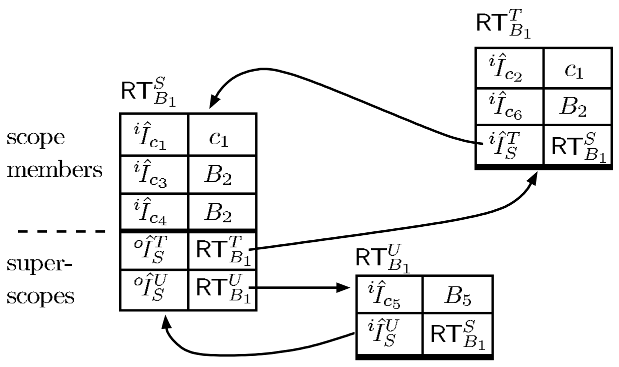
The 'Filter' and 'Destination' columns have still the same semantics as before: an entry indicates that notifications are to be forwarded to the given destination if they match the respective filter. In distinction to the original flat table, however, the new tables store arbitrary mappings instead of just filters. In this way the effective interfaces between components can be tested, including any mappings assigned in the scope graph. Of course, any implementation is free to still store simple filters separate from more complex notification processing functions. For instance, the filter-link pairs of the original routing tables may be transformed into triples of filter sequences and links plus mapping sequences.

The destinations stored in the enhanced tables are either network links or locally stored data structures. The former represents an implementation to communicate with next hop brokers and clients, the latter are the routing tables of next hop nodes in the scope graph. They mix and integrate the two levels of routing between physical brokers, on the one hand, and between scope overlays, on the other hand.

The scoped routing tables $\mathsf{RT}_{B_i}^{S_i}$ govern notification forwarding both within and between scopes, once set up properly. But in order to establish new edges in the scope graph and to create and link the respective routing tables, additional information must be maintained in the broker network. Each broker keeps a *scope lookup table* ST_{B_i} that contains pairs of scope identifiers and network links, indicating in which direction scope brokers of the specified scope can be found. These tables are updated upon scope creation and deletion, as discussed below. For the previous example they look like in Figure 4.15.

4.5.3 Setting Up Routing Tables

Once created, the routing tables are filled when consumers subscribe, and the underlying routing algorithm must forward and register these subscriptions. Chapter 2 described simple routing and covering and merging, which may be applied to accomplish this task. The scoped routing tables themselves and the references between them are set up as reactions to scope graph reconfigurations. In addition to the plain publish/subscribe primitives *pub*, *sub*, and *notify*, Section 3.2.5 on dynamic scopes introduced four new operations: *cscope*(S), *dscope*(S), *jscope*(X, S) and *lscope*(X, S), which create and destroy a scope S, and join X to scope S and remove it, respectively. While the network of brokers is still assumed fixed, the following describes how routing tables are adjusted to reflect these operations. Section 3.8 has suggested tools that support system engineers in this task.



$\mathsf{RT}_{B_1}^{T}$
$^{i}\hat{I}_{c_2}$ | c_1
$^{i}\hat{I}_{c_6}$ | B_2
$^{i}\hat{I}_{S}^{T}$ | $\mathsf{RT}_{B_1}^{S}$
$\mathsf{RT}_{B_1}^{S}$
scope members
$^{i}\hat{I}_{c_1}$ | c_1
$^{i}\hat{I}_{c_3}$ | B_2
$^{i}\hat{I}_{c_4}$ | B_2
super-scopes
$^{o}\hat{I}_{S}^{T}$ | $\mathsf{RT}_{B_1}^{T}$
$^{o}\hat{I}_{S}^{U}$ | $\mathsf{RT}_{B_1}^{U}$
$\mathsf{RT}_{B_1}^{U}$
$^{i}\hat{I}_{c_5}$ | B_5
$^{i}\hat{I}_{S}^{U}$ | $\mathsf{RT}_{B_1}^{S}$

Figure 4.14: Enhanced routing tables of B_1 incorporating scopes

ST_{B_1}	
S	B_1
T	B_1
U	B_1

ST_{B_2}	
S	B_2
T	B_2
U	B_1

ST_{B_3}	
S	B_2
T	B_3
U	B_2

...

Figure 4.15: Scope lookup tables

Adding and Removing Scopes

The primitive *cscope*(S) creates a new scope S if invoked by the system engineer at a specific broker B. If no scope of this name is known before, a new routing table RT_B^S is created and the scope lookup table ST_B is updated. By default the creation is announced as unscoped notification and every broker listening to these kinds of notifications updates its scope lookup table accordingly. If a new scope shall not be made publicly available but only as a member of a specific superscope T, the initial announcement can be postponed until it has joined T. The announcement is then sent within T and its visibility is governed by the installed interfaces. Without such restrictions the full list of all scopes instantiated in the system would be listed in all lookup tables, as it is the case for advertisements or subscriptions in flat pub/sub systems. Applying scope interfaces to restrict the distribution of scope announcements helps limiting the amount of management information kept in the system.

To complete the scope configuration, additional data about its interfaces, transmission policies, or security policies is necessary. This information is also provided by the system engineer and is stored as extension of its routing tables $\mathsf{RT}_{B_i}^S$ in all scope brokers.

A scope is removed from the system by calling *dscope*(S) at one of its scope brokers. Following the entries in its routing table a message is sent to all of its scope brokers to remove its routing tables and any references from routing tables of adjacent scopes. Its members are notified with a corresponding notification.

Joining a Scope

An arbitrary component C is joined to a scope S by calling *jscope*(C, S) at the local or border broker of C. The scope lookup table is used to route a ScopeJoin message to the first scope broker of S. These special messages leave a trail of temporarily stored source-pointers in the visited brokers that allows a response to be routed backwards to C. A scope broker of S that receives a ScopeJoin message takes two steps. It includes the border broker of C as scope broker of S and forwards the interface of the new component to existing scope brokers to get the routing tables updated. The first step requires that the current routing tables is forwarded along the stored trail towards C so that each visited broker creates an initialized routing table for scope S. If security policies are installed in the scope brokers, a join request may be denied, which results in a rejection sent towards C.

A simple component leaving a scope is similar to just unsubscribing to all issued subscriptions. Scope brokers may regularly test if any members are locally connected, and if other scope brokers are reachable via only one link, this scope broker is an unused border broker of the scope overlay and may be shut down. If a scope leaves one of its superscopes, i.e., $\mathit{lscope}(S,T)$, an appropriate message is distributed to the scope brokers of both scopes and the references to the respective other scope are removed from all involved $\mathsf{RT}^{S}_{B_i}$ and $\mathsf{RT}^{T}_{B_i}$ routing tables.

4.5.4 Scoped Routing

Scoped routing uses the enhanced routing tables to forward notification in accordance with the current scope graph. The algorithm basically extends the plain Rebeca algorithm of flat publish/subscribe routing. It is executed in each broker B and operates on a set of enhanced routing tables $\mathsf{RT}^{S_i}_{B}$ of scopes S_i, of which B is currently a scope broker.

Notification Layout

The algorithm needs some additional management information to operate properly. This information is annotated to notifications by the routing implementation and is not accessible to applications.

To prevent loops and infinite forwarding, notifications must not be sent back on links they were received from, both network links and scope graph edges. As in the original Rebeca routing, notifications are annotated in each broker with an identifier of the source network link to prevent it from being sent back in the direction from where it was received. Additionally, each notification carries an identifier of the current scope and of the source component, which are accessed by `get_scope`(n) and `get_source`(n). These identifiers signify the scope in which the notification is currently visible and the (last) component from where it was forwarded into this scope. Note that the latter does not name the original publisher but the last node in the scope graph visited before the current one. The local event broker of the original producer is responsible for setting the identifiers initially.

These component identifiers must be unique with respect to the current scope and its adjacent nodes in the scope graph so that they identify its components or superscopes unambiguously. However, such edge-wise distinct names may not suffice, because many scopes may be hosted in one broker and naming must be unambiguous within a broker. So, besides the simple but restrictive assumption of globally unique identifiers, a scheme similar to the mappings of virtual channel identifiers in Asynchronous Transfer Mode (ATM) networks [185] might be devised that maps identifiers on both sides of a network link to guarantee uniqueness.

The next paragraphs introduces different states of routing that are accessed by `get_state`(n). Of course, all get-functions are accompanied by the respective set methods.

Routing States

Following the discussion about delivery paths in scope graphs and transmission policies, three states of routing are distinguished:

- scope internal routing
  A notification is forwarded to siblings in the same scope.
- downward routing
  An incoming notification is forwarded to scope members.
- upward routing
  An outgoing notification is forwarded to superscopes.

A notification published by a simple component is initially handled in the internal routing state. It may alternate between internal and upward state, but once in downward routing it may not switch back. Adherence to this sequence is mandatory to not break the bipartite nature of delivery paths, that is, notifications are always first sent up in the scope graph before they solely travel down against the edges of the graph. Internal routing is expressly distinguished to facilitate the respective transmission policy, cf. Section 4.5.6.

The Algorithm

Figures 4.16 and 4.17 illustrate the algorithm, which basically extends the plain Rebeca algorithm of flat publish/subscribe routing and is executed in each broker B. The main control loop `main_loop` is triggered whenever new data is appended to the receiving queue, which may either be due to incoming network traffic or via cross scope traffic. The expected pair (n, l) contains the notification to be forwarded and a link from which it was received. The latter may be either a network link or a local routing table, i.e., a routing destination in the enhanced routing tables.

The procedure `scoped_routing` determines the next destinations of a notification currently visible in a scope S. It interprets the current routing state and accordingly queries different parts of the routing table RT_B^S. The function `subscopes`(RT_B^S) returns a routing table that contains all entries that point to a locally stored routing table of a subscope of S. Similarly, `superscopes`(RT_B^S) contains entries of local superscope routing tables. Conversely, `remote_components`(RT_B^S) returns the remaining entries, which are reachable via network connections. In case of $\mathsf{RT}_{B_1}^S$ of Figure 4.14, the three functions return entries of $\{\}$, $\{\mathsf{RT}_{B_1}^T, \mathsf{RT}_{B_1}^U\}$, and $\{c_1, B_2\}$, respectively. First, eligible destinations within the considered scope and then the locally available routing tables of subscopes are determined, both must be done for all routing states. A distinction of states

```
procedure main_loop
  loop
    // the queue is fed from network links
    $(n, l)$ = get_next ($recvQ$)
    scoped_routing($n, l$)
  end
end

procedure scoped_routing ($n, l$)
Input $n$:  notification
      $l$:  source link

  $s$ := get_state($n$)
  $S$ := scope($n$)

  — internal  routing
  $D$ := destinations($n$, remote_components($\mathsf{RT}_B^S$))
  foreach $(n', l') \in D$
    if  $l \neq l'$ then send $(n', l')$
  end

  — downward routing
  $D$ := destinations($n$, subscopes($\mathsf{RT}_B^S$))
  cross_scope($S, D$, "downward")

  — upward routing
  if  not $s$ = "downward" then
    $D$ := destinations($n$, superscopes($\mathsf{RT}_B^S$))
    cross_scope($S, D$, "upward")
  fi
end
```

Figure 4.16: overall routing algorithm

```
function destinations ($n, T$)
Input $n$:  notification
      $T$:  routing table
Output $D$: list of notification-destination pairs

  foreach $(I, d) \in T$
    $n'$ := $I(n)$
    if  $n' \neq \epsilon$ then
      $D := D \cup (n', d)$
    fi
  end
end
```

Figure 4.17: The naïve matching algorithm with mappings

is at this point only necessary when transmission policies are applied, cf. Section 4.5.6. Last, the upward direction is examined to find all locally available routing tables of eligible superscopes, which is only done if routing is not in downward state. Taken together, these steps follow the default delivery and publishing policies of Section 3.4.1 that describe visibility in the scope graph.

The above procedures rely on the function `destinations` to determine all eligible destination in the specified routing table. The naïve matching algorithm,

```
procedure cross_scope ($S, D, s$)
— forward all notifications to next routing tables
Input $S$: current scope
      $D$: list of notification-routing table pairs
      $s$: routing state

  foreach ($n, \mathsf{RT}_B^{S'}$) $\in D$
    if get_source($n$) $\neq S'$ then
      set_source ($n, S$)
      set_scope($n, S'$)
      set_state ($n, s$)
      put_in_front ($recvQ$, ($n, \mathsf{RT}_B^{S}$))
    fi
  end
end
```

Figure 4.18: Inter-scope forwarding

extended with mappings, is given in Figure 4.17 for illustrative purposes. It returns pairs of destinations and notifications to send there, allowing for a seamless integration of mappings in the routing decision. Of course, in practice more efficient matching algorithms, e.g. [104, 303], and a more sophisticated handling of notification copies may be applied.

4.5.5 Crossing Scopes

The scoped routing algorithm relies on `cross_scope` to forward a notification between scopes (Figure 4.18). It is responsible for relaying the current notification to other routing tables stored in the same broker. In fact, an underlying assumption is that scope transitions take place only within a broker. Routing tables of a super- and subscope pair $S \lhd T$ must be collocated at the same broker to enable inter-scope routing. In the above example B_1 is a scope broker of all scopes and may route between S, T, and U, whereas B_2 and B_4 can route between S and T only.

`cross_scope` takes a list D of pairs of eligible destination scopes, whose interfaces match, and notifications that shall be sent there. In this way, the current notification may be forwarded in different representations. With the help of the reference to the source component (`get_source`(n)) the algorithm prevents notifications from being sent back along the scope graph edge they were received from. This does not preclude duplicates because of alternative paths in the scope graph, but it rules out erroneous duplication because of repeated processing, at least in one broker. How to prevent this repetition in different brokers is detailed below.

The procedure sets the source component to the current scope and the intended destination as new current scope and then puts the relayed notification into the incoming queue *recvQ*. This eventually triggers the main loop and starts routing of n in the destination scope. The routing state recorded in each notification is updated according to the specified parameter s that is supplied by the main scoped routing algorithm.

Crossing at Different Locations

Although inter-scope routing is not possible at arbitrary brokers, there still may be multiple brokers where two scopes $S \lhd T$ coincide. And thus a notification might cross a scope boundary repetitively at different brokers, duplicating notifications even along a single edge of the scope graph. Furthermore, security considerations or the implementation of advanced ordering schemes might necessitate a designated broker that bridges all traffic between the respective scopes.In the previous example, a notification published by c_1 is distributed in its scope S, and may enter superscope T at B_1, B_2, or B_4.

Three choices for placing inter-scope routing are distinguished according to the following criteria. First, are the scope-crossing functions applied at only one broker or at several different brokers? Second, if only at one, is it a designated gateway broker or an arbitrary broker that conveys the traffic between the respective two scopes? The following alternatives are available:

1. Transition at designated central gateway
   All inter-scope traffic of a scope S is handled by a single gateway broker B_i of that scope. Only at this gateway the routing table $\mathsf{RT}^S_{B_i}$ contains an entry pointing to sub- and superscopes.

2. Transition anywhere, but only once
   Inter-scope traffic is transfered into its destination scope at the first possible broker, and nowhere else.

The first approach of having a *designated gateway* is the simplest solution. It instantiates the respective scope graph edge at a single point in the broker network. Only at this gateway broker a routing entry for the specific superscope is stored, say $({}^o\hat{I}^T_S, \mathsf{RT}^T_{B_1})$ as part of $\mathsf{RT}^S_{B_1}$ if B_1 is the gateway broker of $S \rightsquigarrow T$. All other scope brokers of S register an entry ${}^o\hat{I}^T_S$ that points towards this gateway broker, e.g., $({}^o\hat{I}^T_S, B_2)$ is stored in B_4. This is necessary to get published notifications matching the output interface forwarded to the gateway broker. Within T all routing table entries pointing to the subscope S are similarly adapted to direct downward traffic to B_1 as well. Each gateway broker links a specific pair of scopes, but generally system engineers may decide to group all gateway brokers at one network node, to group them for each scope, or to place all gateways independently.

A drawback of this strict separation of inter- and intra-scope routing is wasted network bandwidth. Consider c_4 and c_6 connected to broker B_4 in the previous example. Notifications from c_4 to c_6 are routed through broker B_1 to enter T there and go back to B_4 again. The adequate placement of gateway functionality has a major influence on network utilization. On the other hand, the centralized gateway offers full control of the incoming and outgoing traffic at a designated broker. This allows trusted software modules to be employed for cross-scope communication at a single trusted broker, for example, to authenticate all outgoing notifications or to link separate security domains without disclosing other scope brokers. The implementation of transmission policies is simplified, too, as pointed out in the next subsection. In general, if the placement of scope brokers correspond to the physical layout of the underlying network, gateway brokers may also represent the physical gateway between different networks hosting the adjacent scopes.

The second approach allows notifications to cross scope boundaries between two two scopes $S \lhd T$ at the first possible broker that sustains both scopes. When c_1 publishes n, it is forwarded into S, T, and U at B_1, assuming matching interfaces, of course. An appropriate countermeasure must be provided to prohibit repeated scope transitions in B_2 and B_4. This is achieved by testing whether the destination scope T was already seen in the last broker from which n is received, in which case the transition has already happened in a previous broker. Notification forwarding in `cross_scope` is denied if an entry in RT_B^T exists that points towards $\mathtt{link}(n)$. In the example, B_2 has stored an entry $({}^{i}I_{c_2}^{T}, B_1)$ in $\mathsf{RT}_{B_2}^T$ and does not forward n into T again.

Unfortunately, so far each scope transition generates a new notification and the transition at the earliest encountered broker leads to messages being sent on the network that differ only in the annotated current scope they are visible in. In the example, two messages are sent to B_2 and B_4, one visible in S and one in T. A possible improvement is a combined delivery to all eligible superscopes, which are identified by a list of scopes annotated on the notification instead of just one identifier. The multiplicity of messages is replaced by a list of scopes, at least as long as no mappings transform the notification. The routing decision is evaluated as before, only that `scoped_routing` is called multiple times to fill the list of next hop destination. At each broker, the available routing tables are checked and whenever additional scopes are detected and entered the list of visible scopes is updated. In the example, a notification forwarded from S to T is annotated with both scopes and is transmitted only once between B_1, B_2 and B_4.

4.5.6 Transmission Policies

The distinguished routing states directly correspond to the delivery, internal delivery, and publishing policy. The policies are encoded as part of the enhanced routing tables, even if they include general mappings in the routing decision. As discussed in Section 3.5, the policies operate on sets of notifications and must be evaluated after the eligible destinations are determined by the matching algorithm in `destinations`.

The three policies can be inserted into the three parts of the sketched `scoped_routing` algorithm. Internal routing is refined by evaluating

$$D := idp_S(D)$$

on the set of eligible consumers before it is processed in the 'foreach' loop.

Delivery and publishing policy are intended to be applied at scope boundaries, and so they are evaluated in `cross_scope`,

$$D := pp_S(D)$$

for upward routing and

$$D := dp_S(D)$$

for downward routing, again just before sending the notifications in the 'foreach' loop.

4.5.7 Scope Multicast

So far, intra-scope routing has stuck to strict routing where notifications are forwarded only if a matching subscription is available. This prevents notifications from being always sent to all scope brokers of a scope, but induces multiple point-to-point messages and repeated routing decisions. An alternative strategy for routing in a scope S is to send all notifications to all of its scope brokers irrespective of any subscriptions. In a second step the so-called *fan-out* of the broker network to the consumers is implemented via point-to-point communication. The routing tables of S are evaluated in every scope broker of S and each matching and locally connected consumer is notified separately.

If implemented as part of the broker implementation, an application layer multicast scheme is established within the broker network. This approach does not avoid multiple point-to-point messages between the scope brokers, but is readily applicable in most networks. On the other hand, IP multicast offers an established, well-known facility to speed up communication to a group of receivers. The original decision of using point-to-point communication in the broker topology is partially inspired by the assumption that the sets of consumers are rather volatile and vary frequently. A multicast solution that directly communicates to consumers requires frequent group changes and the explicit control of individual delivery is lost. However, IP multicast is a convenient technique to connect scope brokers. The broker topology can be supposed to change less frequently than the consumers and thus does not overwhelm multicast group management. So, intra-scope routing is reduced to a notification being conveyed to all scope brokers with one multicast datagram before it is explicitly directed to any matching consumers. This approach combines multicast efficiency with the full control of notification delivery.

4.5.8 Evaluation

The integrated routing architecture is possibly the most generic scope architecture. It combines the efficiency of a distributed solution, incorporates multicast delivery, and still offers the flexibility to control the hop of notification delivery to consumers. It extends the known routing tables and can built on various existing routing protocols, such as covering- or merging-based routing provided by Rebeca and other notification services. Scoping is here offered as a service of the event infrastructure. The layout of the publish/subscribe network is independent from the actual application structure given by the scope graph.

