6

Scoping

So far, the presented simple event systems merely provide the functionality to distribute notifications, but still fails to offer any support for coping with the complexities of designing and engineering distributed systems. The main deficiency is the missing control of the interaction in the system, which is only given implicitly. The resulting problems were recognized in different contexts, and the means to address the missing control are centered around encapsulation and information hiding, principal engineering techniques that are relevant here, too.

This chapter investigates visibility as central abstraction to cope with engineering complexity and introduces a scoping concept for event-based systems. As an design and engineering tool, scopes offer a module construct to structure applications and compose new functionality. Second, scopes reify aspects of event communication and thus make them adaptable within the composed modules, e.g., access to underlying communication technologies, delivery to module members, forwarding of events out of the module scope, transforming heterogeneous data sources, etc.

The first section analyzes the notion of visibility in event-based systems and relates it to the requirements defined in Sect. 5.1. The scoping concept is defined in Sect. 6.2, including a formal specification of scoped event-based systems that refines the specification of simple systems given in the previous chapter. Scopes reintroduce 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 Sects. 6.3 and 6.4 to include interfaces and mappings; the former further refine visibility control, the latter generalize interfaces to transform notifications at scope boundaries, coping with heterogeneous data models. While communication within scopes is by default like in traditional publish/subscribe systems, the transmission policies presented in Sect. 6.5 adapt the semantics of notification dissemination within scopes. In Sect. 6.6 we sketch a development process for scopes and present a declarative scope language for defining and manipulating scope graphs. Finally, we investigate implementation strategies for scopes in Sect. 6.7 and dis-

cuss combining these They open the publish/subscribe service implementation and allow for the integration of a wide variety of communication techniques.

6.1 Controlling Cooperation

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.

6.1.1 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 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 Sect. 5.1 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 visibility of notifications is essential for the overall function of an event-based system.

[1] Explicit and implicit coordination are also termed objective and subjective coordination in coordination theory [326].

6.1.2 Explicit Control of Visibility

The key to exploiting visibility is to regard it as a 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 are 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 reintegrating 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 because it concerns interaction and communication, but not the computation within the component. Thus, the necessary control must be exerted outside of the components themselves.

6.1.3 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. autonomic 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 architectures of open systems [206]. In terms of coordination theory, administrators are a means of objective coordination providing an exogenous

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

extension of event-based interaction [36], 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 Sect. 5.1, such a control is a prerequisite to solving 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.

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.

6.2 Event-Based Systems With Scopes

This section formally introduces the notion of scoping in event-based systems.[3] It extends the specification of the simple event system presented in Sect. 2.5.2 and is the basis for further extensions and reasoning about scoping functionality.

6.2.1 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.[4]

Scopes have interfaces to regulate the exchange of notification with the remaining system. Scopes forward external notifications to their 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

[3] see also [135, 146].

[4] In fact, in most cases we refer to the entity, which implies the scope of notifications in the former meaning.

higher level scopes and in this way offer a powerful structuring mechanism. Scopes thus act as 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.

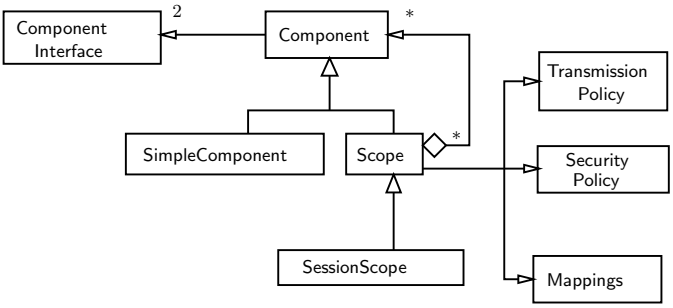


Fig. 6.1. A metamodel of scopes

The concept of scopes as illustrated in Fig. 6.1 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.

6.2.2 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 $\mathcal{S}$, $\mathcal{K} = \mathcal{C} \cup \mathcal{S}$.

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 6.1 (scope graph). *Let $\mathcal{K} = \mathcal{C} \cup \mathcal{S}$ 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 $(C_1, C_2) \in \mathsf{E} \Rightarrow C_2 \in \mathcal{S}$.*

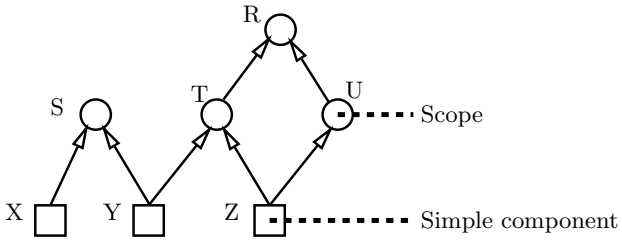


Fig. 6.2. An exemplary scope graph

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.[5] 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 Fig. 6.2, 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 Fig. 6.2, $Y \lhd T$ 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.[6] Informally, component X is visible to Y iff X and Y "share" a common superscope.

Definition 6.2 (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:*

[5] 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. [140].

[6] The more general visibility of individual notifications is discussed in Sect. 6.3.1.

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

In the graph of Fig. 6.2, 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 Def. 2.5 of Sect. 2.5 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 6.3 (scoped event system). *A* scoped event system $ES^{\mathbb{S}}$ *is a system that exhibits only traces satisfying the following requirements:*

- *(Safety)*

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

- *(Liveness)*

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

Definition 6.3 differs only slightly from Def. 2.5 in Sect. 2.5. 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 Def. 6.3 is a generalization of Def. 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 Def. 6.3, resulting in Def. 2.5.