On the other hand, the option to connect scope members to arbitrary brokers may increase network utilization, and the dispersion of components and traffic may increase the complexity of the system. But this is essentially always the case for distributed solutions.

4.6 Combining Different Architectures

The preceding discussion assumed the same type of architecture for all scopes in the system, which is, obviously, a severe limitation of potential application domains. In fact, one of the primary benefits of the scoping concept is its ability to facilitate the customization of the infrastructure. Once groups of components are identified, their scopes can be based on those architectures that fit their respective needs best. The special requirements of their interaction are addressed by employing appropriate implementations of scoped notification dissemination. But yet, the different implementations must be seamlessly integrated.

4.6.1 Architectures and Scope Graphs

In the first place, scopes model application structure. But they are also a tool for determining notification semantics within the application structure. Different types of notifications may demand different quality of service even within a specific scope. For example, consider non-critical timer information sent in bulk (type A in Figure 4.19) and personnel record updates (type B) that are supposed to be encrypted and delivered to authenticated consumers only. While both are consumed in the same part of the application, i.e., in the same scope, these two data types obviously ask for different architectures and communication media that facilitate scalable delivery of the former and secured delivery of the latter.

In principle, several different communication media might be used in one scope to facilitate different QoS. Alternatively, a scope with complex semantics is duplicated in Figure 4.19 and each instance is tailored for a different kind of QoS supported. The interfaces are splitted so that the same notifications are forwarded into T as before. Publishing policies and imposed interfaces assigned to c_1 and c_2 ensure that the traffic within S_1 and S_2 is separated and directed to the scopes that offer the necessary quality of service. In the above example, the timer notifications would be distributed via scope S_1 operating on top of a scalable messaging system, and S_2 would employ encrypted point-to-point connections to meet the security requirements of type B. The edges (c_1, S_2) and/or (c_2, S_1) are necessary if the c_1 and c_2 shall get the same notifications as before, but additional interfaces are necessary to prevent messages from leaking with wrong QoS.

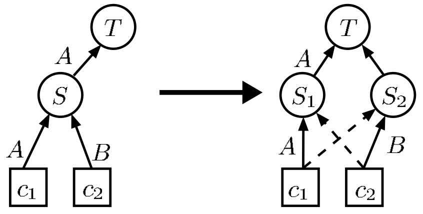


Figure 4.19: Duplicate scopes to separate QoS requirements

Instead of dealing with arbitrary combinations of communication media, dissemination semantics, and scopes, the following assumes a specific scope architecture per scope. For implementation purposes, bridging takes place between connected subgraphs of the graph of scopes that share a common architecture. However, to simplify the discussion only pairs of scopes and the bridging in between are investigated next.

4.6.2 Bridging Architectures

Combining different scope architectures requires a gateway between the different implementations of two scopes $S \lhd T$. The simple components of a scope have as part of their local event brokers an architecture-specific implementation for accessing the underlying communication medium (cf. Figure 4.4). The gateway relies on two local event brokers to bridge the respective implementations of the architectures. Gateway functions are assigned to the considered subscope, S, and enforce the input and output interfaces of S, its publishing and delivery policies, and any mapping applied on the edge (S, T).

Collapsed Filters. The collapsed filters architecture does not instantiate administrative components and so gateway functions must reside in all members of the scope. Due to the required duplication of code in simple components an extra gateway component is preferable. Such a component would not interfere with the internal delivery of notifications. It is similar to the mapping components used in Section 3.4.2 to sketch the feasibility of scoped systems. It acts as an additional producer/consumer in scope S and manually implements the edge to superscope T, being a regular member there as well.

The distinguished *scope destination* and *visibility roots* approaches to annotate notifications and extend subscriptions are hardly different regarding the implementation of the gateway. In both cases a gateway component is instantiated for each bridged scope-superscope pair or for all bridged superscopes collectively. Only their subscriptions must reflect the differences in the lists of annotated scope identifiers: the former lists all reachable scopes while the latter lists only visibility roots on upward paths. Upon receiving a notification, the gateway component tests which of the edges it controls is eligible, applies the assigned output interfaces and publishing policies, and forwards the data, if appropriate.

In the same way, the gateway registers in the superscope(s) and upon receiving a notification from there evaluates the assigned input interfaces and delivery policies. Since delivery policies need cooperative filtering in all consumers, the gateway's functionality depends on the filtering supported in present implementation of the collapsed scope graph.

Scope Address. In the scope address architecture there are administrative components available to execute gateway functions. Cross-scope traffic is matched against the interfaces, publishing and delivery policies are applied as before. The same implementation can be used, only a second local event broker to bridge the different architecture's implementations must be present.

Broker Scopes. Broker scopes are administrative components that represent a specific scope and explicitly controls all internal and external traffic. Thus, they may directly implement any gateway to other architectures.

Integrated Routing. Although no individual representatives of scopes exist, scopes and transitions between scopes are explicitly recorded as routing tables with entries referencing other routing tables. Instead of pointing to other tables, the entries may refer to a second local event broker to access an other's scope architecture. Interface and transmission policies are handled as before—they always explicitly applied. The discussion about locating cross-scope transitions (cf. Section 4.5.5) holds for gateways as well. They can be placed either at a designated broker or at several brokers of the scope, where the first encountered when forwarded a notification is used for the transition.

4.6.3 Integration with other Notification Services

The gateway of a scope may not only bridge different scope architectures within the Rebeca system, but also facilitate coupling of a scoped system with other notification services. The gateway functions simply have to implement another service's API to act as a regular producer/consumer within that service. The traffic flowing between the scoped and the external system is controlled by the gateway functions, i.e., interfaces, mappings, and transmission policies. By creating an 'outside' scope and a gateway that connects other communication services, external data is incorporated into the scoped system without impairing visibility control. On the other hand, this gateway retains the component characteristic of scopes with respect to the outside system. The flow of notifications leaving the scope follows the definition of the scope graph.

Of similar importance is the coupling of scoped and unscoped applications, which are likely to coexist. Consider the integrated routing approach, for instance, and two applications, one scoped and one unscoped. The scoped routing tables are used in addition to a traditional implementation, which is nothing more than a further routing table not connected to the scope routing tables. All scoped clients are assigned to some $\mathsf{RT}_{B_i}^{S_i}$ and the non-scoped ('legacy') clients are still maintained in separate old-style Rebeca routing tables RT_{B_i}. In fact, the overlay of all RT_{B_i} constitute a *default scope* to which every newly created simple component may be assigned. In this way, scoped and non-scoped can interact in a controlled way.

4.7 Discussion

This chapter has presented a range of approaches for implementing scopes. A scope's architecture is distinguished by the communication medium used for transmitting notifications and by the way scopes are distributed. The different architectures all adhere to the visibility constraints defined in the scope graph, but they differ in the amount of control exerted on communication beyond interfaces and filtering. The options covered here range from static deployment and collapsed filters to broker scopes and the integration of content-based routing and scoping. They make the abstract notion of scopes concrete.

The central point made in this chapter is that scoping does not imply a specific implementation per se. Depending on the intended semantics, adaptability, communication efficiency, etc., alternative implementations are applicable. This chapter has shown how a scope graph as a model of an application is transformed into a running system. Scopes thus became a means of implementing and customizing notification dissemination. They allow system engineers to choose those techniques of network communication, data representation, filtering, routing, and data communication security that are appropriate for the considered scopes and conform to their required QoS characteristics. Scopes make applications and the publish/subscribe system itself customizable and they facilitate a model driven and open implementation of middleware.

Chapter 5

Rebeca—An Implementation of Scopes

Contents

5.1 Software Building Blocks **140**
5.2 Scopes **148**

This chapter describes the implementation of scopes as part of the Rebeca notification service. Rebeca is a recursive acronym that stands for *R*ebeca *E*vent-*B*ased *E*lectronic *C*ommerce *A*rchitecture. The system is created as a testbed of notification service implementation and extensions are developed with diverse objectives in mind. The initial approach focused on routing algorithms [213], performance measurements [217], and flexible data and filter models [212] that were not limited to a predefined set of allowed data types and filter expressions. The initial acyclic broker graph was extended to operate on a peer-to-peer (P2P) network that guarantees maximal network depths and self-organizing and fault-tolerant network topologies [291]. Mobility scenarios were addressed in [106, 107, 218, 305]. The initial support of handling notification lifetime was extended in [75, 218], and recently, security issues were addressed based on scopes in Rebeca [117].

The initial implementation of Rebeca [112, 213] has been realized in the Java programming language [17] (J2SE 1.4), and the development currently continues on Java and .Net. Partial implementations were also done in Smalltalk and Python, and a serialization mechanism that allows for the transmission of XML-encoded notifications opens the system for further extensions.

The implementation described in this chapter redesigns core parts of the infrastructure and adds scoping into the more modular software architecture. Section 5.1 presents the core parts of Rebeca and the next sections detail the implementation of scopes.

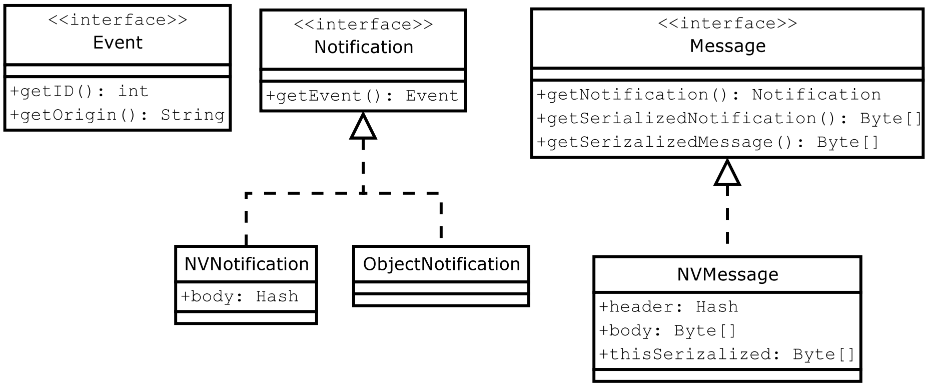


Figure 5.1: Events, notifications, and messages

5.1 Software Building Blocks

The basic infrastructure building blocks are event and subscriptions, the basic API, the broker network, and the individual broker implementation.

5.1.1 Events, Notifications, Messages: Data Models

Important features of notification services are the data and filter models available for notifications and subscriptions, respectively. They determine the expressiveness of the notification service as a coordinating intermediary in the first place. Although this obviously interrelates with application functionality, the data and filter models are often laid down by the service provider. For instance, the Java Message Service acts as a black box using such predefined models.

In Rebeca, events and notifications are distinguished in the implementation (see Figure 5.1). The interface `Event` provides access to information unique to an abstract event. `id` and `origin` are `null` or carry a unique numerical identifier and a string representation, respectively. Essentially the subclasses contain information visible in the infrastructure but not in the application. The application/event data is stored in a `Notification`, which provides an interface to a specific representation of an event. One event can refer to multiple notifications. Initially, when an application publishes a notification, an event instance is created that refers to this first notification. During event processing several notification instances may be created that all stem from the same original event. For instance, if routing is parallelized or if an event is forwarded into several scopes, different notifications are processed that belong to the same event. Events refer only to notifications on the same node, not to remote ones.

Subclasses of `Notification` are provided for different data models, which are independent of any specific events, and for certain types of events in a predefined data model. A name-value pair notification may carry arbitrary data, whereas a `Quote` subclass directly provides a Java implementation of the quote events of the stock market example. In this way, general data models as well as objects as events are supported by REBECA.

In addition to the application data, each notification contains a header that carries management data specific to this notification (cf. Fig. 5.2). The application data contained in the body is used by subscriptions and filters (see below) to select the notifications of interest; the body is not modified by the notification service. The header, on the other hand, is not accessible to applications and contains management information only used in the broker network of REBECA. The header consists of a volatile part, which is not transmitted between brokers, and a non-volatile part, which is maintained until the notification is delivered to the application. Volatile data are broker-local information such as source links, local receive time, etc. The non-volatile part carries data relevant to the routing and delivery of the notification through the whole broker network, e.g., publishing time, scope IDs, sequence numbers, and so forth.

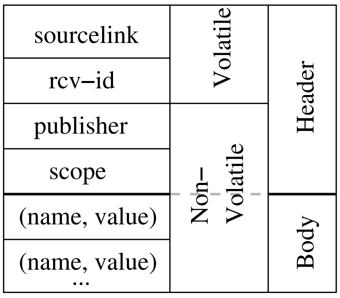


Figure 5.2: Message structure

The flexibility of Rebeca is that the provider of the infrastructure or the application programmer is able to implement and use any form of notification as long as it inherits from `Notification`.

On the lowest level messages are used to send notifications between REBECA processes. The messages are exchanged as part of the link protocol between the processes. This protocol is currently confined to a minimum functionality, including link setup, protocol version tests, and the initialization of the serializer [258]. The latter transforms notifications to the byte stream transmitted on the network, and vice versa. The subclasses of `Message` are specific to the protocol used. Currently, only one implementation is provided, `NVMessage`, which consists of name-value pairs, one of which is named "body" and contains the serialized notification, whereas the others may be used for link sequence numbers or sliding window protocols [243]. `Message`, too, contain management information, which is specific to the data communication on the respective link.

In summary, three different interfaces exist to describe events:

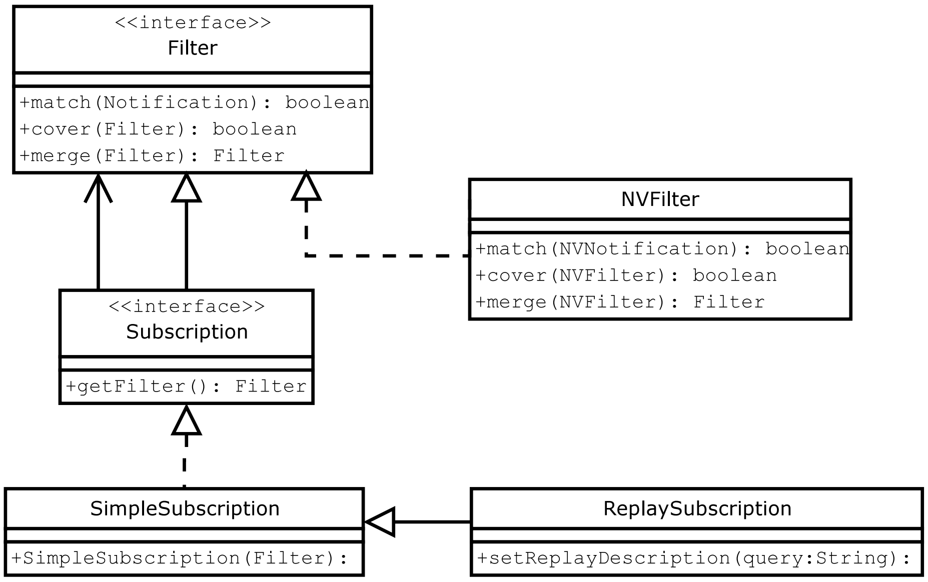


Figure 5.3: Subscriptions and filters

1. `Event` contains global attributes intended mainly[1] for internal use.

2. `Notification` carries the application data in a specific data model and additional internal routing information.

3. `Message` contains a serialized notification (or other Rebeca administrative messages) and link-specific data concerning, e.g., quality-of-service, serialization, etc.

These interfaces correspond to different layers in the implementation, which are, however, not strictly separated. The serialized data sent in messages is deserialized on the receiver side, but it is also cached (`Notification.serialized Form`) to facilitate resending unchanged notifications without repeating the serialization step.

5.1.2 Subscriptions and Filters: Filter Models

The filter model corresponds to a data model and determines the expressiveness of subscriptions (see Figure 5.3 and [212, 213]). The `Filter` interface contains a method `match()` that tests a notification. Currently, the implementation returns a boolean value so that filter concatenation as defined for simple event systems in Section 2.3 must be emulated by looping over a list of filters. The interface also contains three methods `covers()`, `overlaps()`,and `merge()`, which must

[1]Information such as publishing time might be of interest to the application as well.

be implemented if advanced routing algorithms shall be used; the name-value pair filters contain this implementation [213].

In REBECA, clients may issue any filter that inherits from the interface `Filter`; no specific filter model is required. As with the data models above, the provider of the infrastructure or the application programmer is able to implement any kind of filter as long as it inherits from `Filter`.

Filters are not issued directly. Instead, the system expects subscriptions, which are of type `Subscription`. As stated in Section 2.1.3, subscriptions are filters, but additionally may include other information than only the filter expression selecting passing notifications. The simplest form, of course, is the `FilterSubscription`, which simply takes a filter that implements the subscription's `match()` method. Another example is `ReplaySubscription`, which holds a filter and an additional query which is used to retrieve past notifications that shall be delivered in advance of any new ones [75].

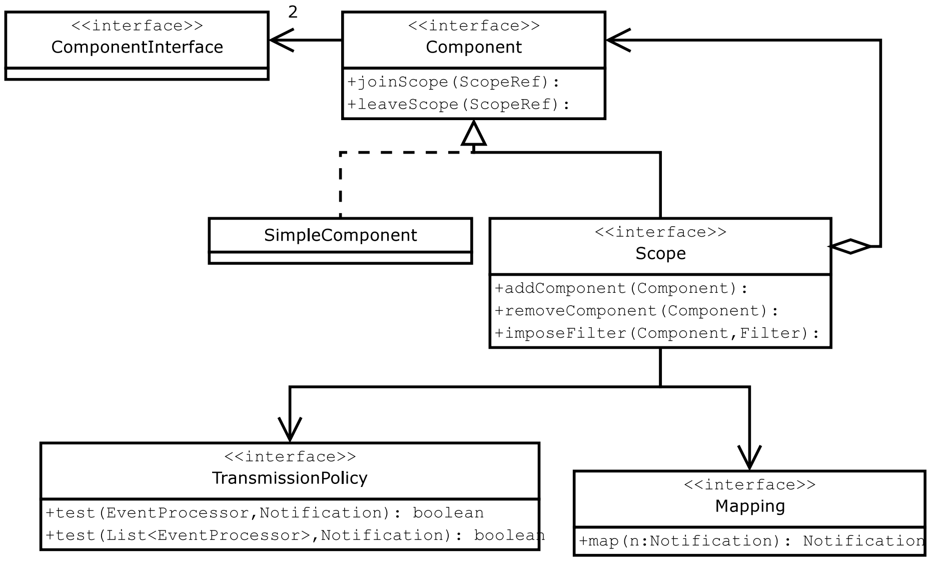
5.1.3 Pub/Sub API

Clients access the REBECA notification service via *local event brokers*, which offers the plain pub/sub API as a library collocated to the client code. Local brokers maintain connections to the event broker network as described in Section 2.3. They also maintain a routing table to store the callbacks for each locally issued subscription. The `subscribe()` method takes a subscription and a callback, which is invoked if a notification arriving at the local event broker matches the specified subscription. These callbacks as well as many other event processing components inherit from `EventProcessor`, which only contains a single method, `process()`. In this way, the routing table can store both local callbacks and connections to remote brokers (see below).

The `publish()` method of the local event broker inserts a new notification into the system; a corresponding event object is created by the broker. Unsubscribe and (un)advertise methods are also available at the local event broker.

In addition to the offered API, which is basically the same as in the original REBECA [213], event-based components are introduced here that hide the explicit handling with the local event broker. A component is implemented through the Interface `Component` (see Figure 5.4). It offers similar functions as the local broker: it can publish, advertise, and subscribe. This type corresponds to the components defined in the scope model in Section 3.2 and it is implemented in `SimpleComponent`. It can be instantiated in any application program and searches for the next event broker at a number of default places (localhost with default port, property file driven JNDI lookup, multicast lookup). Example code for using `SimpleComponent` is given in Figure 5.5. Lists of Subscriptions and advertisement constitute the input and output interface of a component, respectively.

A component may join and leave scopes by calling the respective methods `joinScope()` and `leaveScope()`, cf. Section 5.2.2. `Scope` inherits from `Component` since it acts as component in its superscopes. Additional methods



<<interface>>
ComponentInterface
2
<<interface>>
Component
+joinScope(ScopeRef):
+leaveScope(ScopeRef):
SimpleComponent
<<interface>>
Scope
+addComponent(Component):
+removeComponent(Component):
+imposeFilter(Component,Filter):
<<interface>>
TransmissionPolicy
+test(EventProcessor,Notification): boolean
+test(List<EventProcessor>,Notification): boolea
<<interface>>
Mapping
+map(n:Notification): Notificatio

Figure 5.4: Components and scopes

```
public class DemoPublisher extends SimpleComponent {
  void main (String[] argv) {
    Component c = new SimpleComponent() {
      public void run() {
        publish(new TestNotification(argv[0]));
      }
    };
  }
}
```

Figure 5.5: A demo publisher

enable the configuration of scope features (interfaces, mappings, transmission policies).