6.2.3 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 $\overset{n}{\rightsquigarrow} C$. 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 $X \overset{n}{\rightsquigarrow} Y$, where

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

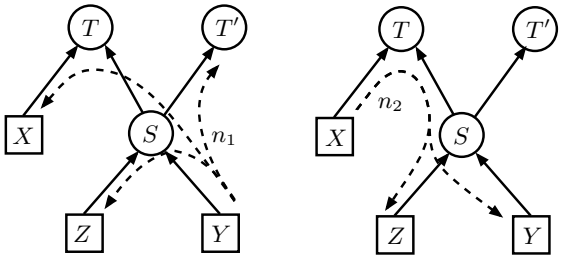
A published notification is made visible in the scopes the producer belongs to. $Y \overset{n_1}{\rightsquigarrow} S$ in Fig. 6.3a, or simply $\overset{n_1}{\rightsquigarrow} S$ to denote the visibility alone if the specific producer is not important. This rule is applied recursively to make notifications visible in all further superscopes; $Y \overset{n_1}{\rightsquigarrow} T$ and $Y \overset{n_1}{\rightsquigarrow} T'$. On the other hand, if a notification is visible within a scope S, $\overset{n}{\rightsquigarrow} S$, it is visible to all its children. Recursively applying this rule yields in Fig. 6.3b $X \overset{n}{\rightsquigarrow} T \Rightarrow X \overset{n}{\rightsquigarrow} S \Rightarrow X \overset{n}{\rightsquigarrow} Y$. 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:

$$\textbf{PP}: \quad X \overset{n}{\rightsquigarrow} S \wedge X \lhd S \lhd T \Rightarrow X \overset{n}{\rightsquigarrow} T \tag{6.1}$$

$$\textbf{DP}: \quad \overset{n}{\rightsquigarrow} T \wedge S \lhd T \Rightarrow \overset{n}{\rightsquigarrow} S \tag{6.2}$$

Consider Fig. 6.3. 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. $X \overset{n_2}{\rightsquigarrow} S$ is at first an external notification to S and is made internal by the delivery policy of Eq. (6.2). 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. 6.3a or from T to S in Fig. 6.3b; 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 Fig. 6.3 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)



(a) outgoing (b) incoming

Fig. 6.3. Outgoing and incoming notifications

siblings are eligible consumers as they are in the same scope, (ii) being a subscope also denotes 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 6.4. *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 6.1. *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 Eq. (6.1). 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 Eq. (6.2). 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, Eq. (6.1) 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. □

6.2.4 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. 6.4). 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.

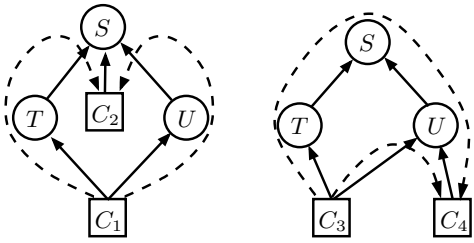


Fig. 6.4. Two ways of generating duplicates

The utilized implementation of scoping determines whether the conceptual replication really results in duplicate deliveries. A broad range of possible implementations of scoping exist,[7] 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 notification processing is idempotent so that duplicate delivery does

[7] Please refer to Sect. 6.7.1 for an overview.

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. 6.4, 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 consumer's 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 offer 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.

6.2.5 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: $cscope(S)$ and $dscope(S)$ to create and destroy a scope S, $jscope(X, S)$ and $lscope(X, S)$ 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. 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

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

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 Def. 6.3 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 Def. 6.3 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:

$$\sigma_5 = sub(Y, F), jscope(X, s), jscope(Y, s), pub(X, n_1), lscope(Y, s), \ldots, jscope(Y, s), pub(X, n_i), lscope(Y, s), \ldots$$

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.

6.2.6 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 Sect. 6.6.

Attributes may carry information about system configuration and management. Section 6.7.1 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. Sect. 6.6).

6.2.7 A Correct Implementation

The following presents a possible implementation of Def. 6.3 as a proof of concept. The implementation uses a simple event system as specified in Sect. 2.5.2 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 Fig. 6.5. 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.

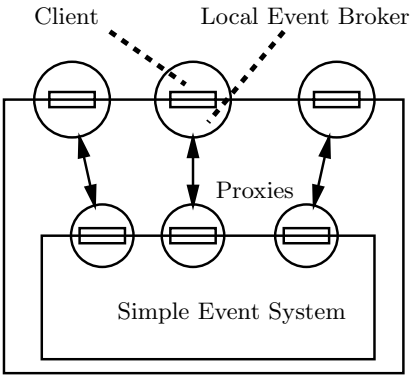


Fig. 6.5. A possible implementation of a scoped event system

Although dynamic scoping is not considered in the specification, the presented algorithm includes dynamic scopes in the style of Sect. 6.2.5. 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.

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

Calls to $sub(X, F)$ and $unsub(X, F)$ 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 $sub(Prox_X, \tilde{F})$.

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 $notify(Y, n)$. Otherwise the notification is discarded.

Correctness

In order to show that Def. 6.3 is satisfied, the presented implementation must obey the visibility $v(X, Y)$ of the safety condition and the additional precondition $\Box v(X, Y)$ of the liveness condition. The remaining part is satisfied by using the simple event system which satisfies Def. 2.5.

Lemma 6.2. *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 $v(X, Y)$. □

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 $notify(Y, n)$ 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. 6.3.

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

To prove the second and the third conjuncts, assume that the local broker issues $notify(Y, n)$ 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 6.2 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 $sub(Y, F)$ 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 6.2 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.