5.1.4 Broker Network

The broker network consists of implementations of `EventBroker`. The interface is implemented in the `LocalEventBroker` class (Figure 5.6), which is typically colocated to the clients of the notification service and serves as access point to the infrastructure. The core network consists of `EventRouter` implementations, which inherits from `EventBroker` the methods to access the routing table/ engine and the current list of links to other brokers. The router adds listener management to allow other brokers to connect to the EventRouter instance. Current listeners are, e.g., `ServerSocketListener` and `PipeListener`, which enable TCP and node-local connections to be established. These routers are

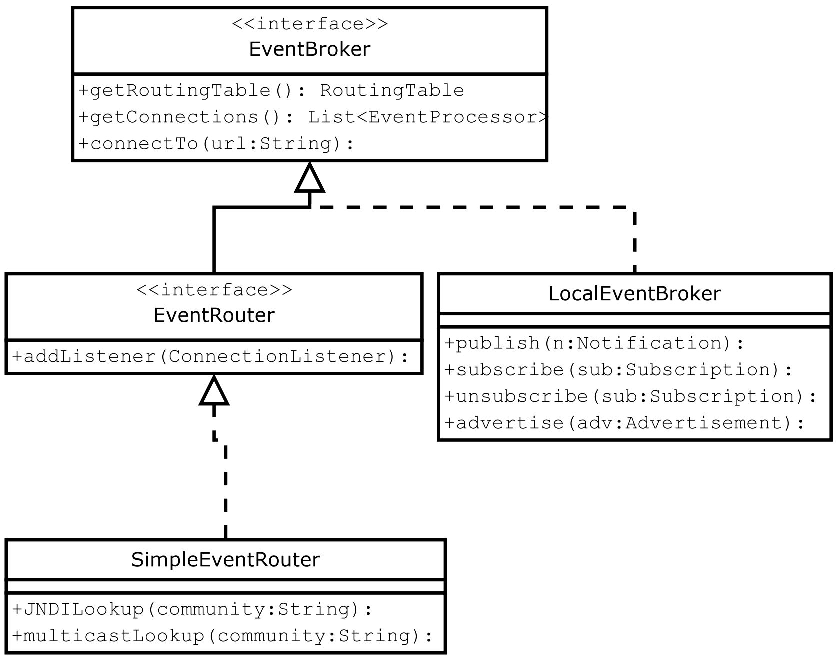

<<interface>>
EventBroker
+getRoutingTable(): RoutingTable
+getConnections(): List<EventProcessor>
+connectTo(url:String):
<<interface>>
EventRouter
+addListener(ConnectionListener):
LocalEventBroker
+publish(n:Notification):
+subscribe(sub:Subscription):
+unsubscribe(sub:Subscription):
+advertise(adv:Advertisement):
SimpleEventRouter
+JNDILookup(community:String):
+multicastLookup(community:String):

Figure 5.6: Rebeca brokers

instantiated as separate processes on arbitrary machines in the network.

Currently, brokers are connected using point-to-point TCP connections. Although multicast support would improve communication efficiency, the TCP solutions is the most general, and optimizations are left for future work. New brokers are connected to existing brokers by specifying an address with the help of a Uniform Resource Locator (URL). Helper functions, as in `SimpleEventRouter`, can exploit the Java Naming and Directory Interface (JNDI) or multicast to look up running brokers.

The currently available routing algorithms require the network to be acyclic.

5.1.5 Broker Implementation

The basic functionality of a broker is rather simple. Each incoming notification is tested against the filters stored in the routing table and it is forwarded along all links having matching filters. In Rebeca, the `RoutingEngine` encodes the routing algorithm used to maintain the routing tables (Figure 5.7 and [213]). So far, it also includes the filter tests, although routing and matching are independent in principle; a more elaborate implementation is left for future work.

A routing table contains pairs of filters and event processors. It can be used to store local callbacks as well as network links, because both inherit from

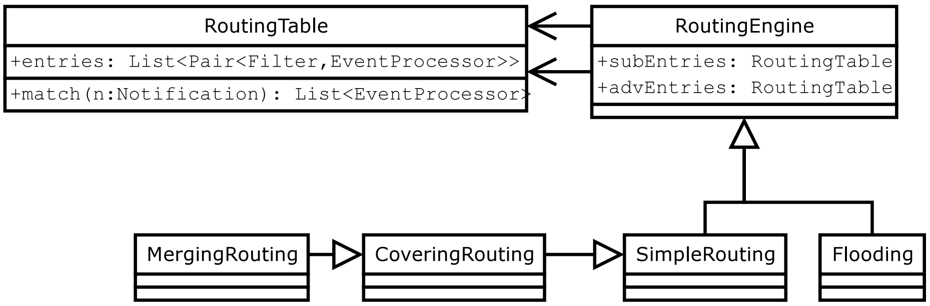

RoutingTable
+entries: List<Pair<Filter,EventProcessor>>
+match(n:Notification): List<EventProcessor>
RoutingEngine
+subEntries: RoutingTable
+advEntries: RoutingTable
MergingRouting
CoveringRouting
SimpleRouting
Flooding

Figure 5.7: REBECA routing classes

`EventProcessor`. Network links are encapsulated in `EventTransport`, an interface for bidirectional transmission of notifications. Such a transport implements the REBECA protocol for link setup, it accesses a serialization mechanism to stream notifications, and it transmits messages containing serialized data and other control information. In local event brokers an `EventProcessor` instance may be a callback of a registered consumer. In any case this abstraction can be exploited to build arbitrary sequences of event processing modules that comprise the broker functionality, which is used in a re-design of the original broker implementation.

REBECA messages transmitted between brokers may contain (a) control messages, like subscriptions or other administrative messages, or (b) notifications, which consist of a management header and the notification data. Messages of type (b) are handled by the routing table. A number of subtypes of admin messages are differentiated and specialized message handlers can be registered for their processing. Thus, new message formats with the respective processing functionality can be easily added.

The original broker implementation relied on a number of subcomponents that were hard-wired. Now, the re-implemented brokers are instantiated and configured within a software container that supports the dependency injection pattern [121, 149]. Brokers become customizable *software containers* and the implementation of the routing engine, transmission protocols, connection pooling, and message handlers is specified at deployment time of the broker. The container starts the components specified in an configuration file (Figure 5.8) and starts those having implemented the `Startable` interface.

Each broker of the network can be instantiated with a JMX (Java Management Extension) agent assigned to it. This opens the instance for online monitoring and reconfiguration of the current state at runtime. Currently, the JMX implementation and the HTTP connectors of JBoss are used [159]. A JMX-enhanced broker accepts requests to instantiate local simple components, or scopes as detailed below.

```
<?xml version="1.0"?>
<components>
  <component type="rebeca.container.configuration.ConfigurationBuilder"
             class ="rebeca.container.configuration.XMLConfigurationBuilder"/>

  <component type="org.apache.commons.logging.LogFactory"
             class ="org.apache.commons.logging.impl.LogFactoryImpl"/>
  <component class="rebeca.network.transport.EventSocketTransportManager">
    <local-port>8020</local-port>
  </component>
  <component type="rebeca.routing.RoutingEngine"
             class ="rebeca.routing.Flooding">
  </component>
  <component type="rebeca.security.SecurityManager"
             class ="rebeca.security.PKSecurityManager">
    <key>
      <path="to/file"/>
    </key>
  </component>
  <component type="rebeca.network.EventRouter"
             class ="rebeca.network.SimpleEventRouterImpl">
  </component>
</components>
```

Figure 5.8: An event broker configuration file

5.1.6 Channels

For a number of application scenarios and internal management tasks it would be benefical to identify a specific route between sender and receiver of a message. The JMS standard includes a mechanism to send reply messages on the application level. The processing of scope join 'requests' presented in the next section relies on this ability. And the implementation of histories replaying past notifications uses direct connections within the broker network, but would profit from such a mechanism as well. However, the use of directed communication is generally questionable in event-based systems. The solution implemented in Rebeca does not disclose sender IDs, it rather identifies the route a notification took in the network, and makes that accessible for later routing in either direction.

A prelimary implementation of *channels* works as follows. A channel is a specific path in the broker network, which may split and join at any broker. Each involved broker stores a the next hop brokers of the channel. A notification can be tagged with a channel ID and if the corresponding channel is known, the notification is forwarded to the stored next hops. If the ID is not known, routing is carried out according to the normal routing tables, but additionally a new channel with this ID is registered that points to the originating and all destination directions.

The configured channels are stored in the `ChannelTable` in each `EventRouter`, i.e., broker of the network. Currently, a `Channel` instance is stored for every next hop of a channel, so that multiple instances are created referencing the same `ChannelID` object. Since channels are added via calls to the channel table, alternative implementations may use only one channel object that holds all next hops. The channel IDs can be be universally unique identifiers[2] or can be generated unique to a specific broker. In the latter case the tag assigned to a notification must be transformed when forwarding it to the next hop (`Channel.nextID()`. Similar to ATM (Asynchronous Transfer Mode) virtual paths [185], IDs a used to distinguish channels within one broker.

Obviously, after creating the channels, they must be taken down after some time to keep the size of the channel tables small. Depending on the application scenario, a different rules can be applied to determine outdated channels. The channels have a field `Channel.relevance` that is an integer counted down to zero, which indicates that the channel is not used anymore. The simple approach to implement a back-channel for notifications is to count down whenever a notification passes along the channel; which is used for scope administration detailed below. If the relevance is initially set to one, exactly one notification can be sent in the channel. Alternatively, a timer associated with the channel table could be used to count down the relevance.

5.2 Scopes

The scope implementation in Rebeca consists of: the generic scope interfaces, two imlementations (broker scopes and integrated routing), and the management interfaces of running instances.

5.2.1 Scope Interface

As depicted in Figure 5.4 on page 144, the `Scope` interface is derived from the general `Component` interface. Being a component, scopes have methods `joinScope` and `leaveScope` to ask for joining or leaving an other existing scope. Actually, the presentation is simplified in the figure, because these methods are overloaded and can take different parameters such as filters and credentials to enable admission control [117]. The `ScopeRef` interface taken as parameter for joining and leaving encapsulates a reference to a running scope instance. Subclasses are defined specific to the scope architectures. In the broker scope implementation, it contains IP address and port of the running scope instance. In integrated routing, it contains a reference to the *scope lookup table* ST_{B_i} which gives the next hop towards the scope instance. Scope references can be retrieved from a lookup service or directory—but remind that components are generally not intended do join scopes manually.

Additionally, the scope interface provides the minimal methods needed in a scope. `imposeFilter()` enforces a filter on one of the scope's components, shaping all traffic to and from the specified component to conform with the specified filter. `addComponent()` and `removeComponent()` process join and leave requests from components.

[2]J2SE 1.5 will contain helper classes for UUIDs.

Transmission policies can be incorporated following the strategy pattern [129]. A `Policy` interface should comprise init and shutdown methods to manage its own resources, and a test method that takes a notification and indicates to which components it should be forwarded. Instances of a policy are called after interface filters have been applied and before the message is actually forwarded.

5.2.2 Implementation: Broker Scope

The following section describes an exemplary scope implementation and its scope lifecycle, i.e., details on how scopes and scope graphs are created, how components join and leave scopes, and how scopes are deleted.

The `BrokerScope` class implements the `Scope` interface and follows the broker scope architecture of Section 4.4. One instance of this class represents a specific scope of the scope graph, and it acts as broker in the broker network. Technically, the broker network is instantiated first and then a `BrokerScope` instance is collocated to an existing broker. The scope functionality is linked into this broker: the routing table is wrapped to intercept calls to distinguish scoped traffic from unscoped traffic. The entries of the routing table are now separated in three parts, the list of links to superscopes, links to sub-components, and entries of unscoped destinations. In the first two cases, the network links of the broker correspond to the edges of the scope graph.

With respect to a broker the scope instance is a singleton, governed by a static hash that governs the one-to-one relationship within a Java virtual machine.

Creating a new scope

A `BrokerScope` object can be instantiated in any VM where an event broker is already running. It is constructed as singleton with a static member referencing the broker instance, thus guarding against multiple instantiations. In fact, the member holds a list of brokers so that multiple brokers can be instantiated in one VM and the `BrokerScope` is a singleton with respect to one broker. A `BrokerScope` object is instantiated by calling an event broker via the JMX management API, using either the HTTP adaptor or RMI calls to the respective methods.

Joining an existing scope

A running scope instance S has no connections to other *components* at first; the links to neighbor brokers are not part of the scope graph. Two cases must be distingsuished: i) a component asks for joining this scope instance, and ii) this instance wants to join an other scope.

In the first case, a simple component can join a `BrokerScope` instance if it is directly connected to this instance. A call to `Component.joinScope()` is implemented by having the local event broker send an admin message (`ScopeJoin-Message`) into the broker network towards the border broker to which it is connected. If the border broker hosts the requested scope the join request is processed by the scope instance. If the scope admits the join, it updates its list of sub-components and acknowledges the request; otherwise the request is rejected by sending a negative ScopeJoinMessage back to the client.[3] Scope join requests of other scopes are handled in the same way.

The second case considered is how to the broker scope instance S asks for joining an other existing scope. The administrative interface of S gives access to the `Component.joinScope()` method implemented by `BrokerScope`. Similar to the implementation in simple components, this method prepares a `ScopeJoinMessage` and sends it to neighbor brokers. It is send to all next hop brokers if no specific destination is given. Or it is send to only one specific broker, e.g., if the administrator initiating this request knows of the scope running there. On a positive acknowledgement the list of links pointing to superscopes is updated to reflect the locally known scope graph.

So, the scope graph implemented in the system is build up by connecting broker scope instances to each other and simple components to border brokers. The subgraph known to the broker scope instance, i.e., itself plus egdes to adjacent nodes, is stored in the lists of links to superscopes and to sub-components. The mapping of the scope graph to the broker network is done manually by the administrator, which is infeasible for large graphs. But it is advantageous if full control of communication is required, e.g., at gateways between organizational boundaries.

Scope Join Notification

The process of joining a scope can be seen as a sequence of events, describing the initial interest to join and the reaction of the scope instance. In the broker scope approach they are send as administrative messages. In the scope address architecture, which less strictly correlates scope graph and broker network layout, these notifications are actually published in the network. In any case they carry the data necessary to identify scope and component, the status of the join process, and additional information about the component (see Figure 5.9). The notifications are not only used to inform about the progress, they drive it. The `Scope` and `Component` fields carry names that uniquely identify joined and

`Scope`	Name (or ID) of the destination scope	Mandatory
`Component`	Name (or ID) of the joining component	
`Status`	Request, Approval, or Rejection	
`Interface`	with which admission is applied	Optional
`Credentials`	List of certificates for the interface	

Figure 5.9: Structure of a ScopeJoin notification

[3]In other scope architectures, e.g., scope address (Sect. 4.3), the join messages may be published as notifications themselves to leverage space decoupling.

joining components. Uniqueness is only required in terms of the reach of the notification. If it is potentially distributed in the complete system, it must be universally unique. In the broker scope approach, the messages are sent only to the next hop brokers anyway. Generally, applications are able to control the distribution of ScopeJoin notifications with their scope interfaces. Hence, the administrator must ensure that these names are unique within intended area of distribution.

The `Status` field communicates the process status, which is one of three stages: *Request*, *Approval*, or *Rejection*. In *Request* state, the join has been initiated but no decision is made yet. The joining component requests participation with the interface specified in the respective field and the provided credentials may be used to determine admittance by the scope membership control (see below). The *Approval* state confirms successful join with the interface given in the respective field of the approval notification. Note that the given interface may deviate from the originally requested one. If the interface is not specified in the approval, the originally requested one is allowed. Otherwise, the imposed interface is given in the approval (cf. Section 3.3). A *Rejection* state finally disapproves the request, indicating that this component is not allowed to join with the given credentials currently. Interface and Credentials fields may be emtpy or contain the specific entries that led to the refusal.

Membership Control

Who is going to decide on the join request? As mentioned before, scopes lend themselves to implement access control mechanisms in event-based systems. A broker scope instance can accept or reject a join request based on the information provided in the request. A security manager is configured and started at deployment time. The interface `SecurityManager` has methods to test join requests, subscriptions, and advertisement for compliance with scope-specific security policies. These methods are invoked from the respective message handlers. The example implementation (cf. [117]) utilizes signed filters, which are issued and signed offline with a scope-specific key so that they can be checked online by the scope instance. In addition to link setup at the broker network level, additional application-specific security measures are thus implemented at the scope level.

Leaving a Scope

A leave request is issued as notification in the scope to be left. Any management information about the leaving member stored in the scope instance is deleted.

Destroying a Scope

The management interface of a broker scope instance can be used to request a shutdown of the instance.

5.2.3 Management Interfaces

Management interfaces are provided to allow system administrators to monitor, configure, and deploy infrastructure and application components at runtime. For this reason, event brokers, routers, scope instances, and simple components can be decorated with management interfaces. These interfaces rely on the Java Management Extensions (JMX) [283], which are part of the JAVA SDK as of J2SE 1.5 [160]. A JMX agent runs in a virtual machine registers local MBeans, offers protocol adaptors to access their interface with HTTP, and uses connectors to transparently forward calls from remote proxies to the local MBean instances.

Standard MBeans are the simplest way to export fields and methods of managed classes. The class must implement an interface *Classname*`MBean` and every (getter/setter) method of the class that is also defined in this interface is accessible through the JMX agent. In order to make the existing classes manageable either the inheritance hierarchy has to be modified or new classes has to delegate calls to the existing ones. Another elegant solution, which does not require to always instrument all classes with management code, is aspect-oriented programming (AOP) [99, 172]. A management aspect of the form given in Figure 5.10 adds these features without having to change existing code. After a first evaluation of the delegate approach, AspectJ [171] was used to implement the management extensions.

```
public EventBrokerMBean extends EventBroker {...}
...
public aspect JMX {
  declare parents:
     (! EventBroker && EventBroker+) implements EventBrokerMBean;

  pointcut mbeanCreate (Object o) :
     execution(*MBean.new(...) && this(o);
  after (Object o): mbeanCreate (o) {
     // register to agent
  }

  public JMX() {
     // get reference to agent, start adaptors
  }
}
```

Figure 5.10: Adding a management aspect

Scope management deals with monitoring and controlling the state of the scope graph. It covers scope creation and deletion, and the lifecycle management of simple components, their connecting to and disconnecting from scopes. Scopes are created by accessing the event broker that shall host their instance (cf. Figure 5.11. The MBean interface of an event broker provides a call `EventBrokerMBean.createComponent()` that takes a class name and additional constructor parameters and utilizes Java class loading to load and instantiate a class of the given name. The configuration parameters can either be directly supplied to the constructor or a `Config` instance is given that carries all the configuration parameters as a key-value map. `Config` is inspired by the software container approach to configuration as it is used in the Apache Avalon project.

```
public interface EventBrokerMBean {

  Collection getLinks();
  Collection getClients();

  void createSimpleComponent(String className);
  void createSimpleComponent(Config config);

  void createScope (String className);
  void createScope (Config config);

  void destroyComponent (String ref);
  ...
}
```

Figure 5.11: EventBroker management interface

The management interface of individual scope instances, Figure 5.12, provides the current list of connected components and superscopes, and generally informs about the current configuration of scope features, even if they are not adaptable at runtime. A likely future extension is to capitalize on the software container approach to make the constituents of the scope instance more adaptable. The functions to join scopes is inherited from the `Component` interface.

```
public interface ComponentMBean extends Component {
  // makes all methods of Component manageable
}

public interface ScopeMBean extends ComponentMBean, Scope {
  // makes all methods of Component and Scope manageable, plus

  Collection getComponents();
  Collection getSuperScopes();
  ...
}
```

Figure 5.12: Scope management interface

5.2.4 Implementation: Integrated Routing

Integrated routing reconciles distributed notification routing with the visibility constrains defined by the scope graph (cf. Sect. 4.5). The original routing table is broken into multiple tables, one for each locally available scope. Thus, for each scope a connected subset of event brokers constitute an overlay within the broker network that conveys scope-internal traffic. In order to simplify scope joins, another routing table is stored in each broker. The scope routing table `scopeRT` records scope-link pairs in each broker to signify in which directions brokers of the respective scope can be found.