6.3 Event-Based Components

6.3.1 Component Interfaces

So far, visibility is an only two-level hierarchy induced by the topmost superscopes, the visibility roots of the graph G. Any two components are either able to see all of their published notifications or none 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 in Sect. 3.1, 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.[8]

6.3.2 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.

[8] The relationship between scopes and simple components is shown in the UML class diagram in Fig. 6.1.

The base interface I_C of a component C contains two sets of filters, ${}^{i}F_C$ and ${}^{o}F_C$, 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 ${}^{i}F_C$ for incoming notifications or ${}^{o}F_C$ 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 (Fig. 6.6). While the next paragraphs discusses the different forms of interfaces, the formal definition of a scoped event system with interfaces is given in Sect. 6.4.1.

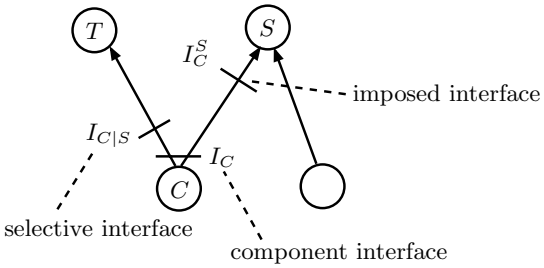


Fig. 6.6. Different scope interfaces

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* $I_{C|T}$ controls 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 Fig. 6.6, 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 toward 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. 6.6 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_C^S in Fig. 6.6 restricts the interaction of C with S, without affecting the other children in S. If I_C^S 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_C^S.

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 $\hat{I}_C^S$ of a component $C \lhd S$ iff it matches I_C *and* $I_{C|S}$ *and* I_C^S *and* I^S.

6.3.3 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 [369]. 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 (Sect. 6.6).

6.3.4 Example

The example stock trading application introduced in Sect. 5.1.2 is expanded to illustrate the use of scopes (Fig. 6.7). There are two main scopes, `M1` and `M2`, denoting two different stock markets. Within each market customers are grouped into subscopes 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's 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.

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. 6.8.

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

[9] Delayed forwarding is discussed in Sect. 6.5.

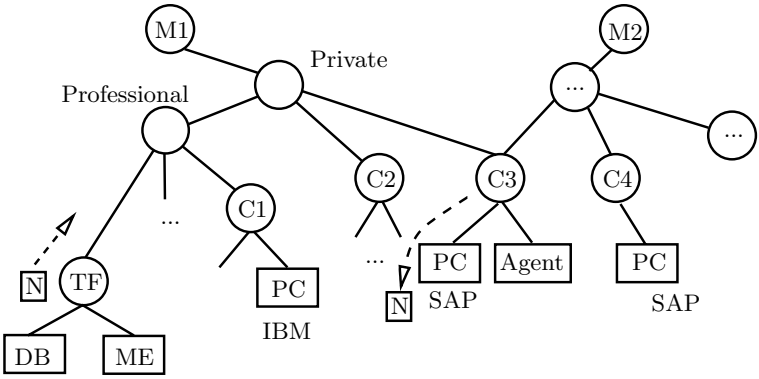


Fig. 6.7. The graph of the stock application

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)[9]
`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	–

Fig. 6.8. Interfaces of the components in the example application

sends out *Quotes*, informing about successfully executed orders. The trading floor itself is 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 ten 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.

6.4 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 Sect. 5.1. 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 Sect. 5.1 and facilitate construction and maintenance of large event-based systems.

6.4.1 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

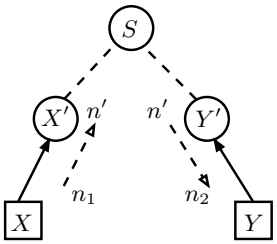


Fig. 6.9. Recursive definition of the relation $(n_1, X) \rightsquigarrow (n_2, Y)$

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 6.5. *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*:*

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

The recursive definition of $(n_1, X) \rightsquigarrow (n_2, Y)$ is illustrated by Fig. 6.9. 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 Sect. 6.2, 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 6.6 (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)*

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

- *(Liveness)*

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

The difference between this definition and that of scoped event systems (Def. 6.3) 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 nonmatching 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 6.7 (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 6.8 (interface). *An* interface I *consists of an input mapping* ${}^{i}I$ *and an output mapping* ${}^{o}I$*:* $I = ({}^{i}I, {}^{o}I) \in \mathcal{M} \times \mathcal{M}$*. The base interface* I_C *of a component* C *represents the sets of open subscriptions and advertisements of* C*:*

$$\begin{aligned} I_C &= ({}^{i}I_C, {}^{o}I_C) \in \mathcal{M} \times \mathcal{M} \\ &\triangleq ({}^{i}F_C, {}^{o}F_C) = \big\{\{F_1, F_2, \ldots, F_k\}, \{F'_1, F'_2, \ldots, F'_l\}\big\} \in P(\mathcal{F}) \times P(\mathcal{F}) \end{aligned}$$

where ${}^{i}I_C$ *and* ${}^{o}I_C$ *are defined as*

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

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

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 6.9 (concatenation of interfaces). *Two interfaces* I_1 *and* I_2 *are concatenated by*

$$I_1 \circ I_2 = ({}^{i}I_1 \circ {}^{i}I_2, {}^{o}I_2 \circ {}^{o}I_1).$$

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 no longer commutative. 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 Sect. 6.3.2.

Definition 6.10 (effective interface). *The* effective interface $\hat{I}^S_C$ *between two components* $C \lhd S$ *is given by concatenating base interface, selective interface, mapping, and imposed interface:*

$$\hat{I}^S_C = I_C \circ I_{C|S} \circ M^S_C \circ I^S_C \circ I^S$$

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

$$\begin{aligned} n_1 \searrow n_2 &\Leftrightarrow (I_C \circ I_{C|S} \circ {}^{i}M_C^S \circ {}^{i}I_C^S \circ {}^{i}I^S)(n_1) = n_2 \\ &\Leftrightarrow {}^{i}\hat{I}_C^S(n_1) = n_2 \\ n_1 \nearrow n_2 &\Leftrightarrow ({}^{o}I^S \circ {}^{o}I_C^S \circ {}^{o}M_C^S \circ {}^{o}I_{C|S} \circ {}^{o}I_C)(n_1) = n_2 \\ &\Leftrightarrow {}^{o}\hat{I}_C^S(n_1) = n_2 \end{aligned}$$

The rules of notification forwarding in the scope graph given by the publishing and delivery policies in Eqs. (6.1) and (6.2) can be refined corresponding to the above discussion:

$$\textbf{PP}: \quad X \overset{n_1}{\rightsquigarrow} S \wedge X \lhd S \lhd T \wedge {}^{o}\hat{I}_S^T(n_1) = n_2 \Rightarrow X \overset{n_1}{\rightsquigarrow}_{n_2} T \tag{6.3}$$

$$\textbf{DP}: \quad \overset{n_1}{\rightsquigarrow} T \wedge S \lhd T \wedge {}^{i}\hat{I}_S^T(n_1) = n_2 \Rightarrow \overset{n_2}{\rightsquigarrow} S \tag{6.4}$$

Despite the integration of interfaces and mappings, the scope overview in Fig. 6.1 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 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 book, 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 syntactic and semantic transformations that are applicable here [46, 79, 232]. 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 [146, 406].

6.4.2 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

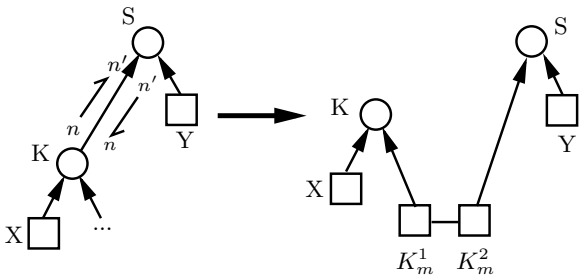


Fig. 6.10. Transformation of mappings into components

$ES^{\mathcal{M}}$ 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 6.10 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 6.11 describes the architecture of the implementation for the example system in Fig. 6.10. 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 $ES^{\mathcal{M}}$.

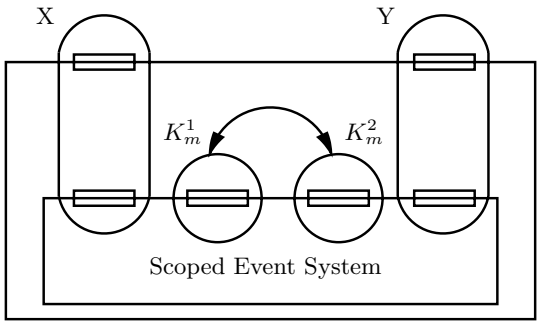


Fig. 6.11. Architecture of scoped event system with mappings

In general, if a scope K is to be joined to a superscope S by calling $jscope(K, S)$, two mapping components K_m^1 and K_m^2 are created that communicate directly via a point-to-point connection. K_m^1 joins K, subscribes to all notifications published in K, and 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_m^2 joins S and republishes all notifications it gets from its peer K_m^1. It subscribes in S according to the subscriptions forwarded by K_m^1, transforms any notifications received out of S, again with externally supplied functions, and forwards them to K_m^1, which republishes them into K.

Correctness

The algorithm from the previous section has to satisfy the requirements given in Def. 6.6 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 6.3. *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.*
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. 6.6 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 conjuncts.

The second conjunct is proved by a backward induction on the path guaranteed by Lemma 6.3. 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^{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 6.3 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 6.3 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.

6.4.3 Example

Returning to the stock exchange example, mappings can be exploited to convert between different currencies.[10] Quotations are typically given in a local currency which needs to be transformed at the boundary of the local scope in order to achieve comparability. As another example for the usefulness of mappings, consider XML languages like FIXML [273] 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.

6.5 Transmission Policies

The discussion of engineering requirements in Sect. 5.1 argued not only for the heterogeneity of data models but also emphasized the necessity to adapt noti-

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

fication 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.

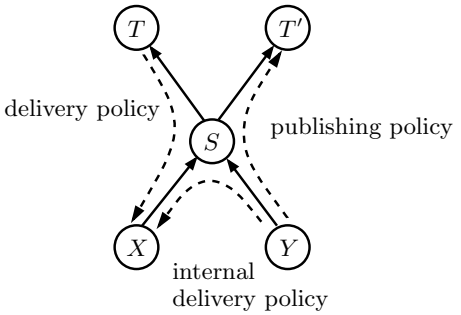


Fig. 6.12. Three important transmission policies in scope graphs

6.5.1 Publishing Policy

A *publishing policy* is associated with a component and controls into which superscopes an outgoing notification is forwarded. In Fig. 6.12, a publishing

policy at S can prevent a notification $Y \overset{n_1}{\rightsquigarrow} S$ from being forwarded to T, even 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 is 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 (PP) does not object to T. That is,

$$\textbf{PP}: \quad \underbrace{Y \overset{n_1}{\rightsquigarrow} S \wedge Y \lhd S \lhd T}_{\substack{\text{component}\\ \text{visibility}}} \wedge \underbrace{{}^o\hat{I}_S^T(n_1) = n_2}_{\substack{\text{interface}\\ \text{mappings}}} \wedge \underbrace{T \in pp_S(n_1)}_{\substack{\text{publishing}\\ \text{policy}}} \Rightarrow S \overset{n_1}{\rightsquigarrow}_{n_2} T \quad (6.5)$$

This definition of PP refines the previous one of scoped delivery with mappings given in Eq. (6.3). 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 Eqs. (6.5) and (6.3) 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 metadata. 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. 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.