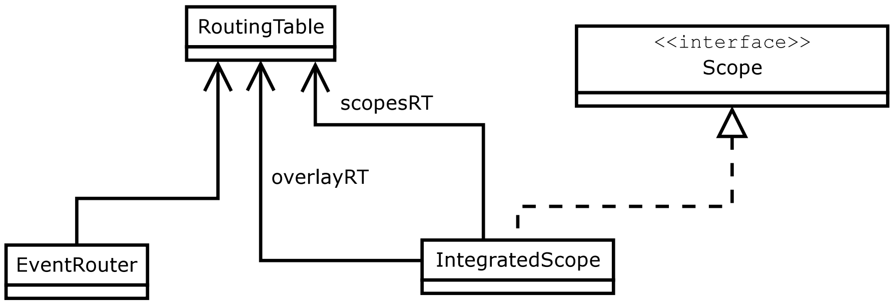


Figure 5.13: Integrated routing scope classes

This kind of scope architecture is implemented by the `IntegratedScope` class (Fig. 5.13). The overlay routing table (`IntegratedScope.overlayRT`) holds the routing information of scope members and superscopes. The static field `IntegratedScope.scopesRT` stores the direction towards other scopes in a routing table that is shared by all instances of this scope architecture.[4]

To create a scope of the integrated routing architecture, an instance is created via the `EventBrokerMBean.createComponent()` call at an arbitrary broker. Each scope is represented by one instance, which has a routing table of its own in addition to the one maintained by the event broker for the unscoped traffic. The static constructor of `IntegratedScope` registers new message handlers with the event broker to process administrative messages such as join requests. The creation of the scope is announced with a notification, which is distributed in the network to update the scope routing tables. This announcement can be deferred until the newly created scope has joined other scopes. In this way, the distribution of the scope creation events is limited by the overlay of the superscopes and their output interfaces. And so the scope routing tables do not have to store references to any socpe in the system.

The scope overlay can either be extended manually via the management interface to preset a certain extent of the overlay, or it is extended dynamically when other components are to join the scope. In the second case, a scope *join request* is issued at a broker currently not part of the overlay. A request is traveling in the direction stored in the scope routing table, leaving a temporary trail of references to the requesting source (using channels described in Sect. 5.1.6). The first broker encountered that is part of the requested scope, processes the join request and sends a reply back along the channel. If affirmative, instances of `IntegratedScope` are created along the way; they become part of the scope overlay. Alternatively, the join request or the replying scope may opt to use the channel just created to tunnel all scope internal traffic towards the requesting source. This mechanism is used in Fiege et al. [117] to tunnel encrypted message

[4]Although valid for any scope architecture, this table is currently implemented in this class as it is not used elsewhere.

payload through untrusted brokers.

The transition of notifications between two scopes requires the two scope overlays to share at least one broker. The edge between two adjacent nodes in the scope graph is implemented as references between two `IntegratedScope` instances. The input interface (subscriptions) of the subscope is stored in the superscope's routing table with a message handler of the subscope as destination. Conversely, the output interface (advertisements) of the subscope is stored in its own routing table. The associated handler of the superscope may add any imposed filters on the traffic.

Chapter 6

Related Work

Contents

6.1 Distributed Systems 159
6.2 Notification Services 164
6.3 Rule-Based Systems 171
6.4 Data Management 171
6.5 Coordination Models 173
6.6 Software Engineering 175
6.7 GUI Design 179

The paradigm of event-based computing is used in nearly all areas of computer science. And although terminology and focus may vary, there is often a common ground. In this chapter, existing results and solutions, which originate not only from the obvious areas of notifications services and distributed systems, are related to the key characteristics of the scoping approach. However, there is hardly any work that focuses on event visibility in general and addresses the different aspects involved:

Message Visibility: The basic goal is to control the visibility of notifications, limiting their distribution. Any communication substrate that supports one-to-many communication faces the need to determine the receivers of sent data. The approaches taken can be related to scopes and their ability to group eligible receivers along multiple dimensions.

Engineering: In order to cope with the complexity of large applications, abstractions and modules exist in all areas of computer science that hide details of implementation and internal structure. Building up structures and the coordination of composed members has turned out to be the major engineering issue. It consists of an application and an infrastructure related part.

- **Application:** Decomposing an application and identifying its structure enables engineers to modularize and reuse functionality. The inherent problem is to first delimit modules and then organize their interaction. A module construct for event-based systems might draw from existing work on component engineering in these points.
- **Infrastructure:** The delimitation of application modules becomes an infrastructure issue when regarding effectiveness, scalability, and administration. In view of heterogeneous applications it is clear that no one-size-fits-all approach of event notification will fit the needs. Instead of manually bridging different domains one may want to structure the service implementation itself, without giving up the single model of the system. Existing work can be compared with respect to its ability to identify system structure (in event-based systems), integrate varying implementations, and be adapted to application needs.

Management: Management support is more difficult and more important to achieve in event-based systems as it is in other kinds of systems, because of the implicit interaction between components. Structuring both application and infrastructure can help to locate management functionality. In this way, issues of administrative domains, low-level implementation details, runtime control, etc., can be bound to this structure. Existing work can be compared with respect to its ability to support such management tasks.

Security: Still an open problem is how to control/enforce security policies, since interaction is only implicit here and cannot be bound to pairs of interacting entities as it is typically done in request/reply systems. Although this thesis does not investigate security in detail, the scoping approach obviously bears the potential to integrate security measures into event-based systems, and it is mentioned here to stress that further work is missing in this important area of event-based systems.

The related work presented in this thesis is compared to the above objectives and is drawn from the following areas:

- Distributed systems
  Related communication paradigms: multicast networking [91], group communication [250].

- Notification services
  The primary source of work directly related to this thesis [57]. Since details of the relevant work are referenced throughout the thesis, this section merely summarizes the characteristics of the main contributions.

- Rule-based systems
  A more abstract, often centralized, viewpoint with focus on specification of and programming with events and triggered actions [154].

- Active database systems
  An application of rule processing within database systems [242], integrated with transaction processing and database languages.

- Coordination models
  This field addresses the general problem of orchestrating autonomous components [197, 237].

- Software engineering
  A wealth of different topics is related to event-based systems: implicit invocation [132], software architecture [134, 192], distributed debugging [29], component models [27], etc.

- GUI design
  An early source of event usage [178, 267], at least as far as the term 'event' is concerned, although primarily callbacks are used.

6.1 Distributed Systems

The basic point of view taken in this thesis is mainly that of distributed systems. This field is obviously a major source of related work which addresses the objectives of managing application and system structure and controlling the distribution of messages. The following topics are considered: managing structure in middleware, related communication mechanisms (multicast, group communication, peer-to-peer, direction diffusion, etc.), and the use of events in specific distributed application scenarios.

6.1.1 Middleware

The two most important standardized middleware platforms are Corba and Java 2 Enterprise Edition. The issues related to notification services and component models are discussed in 6.2 and 6.6, respectively.

Corba

Besides the Corba event and notification services, which are described in 6.2.1, the Common Object Request Broker Architecture [224] provides a number of mechanisms to organize and structure distributed systems. Corba domains [249] bundle sets of objects, which may be grouped by some common characteristic of administrative or technical nature. These domains are used, for example, to install security policies in a specific set of brokers, and to delimit implementation details like inter-ORB protocols and implementation repositories. A domain is represented by an object and can be stacked recursively to include subdomains. Bridges handle the traffic crossing domains. They control and restrict inter-domain communication and transform object references necessary to access an object, and hence control any call on these references. Corba domains pursue objectives comparable to the scoping approach presented in this thesis. However, Corba domains are mainly a means to cope with infrastructure differences, such as protocols, representation, and security domains, and they are constrained to handling object calls. They are not targeted at engineering issues of application design, and more importantly, they do not integrate notification service bridging.

Corba's management service facilities comprise the managed set service, the instance management service, and the policy management service. The first is used to manage groups of objects, which have to implement the service' `Member` interface, the instance management service offers lifecycle management of these objects, and more related to the ideas in this thesis, the policy management service binds policies to object domains. The latter is used to set initial values in managed objects and to control their evolution at runtime. All of these are related to the idea of configuring distributed application components in general, but they do not address the problem of composing new, *event-based* components out of existing ones.

J2EE

The enterprise edition of Java (J2EE, [282]) specifies an execution environment that contains a component model and a number of standardized services, including a notification service (JMS), which are detailed in sections 6.6.3 and 6.2.2, respectively. In terms of managing application components, the Java Management Extensions (JMX) defines APIs for a (partially) standardized way to application and network management and monitoring [283]. The managed entities are called Managed Beans (MBeans), which are either Java objects or external resources controlled by a JMX agent. JMX also defines publisher and subscriber interfaces that allow MBeans to notify other entities in the management framework. Unfortunately, the necessary interface to event services is not part of the specification. But similar to the Corba services presented above, this functionality may be used to implement some of the ideas presented in this thesis, but they do not address the problems themselves.

6.1.2 Communication Paradigms

Communication paradigms other than the plain publish/subscribe approach [233] are also relevant when considering the implementation of event-based systems. Once deployment environment and structure of an application is known, one might reasons whether a plain pub/sub service (Section 6.2) fits the needs best or whether multicast, group communication or even some peer-to-peer substrate offers the appropriate combination of quality of service and performance. But can those paradigms be used to implement features of scopes? What kinds of structuring mechanisms are available?

Multicast

When considering the implementation of event-based systems and application structures exposed in scope graphs, multicast communication techniques are an apparent candidate. Multicast techniques were originally explored on bus networks, like Ethernet [86, 208], they were incorporated in the Internet Protocol [90], and are recently also considered on the application layer to improve configurability, and because of the limited availability of the Internet Multicast backbone *MBONE* [25, 157, 255].

Multicast primitives offer one-to-many communication, transmitting a single message to (potentially) multiple receivers. The original network level multicast exploits broadcast abilities and tree-based routing to ensure that a message is sent at most once over any physical link, which cannot be guaranteed by application layer schemes. Deering and Cheriton [91] presented a good introduction to using multicast in wide area networks (WAN) and to the early routing protocols. Even more than in group communication (see below), multicast concentrates on the efficient dissemination towards sets of receivers, so called multicast groups. In IP Multicast, like in many other schemes, these groups are independent, without any relation between the sets. They are, therefore, per se not able to model application structure. Another problem arises due to the fact that routing is based on group identifiers alone, prohibiting a combination of consumer interests and application structure. Multicast groups are arranged according to either content or structure; similar problems exist with the subject trees of subject-based publish/subscribe [212]. However, as was shown in Sect. 4.5.7, multicast is an appropriate communication technique to deliver notifications to edge routers of a specific scope, which implement the so-called fan-out of the routing network and deliver notifications via point-to-point communication to consumer nodes.

There are a number of extensions to this simple model providing more control of the distribution of messages. Multicast scope control, as Deering and Cheriton [91] call it, can be accomplished by using the time-to-live fields (TTL) of IP packets. Routers reduce the TTL on each hop and discard packets if their TTL is zero, but TTL scoping is orthogonal to application needs and has proven to be difficult to understand and use reliably. To define multicast scoping explicitly, administratively scoped IP Multicast was proposed [209], defining a range of IP addresses (239.0.0.0/8) that can be subdivided to represent hierarchical administrative boundaries. Each (sub)range of addresses constitutes a *multicast scope* and covers a set of interconnected adjacent routers and network nodes, and thus has a topological meaning. Boundary routers discard all packets of this range that are about to enter or leave the set. They limit the distribution of packets, while address allocation [151] according to application needs within these ranges is only affected by the now reduced number of available addresses. Multicast scopes bundles network nodes, but does not support communication between scopes and require static configuration within the IP network routers. Nevertheless, they may well be used to implement those scopes in event-based systems that are rather static and correspond to network layout. Interestingly, mapping a subset of the scope graph on administratively scoped IP Multicast allows to transparently integrate existing work on implementing publish/subscribe on top of multicast [22, 235, 277].

Group Communication

Multicast and group communication [250] constitute two ends of a spectrum of communication systems, which are most notably demarcated by the works of Deering [90] and Birman and Joseph [39]. Group communication is based on the notion of groups of processes, and deals with their management and the communication within groups. It provides a generic abstraction to reduce the complexity of handling multiple recipients. The actual set of receivers within an addressed group is determined by group membership and the protocol used to deliver the message.

Groups are mainly understood as a means to delimit sets of receivers. One of the first works on process groups is the V Kernel [69], which allows processes to be in one or more flat groups, but no nested groups are supported. In the renowned ISIS system [38] nested groups are available, but only to provide a multi-level set inclusion. A message sent to a specific group is always delivered in all subgroups and not into any parent groups at all. In the terminology of this thesis, hierarchical process groups predefine static interfaces that accept all incoming and block all outgoing traffic, and thus are more restricted than scopes.

An interesting, different notion of subgroups is presented by Jenkins et al. [164]. A gossip protocol considers probabilities of infectivity and susceptibility between any two pairs of communicating nodes to determine in which directions a message is forwarded. Groups emerge as clusters of members whose probability is high for internal and low for external communication.

A major contribution of group communication are protocols that guarantee reliable delivery in case of failures, atomic delivery, and the consistent handling of group membership [39]. Other issues of implementation are closely related to multicast—the notion of host groups [68] has identified commonalities between the two.

To summarize, part of the work on group communication nicely complements the presented scoping model. While scopes provide means of engineering superior to groups, the work on reliable delivery *within* specific groups can be used, for example, to implement advanced delivery policies.

Peer-to-peer systems

In order to facilitate self-managed networks the idea of combining publish/subscribe architectures with peer-to-peer systems (P2P) has recently attracted considerable attention [65, 244, 291, 295]. The efficiency of publish/subscribe is closely tied to the topology of the underlying network, the design and management of which, however, has been neglected so far. P2P systems are designed to resiliently cope with frequent node failures and changing participation, offering inherently bounded delivery depths, load sharing, and self-organization. These features may be exploited for intra-scope communication, but the efforts to introduce structure in peer-to-peer systems, which improve routing and management, are not yet applied on any combination of pub/sub and P2P.

6.1.3 Other Notions of Communication

The principal ideas of event-based computing and publish/subscribe are known and used under various terms. Directed diffusion[161] is a term used in sensor networks [10] to denote publish/subscribe communication. It relies on a model of routers, subscriptions, and routing protocols very similar to those used in the Rebeca notification service presented in Section 2.4, but with more emphasis on energy efficiency. The dissemination of subscriptions can be limited according to a topological scope.

Other approaches to communication resembling pub/sub exist, but they mostly resort to some static configuration of multiple instances for delimiting traffic; take, for example, push services such as broadcast disks [4], web caching [210] and content delivery networks (CDN), both in industry [9, 261] and academia [254].

6.1.4 Application Scenarios

Various scenarios of event-based communication usage exist that demand certain system structures for consistent operation. As examples, network management and workflow systems are sketched.

Network Management

Network management has a strong need for sophisticated monitoring capabilities of runtime statistics, alerts, and configuration changes [166]. All of these are subject to filtering and independent processing in different management applications. Obviously, event-based communication of the necessary data is indicated. The OSF Distributed Management Environment (DME, cf. [120]) defined a set management service and the International Standards Organization (ISO) standardized with Open System Interconnection (OSI) a whole suit of network communication and management protocols—both include event management. Both approaches failed to have major practical impact [259], whereas the Simple Network Management Protocol (SNMP, [260]) is prevalent today. It uses *communities* to delimit management topologies and tasks [64], which are but strings in the management data that are filtered as needed. Although similar to the collapsed scope architecture presented in Chapter 4.2, there is no explicit delimitation of traffic as it is possible with scopes. The resulting side effects were considered one of the major architectural drawbacks, which are addressed in the recently passed SNMPv3 [152]. A number of SNMP Entities can now be federated in a network and applications register at one of the entities [63]. Managed objects can be grouped in 'scopes', which are, however, separated from each other and have no equivalent in the underlying point-to-point communication.

Workflow Systems

Until recently, workflow systems [142] were typically built around centralized data stores, incapable of supporting inter-organizational processes that typically evolve separately and are based on different workflow engines [11]. In order to distribute workflow control and execution the individual tasks must be decoupled. This is accomplished by taking event condition action (ECA) rules as basic execution model. ECA rules were developed for active database management systems (cf. Sect 6.4, [5, 88, 242]) and trigger an assigned action only if the specified event occurred and the condition is valid. Event notification services are a building block of such distributed activity services, which are investigated in themselves [74] and as part of distributed workflow systems [51, 82, 167, 184]. They facilitate the decoupled operation necessary in these scenarios. However, typically only simple event mechanisms are applied, raising the problems outlined in the introduction of this thesis.

For example, Geppert and Tombros [143] introduce *EVE*, an event-based infrastructure to build distributed workflows. The distributed set of EVE-servers relay notifications according to announced interest, that is, similar to the simple event-based system model with no explicit structure in the system. Only in event composition the set of base events can be limited to events of the same workflow, which resorts to filters on workflow IDs. The problems arising from missing structures in workflow systems are pointed out by Eder and Panagos [98], and their arguments are covered by the discussion given in Sect. 3.1.2. Their system connects engines from multiple sites with the READY notification service [147]. It supports administrative domains on a topological basis (see below).

6.2 Notification Services

The work presented in this thesis originated from the area of event notification systems in which a considerable amount of work exists and many concrete systems have been designed and implemented, both in industry and academia. A comparison of the wide variety of existing notification services would require a formalization of the provided function. In most systems, both commercial and academic ones, practitioner's approaches dominate. A precise definition of semantics is at most given for subscription languages, e.g., [7], neglecting the semantics of the event service itself.

A thorough comparison would require a formal specification of the functionality of these systems, which is beyond the 'scope' of this thesis and for which adequate tools in form of a sound theoretical abstraction of event-based systems are still to be developed. The formalization presented in Section 2.3, and in

[114, 213, 297], facilitates reasoning about event-based semantics and correctness of implementation, but a detailed comparison based on this or a similar formalism is an open issue of future work. The following discussion continues to favor the practical aspects and points out the most characteristic differences.

The following paragraphs consider the widely used API specifications of the CORBA Notification Service and Java Messaging Service, some commercial systems, and research prototypes.

6.2.1 Corba Notification Service

The CORBA Notification Service is an API specification as part the Common Object Services of the CORBA platform [231].[1] It relies on channel-based addressing, i.e., publishers have to get a reference to a specific channel, which is a regular CORBA object, and send their notification data into this channel. Conversely, consumers choose a channel and may additionally specify a content-based filter to select a subset of notifications sent through this channel. The channels offer a structuring mechanism in that notifications are originally visible only within the channel they were published in, while any content-based subscriptions are subsequently applied to those notifications visible to the consumer. Channels thus partition the visible notifications, and if they do not just classify notifications according to their content or type, they may, to some extent, model structure in the same way as scopes do.

Still, there are a number of problems inherent to a channel-based solution. Producer and consumers, that is, the application components, have to deal with channels explicitly. They have to select the right ones moving information about application structure into the components—there is no support for the role of an administrator to arrange channels, producers, and consumers from a system's point of view. And this also limits system evolution, since the set of channels referenced by applications is static, a problem which is only recently addressed by reflective middleware [85]. Regarding the structure of a system, CORBA channels cannot reflect any hierarchy because their traffic is completely separated. Although event management domains [229] support the federation of multiple channels in arbitrary topologies, they do not offer any filtering of notifications between coupled channels.

As mentioned in 6.1.1, Notification Service instances may be federated with the help of ORB domains. However, the necessary bridging between these domains has to be set up manually. The domains are mostly seen as means to model the network and broker infrastructure [249], they are not targeted at engineering issues of application design. So, in the end one can only assess that the standardized API does not support visibility control and system management sufficiently well, but the CORBA Notification Service may serve as a communication technique to realize a subset of a scope graph.

[1]It subsumes and obsoletes the CORBA Event Service [227].

6.2.2 Java Message Service

The Java Message Service (JMS) is an API specification as part of the Java 2 Enterprise Edition (J2EE) [282, 284]. Different to the CORBA solution, it can be used without the enterprise object platform it is part of. JMS coined the term topic-based subscription, which stands for message grouping according to abstract topics plus content-based filtering on a set of header fields and properties, similar to CORBA's channels. And this is the main design objective, namely, to provide an API layer that can be put on top of many industry messaging and pub/sub products, including the CORBA Notification Service. So, the specification leaves open how to define topics or how they are interrelated. An API implementation, called JMS provider, is responsible for determining the exact semantics of topics, their syntax and management; JMS relies on an abstract `Destination` object only. If topics are grouped in hierarchies, e.g., with a dotted notation in subject trees like `stocks.technology.europe`, JMS providers can opt for topics to represent only leaves or whole subtrees of a subject tree.

The Java Message Service is perhaps becoming the dominating messaging API, and lots of commercial, academic, and open source notification services are implementing this API. It supersedes the CORBA Notification Service. But as a very generic interface, JMS does not provide any structuring mechanism or means for managing event-based systems as part of the specification. An implementation of the publish/subscribe part of the API is also available on top of REBECA, offering engineering and management capabilities in a standard JMS environement [95].