6.5.2 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 Fig. 6.12, a delivery policy at S might direct a notification $T \overset{n}{\rightsquigarrow} S$ to X, prohibiting the delivery to Y even if the notification conforms to the effective input interface ${}^{i}\hat{I}_{Y}^{S}$. 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:

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

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 Fig. 6.12 and a notification visible to S in T, $T \overset{n_1}{\rightsquigarrow} S$. The visibility of the notification within S is then determined by

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

Please note that this equivalence not only guides forwarding of incoming notifications but also of internal notifications published by scope members; in the example, $T \overset{n_1}{\rightsquigarrow} S$ and $Y \overset{n_1'}{\rightsquigarrow} S$ 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[11] notification that is made visible by a superscope T, i.e., $X \lhd S \lhd T$ and $T \overset{n_1}{\rightsquigarrow} S$. 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 considered consumer X, $X \lhd S \rhd Y$ and $Y \overset{n_1'}{\rightsquigarrow} S$.

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

The rule of downward notification delivery (p. 173) is thus given as follows:

[11] The term “internal” and “incoming” notifications are also discussed on page 157 in Fig. 6.3a.

$$\textbf{DP}: \quad \underbrace{T \overset{n_1}{\rightsquigarrow} S \wedge X \lhd S \lhd T}_{\substack{\text{component visibility}\\ \text{incoming notification}}} \wedge \underbrace{{}^{i}\hat{I}_X^S(n_1) = n_2}_{\substack{\text{interface}\\ \text{mappings}}} \wedge \underbrace{X \in dp_S(n_1)}_{\substack{\text{delivery}\\ \text{policy}}} \Rightarrow S \overset{n_1}{\rightsquigarrow}_{n_2} X \tag{6.6}$$

$$\textbf{iDP}: \quad \underbrace{Y \overset{n_1}{\rightsquigarrow} S \wedge X \lhd S \rhd Y}_{\substack{\text{component visibility}\\ \text{internal notification}}} \wedge \underbrace{{}^{i}\hat{I}_X^S(n_1) = n_2}_{\substack{\text{interface}\\ \text{mappings}}} \wedge \underbrace{X \in idp_S(n_1)}_{\substack{\text{internal}\\ \text{delivery policy}}} \Rightarrow S \overset{n_1}{\rightsquigarrow}_{n_2} X \tag{6.7}$$

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

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.

6.5.3 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 metaobject protocols [221] 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.

6.5.4 Influencing Notification Dissemination

Transmission policies are a means to adapt the event-based dissemination within scopes, i.e., to tailor the quality of service (QoS). They make the interaction in the graph programmable.

To some extent transmission policies bear similarities to metaobject protocols (MOP) known in object-oriented programming [221]. Metaobject protocols offer the ability to redirect or transform messages sent as method calls, and this control allows one 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 Def. 6.6, 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 metadata. 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 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 toward 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.[12]

6.6 Engineering With Scopes

Scopes are an engineering abstraction for event-based systems. 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* scoped event system, describes a running 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. Sect. 6.2.6) 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 [245].

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

6.6.1 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

[12] Let us call the releasing notifications *commit* and *abort* and you see the link to transactions.

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 chosen, 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.

6.6.2 Scope Graph Handling

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 here: scope API, XML description, and SQL-like language. A basic programming API, e.g., in Java, is easily conceivable. A scope class is the base class with a default implementation of scope, which can be specialized in subclasses. On the programming language level, scope classes are part of the descriptive scope graph and objects constitute the instantiated scope graph. However, the scope concept is too generic to come up with exactly one API proposition; an example can be found in [135].

The context of a scope is a list of requirements that is better encoded in a descriptive language, like XML or SQL. An XSchema definition of scope graphs defines the entities that compose a descriptive scope graph in form of

an XML document. It includes descriptions of single scopes and their dependencies in a scope graphs, but may also contain information about network layout and broker networks; an example is available in [268].

The specification of dependencies to other points are called *coupling points*, which is a variation of UML (Unified Modeling Language) ports and interfaces. 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 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 later in this section facilitates the definition of scopes and their features, and includes coupling points to express dependencies, 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 the loose coupling of application components, they 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 for different scopes. We can use abstract scopes to define different administrative domains [268].

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.

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

[13] The use of ontologies like in concept-based publish/subscribe [79] is an example of such externally provided agreements.

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. Unfortunately, there are no general guidelines available so far.

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.

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 (ECA) 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, [154]). For example, annotations of required quality of service attributes may govern the following deployment step, hinting at appropriate implementation techniques.[14]

Scope Graph Deployment

Scope deployment creates or extends an instantiated scope graph, which contains all scopes currently running in the system. This step deploys pre-configured 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

[14] Obviously, the stepwise transformation and deployment of the scope graph resembles the ideas of model-driven development [155].

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. 6.1.

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 covers 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 reacts 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 [29].

Such rules can be used to define scopes that automatically include all components conforming to a certain condition. One example is mobile systems, which are an apparent 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 [142].

[15] Grouping always implies a common context, and scoping thus may contribute to the discussion about context in mobile systems [335].

6.6.3 Scope Graph Language

In order to support the development process a specification language for scope graphs is defined next. Corresponding to the generic nature of the scope concept, the language definition is intended to be open for further refinements, which are probably domain dependent. 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
```

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 superscopes 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 later in this section.

```
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, for example

```
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 chosen 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 that evaluates 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. Sect. 6.3.1), and so a syntax corresponding to the available filter model must be chosen.

```
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. Sect. 6.3.2).

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.[16] 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 define 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.[17]

```
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
```

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

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

```
  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, but 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 [349] for a scenario of wireless sensor networks: application code is assigned to network nodes based on scoped definitions.

Updates

The update clause defines 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, and `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 notification 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 Sect. 6.7.1. 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 ]
```

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
```

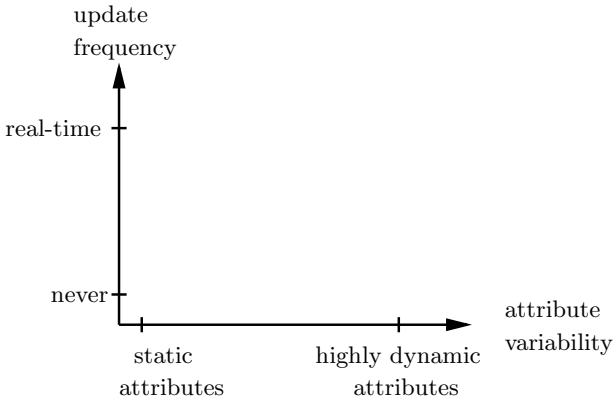


Fig. 6.13. Scope definition accuracy

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%? Fig. 6.13 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.

We assume that queries are evaluated at deployment time only and that their result is not automatically updated afterwards. This corresponds to the lower left point in Fig. 6.13. However, the update clauses in scope definitions allow system engineers to install custom ECA rules to maintain accuracy.

6.7 Implementation Strategies for Scoping

The concept of scopes can be implemented on top of a variety of techniques. In fact, the ideas underlying the scope concept are quite common, but visibility control is often implemented only partially and in an ad hoc manner.

This section investigates a number of approaches for implementing scopes. They differ in the characteristics of the communication media used to convey messages and in the strategies for scope graph distribution. The resulting *scope architectures* are the blueprints of the implementation. All the architectures implement the visibility constraints defined by scopes, but they diverge

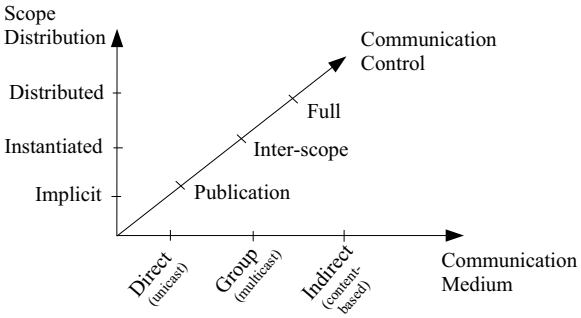


Fig. 6.14. Design dimensions of scope architectures

in their support of other quality of service parameters, like communication reliability and performance, and they also influence system extensibility and adaptability. They emphasize different aspects of the visibility abstraction and are therefore eligible for different application environments.

6.7.1 Scope Architectures

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 (Fig. 6.14): communication medium and scope implementation. The combination of these dimensions gives rise to a number of *scope architectures* that determine the principal layout of the scoped event service (cf. Fig. 6.16). The third dimension turns out to classify the architectures' ability to control communication. This section details the architectural choices and defines a metric for comparing the architectures presented later.

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 [370] connections to IP multicast [106] and peer-to-peer networks [310, 374]. Moreover, existing publish/subscribe services, database management systems [172], and tuple spaces [174] 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 Sect. 6.8 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, rather 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, while 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* later in this section). 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 are 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 [319] 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 [258, 320], where messages are sent only to receivers that are physically close by, i.e., addressees are implicitly determined by location metadata.

Communication Media, Publish/Subscribe, 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 publish/subscribe API, group and indirect addressing is related to the discussion on filter models (channel-, subject-, and content-based filtering) given in Sect. 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 [289, 380], which is, to some extent, also supported by IP multicast [259].

Group-based multicast media establish visibility constraints in that they encapsulate intragroup 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 viewpoints with the help of subgroups, the exponentially growing number of necessary groups limits practical applicability (see Sect. 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 publish/subscribe, implementations based on database management systems (DBMS), and tuple spaces, they are able to carry different viewpoints (content vs. structure) simultaneously. Available products/prototypes are able to offer only a few of the features of scopes, but they are an ideal basis for their implementation.

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 capabilities, 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. 6.3 in Sect. 6.2.2.

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

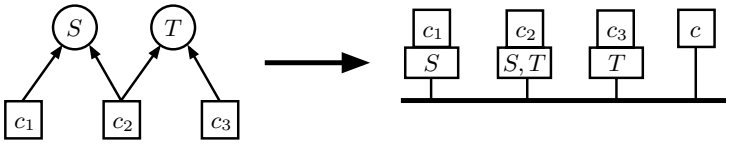


Fig. 6.15. Implicit implementation shifts visibility control into application components

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 Fig. 6.15. Every component is connected to the same medium, and conventions must determine 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 systems, because it simplifies notification routing and access control, but comes at

the expense of scalability and diversity support. Examples range from centralized databases [172, 292] (see later in this section) to content delivery networks [333], 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 Sect. 6.8). 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 intrascope 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.

Example Architectures