6.2.3 Commercial Systems

Event-based systems are in use in industry for a long time, sometimes with an infrastructure that is not specially optimized for event-based operation. On the other hand, message-oriented middleware (MOM) and publish/subscribe services are often deployed to alleviate some shortcomings of request/reply, but not to deliberately realize event-based communication. Major areas include financial industry [23, 306], control systems [188], enterprise application integration (EAI) [165], and more generally news and content distribution [9, 261]. Lots of products exist that implement (and extend) the Java Message Service, CORBA Notification Service, or some proprietary interface. Interestingly, the need to structure systems was early recognized in commercial systems, mainly due to the more relevant demand for controlling data dissemination and security. Available products consequently concentrate on setting up management domains for these objectives, often manually and more or less statically. The typical main omission is any support for engineering event-based systems and adapting their semantics.

TIBCO's Information Bus (TIB, later renamed to TIBCO Rendezvous) is one of major pub/sub products [292, 293], which is based on the ideas of Oki et al. [233]. TIBCO uses, and has patented [269], subject-based addressing. Subscriptions select from a hierarchy of subjects single nodes or entire subtrees with simple pattern matching on a dotted notation of subjects, e.g., `stocks.technology.*`. The mapping from subjects to underlying transport protocols, in particular to specific IP Multicast addresses, has to be done manually, and it is statically encoded in every producer and consumer.

Although the inherent communication efficiency of multicast is appealing, it comes at the cost of a rather static configuration, which not only complicates maintenance, but also restricts configurability and integration, and thus the range of possible application domains [294].

In order to prevent interference between several applications running in the same network, a service parameter configures every data transport to belong to a distinct, completely independent application. This is obviously only a very weak mechanism to control dissemination, which is equivalent to a mandatory additional filter or channel-based publish/subscribe. Additionally, independent networks of Rendezvous instances can be connected with routing daemons. They forward messages between networks so that subscribers can transparently listen for subject names and receive messages from other networks. Administrators managing the daemons have control over the subject names (and associated messages) that are relayed and flow in or out of a network. These daemons offer a basic means of structuring, even though mainly for administrative purposes like CORBA domains.

Oracle Advanced Queuing (AQ) is a queuing and notification service on top of the Oracle database management system [236]. It offers a JMS implementation and proprietary point-to-point and point-to-multipoint queues, the equivalent of topics in AQ. Queues are implemented on top of database tables, i.e., data sent to a queue is appended to the corresponding DB table. AQ uses content-based routing in that a subscription to a queue is mapped to a query on the queue's table. Newly appended data that matches this query is delivered to the subscribers. Since subscribers can itself be queues these filters can be used to connect queues with a specified interface between them. This is similar to the presented notion of scoping with interfaces between them. However, queues are typed and therefore are mainly a means to classify notification content, not application structure. The offered control focuses on data dissemination and access control, but not from a system engineering point of view.

Furthermore, Oracle AQ exploits all the functionality offered by the database, like transactions, triggers, consistency constraints, logging, replication, authentication, access control, etc., and applies it to the publish/subscribe communication as well. To summarize, Oracle Advanced Queuing is a feature-rich notification service that has many interesting characteristics, but because of its concentration on centralizing databases and relational queries, its focus is more on advanced QoS features than on a lean implementation. If these constraints match the requirements in a specific applications, or probably in only a subset of a scope graph, Advanced Queuing is a proper candidate for implementing scoped communication.

Other commercially available pub/sub implementations include IBM's Web-

Sphere MQ [78], previously known as MQseries, BEA's WebLogic [30], which also contains a JMS implementation, and Vitria's BusinessWare and other process support systems that often include a messaging component. Typically, these systems recognize the need to administer entities of messaging functionality like channels and server instances, but they do not support hierarchical engineering of applications nor the adaptation of delivery semantics as is available with transmission policies in scopes. As pointed out at the beginning of the chapter, a number of issues of controlling visibility are identified and addressed, but an approach for designing and engineering event-based systems is not available, mainly because visibility is not taken as basic principle for the design of these systems.

6.2.4 Research Prototypes

Siena

The Siena notification system [62] is probably the most widely known research prototype. Based on the thesis of Carzaniga [56] a number of publications detail the concept of content-based routing in networks of event brokers. Siena has formally defined the semantics of its subscription mechanisms. However, the semantics of notification distribution is not clearly specified and has several flaws. For example, according to the specification a notification is only delivered to a client if the client had a matching subscription at the time the notification was published; clearly an imprecise formulation. Moreover, clients are required to accommodate to race conditions. For example, notifications may be delivered after cancellation of the respective subscriptions. Finally, in Siena, a client that unsubscribes to a filter implicitly unsubscribes to all filters that are covered by the former filter, too. This approach burdens the client with keeping track of relations among the issued subscriptions. In contrast to that, the basic formalism presented in Chapter 2 serves as a basis for modeling visibility, and also for a sophisticated model of content-based routing in Mühl's thesis on Rebeca [213].

Siena concentrates on filter semantics and design choices of the distributed architecture [60, 61], focusing on network bandwidth efficiency but neglecting any support of engineering or structuring event-based systems. As with all other content-based filtering approaches, filters can be used to realize visibility constraints, as described in 4.1, but these issues are not explicitly addressed in Siena, even though the assumed global, flat namespace of notification attributes calls for visibility structures to not impede scalability.

Gryphon

The Gryphon notification system [158] is being developed at IBM Research. It was originally centered on the abstraction of information flow graphs [24, 274], which are used to model event-based systems. Nodes in these graphs are information spaces containing events of a specific type and edges select or transform events. In the Gryphon system, the information flow graph is implemented by distributing the spaces on a network of brokers.[2] Content-based routing, partly on top of IP Multicast [235], can be used for communication. But since the flow of information is specified in the graph, every edge may either be implemented using a (bundle of) point-to-point or a content-based routing communication. To facilitate this decision, a newer paper [37] introduced the knowledge graph structure that abstracts the information flow graph. It bundles information spaces into virtual brokers and interconnects them with virtual links. The authors then use an implementation strategy to map each of the virtual brokers to the physical network. So, the bundling in the knowledge structure gives hints for the actual mapping of virtual to physical brokers and the communication technique used to connect them.

Gryphon's knowledge structure incorporates some of the ideas that underlie scoping. Yet Gryphon does not focus on engineering issues. The structuring mechanisms are applied to govern the physical deployment of systems and facilitate quality of service handling and controlled utilization of network resources, similar to CORBA domains. Whereas scopes serve as a more general module construct that may similarly govern the mapping of structures to a specific deployment environment, as pointed out in 4.1, but which also offers an abstraction for reuse and adaptation. Furthermore, the available information on Gryphon addresses only static configurations, does not allow for hierarchical structures, and does not investigate the general problem of implementing these structures.

READY

The READY event notification service [146] introduced *event zones* to partition components based on (either) logical, administrative, or geographical boundaries and to delimit the visibility of events. Boundary brokers connect zones and control the communication between them, and may enforce security policies on connected clients. Although similar to scoping, zones resemble more the domain idea of CORBA as it mainly addresses control on the physical routing network, the engineering aspect is lacking. For instance, in READY a component belongs to exactly one zone so that there is only a two-level hierarchy. The system is structured only based on one specific point of view, prohibiting composition and mixing of aspects [153]. Heterogeneity issues are only mentioned in READY: boundary brokers could apply transformations on crossing notifications. Following the idea of CORBA domains, brokers operate here on a rather coarse and static granularity, whereas event mappings (Sect. 3.4) allows for syntactic and semantic mappings in the formal model and at every layer of abstraction in a scoped system, outlining a development path to integrate existing work on data integration, cf. Sect. 6.4.3. READY is a prime example of an initial study of visibility issues that identifies the problem but fails to combine the different aspects of visibility.

[2]Although it is unclear whether the mapping is done automatically or manually.

6.2.5 Other Related Work

A number of other event notification services exist that do not address visibility directly, yet offer some functions with which visibility constraints may be implemented. The JEDI notification service [83] uses content-based filters on tuples of data and operates in networks of event routers. It may adhere to visibility constraints expressed in its filter language, but similar to SIENA it assumes a global naming scheme for notification representation. All clients of such a service are forced to agree on the same data model used for all notifications. This impedes system integration and limits scalability.

The JECho pub/sub service [307] introduces eager handlers that divide traffic shaping and filtering into two parts, one resides at consumer nodes and one is moved to producer(s). This technique is similar to the more general interceptor concept [118, 265], but has in this context two major flaws. Regarding the management of propagated handlers, this approach is equal to direct registration of each subscriber at every eligible producer, accumulating lots of handlers at producers and inevitably leading to scalability problems. On the other hand, event handlers are defined by consumers and they are not externally administrable. So, even though interceptors are a way to implement visibility constraints on producers and consumers, there is no support for the role of the administrator—aspect-oriented programming and reflective middleware seem more appropriate for this purpose (see below).

DEEDS [96] is a channel-based pub/sub service that concentrates on an extensible software architecture for propagating events. It employs system routing assistants as plug-ins in channel implementations, which were intended as wrappers of the underlying communication infrastructure but may as well be used to implement visibility constraints within a channel. They adapt the interceptors concept and make it accessible to administrators.

An event-service for virtual reality applications is proposed by O'Connell et al. [232]. The authors define a hierarchy of *zones* to limit the distribution of events for efficiency reasons. Events can only cross the boundary in downward direction and no other features of scopes are mentioned. COBEA [194] runs a publish/subscribe service in a CORBA environment, and may exploit available if limited structuring mechanisms provided by CORBA domains. Hermes [245] does not only investigate peer-to-peer topologies, but also introduce a role-based security model for event services. Elvin [266] is another renowned notification service which has structuring capabilities only in form of manually established links between routing domains.

Mansouri-Samani and Sloman [199] present an event service that concentrates on event composition and they do not explicitly consider distributed event services or visibility. There is an 'event disseminator' component responsible for sending events. However, their system is based on Regis/Darwin [196], a distributed object management tool, and it facilitates runtime deployment and configuration of components. Such an integration of scopes into system management platforms is an important direction of possible future work.

6.3 Rule-Based Systems

A rule-based system (RBS) consists of a set of if-then rules, a set of facts, and an interpreter controlling the application of the rules [154]. The evaluation and matching of conditions ('antecedents') and execution of an assigned action ('consequents') is similar to the event-based approach of publishing notifications, matching subscriptions and delivering notification data to the application program.

Lots of work exist on RBS, but they typically concentrate on condition detection [119] and rule languages [211], whereas event services look at the flow of notifications in distributed systems. On the other hand, these areas seem to complement each other with respect to rule or subscription construction and management, i.e., visibility control. Rule groups and modularity are, for example, investigated by Browne et al. [43] and are used in a number of systems. However, rule groups are connected by explicit procedural invocation and the modularity cannot be used to create new event-based components.

Based on a formal description of rule systems, Wu et al. [302] offer a thorough formal presentation of a declarative rule decomposition mechanism that enables parallel rule processing.

6.4 Data Management

Database research is related to event-based systems and scoping in that it provides an event-based programming model: (active) database management has generalized the aforementioned rule concept (Sect. 6.4.1), which is lately being considered separate from the database system (6.4.2). Based on schema descriptions, techniques for data integration are exploited to map different representations (6.4.3).

6.4.1 Active Database Management Systems

Active database management systems (ADBMS) extend classical databases with a reactive functionality [242]. Event condition action (ECA) rules are used to encode reactive behavior [88, 89]: If the specified event occurs and the condition holds, the associated action is executed. For instance, fund withdrawal in banking applications result in change events once the amount is subtracted from the account. A rule on such events may prevent withdrawals above a certain amount or may control loan limits, and if the limit is reached, the action of the ECA rule aborts the transaction, revoking the transfer. Rule processing is controlled by an execution model that determines execution dependencies between the triggering and the triggered (trans-)actions, i.e., transactional contexts, coupling modes, and consumption modes [47].

ECA rules control and react to data changes and they decouple application computation from the control over shared data, called 'knowledge independence' [126] in this context. They make it possible to bind reactions to data changes without affecting application code. This obviously relates ECA rules to publish/subscribe, but also to the more general coordination paradigm [237] considered in Section 6.5, and, interestingly, to the notion of aspect-oriented programming [76], cf. Sect. 6.6.5.

Research on rule management concentrated on termination and confluence properties of rules [8]. Only little work is published that addresses the engineering complexity of ECA rule creation and management, although it is known that rule analysis and control of side effects is hard [20, 26]. Baralis et al. [26] propose stratification of rules as a modularization technique in order to simplify termination testing. Each rule is assigned to one stratum according to a given metric and the strata are ordered by priority so that rules with lower priority will not influence those of higher priority; termination testing can now be done for each isolated stratum in turn. The presented metrics are behavioral, assertional, and event-based stratification. The first subsumes the other two, and it groups rules based on functionality, separating levels of abstraction in a way. Assertional stratification tries to ensure a predefined postcondition with the rules of the stratum. Finally, event-based stratification orders rules so that lower strata do not produce events consumed by the upper ones. This approach can be seen as a simple design methodology, which also lends itself to the design in arbitrary event-based systems.

Kappel et al. [168] propose rule patterns, which offer a mechanism to bundle a parameterized set of rules. They facilitate constructing complex reactive behavior out of simpler rules and in that offer a modularization technique, although recursive bundling of rules and the creation of new, first class components is not directly supported.

6.4.2 Reactive Functionality

Reactive functionality available from ADBMS is coupled with the database management system itself. As consequence of the increasing demand for reactive behavior, research efforts started to draw event handling out of the databases. The complex DBMS was unbundled to ease reconfiguration and reuse individual services [127, 137]. Motivated by this development, independent reactive functionality services were developed [71, 176]. Both this approach and the proliferation of event notification services stem from the same idea: separate reaction management from application code for increased flexibility.

These services provide means to detect and compose events, to specify reactions, and to control their execution. However, in search of an infrastructure service that is independent of specific applications, engineering issues were neglected and the services miss to identify and sustain application structures directly. They rely on underlying notification mechanisms and concentrate more on reaction handling than notification visibility issues.

6.4.3 Heterogeneity and Data Integration

As event services are the basis for application integration and evolution, they cannot be expected to run in homogeneous environments. System parts evolve independently and have different objectives so that their demands on the quality of service and the content of data is varying. Unfortunately, while known and understood in traditional request/reply systems, heterogeneity issues are seldom considered in event systems. Database research can contribute to the necessary syntactic and semantic data mappings (federated databases [268], event composition in ADBMS [67, 139, 304], ontology-based transformations [41]), as well as existing approaches in notification services [189, 246]. The event mappings of scopes are a means to integrate these results into a scoped event-based system.

Renowned notification services like SIENA, JEDI, and Elvin do not address heterogeneity, whereas CORBA domains and similar concepts in notification services, like READY zones and TIBCO routing bridges, allow for *some* support along their rather coarse structures. Practical solution are given by Bates et al. [28] and Cilia et al. [74], for example. The latter is based on a model describing semantic mappings, the MIX model [41], and introduces the notion of concept-based publish/subscribe. It allows clients using divergent data formats to be seamlessly integrated by applying externally provided interpretations and transformations to the exchanged notifications. Since their proposal is lacking an abstraction of application structure, a promising combination of scopes and concept-based pub/sub could identify homogeneous groups that agreed on a common data format (scopes) and transform crossing notifications between the applied concepts [72].

Besides the heterogeneity of data representations, differences in the required and the provided quality of service are the second major source of incompatibilities. Note that this problem is only partially addressed by parameterized APIs, for the API itself and its implementation is subject to change and evolution. Multiple channels might be used to wrap different implementations [96], but with the need to deliberately access the right channel the decoupling of producers, consumers, and their interaction is diluted. For the first time, scopes provide the engineer with a tool to tailor the implementation of a specific *part* of a system, with well-defined interfaces and without need to affect client components. They open the implementation of notification services in the vein of Kiczales [169] to facilitate adaptation of event-based systems.

6.5 Coordination Models

The field of *coordination theory* investigates scenarios and techniques for managing the dependencies between a set of active components [197]. The coordination paradigm differentiates computation from coordination [237] and it makes the interaction between components explicit in *coordination media* [70], which offer means to control and adapt a system's configuration. Event-based communication directly corresponds to this viewpoint, and events are only one form of coordination out of the wealth of models proposed in literature [15, 50]. But not all of them reduce component interdependencies, let alone offer scalability. For example, it was criticized that race conditions are possible in Linda [54, 140] and its variants, resulting from the inherent concurrency of the model [6].

From a coordination point of view, scopes reify the structure of event-based applications, and as a coordination medium each scope functions both as a communication medium transferring data as well as a coordination space controlling the interaction of the grouped components (e.g., via transmission policies). Scopes introduce an external control of event-based systems, termed objective coordination[3] [257] as opposed to the subjective coordination of the plain event model, where delivery of notifications is determined by the issued subscriptions only. In fact, scopes combine the two modes of coordination, because within a scope the plain, subjective model applies, while the interaction between scopes is controlled by an outside entity, i.e., the administrator.

Research on Linda-like systems investigated structures of components. However, the need to specify names or identities of tuple spaces is a major characteristic/drawback of many works on hierarchical tuple spaces [55, 141]. In this way, the same negative arguments hold as for the manual selection of event channels and subjects, both draw coordination control into the application components. Lime [221] realizes a transparent access to multiple tuple spaces, although the approach is limited to a three-level hierarchy bound to the physical layout of the system. It is focused on the intended application domain of mobile agents and does not offer a general solution.

Agha and Callsen [6] propose ActorSpaces to limit the distribution of messages. The basic drawback of their approach is that even though previously unknown objects are intended to cooperate, senders have to specify destination addresses. The sketched implementation is rather limited. Merrick and Wood introduce scopes to limit the visibility of tuples in Linda, but again, senders have to specify destination scopes [207]. Furthermore, scope nesting is restricted to two levels. Tuple Centres [234] enable customization and adaptation of the semantics of tuple space operations, following an idea similar to the transmission policies of scopes. Their general programmability and the automatic triggering of reactive behavior might position tuple centres as an appropriate technique to implement scopes, at least if the event-based behavior is not only used internally but also extended into the application components. With their general approach to implement tuple spaces they resemble the connectors idea of Sullivan and Notkin [276], whereas scopes can (partially) be seen as a more concrete instance of such models (see Sect. 6.6).

Different from Linda, event-based coordination media directly support the event-based style [49, 50, 297]. Moreover, control-driven models such as Manifold more strictly separate the coordination from communication [237, 238], although the event-based style is not directly supported, and in the case of Manifold, events are only used for configuration control not for application communication.

[3]or endogenous vs. exogenous coordination [34]

Commercially available are JavaSpaces [281], which also implement notification mechanisms to notify about newly inserted tuples matching a previously defined pattern, but they, too, fail to support structure apart from manually federated spaces.

6.6 Software Engineering

Both events and structuring of software systems are long known in software engineering. Events are the basis of flexible and evolvable architectures, and the need to structure complex system is a basic engineering principle that is reflected in concepts like information hiding and modularization [240].

6.6.1 Software Architecture

The field of software architecture is concerned with the overall organization of a software systems [134]. It corresponds to the coordination paradigm, since both deal with the high-level interaction of system components. The architectural point of view focuses more on the static, immutable characteristics of these constellations. Architecture definition languages (ADLs)[4] are employed to describe the high-level conceptual architecture consisting of components, connectors, and specific configurations [205] of these. Typical, well understood arrangements of connectors and configurations are identified as *architectural styles* [3], the patterns of software architecture, and events and implicit invocation is one of them. The event-based architectural style comprises exactly the concepts given in Sect. 2.1, featuring the independence of producer and consumer components [57, 134]. In fact, Garlan et al. [131] identified early the prominent importance of using events for the construction of flexible software architectures.

Rapide [192, 193] is an event-based architecture definition language, which defines the system's overall behavior by means of the events that components publish and consume. An architecture surrounds its constituent components with in- and out-events[5] and can therefore serve as a component itself, similar to the notion of scope interfaces. Connectors convey the events, and the set of receivers is determined with rules, which coincide with ECA rules of ADBMS. The rules explicitly interconnect the internal communication ports and also internal to external ports of the enclosing architecture. With their general capability to connect communication ports Rapide does not address event-based systems specifically. A particular architectural style, such as event-based communication not only between component and connector but within sets of components, is a function that must itself be represented by an extra component. Interestingly, these kinds of specification are directly implemented by the control-driven coordination approach taken by Manifold [238] and the more general Tuples Centres [234]. So, the semantics of scope interfaces and transmission policies

[4]also: Architecture Description Language

[5]In- and out-functions are available in addition to events.

can be described with ADLs like Rapide, which, however, do not attend to any form of implementation in distributed event-based systems.

A less abstract form of architecture are execution architectures [296], which describe the organization of the actually instantiated set of components in a running system. Architectural events describe the evolution in terms of newly created or destroyed components and connectors, and thus monitors system execution. This recapitulates the early application of event-based style in distributed debugging [29, 191], and might be used to visualize scoping.

6.6.2 Software Integration

As mentioned above, Garlan et al. [131] emphasize the importance of events for flexible software systems, which is corroborated by [275] and others. One of the first contributions is the Field environment [256], an early work on tool integration that is built around a centralized server that distributes messages. Messages sent to the server are selectively re-broadcast to receivers that have registered patterns matching the message. The original approach realizes content-based filtering in a flat space of notifications. With the Field Policy Tool added later it is possible to adapt delivery semantics with manually provided mappings of messages to sets of receivers. But since these mappings are not associated with the application structure and operate in the flat system, they are very hard to control, cf. ECA rules.

The InfoBus [77] is a small Java API that facilitates communication between several JavaBeans or cooperating applets on a Web page. Multiple instances of InfoBus might be manually connected with bridges, providing a limited means of structuring, but without any inherent interfaces or composition support. Matching of messages is done by names, i.e., string matching. Besides being limited to one virtual machine, it is a tool for connecting components not for composing new ones.

Ported objects in CodA [204] are objects which communicate by processing messages arriving at ports. A port is connected to data streams by an external binding that is not controlled by the object itself, similar to control-driven coordination. A compound ported object encapsulates a number of ported objects and hides the data flow inside, the basic mechanism of modularization that is also applied in grouping producers and consumers in scopes. The presented meta-level configuration enables the adaption of compound behavior, drawing from the more general meta object protocol [170] that reifies interaction in object-oriented programming. These ideas influenced the notion of transmission policies and event mappings in scopes to facilitate handling of notifications in a similar manner.

Sullivan and Notkin introduce mediators [276] as a design approach that explicitly instantiates and expresses integration relationships and separates them from component function. An implicit invocation (i.e., event-based, see below) abstraction is used to bundle components and mediators, and, with its own interface, to compose new components. A similar approach regarding visibility is used as in the scoping model, but no default semantics is outlined so that they 'only' suggest a framework that facilitates design without identifying features that are attached to visibility: transmission policies, security, let alone implementation in distributed systems. In a less general approach, Evans and Dickman defined *zones* to support partial system evolution [103].

Barrett et al. [27] present the event-based integration (EBI) framework as a common ground for comparing event-based integration approaches. The framework consists of an abstract class hierarchy to compare different implementations thereof in existing systems. Unfortunately, the classes are defined only informally and often in an inaccurate way. The framework includes: participants (informers and listeners), brokers forwarding messages, and message transforming functions that map and filter notifications. The framework does not differentiate centralized and distributed broker implementations. From a functional, black box point of view, this may make no fundamental difference in 'classic' notification services where brokers are distributed only for efficiency reasons. But completely disregarding distribution aspects also excludes many other related features of implementation (like reliability, communication efficiency, etc.). This is the main difference to the scoping model presented here, which makes the system aware of its structure and may comprise various implementation approaches. Delivery constraints are sketched and are meant to shape message streams and modify delivery semantics on per-consumer basis only, e.g., ordering and priority issues, while their role within the routers is not specified. In order to support hierarchical integration, a limited form of grouping of participants is identified, where each group is centered around a single common broker. Concluding, the EBI framework fails to identify visibility and application structure as a fundamental concept for both design and implementations. Brokers offer some of the functionality inherent to scopes, and constrain the physical layout of the implementation while scopes are also a design tool that can be mapped on multiple, different implementations. In the end, the framework falls short of coping with issues specific to distributed systems.

6.6.3 Component Models

The discussion of CORBA and J2EE in Sect. 6.1.1 covered the management capabilities of these component models. Here, the programming support for event-based applications and components is investigated. J2EE defines three relevant interfaces: Java Management Extensions (JMX), Java Message Service (JMS), and the Message Driven Beans (MDB) in the Enterprise Java Beans (EJB) component model. JMX provides for an external (management) access to components and JMS offers communication facilities. Both are described above and are the basis of message driven beans. MDBs are asynchronous message consumers[6] that are instantiated and managed by EJB containers; the EJB specification deprecates using entity or session beans as JMS consumers to not

[6]As of EJB 2.1 messaging is not restricted to JMS alone and MDBs can be built on top of other services as well.

undermine container-managed access to these beans. An EJB container pools multiple instances of a deployed MDB and subscribes itself to the JMS topic given in the MDB deployment descriptor. For each incoming message the container selects arbitrarily one of the pooled instances and forwards the message to it. Hence, MDBs are stateless for other clients in the sense that instances are interchangeable and any two messages are not guaranteed to be processed by the same MDB instance. MDBs enable system engineers to exploit EJB container facilities like replication and management for message consumers. On the other hand, they are the only way to put message consumption, and only the consumption, under the control of the application server. Message distribution, and thus visibility aspects, are not considered.

Corba also contains a component model [230], which was inspired and is closely related to J2EE Enterprise JavaBeans.

6.6.4 Programming Loosely Coupled Systems

The event-based architectural style is also known as *implicit invocation* mechanism [94, 132]. This term describes loose coupling in the context of classic procedure calls. The invocation of a procedure is divided into three parts: (i) a call on the caller's side is bound at runtime to a set of procedures, introducing a one-to-many indirection, (ii) the bound procedures are invoked concurrently, and (iii) the (possibly) multiple replies are handled. Implicit invocation is a mere implementation mechanism that may be used to realize either anonymous request/reply or event-based communication. Garlan and Scott presented delivery policies for implicit invocation systems [133]. Four different policies are distinguished: full (broadcast) and single delivery (1-of-n), parameter-based selection (filter), and a state-based policy. These policies are a subset of the transmission policies described in Sect. 3.5.

In [28] domains of common event semantics and transformation functions are mentioned. As with most other event systems one has to identify the boundaries and define the mappings manually, they are no first-class citizens. All event services can be used in this way: they serve as internal scope implementation and the scope structure can be implemented on top of the provided functionality.

Cardelli and Gordon propose a process calculus for mobile ambients [53]. It facilitates the management of a tree of ambients whose intended purpose of grouping computation resembles our graph of scopes. The calculus might be used to model scope graph dynamics, but communication across ambients is only indirectly supported and destination identities must be known, similar to the approach of Bauhaus Linda [55].

In terms of programming abstractions and languages for event-based computing a number of proposals exist that support or make use of event-based communication, for example, [87, 101, 102, 155]. Furthermore, design patterns have been used to encode loose coupling [129]. The Mediator pattern simply reifies bidirectional decoupling of interactions. In contrast to that, the Facade pattern offers an abstraction of a set of entities, delegating calls to the proper destination, which is partially the intent of scopes, too. The observer pattern is considered to describe a publish/subscribe interaction, in which a subject manages a set of observers that are called whenever the subject changes. In contrast to a ‘real’ event-based architectural style, it requires consumers to be registered directly with each producer. Though applying an event-based style, none of the approaches considers visibility explicitly, let alone support it.

6.6.5 Aspects and Reflection

The above mentioned interceptor approach to adapt a component’s function is generalized in the emerging field of aspect-oriented programming (AOP, [173, 308]). AOP provides programmers with tools to separate code that deals with different aspects of a program, classic examples are application functionality versus logging or persistence. At compile- or runtime, the separated code is woven together to accomplish the combined functionality. Interestingly, AOP share a common problem understanding with other techniques supporting cross-cutting functionality, like ECA rules of databases [76]. Similar ideas are investigated in reflective middleware [79], but from a distributed systems’ point of view. Based on the idea of computational reflection [195], reflective middleware makes system features and functions tangible and adaptable. Obviously, the programming language focus of (classic) AOP can be exploited to implement such middleware [92, 252][7]. Likewise, these techniques are candidates for implementing scopes, as well. On the other hand, transmission policies, mappings, and tailored implementation techniques shape event-based communication and separate communication control from component code. They may thus serve as a tool to deal with aspects of event-based communication, pointing out an open issue of future research.

6.7 GUI Design

The design and implementation of graphical user interfaces (GUIs) was one of the first areas applying event-based interaction schemata. The model view controller [144] and observer patterns [129] are examples for that. However, mostly a callback mode of interaction is used, in which an observer directly connects itself to a previously known source of events. Furthermore, these systems are typically not distributed. An exception is Taylor et al. [289] who decouple applications from GUI toolkits using an event-based style and make even distributed deployment possible.

[7]Future versions of JBoss, a popular Java application server, will incorporate a runtime aspect weaving tool to adapt container functionality [48].

Chapter 7

Conclusions and Future Work

Events are of increasing importance in modern distributed systems. Growing interconnectivity and continuous evolution demand a loose coupling of communicating parties that traditional paradigms like request/reply can hardly provide. The event-based computing paradigm offers the required flexibility, but existing work has focused so far on scalability issues in terms of communication efficiency and size. Problems of system engineering and management, however, were often neglected and the targeted application scenarios are of rather simple structure, e.g., one-way information dissemination like stock monitoring.

This gives rise to two major problems. First, today's event-based systems mostly rely on flat design spaces and they are unstructured. Notification services convey data by matching it with issued subscriptions, not distinguishing subsets within the complete set of consumers. Any consumer may, in principle, receive any notification, and even without considering security concerns, the complexity to build, understand, and manage such systems easily becomes intractable. There are no means to predict the effects of creating or removing producers of notifications but to analyze *all* participating components. This approach is to some extent comparable to having programming languages with global variables only, where no inherent structure prevent (un)wanted access and effect.

Consequently, unclear side effects prevent complex applications beyond simple, uni-directional dissemination. This leads to the second problem, the currently limited applicability of the event-based paradigm. Without any effective control of notification distribution or any means of decomposition, the whole system acts as black box, which is configured by subscriptions only. The inability to tailor the service and adapt its semantics makes the integration of heterogeneous systems an error-prone and complicated task. And so, in many application scenarios the benefits of event-based interaction could not be fully exploited in the past.

Contribution of the Thesis

This thesis presents a scoping concept for event-based systems. Scopes reify system structure and hierarchically arrange groups of producers, consumers, and other scopes in directed acyclic graphs. They limit the visibility of notifications and govern their distribution within this structure. Scopes provide both a design-time tool and an implementation framework that addresses the aforementioned problems; they control side effects, decompose the system, and allow for adaption and management of its parts.

The approach taken in this thesis is to first investigate the abstract notion of notification visibility. All mentioned problems can be broken down to visibility issues, and while existing work addressed individual aspects thereof, the full implications have not been realized before. A thorough analysis of event-based interaction and its characteristics infers the need to support the role of an administrator, which is responsible for system engineering beyond individual producers and consumers (chapters 2 and 3). The control of interaction, which is extracted out of the components by the event-based style, is re-introduced as part of the administrator's role without destroying the loose coupling of producers and consumers.

Scope Model

Scopes are a tool that specifically support the administrator role. They make it possible to intercept notifications at scope boundaries and guide their further forwarding. Individual components are not affected by this scope control and the loose coupling is not impaired, avoiding the disadvantages of other structuring mechanisms in event-based systems. A formal specification of event-based systems is introduced, which covers the basic function of (existing) notification services, and it is extended to define visibility and scopes (Chapter 3). This definition includes scope interfaces that filter notifiations and determine those allowed to cross scope boundaries. Interfaces are composed of subscriptions and advertisements and make scopes behave like other producers and consumers in the system. Thus, scoping offers a component model for event-based systems [285].

The structure of a system reified with scopes is the ideal place to localize engineering and management tasks, which otherwise would be mixed into application components. *Notification mappings* are applied at scope boundaries to transform the representation of notifications. The need to agree on a common data model is avoided and each group of components may use its own appropriate model. A mapping is assigned to a scope and translates between the internal representation used within the scope and the external used outside in superscopes.

Furthermore, *transmission policies* customize notification forwarding at scope level, and thus facilitate scope programming. One form, delivery policies, direct notification delivery to only a subset of eligible consumers within a scope. They offer system engineers the ability to refine the default "broadcast" semantics of sending notifications to all consumers with a matching subscription. Publishing policies apply to notifications leaving a scope and they can delay or prevent forwarding to certain superscopes. They refine the interface and advertisements of a scope so that the filtering decision is not only based on individual notifications, but on other additional data sources such as current time or other notifications. This can be used to implement conditional events that are forwarded only within a specific time window, after a certain delay, or if another notification acknowledges the publication. This flexibility and extensibility is utilized by session scopes, which apply scoping to sets of notifications. Sessions identify the relationship between consecutive, dependent notifications. Transmission policies and interfaces are used to distinguish groups of related notifications, structuring the dynamic behavior of a system in addition to the architectural layout reified in the scope graph.

All these scope features are made available through a specification language that supports the definition of scope types as well as their instantiation and runtime modification.

Scope Implementation

Scopes are also a framework for the implementation of notification services. Their primary benefit, the delimitation of components, makes it possible to draw from a broad range of techniques to implement individual scopes. As long as the different scope instances are bridged and the visibility specification is met, techniques from network communication, data representation, filtering and routing, and data communication security can be employed to tailor the service implementation towards the needs of a specific (part) application. The thesis presents a number of different architectures for the implementation of scopes in distributed systems. Four architectures are investigated in detail. While all adhere to the visibility specification, they differ in their support for the mentioned implementation techniques, and thus in their quality of service parameters beyond visibility constraints.

Two architectures, *collapsing scope graphs* and *scope address*, can be built on top of existing communication techniques, e.g., content-based publish/subscribe services. The former collapses the graph and does not instantiate any scope representatives, using the underlying infrastructure as a black box. Eligible delivery paths in the scope graph are condensed into complex filter expressions. The expressions are partially evaluated in producers and consumers if content-based filtering is not available. This approach is applicable especially when existing systems are stepwise extended with scoping. The scope address approach builds on any multicast facility and addresses a scope and all of its members with one call to the communication substrate. An extra administrative component represents each scope and forwards outgoing and incoming notifications to/from superscopes. The explicit scope instance allows for physical delimitation of communication and admission control. Interestingly, both architectures can be built on database management systems as communication substrates. Despite its heavy footprint this approach has a number of advantages, because the functionality of DBMSs regarding reliability and access control can be exploited for the delivery of notifications.

The two other presented architectures, *broker scope* and *integrated routing*, influence and extend the notification service infrastructure itself. The former establishes a broker for each scope, transforming the scope graph into the broker graph that comprise the notification service. This approach combines system modeling and implementation, and it is suited for network centric designs. Finally, the integrated routing architecture extends traditional content-based routing. It provides scoping as an inherent service of a distributed infrastructure. Separate routing tables for each scope delimit notification visibility without affecting their basic routing capabilities. Integrated routing is the most general realization of scoping. The system structure reified in scopes is instantiated and accessible to system engineers, all features of scopes are available, and the scalability of notification routing is not impaired.

The feasibility of the scoping concept is demonstrated with a prototype implementation extending the Rebeca notification service.

Influence on Event-Based System Design and Implementation

The contribution of the presented scoping concept for event-based systems is twofold. It provides a design tool to model and specify event-based systems, and it is an implementation framework for the construction of customizable notification services. Taken together, the engineering of event-based systems is made possible where formerly only the plain pub/sub API was available.

As a design tool, scopes facilitate describing, specifying, and understanding event-based systems and their functionality. They introduce a structuring mechanism that delimits and controls interdependencies of application components. They allow system engineers to compose new functions and abstract from implementation details. And this is achieved without abandoning the inherent loose coupling of events. This is clearly a substantial advancement to existing systems, which are flat and unstructured and require manual visibility control.

As an implementation tool, scopes open the black box of notification services [169]. Scopes provide the means to seamlessly integrate various implementation techniques in one system and to tailor the semantics of notification dissemination in well-defined parts. On the level of scope implementation, the presented scope architectures offer a variety of quality of service options. The architectures differ in what parts of a scope graph are explicitly instantiated and in the amount of control they exert on communication. From an application point of view, the system structure identified with scopes is the obvious place to localize system programming and management tasks. At the level of scopes, notification mappings, transmission policies, and application-dependent admission control and security policies can be installed to program scope components.

Future Work

The scoping concept is a starting point for future work. Visibility in event-based systems, from which scoping is derived, is at the heart of many problems, and scoping provides a way to address these problems. This thesis has shown that scopes have the ability to integrate existing results from data integration, routing, and communication protocols. They separate coordination from computation and allow for model-driven development of distributed event-based systems.

Theory. The presented specification of event-based system can be extended, e.g., to include aspects like ordering, timing, etc. Furthermore, the theoretical tools should be refined to better address distribution aspects and to express conditions with respect to the visibility of notifications. Of course, it would be beneficial to map other existing systems to this specification to make their characteristics comparable.

Component programming. Future work should investigate applicable policies, their specification, and implementation with respect to the various scope architectures. Delivery policies govern the internal delivery of notifications and are a means of component programming. They enable engineers to tailor intra-scope distribution: group communication protocols could enforce ordering guarantees, consumers might be selected to implement load balancing, each notification may be delivered to a list of consumers in a predefined order (cf. workflows, escalation strategies). It would be rewarding to draw from other relevant areas such as composition and coordination languages, communication protocols, and others. Altogether, scopes coordinate groups of components and determine aspects of event-based interaction that are not encoded in simple application components.

Quality of Service. This term covers a broad range of issues from bandwidth and real-time constraints to reliability and transactional processing. The ability to utilize different communication techniques and architectures makes it possible to address QoS beyond visibility within individual scopes. For instance, the use of network level multicast or database systems as communication medium exhibit totally different QoS charactistics. Furthermore, the discussion about session scopes and activities has demonstrated the flexibility of scopes and transmission policies. Future extensions should develop the notion of activities and transactions in event-based systems.

Heterogeneity. Practical, large systems must cope with heterogeneity. Syntax and semantics of notifications are likley to vary in different parts of the system. Scoping and notification mappings provide the technical basis to address these issues, but so far require manual configuration. Future work should exploit this technical basis and apply existing approaches of data integration and mapping [71].

Event composition and lifecycle. Transmission policies were introduced to constrain the set of eligible consumers of a notification. More generally, they might be considered as notification processors that may correlate and even produce new notifications. They would thus enable event composition at scope boundaries.

A related issue is the notification lifecycle. When does published data disappear from the system? Are producers and consumers required to be active at the same time or are they decoupled in time? Caches [75] and histories [213] can be installed to make old notifications accessible to consumers.

Security. Security was mentioned several times to be a prominent open issue in event-based systems. Since any form of trust corresponds to some correlation of the participants, scoping suggests itself as a mechanism to incorporate security policies and implementation [117]. Scopes can perform admission control, protect intra-scope traffic from eavesdropping with encryption in the appropriate communication medium (preferably point-to-point communication). The broker scope architecture directly instantiates scopes and thus security domains, whereas integrated routing must also decide what part of the broker network is trustworthy.

Tools. A companion master thesis [219] successfully used Eclipse as an integration platform for design and management plugins. This is a promising starting point for more sophisticated tools for the design, programming, deployment, and management of scoped event-based systems.

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993. URL .

[3] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993. URL .

[4] S. Acharya, R. Alonso, M. J. Franklin, and S. B. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 199–210, San Jose, California, 22–25 May 1995.

[5] ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *SIGMOD Record*, 25(3): 40–49, Sept. 1996.

[6] G. Agha and C. J. Callsen. ActorSpace: an open distributed programming paradigm. *ACM SIGPLAN Notices*, 28(7):23–32, July 1993. URL .

[7] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 53–61, Atlanta, GA, USA, 1999. URL .

[8] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. In M. Stonebraker, editor, *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1992)*, pages 59–68, San Diego, CA, USA, 1992. ACM Press. URL .

[9] Akamai Technologies, Inc. Content and application delivery. Online information: http://www.akamai.com/en/html/services/content/application/delivery.html, 2003. URL .

[10] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002. URL .

[11] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. *IEEE Expert*, 1997. URL . Special Issue on Cooperative Information Systems.

[12] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[13] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering (ICSE'76)*, pages 562–570, Long Beach, CA, USA, Oct. 1976. IEEE Computer Society Press. URL .

[14] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000. URL .

[15] L. Andrade and J. L. Fiadeiro. Coordination primitives for event-based systems. In Bacon et al. [19]. URL . Published as part of the ICDCS '02 Workshop Proceedings.

[16] F. Arbab and C. Talcott, editors. *5th International Conference on Coordination Models and Languages (COORDINATION 2002)*, volume 2315 of *LNCS*, York, UK, 2002. Springer. URL .