Figure 6.16 shows possible architectures that are defined as specific combinations of scope distribution and communication medium. They are sketched in the following and two of them are detailed in Sects. 6.7.3 and 6.7.4.

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

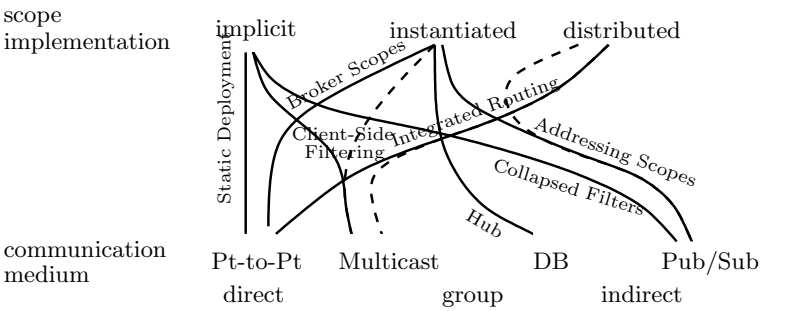


Fig. 6.16. A comparison of scope architectures

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 toward 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, which 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 the local event brokers without affecting the publish/subscribe 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 [296, 297]), which connect input/output ports of coordinated entities, and even an implementation using TCP/IP to connect the participants is eligible, particularly if the system footprint has to be kept small. Interestingly, the JavaBeans programming model [84, 359] and component-oriented programming in general [234, 369] 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 publish/subscribe 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 earlier in this section, 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 [87, 291]. 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, the visibility constraints expressed in the scope graph are merged into the 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 publish/subscribe. Extra effort is necessary on both the producer and consumer sides. Producers, i.e., their local event broker, annotate notifications and add data necessary for visibility filtering. Consumers have to extend their subscriptions to test as much of the imposed visibility constraints as possible. If the filter model is not expressive enough, they must locally evaluate the remaining filters on every received notification.

The collapsed filters approach is a simple implementation of scoping as a layer on top of an existing communication infrastructure. But 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, because application components are not easily reconfigurable and changes to the graph have to be consistently distributed to all affected components.

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. A corrupted or malevolent component may publish or eavesdrop in any scope. The discussion on combining different scope architectures in Sect. 6.8 leads to a possible solution when gateway components bridge two separated subgraphs and provide an explicit encapsulation of visibility constraints.

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 it easily offers sophisticated quality of service guarantees in addition to the basic safety and liveness requirements of scoped event systems.

Using databases for implementation blurs the distinction between the collapsed filter and the central hub scope architectures. Similar to the content-based publish/subscribe medium assumed above, a database table can hold all published notifications, and subscriptions are merely queries to this table. In fact, database technology provides a wide spectrum of functionality [172] that may be exploited to extend the quality of service offered by the event system beyond the definitions given in Chap. 2. On the other hand, there are drawbacks like their maintenance complexity, resource consumption, and acquisition and operation costs.

Addressing Scopes

Addressing scopes is an extension of the client-side filtering approach that no longer relies on multicast but instead on content-based publish/subscribe. Each scope has a unique name that is appended to published notifications. Every subscription is extended to accept notifications only if they are issued in the consumer's scope. 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 interscope communication than the collapsed scopes.

Scoping is still implemented on a shared multicast medium and the implementation is not aware of the underlying network layout. In fact, intrascope communication is not directly governed by the administrative 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 intrascope 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 on a well-known tradeoff between scalability and expressiveness [69]. 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.

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. Sect. 2.4). This approach is detailed in Sect. 6.7.3.

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 Sect. 6.7.4.

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 6.17 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.

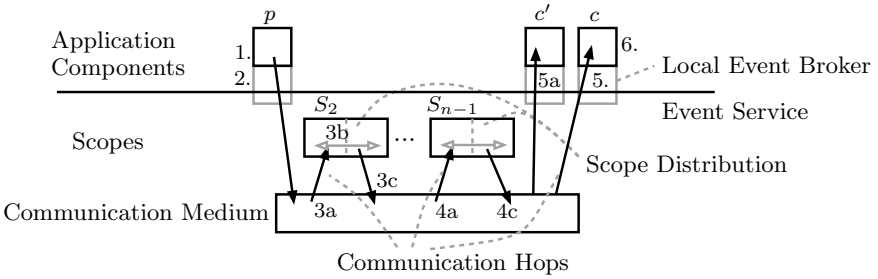


Fig. 6.17. Steps of scoped notification delivery

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 toward 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.

	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

Fig. 6.18. Types of architectures, their characteristics, and examples

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 Fig. 6.17, whereas communication between instances of the same distributed scope (step 3b) is not counted, since 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. 6.18.

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 toward 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 nondiscriminated 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.

6.7.2 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 publish/subscribe 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 Fig. 6.19.

	impact on			ability to		
	infrastr.	components	overhead	reliability	reconfigure	customize
collapsed filters	+	–	○	–	–	–
hub	+	○	○	+	○	+
static deploy.	+	–	+	○	–	○
addressing scopes	+	○	○	○	+	○
broker scopes	○	+	○	+	+	+
integrated routing	–	+	+	+	+	+

Fig. 6.19. Comparison of scope architectures (+ means low impact and overhead, and high ability to achieve reliability, reconfiguration, and customization)

6.7.3 Implement 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 intrascope communication. It directly implements the structure of the scope graph in the sense that publishing within a scope first requires accessing 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 Fig. 6.17, 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.

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 intrascope 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 Sect. 2.4, only that a strict treelike 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 degradation 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 intrascope 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.