[17] K. Arnold, J. Gosling, and D. Holmes. *The Java(TM) Programming Language.* Addison-Wesley, 3rd edition, 2000. URL .

[18] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications. In *IEEE SDNE Services in Distributed and Networked Environments*, pages 148–155, Whistler, British Columbia, June 1995.

[19] J. Bacon, L. Fiege, R. Guerraoui, H.-A. Jacobsen, and G. Mühl, editors. *1st Intl. Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, 2002. IEEE Press. URL . Published as part of the ICDCS '02 Workshop Proceedings.

[20] J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In A. Mendelson and J. Paredaens, editors, *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, pages 264–273, Seattle, WA, USA, 1998. ACM Press. URL .

[21] J. Bailey, A. Poulovassilis, and P. T. Wood. An event-condition-action language for xml. In D. Lassner, D. De Roure, and A. Iyengar, editors, *Proceedings of the Eleventh International Conference on World Wide Web*, pages 486–495, Honolulu, HI, USA, 2002. ACM Press. URL .

[22] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–272, 1999. URL .

[23] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In P. Jayanti, editor, *13th International Symposium on Distributed Computing (DISC'99)*, volume 1693 of *LNCS*, pages 1–17. Springer-Verlag, 1999. URL .

[24] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information flow based event distribution middleware. In W. Sun, S. Chanson, D. Tygar, and P. Dasgupta, editors, *ICDCS Workshop on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, Austin, TX, USA, 1999. URL .

[25] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In M. Mathis, P. Steenkiste, H. Balakrishnan, and V. Paxson, editors, *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 205–217, Pittsburgh, PA, USA, 2002. ACM, ACM Press. URL .

[26] E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Transactions on Database Systems (TODS)*, 21 (1):1–29, 1996. URL .

[27] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, Oct. 1996. URL .

[28] J. Bates, J. Bacon, K. Moody, and M. Spiteri. Using events for the scalable federation of heterogeneous components. In P. Guedes and J. Bacon, editors, *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, pages 58–65, Sintra, Portugal, Sept. 1998. URL .

[29] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, Feb. 1995. URL .

[30] I. Bea. Weblogic, 2005. http://www.bea-sys.com.

[31] S. Behnel. Abstractions for overlay software design. submitted, 2005.

[32] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based access control for publish/subscribe middleware architectures. In Jacobsen [163]. URL .

[33] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* North-Holland, 2001.

[34] M. Bernardo and F. Franzè. Exogenous and endogenous extensions of architectural types. In Arbab and Talcott [16], pages 40–55. URL .

[35] P. A. Bernstein. Transaction processing monitors. *Communications of the ACM*, 33(11):75–86, Nov. 1990. URL .

[36] B. Betts and C. Heinrich. *Adapt or Die: Transforming Your Supply Chain into an Adaptive Business Network.* John Wiley, 2003.

[37] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In J. Fabre and F. Jahanian, editors, *The International Conference on Dependable Systems and Networks (DSN'02)*, Washington, D.C., USA, jun 2002. IEEE Press. URL .

[38] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993. URL .

[39] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1): 47–76, 1987. URL .

[40] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1):39–59, Feb. 1984. URL .

[41] C. Bornhövd and A. P. Buchmann. A prototype for metadata-based integration of internet sources. In M. Jarke and A. Oberweis, editors, *11th International Conference on Advanced Information Systems Engineering (CAiSE'99)*, volume 1626 of *LNCS*, pages 439–445, Heidelberg, Germany, 1999. Springer-Verlag. URL .

[42] C. Bornhövd, M. Cilia, C. Liebig, and A. P. Buchmann. An infrastructure for meta-auctions. In *Second International Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'00)*, San Jose, California, June 2000. URL .

[43] J. C. Browne et al. Modularity and rule-based programming. *Intl. Journal on Artificial Intelligence Tools*, 4(1):201–218, 1995.

[44] M. Broy and E.-R. Olderog. Trace-oriented models of concurrency. In Bergstra et al. [33], chapter 2.

[45] A. Buchmann, C. Bornhövd, M. Cilia, L. Fiege, F. Gärtner, C. Liebig, M. Meixner, and G. Mühl. Dream: Distributed reliable event-based application management. In M. Levene and A. Poulovassilis, editors, *Web Dynamics—Adapting to Change in Content, Size, Topology and Use*, pages 319–349. Springer-Verlag, 2004. ISBN 3-540-40676-X. URL .

[46] A. Buchmann, F. Casati, L. Fiege, M.-C. Hsu, and M.-C. Shan, editors. *Third International Workshop on Technologies for E-Services (TES '02)*, volume 2444 of *LNCS*, Hong Kong, China, 2002. Springer-Verlag. URL .

[47] A. P. Buchmann. Architecture of active database systems. In N. W. Paton, editor, *Active Rules in Database Systems*, chapter 2, pages 29–48. Springer-Verlag, 1999. URL .

[48] B. Burke and A. Brock. Aspect-oriented programming and jboss. *OnJava.com*, May 2003. URL .

[49] N. Busi, A. Rowstron, and G. Zavattaro. State- and event-based reactive programming in shared dataspaces. In Arbab and Talcott [16], pages 111–124. URL .

[50] N. Busi and G. Zavattaro. On the expressiveness of event notification in data-driven coordination languages. In G. Smolka, editor, *Proceedings of 9th European Symposium on Programming Languages and Systems (ESOP 2000)*, volume 1782 of *LNCS*, pages 41–55, Berlin, Germany, 2000. URL .

[51] C. Bussler and S. Jablonski. Implementing agent coordination for workflow management systems using active database systems. In J. Widom and S. Chakravarthy, editors, *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering: Acticve Database Systems (RIDE-ADS 1994)*, pages 53–59, 1994. URL .

[52] L. Capra, W. Emmerich, and C. Mascolo. Middleware for mobile computing (a survey). Research Note RN/30/01, University College London, July 2001.

[53] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155, Lisbon, Portugal, 1998. Springer-Verlag. URL .

[54] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, Apr. 1989. URL .

[55] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems, ECOOP'94 Workshop*, volume 924 of *LNCS*, pages 66–76, Bologna, Italy, 1995. Springer-Verlag.

[56] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998. URL .

[57] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In *Proceedings of the Third International Workshop on Software Architecture (ISAW '98)*, pages 17–20, Orlando, FL, USA, 1998. URL .

[58] A. Carzaniga and P. Fenkam, editors. *3rd Intl. Workshop on Distributed Event-Based Systems (DEBS'04)*, Edinburgh, Scotland, UK, May 2004. IEE. URL .

[59] A. Carzaniga, D. R. Rosenblum, and A. L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In W. Emmerich and V. Gruhn, editors, *ICSE '99 Workshop on Engineering Distributed Objects (EDO '99)*, May 1999. URL .

[60] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, Univ. of Colorado at Boulder, USA, 1998. URL .

[61] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 219–228, NY, July 16–19 2000. ACM Press. URL .

[62] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001. URL .

[63] J. Case, D. Harrington, R. Presuhn, and B. Wijnen. RFC 3412: message processing and dispatching for the simple network management protocol (snmp), Dec. 2002. URL . Status: Standard.

[64] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. RFC 1901: introduction to community-based snmpv2, Jan. 1996. URL . Status: Historic.

[65] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), 2002. URL .

[66] A. Celik, A. Datta, and S. Narasimhan. Supporting subscription oriented information commerce in a push-based environment. *IEEE Transactions on Systems, Man and Cybernetics*, 30(4):433–445, July 2000.

[67] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 606–617, Santiago, Chile, July 1994. Morgan Kaufmann. URL .

[68] D. R. Cheriton and S. E. Deering. Host groups: a multicast extension for datagram internetworks. In W. P. Lidinsky and B. W. Stuck, editors, *Proceedings of the Ninth Symposium on Data Communications*, pages 172–179, Whistler Moutain, British Columbia, Canada, 1985. ACM Press. URL .

[69] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems (TOCS)*, 3(2):77–107, 1985. URL .

[70] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys (CSUR)*, 28(2):300–302, 1996. URL .

[71] M. Cilia. *An Active Functionality Service for Open Distributed Heterogeneous Environments.* PhD thesis, TU Darmstadt, 2002. URL .

[72] M. Cilia. Personal communication, 2003.

[73] M. Cilia, M. Antollini, C. Bornhoevd, and A. Buchmann. Dealing with heterogeneous data in pub/sub systems: The concept-based approach. In Carzaniga and Fenkam [58]. URL .

[74] M. Cilia, C. Bornhövd, and A. P. Buchmann. Moving active functionality from centralized to open distributed heterogeneous environments. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 195–210, Trento, Italy, 2001. Springer-Verlag. URL .

[75] M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. Buchmann. Looking into the past: Enhancing mobile publish/subscribe middleware. In Jacobsen [163]. URL .

[76] M. Cilia, M. Haupt, M. Mezini, and A. P. Buchmann. The convergence of aop and active databases: Towards reactive middleware. In F. Pfenning and Y. Smaragdakis, editors, *Proceedings of the International Conference on Generative Programming and Component*

Engineering (GPEC'03), volume 2830 of *LNCS*, pages 169–188, Erfurt, Germany, 2003. Springer-Verlag. URL .

[77] M. Colan. *InfoBus 1.2 Specification.* Lotus, 1999. URL .

[78] I. Corp. Websphere mq (mqseries), 2003. http://www.software.ibm.com/mqseries.

[79] F. M. Costa, H. A. Duran, N. Parlavantzas, K. B. Saikoski, G. Blair, and G. Coulson. The role of reflective middleware in supporting the engineering of dynamic applications. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *LNCS*, Denver, CO, USA, Nov. 2001. Springer-Verlag. URL .

[80] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In Jacobsen [163]. URL .

[81] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Efficient query subscription processing in a multicast environment. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, page 83, 2000. URL .

[82] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 261–270. IEEE Computer Society Press / ACM Press, 1998. URL .

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

[84] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002. URL .

[85] E. Curry, D. Chambers, and G. Lyons. Reflective channel hierarchies. In *The 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil, 2003. URL .

[86] Y. K. Dalal and R. M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, 1978. URL .

[87] J. Dávila. Reactive pascal and the event calculus: A platform to program reactive, rational agents. In U. C. Sigmund and M. Thielscher, editors, *Proceedings of the Workshop at FAPR'95: Reasoning about Actions and Planning in Complex Environments*, 1996. URL .

[88] U. Dayal, B. T. Blaustein, A. P. Buchmann, U. S. Chakravarthy, M. Hsu, R. Ledin, D. R. McCarthy, A. Rosenthal, S. K. Sarin, M. J. Carey, and R. J. Miron Livny. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, 1988. URL .

[89] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: a knowledge model for an active, object-oriented database system. In *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, volume 334 of *LNCS*, pages 129–143. Springer, 1988. URL .

[90] S. Deering. Host Extensions for IP Multicasting. Request for Comments (RFC) 1112, Aug. 1989. URL .

[91] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990. URL .

[92] J. Dempsey and V. Cahill. Aspects of system support for distributed computing. In *ECOOP '97 Workshop on Aspect-oriented Programming*, 1997. URL .

[93] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Dissemination of dynamic web data. In *10th International World Wide Web Conference*, Hong Kong, May 2001. URL .

[94] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proceedings of of the 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, pages 209–221, Lake Buena Vista, FL, USA, Nov. 1998. ACM Press. URL .

[95] D. Dobre. A framework for engineering j2ee-based publish/subscribe applications with scopes. Master's thesis, TU Darmstadt, 2004.

[96] S. Duarte, J. L. Martins, H. J. Domingos, and N. Preguia. Deeds - a distributed and extensible event dissemination service. In *Proceedings of the 4rd European Research Seminar on Advances in Distributed Systems (ERSADS)*, Forli, Italy, 2001. URL .

[97] S. Duarte, J. L. Martins, H. J. Domingos, and N. Preguiça. A case study on event dissemination in an active overlay network environment. In Jacobsen [163]. URL .

[98] J. Eder and E. Panagos. Towards distributed workflow process management. In C. Bussler, P. Grefen, H. Ludwig, and M.-C. Shan, editors, *Proceedings of the Workshop on Cross-Organisational Workflow Management and Coordination*, San Francisco, CA, USA, 1999. URL .

[99] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001. URL . Special Issue on Aspect-Oriented Programming.

[100] M. Endler and D. C. Schmidt, editors. *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, volume 2672 of *LNCS*, Rio de Janeiro, Brazil, 2003. Springer-Verlag. URL .

[101] P. T. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. In L. Northrop and J. Vlissides, editors, *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming Systems Languages and Applications*, pages 254–269, Tampa Bay, FL, USA, 2001. ACM Press. URL .

[102] P. T. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In E. Bertino, editor, *European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *LNCS*, pages 252–276, 2000. URL .

[103] H. Evans and P. Dickman. DRASTIC: A run-time architecture for evolving, distributed, persistent systems. In M. Akşit and S. Matsuoka, editors, *European Conference for Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 243–275, Jyväskylä, Finnland, 1997. Springer-Verlag. URL .

[104] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In T. Sellis and S. Mehrotra, editors, *Proceedings of the 20th Intl. Conference on Management of Data (SIGMOD 2001)*, pages 115–126, Santa Barbara, CA, USA, 2001. URL .

[105] L. Fiege. Architekturelle Unterstützung von Electronic Commerce Anwendungen. In *Informatiktage 2000*. Konradin Verlag, 2000. in German.

[106] L. Fiege, F. C. Gärtner, S. B. Handurukande, and A. Zeidler. Dealing with uncertainty in mobile publish/subscribe middleware. In *1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 03)*, Rio de Janeiro, Brazil, 2003. URL .

[107] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In Endler and Schmidt [100], pages 103–122. URL .

[108] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. Technical Report IC/2003/11, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, Mar. 2003.

[109] L. Fiege, M. Mezini, G. Mühl, and A. Buchmann. Komponenten in ereignisbasierten systemen. *Thema Forschung*, 1:108–114, 2003. ISSN 1434-7768. URL .

[110] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 309–333, Malaga, Spain, June 2002. Springer-Verlag. URL .

[111] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Visibility as central abstraction in event-based systems. In A. Beugnard, S. Sadou, L. Duchien, and E. Jul, editors, *Concrete Communication Abstractions of the Next 701 Distributed Object Systems (ECOOP 2002 Workshop)*, volume 2548 of *LNCS*, Málaga, Spain, 2002. Springer-Verlag. URL .

[112] L. Fiege and G. Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. URL . http://www.gkec.informatik.tu-darmstadt.de/rebeca.

[113] L. Fiege, G. Mühl, and A. Buchmann. An architectural framework for electronic commerce applications. In *Informatik 2001: Annual Conference of the German Computer Society*, 2001.

[114] L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press. URL .

[115] L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, 2003. URL .

[116] L. Fiege, G. Mühl, and U. Wilhelm, editors. *Second International Workshop on Electronic Commerce (WELCOM 2001)*, volume 2232 of *LNCS*, Heidelberg, Germany, 2001. Springer-Verlag, Berlin. URL .

[117] L. Fiege, A. Zeidler, A. Buchmann, R. Kilian-Kehr, and G. Mühl. Security aspects in publish/subscribe systems. In Carzaniga and Fenkam [58]. URL .

[118] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Communications of the ACM*, 45(1): 116–122, 2002. URL .

[119] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[120] O. S. Foundation. OSF Distributed Management Environment (DME) Rationale, Oct. 1991. URL . O-DME-RD-1.

[121] M. Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html#InversionOfControl, Jan. 2004. URL .

[122] M. Fowler. *UML Distilled*. Addison-Wesley, 2004.

[123] D. S. Frankel. *Model Driven Architecture*. John Wiley & Sons, 2003.

[124] M. J. Franklin and S. B. Zdonik. A framework for scalable dissemination-based systems. In A. M. Berman, M. Loomis, and T. Bloom, editors, *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, pages 94–105, Atlanta, GA, USA, Oct. 5–9, 1997. URL .

[125] M. J. Franklin and S. B. Zdonik. "Data In Your Face": Push Technology in Perspective. In L. M. Haas and A. Tiwary, editors, *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 516–519, Seattle, USA, 1998. ACM Press. URL .

[126] O. Friesen, G. Gauthier-Villars, A. Lefebvre, and L. Vieille. Applications of deductive object-oriented databases using del. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 1–22. Kluwer Academics, 1994.

[127] H. Fritschi, S. Gatziu, and K. R. Dittrich. FRAMBOISE: an approach to framework-based active database management system construction. In G. Gardarin, J. French, N. Pissinou, K. Makki, and L. Bouganim, editors, *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM-98)*, pages 364–370, New York, Nov. 3–7 1998. ACM Press.

[128] L. Fuchs. Area: A cross-application notification service for groupware. In S. Bødker, M. Kyng, and K. Schmidt, editors, *The 6th European Conference on Computer Supported Cooperative Work (ECSCW 1999)*, pages 61–80, Copenhagen, Denmark, 1999. Kluwer Academic Publishers. URL .

[129] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[130] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, Nov. 1995.

[131] D. Garlan, G. E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, 25(6):30–38, June 1992.

[132] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In S. Prehn and W. J. H. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 31–44, Noordwijkerhout, The Netherlands, 1991. Springer-Verlag.

[133] D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In V. R. Basili, R. A. DeMillo, and T. Katayama, editors, *Proceedings of the 15th Intl. Conference on Software Engineering (ICSE '93)*, pages 447–455, Baltimore, MD, USA, 1993. IEEE Computer Society Press / ACM Press. URL .

[134] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993. URL .

[135] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, Mar. 1999. URL .

[136] F. C. Gärtner. *Formale Grundlagen der Fehlertoleranz in verteilten Systemen.* PhD thesis, TU Darmstadt, 2001.

[137] S. Gatziu, A. Koschel, G. von Bültzingsloewen, and H. Fritschi. Unbundling active functionality. *SIGMOD Record*, 27(1), Mar. 1998.

[138] D. Gawlick and S. Mishra. Information sharing with the oracle database. In Jacobsen [163]. URL .

[139] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In L.-Y. Yuan, editor, *Proceedings of the 18th International Conference on Very Large Data Bases VLDB'92)*, pages 327–338. Morgan Kaufmann Publishers, 1992. URL .

[140] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985. URL .

[141] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE '89)*, volume 366 of *LNCS*, pages 20–27, Eindhoven, The Netherlands, 1989. Springer-Verlag. URL .

[142] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, Apr. 1995. URL .

[143] A. Geppert and D. Tombros. Event-based distributed workflow execution with EVE. In N. Davies, K. Raymond, and J. Seitz, editors, *Middleware ´98*. Springer-Verlag, 1998. URL .

[144] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[145] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[146] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In P. Dasgupta, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Middleware Workshop*, Austin, TX, USA, May 1999. URL .

[147] R. Gruber, B. Krishnamurthy, and E. Panagos. READY: A high performance event notification service. In *Proceedings of the 16th International Conference on Data Engineering*, 2000. URL .

[148] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001. URL .

[149] P. Hammant, A. Hellesoy, and J. Tirsen. Picocontainer—a lightweight embeddable container for dependency-injector components, 2004. URL . http://www.picocontainer.org/.

[150] S. Handurukande, P. T. Eugster, P. Felber, and R. Guerraoui. Event systems: How to have ones cake and eat it too. In Bacon et al. [19]. URL . Published as part of the ICDCS '02 Workshop Proceedings.

[151] S. Hanna, B. Patel, and M. Shah. RFC 2730: multicast address dynamic client allocation protocol (madcap), Dec. 1999. URL . Status: Proposed Standard.

[152] D. Harrington. The evolution of architectural concepts in the snmpv3 working group. *The Simple Times*, 5(1), 1997. URL .

[153] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In A. Paepcke, editor, *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, Washington, WA, USA, 1993. URL .

[154] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9): 921–932, Sept. 1985. URL .

[155] R. Hayton, J. Bacon, J. Bates, and K. Moody. Using events to build large scale distributed applications. In Herbert and Tanenbaum [156], pages 9–16. URL .

[156] A. Herbert and A. S. Tanenbaum, editors. *Proc. of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996. URL .

[157] Y. hua Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Selected Areas in Communications*, 20(8), 2002. URL .

[158] IBM. Gryphon: Publish/subscribe over public networks. Technical report, IBM T.J. Watson Research Center, 2001. URL .

[159] J. Inc. http://www.jboss.org, 2004. URL .

[160] S. M. Inc. Java 2 platform, standard edition, v 1.5.0, 2004. URL . http://java.sun.com/j2se/1.5.0/.

[161] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking (TON)*, 11(1):2–16, 2003. URL .

[162] ISO/IEC. Open distributed processing–reference model. International Standard ISO/IEC IS 10746, May 1995.

[163] H.-A. Jacobsen, editor. *2nd Intl. Workshop on Distributed Event-Based Systems (DEBS'03)*, San Diego, CA, USA, June 2003. ACM Press. URL .

[164] K. Jenkins, K. Hopkins, and K. Birman. A gossip protocol for subgroup multicast. In M. Raynal and L. Rodrigues, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*, Phoenix, AZ, USA, 2001. IEEE Press. URL .

[165] T. Joseph. A messaging-based architecture for enterprise application integration. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 62–63, Sydney, Australia, 1999. URL .

[166] M. Kahani and H. W. P. Beadle. Decentralised approaches for network management. *ACM SIGCOMM Computer Communication Review*, 27 (3):36–47, 1997. URL .

[167] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. Coordination in workflow management systems—a rule-based approach. In W. Conen and G. Neumann, editors, *Coordination Technology for Collaborative Applications (ASIAN 1996 Workshop)*, volume 1364 of *LNCS*, pages 99–120. Springer, 1998. URL .

[168] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen. From rules to rule patterns. In P. Constantopoulos, J. Mylopolous, and Y. Vassiliou, editors, *Proc. of the* 8^{th} *Intl. Conference on Advanced Information Systems Engineering (CAiSe'96)*, volume 1080 of *LNCS*, pages 99–115, Heraklion, Crete, Greece, 1996. Springer-Verlag. URL .

[169] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, Jan. 1996. URL .

[170] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[171] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *CACM*, 44(10):59–65, Oct. 2001. URL . Special Issue on aspect-oriented programming.

[172] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154, 1996. URL .

[173] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[174] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002. URL .

[175] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, volume 563 of *LNCS*, pages 87–101. Springer-Verlag, 1991. URL .

[176] A. Koschel, R. Kramer, G. von Bültzingsloewen, T. Bleibel, P. Krumlinde, S. Schmuck, and C. Weinand. Configurable Active Functionality for CORBA. In *ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation*, 1997.

[177] M. Koubarakis. Textual information dissemination in distributed event-based systems. In Bacon et al. [19]. URL . Published as part of the ICDCS '02 Workshop Proceedings.

[178] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988. URL .

[179] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, Mar. 1977.

[180] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, Amsterdam, 1983. North-Holland.

[181] L. Lamport and N. Lynch. Distributed computing: Models and methods. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1157–1199. Elsevier, 1990. URL .

[182] F. Lange, R. Kröger, and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), 1992. URL .

[183] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. W3C Recommendation, Feb. 1999. http://www.w3.org/TR/REC-rdf-syntax.

[184] A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler. The wise approach to electronic commerce. *International Journal of Computer Systems Science & Engineering*, 15(5), Sept. 2000. URL .

[185] J. Le Boudec. The Asynchronous Transfer Mode: a tutorial. *Computer Networks and ISDN Systems*, 24:279–309, 1992.

[186] G. T. Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems.* Cambridge University Press, 2000.

[187] J. Liberty. *Programming C#.* O'Reilly, 3rd edition edition, 2003. URL .

[188] C. Liebig, B. Boesling, and A. Buchmann. A notification service for next-generation it systems in air traffic control. In *GI-Workshop: Multicast-Protokolle und Anwendungen*, Braunschweig, Germany, May 1999. URL .

[189] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *Proceedings of the 4th Intl. Conference on Cooperative Information Systems (CoopIS '99)*, Sept. 1999. URL .

[190] C. Liebig, M. Malva, and A. Buchmann. Integrating notifications and transactions: Concepts and X^{2}TS prototype. In W. Emmerich and S. Tai, editors, *Proceedings of Second International Workshop on Engineering Distributed Objects (EDO 2000)*, volume 1999 of *LNCS*, Davis, CA, USA, Nov. 2000. Springer-Verlag. URL .

[191] C.-C. Lin and R. J. LeBlanc. Event-based debugging of object/action programs. In R. L. Wexelbalt, editor, *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24(1) of *SIGPLAN Notices*, pages 23–34, Madison, WI, USA, 1988. ACM Press. URL .

[192] D. Luckham. *The Power of Events.* Addison-Wesley, 2002.

[193] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept. 1995. URL .

[194] C. Ma and J. Bacon. COBEA: A CORBA-based event architecture. In J. Sventek, editor, *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pages 117–132, Santa Fe, NM, USA, 1998. USENIX Association. URL .

[195] P. Maes. Concepts and experiments in computational reflection. In N. Meyrowitz, editor, *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 147–155, Orlando, FL, USA, Oct. 1987. ACM Press. ISBN 0-89791-247-0. URL .

[196] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering*, 1(5):304–312, 1994. URL .

[197] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994. URL .

[198] T. W. Malone and R. J. Laubacher. The dawn of the e-lance economy. *Harvard Business Review*, pages 145–152, Sept. 1998.

[199] M. Mansouri-Samani and M. Sloman. A configurable event service for distributed systems. In *Third International Conference on Configurable Distributed Systems*, pages 210–217, 1996. URL .

[200] R. C. Martin. The Dependency Inversion Principle. *C++ Report*, 8(6): 61–66, June 1996. URL .

[201] D. Mason and D. Woit. Problems with software reliability composition. In *Proceedings of 1998 International Symposium on Software Reliability Engineering (ISSRE'98 Fast Abstracts)*, 1998. URL .

[202] F. Mattern. The vision and technical foundations of ubiquitous computing. *Upgrade*, II(5), 2001. URL . Special issue on Ubiquitous Computing.

[203] N. Maxemchuk and D. Shur. An internet multicast system for the stock market. *ACM Transactions on Computer Systems*, 19(3):384–412, 2001. URL .

[204] J. McAffer. Meta-level programming with CodA. In W. Olthoff, editor, *Proceedings of the European Conference for Object-Oriented Programming (ECOOP '95)*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, 1995. Springer-Verlag. URL .

[205] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 60–76. Springer / ACM Press, 1997. URL .

[206] R. Meier, M.-O. Killijian, R. Cunningham, and V. Cahill. Towards proximity group communication. In Banavar:2001:MobileMiddleware, editor, *Advanced Topic Workshop Middleware for Mobile Computing (Middleware 2001)*, 2001. URL .

[207] I. Merrick and A. Wood. Coordination with scopes. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2000)*, pages 210–217, Como, Italy, Mar. 2000. URL .

[208] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7): 395–403, July 1976. URL .

[209] D. Meyer. RFC 2365: Administratively scoped IP multicast, July 1998. URL . Status: Best Current Practice.

[210] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web caching: Towards a new global caching architecture. *Computer Networks and ISDN Systems*, 30:2169–2177, Nov. 1998. URL .

[211] D. P. Miranker, L. Obermeyer, L. Warshaw, and J. C. Browne. Venus: An object-oriented extension of rule-based programming. Technical report, University of Texas at Austin, 1998. URL .

[212] G. Mühl. Generic constraints for content-based publish/subscribe systems. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 211–225, Trento, Italy, 2001. Springer-Verlag. URL .

[213] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, 2002. URL . http://elib.tu-darmstadt.de/diss/000274/.

[214] G. Mühl and L. Fiege. Supporting covering and merging in content-based publish/subscribe systems: Beyond name/value pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001. URL .

[215] G. Mühl, L. Fiege, and A. P. Buchmann. Evaluation of cooperation models for electronic business. In *Information Systems for E-Commerce, Conference of German Society for Computer Science / EMISA*, pages 81–94, Nov. 2000. ISBN 3-85487-194-5.

[216] G. Mühl, L. Fiege, and A. P. Buchmann. Filter similarities in content-based publish/subscribe systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *Lecture Notes in Computer Science*, pages 224–238, Karlsruhe, Germany, 2002. Springer-Verlag. URL .

[217] G. Mühl, L. Fiege, F. C. Gärtner, and A. P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In A. Boukerche, S. K. Das, and S. Majumdar, editors, *The Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, pages 167–176, Fort Worth, TX, USA, October 2002. IEEE Press. URL .

[218] G. Mühl, A. Ulbrich, K. Herrmann, and T. Weis. Disseminating information to mobile clients using publish/subscribe. *IEEE Internet Computing*, 8(3), May 2004. URL .

[219] M. Mühleisen. Programming and administration of publish-subscribe systems (in german). Master's thesis, Technische Universität Darmstadt, 2005.

[220] S. Mullender, editor. *Distributed Systems.* Addison-Wesley, 2nd edition, 1993.

[221] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In F. Golshani, P. Dasgupta, and W. Zhao, editors, *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 524–533, Phoenix, AZ, USA, May 2001. URL .

[222] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, Sept. 1994.

[223] *FIXML - A Markup Language for the Financial Information eXchange (FIX) protocol.* Oasis, July 2001. http://www.oasis-open.org/cover/fixml.html.

[224] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Version 2.3. Object Management Group, Framingham, MA, USA, 1998.

[225] Object Management Group. *CORBA Components.* OMG, Framingham, MA, USA, 1999. orbos/99-07-01.

[226] Object Management Group. Corba notification service. OMG Document telecom/99-07-01, 1999. URL .

[227] Object Management Group. CORBA event service specification, version 1.0. OMG Document formal/2000-06-15, 2000. URL .

[228] Object Management Group. Corba transaction service v1.1. OMG Document formal/00-06-28, 2000. URL .

[229] Object Management Group. Management of event domains. Version 1.0, Formal Specification, 2001. URL . formal/01-06-03.

[230] Object Management Group. Corba components, v3.0 full specification. formal/02-06-65, June 2002. URL .

[231] Object Management Group. CORBA notification service, version 1.0.1. OMG Document formal/2002-08-04, 2002. URL .

[232] K. O'Connell, T. Dinneen, S. Collins, B. Tangney, N. Harris, and V. Cahill. Techniques for handling scale and distribution in virtual worlds. In Herbert and Tanenbaum [156], pages 17–24. URL .

[233] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus—an architecture for extensible distributed systems. In B. Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 58–68, Asheville, NC, USA, Dec. 1993. ACM Press. URL .

[234] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001. URL .

[235] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In J. Sventek and G. Coulson, editors, *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, volume 1795 of *LNCS*, pages 185–207. Springer-Verlag, 2000. URL .

[236] I. Oracle. Introduction to oracle advanced queuing (aq). Application Developer's Guide, July 2001. URL .

[237] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, Aug. 1998. URL .

[238] G. A. Papadopoulos and F. Arbab. Modelling Activities in Information Systems using the coordination language Manifold. In K. M. George and G. B. Lamong, editors, *Proceedings of the ACM Symposium on Applied Computing (SAC '98)*, pages 185–193, Atlanta, GA, USA, 1998. ACM Press. URL .

[239] G. A. Papadopoulos and F. Arbab. Configuration and dynamic reconfiguration of components using the coordination paradigm. *Future Generation Computer Systems*, 17(8):1023–1038, June 2001. URL .

[240] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[241] C. Partridge, T. Mendez, and W. Milliken. RFC 1546: host anycasting service, nov 1993. URL . Status: Informational.

[242] N. W. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999. URL .

[243] M. Pauly. Event-based meta-auctions on mobile devices. Master's thesis, Departement of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany, Apr. 2004. (in German).

[244] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In Bacon et al. [19]. URL . Published as part of the ICDCS '02 Workshop Proceedings.

[245] P. R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, February 2004. URL .

[246] P. R. Pietzuch, B. Shand, and J. Bacon. A framework for event composition in distributed systems. In Endler and Schmidt [100], pages 62–82. URL .

[247] D. Platt. The COM+ event service eases the pain of publishing and subscribing to data. *Microsofts Systems Journal*, September 1999.

[248] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[249] A. Pope. *The CORBA Reference Guide*. Addison-Wesley, Reading, MA, USA, 1997.

[250] D. Powell. Group communication. *Communications of the ACM*, 39(4): 50–53, Apr. 1996. URL .

[251] R. Prakash and R. Baldon. Architecture for group communication in mobile systems. In *The 17th IEEE Symposium on Reliable Distributed Systems (SRDS '98*, pages 235–242, Oct. 1998. URL .

[252] E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in distributed environments. In K. Czarnecki and U. Eisenecker, editors, *Generative and Component-Based Software Engineering, GCSE'99*, volume 1799 of *LNCS*, Erfurt, Germany, Sept. 1999. Springer-Verlag. URL .

[253] B. Quinn and K. Almeroth. RFC 3170: ip multicast applications: Challenges and solutions, Sept. 2001. URL . Status: Informational.

[254] K. Ramamritham, P. Deolasee, A. Katkar, A. Panchbudhe, and P. Shenoy. Dissemination of dynamic data on the internet. In S. Bhalla, editor, *Databases in Networked Information Systems (DNIS 2000)*, volume 1966 of *LNCS*, pages 173–187, Aizu, Japan, 2000. Springer-Verlag. URL .

[255] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In J. Crowcroft and M. Hofmann, editors, *Proceedings of the Third International COST264 Workshop (NGC 2001)*, volume 2233 of *LNCS*, pages 14–29. Springer-Verlag, Nov. 2001. URL .

[256] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[257] A. Ricci, A. Omicini, and E. Denti. Objective vs. subjective coordination in agent-based systems: A case study. In Arbab and Talcott [16], pages 291–299. URL .

[258] D. Riehle, W. Siberski, D. Bäumer, D. Megert, and H. Züllighoven. Serializer. In R. Martin, D. Riehle, , and F. Buschmann, editors, *Pattern Languages of Program Design 3*, chapter 17, pages 293–312. Addison-Wesley, 1998. URL .

[259] M. T. Rose. Network management: Status and challenges. Keynote Presentation at the Third International Symposium on Integrated Network Management (ISINM'93), Apr. 1993. URL .

[260] M. T. Rose. *The Simple Book: An Introduction to Internet Management.* P T R Prentice-Hall, second edition, 1994.

[261] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of internet content delivery systems. *ACM Operating Systems Review*, 36(SI):315–327, 2002. URL .

[262] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994. URL .

[263] D. Schmidt and S. Vinoski. Time-Independent Invocation and Interoperable Routing. *C++ Report*, 11(4), Apr. 1999.

[264] D. C. Schmidt and S. Vinoski. Programming Asynchronous Method Invocations with CORBA Messaging. *C++ Report*, 11(2), Feb. 1999.

[265] D. C. Schmidt and S. Vinoski. Portable interceptors concepts and components. *C++ Experts Forum*, Mar. 2003. URL . Object Interconnections: CORBA Metaprogramming Mechanisms, Part 1.

[266] W. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia, September 1997.*, 1997. URL . http://elvin.dstc.edu.au/doc/papers/auug97/AUUG97.html.

[267] Y.-P. Shan. An event-driven model-view-controller framework for Smalltalk. In N. Meyrowitz, editor, *OOPSLA'89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 347–352. ACM Press, 1989. URL .

[268] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, Sept. 1990.

[269] M. D. Skeen and M. Bowles. Apparatus and method for providing decoupling of data exchange details for providing high performance communication between software processes. United States Patent No. 5,557,798, Sept. 1996. URL .

[270] M. Stal. Web services: Beyond component-based computing. *Communications of the ACM*, 45(10):71–76, 2002. URL .

[271] J. Steffan and L. Fiege. Scoping as a general node selection concept for wireless sensor networks. submitted, 2005.

[272] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann. Scoping in wireless sensor networks. In *2nd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Toronto, Canada, 2004. ACM.

[273] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *Int'l Symposium on Software Reliability Engineering*, 1998. URL .

[274] D. Sturman, G. Banavar, and R. Strom. Reflection in the gryphon message brokering system. In *In Reflection Workshop of the 13th ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, 1998. URL .

[275] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. In R. N. Taylor, editor, *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 22–33, Irvine, CA, USA, 1990. ACM Press. URL .

[276] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions of Software Engineering and Methodology*, 1(3):229–269, July 1992.

[277] Q. Sun. Reliable multicast for publish/subscribe systems. Master's thesis, Massachusetts Institute of Technology, 2000. URL .

[278] Sun Microsystems, Inc. Javabeans version 1.01. Proposed Final Draft, 1997. http://java.sun.com/products/javabeans.

[279] Sun Microsystems, Inc. Java Message Service (JMS) Specification 1.0.2, 1999. URL .

[280] Sun Microsystems, Inc. Enterprise javabeans specification, version 2.0. Proposed Final Draft, 2000. http://java.sun.com/products/ejb/index.html.

[281] Sun Microsystems, Inc. JavaSpaces Service Specification version 1.1, 2000. URL .

[282] Sun Microsystems, Inc. Java 2 Platform Enterprise Edition Specification, v. 1.3, July 2001.

[283] Sun Microsystems, Inc. Java management extensions. Instrumentation and Agent Specification, v1.2, Oct. 2002. URL .

[284] Sun Microsystems, Inc. Java Message Service (JMS) Specification 1.1, 2002. URL .

[285] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1997.

[286] S. Tai, T. A. Mikalsen, I. Rouvellou, and S. M. Sutton Jr. Dependency-spheres: A global transaction context for distributed objects and messages. In *Proceedings of the 5th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2001)*, Seattle, Washington, USA, Sept. 2001. URL .

[287] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, third edition, 1996.

[288] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002. ISBN 0-13-066102-3.

[289] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996. URL .

[290] W. W. Terpstra, S. Behnel, L. Fiege, J. Kangasharju, and A. Buchmann. Bit Zipper Rendezvous—Optimal data placement for general P2P queries. In C. Meghini, N. Spyratos, and Y. Tzitzikas, editors, *EDBT 04 Workshop on Peer-to-Peer Computing & DataBases*, volume ? of *LNCS*, Heraklion, Crete, Greece, 2004. Springer-Verlag. URL .

[291] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In Jacobsen [163]. URL .

[292] TIBCO, Inc. TIB/Rendezvous. White Paper, 1996. http://www.rv.tibco.com/.

[293] TIBCO, Inc. *TIBCO Rendezvous: Concepts*, 2002.

[294] P. Timberlake. The pitfalls of using multicast publish/subscribe for EAI. IBM MQseries Whitepaper, 2002. URL . also published on messageQ.com.

[295] R. van Renesse, K. Birman, A. Bozdog, D. Dumitriu, M. Singh, and W. Vogels. Heterogeneity-aware peer-to-peer multicast. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, 2003. URL . Submitted.

[296] J. Vera, L. Perrochon, and D. C. Luckham. Event-based execution architectures for dynamic software systems. In *Proceedings of the First Working IFIP Conf. on Software Architecture*, San Antonio, TX, USA, Feb. 1999. IEEE. URL .

[297] M. Viroli and A. Ricci. Tuple-based coordination models in event-based scenarios. In Bacon et al. [19]. Published as part of the ICDCS '02 Workshop Proceedings.

[298] J. Vitek, N. Horspool, and A. Krall. Efficient type inclusion tests. *ACM SIGPLAN Notices*, 32(10):142–157, Oct. 1997. ISSN 0362-1340. URL .

[299] W3C. Simple object access protocol (SOAP) 1.2. Recommendation, June 2003. URL . http://www.w3.org/TR/SOAP/.

[300] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security issues and requirements for Internet-scale publish-subscribe systems. In *Proceedings of the Thirtyfifth Hawaii International Conference on System Sciences (HICSS-35)*, Big Island, Hawaii, Jan. 2002. URL .

[301] M. Weiser. The computer for the 21st century. *Scientific American*, 265 (3):94–104, 1991. URL .

[302] S.-Y. Wu, D. P. Miranker, and J. C. Browne. Decomposition abstraction in parallel rule languages. *Transactions on Parallel and Distributed Systems*, 7(11):1164–1184, 1996. URL .

[303] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, 1994. URL .

[304] S. Yang and S. Chakravarthy. Formal semantics of composite events for distributed environments. In *Proceedings of the 15th International Conference on Data Engineering (ICDE '99)*, pages 400–407. IEEE Computer Society Press, 1999. URL .

[305] A. Zeidler and L. Fiege. Mobility support with REBECA. In J. Wu, editor, *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS) Workshop on Mobile Computing Middleware (MCM 03)*, pages 354–361, Providence, RI, USA, 2003. IEEE Press. URL .

[306] H. Zeller. Nonstop sql/mx publish/subscribe: Continuous data streams in transaction processing. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *Proceedings of SIGMOD'03*, page 633, San Diego, CA, USA, 2003. URL .

[307] D. Zhou, K. Schwan, G. Eisenhauer, and Y. Chen. Jecho–interactive high performance computing with java event channels. In *Intl. Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, USA, 2001. URL .

[308] J. Zinky and R. Shapiro. The aspect-oriented interceptors' pattern for crosscutting and separation of concerns using conventional object oriented programming languages. In *The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2003. URL .

Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades „Dr.-Ing.“ mit dem Titel „Visibility in Event-Based Systems“ selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 04. April 2005 Ludger Fiege