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 Fig. 6.17. 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 [11] or operating in parallel independently 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 administrative 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 administrative 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 exploit 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 re-

duced once the connected administrative 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 are mixed, which is investigated in Sect. 6.7.4.

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 interscope communication is done locally. Although closely related to the central hub approach, cf. Sect. 6.7.1, the scope graph is explicitly instantiated here, only that interscope communication is implemented by interprocess communication (IPC). Separate administrative components can still evolve independently, they just happen to be collocated, so to speak, to improve efficiency, auditability, or other global constraints.

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 it is the most generic one. It is not a service of the publish/subscribe infrastructure, but instead 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 Sect. 6.8. 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.

6.7.4 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.

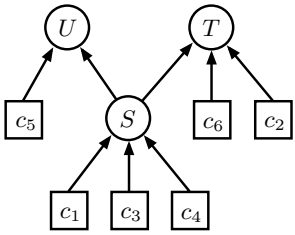


Fig. 6.20. An exemplary scope graph

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 Figs. 6.20 and 6.21. Brokers B_1, B_2, and B_4 are part of scope T, that is, they are *scope brokers*[18] of T. Together with B_3, they are also scope brokers of S. B_2 is in both cases an intermediate broker that currently does not have any directly connected scope members. B_1 and B_5 are scope brokers of U.

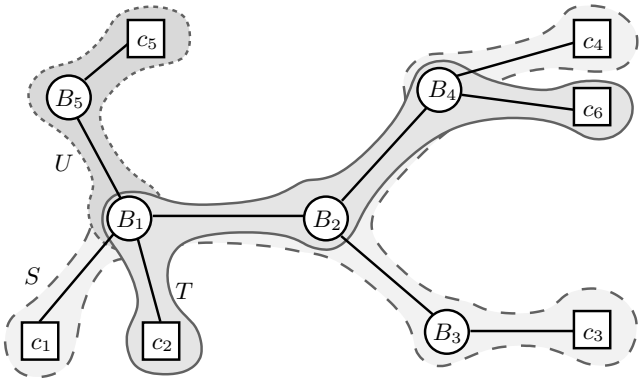


Fig. 6.21. Scopes as overlays within the broker topology

[18] 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 architecture and a different way to implement the scope graph (Sect. 6.7.3).

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

Fig. 6.22. A flat routing table for broker B_1

The main idea is to rely on any of the existing routing schemes, e.g., those offered by REBECA (Sect. 2.4), as before, but to use it for intrascope 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 Fig. 6.21. On the other hand, the separation of scope-internal routing necessitates a special handling of interscope transitions. In Fig. 6.21, 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: intrascope within a specific scope and interscope routing between scopes adjacent in the scope graph. In intrascope routing each scope 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 publish/subscribe service in which no visibility constraints exist. In interscope 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.

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 6.22 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 Fig. 6.23. 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 separately 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 $\mathsf{RT}^{S_i}_{B_i}$ 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 Fig. 6.24.

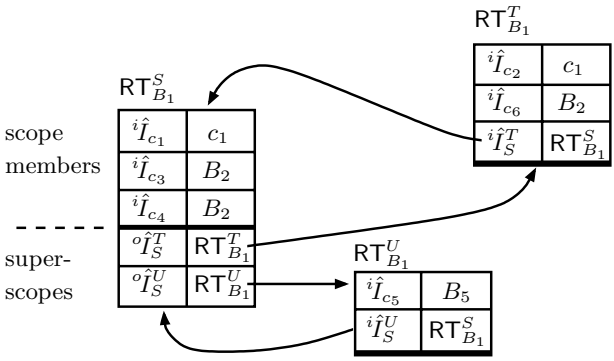


Fig. 6.23. Enhanced routing tables of B_1 incorporating scopes

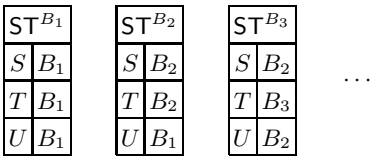


Fig. 6.24. Scope lookup tables

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*, Sect. 6.2.5 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 6.6 has suggested tools that support system engineers in this task.

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 is the case for advertisements or subscriptions in flat publish/subscribe systems. Applying scope interfaces to restrict the distribution of scope announcements helps limit 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 an 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 rout-

ing 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 table is forwarded along the stored trail toward 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 toward 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., $lscope(S, T)$, 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 $\mathsf{RT}_{B_i}^{S}$ and $\mathsf{RT}_{B_i}^{T}$ routing tables.

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 $\mathsf{RT}_{B}^{S_i}$ 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), respectively. 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 edgewise 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 [233] might be devised that maps identifiers on both sides of a network link to guarantee uniqueness.

The next paragraphs introduce 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 states, 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. Sect. 6.7.4.

The Algorithm

Figures 6.25 and 6.26 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

```
        procedure main_loop
          loop
            // the queue is fed from network links
 4          (n, l) = get_next (recvQ)
            scoped_routing(n, l)
          end
        end

 9      procedure scoped_routing (n, l)
        Input n: notification
              l: source link

          s := get_state(n)
          S := scope(n)
14
          --- internal routing
          D := destinations(n, remote_components(RT_B^S))
          foreach (n', l') ∈ D
            if l ≠ l' then send (n', l')
19        end

          --- downward routing
          D := destinations(n, subscopes(RT_B^S))
          cross_scope(S, D, "downward")
24
          --- upward routing
          if not s = "downward" then
            D := destinations(n, superscopes(RT_B^S))
            cross_scope(S, D, "upward")
29        fi
        end
```

Fig. 6.25. Overall routing algorithm

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 the case of $\mathsf{RT}_{B_1}^S$ of Fig. 6.23, 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

```
function destinations (n, T)
Input n: notification
       T: routing table
Output D: list of notification-destination pairs

  foreach (I, d) ∈ T
    n' := I(n)
    if n' ≠ ϵ then
      D := D ∪ (n', d)
    fi
  end
end
```

Fig. 6.26. The naïve matching algorithm with mappings

subscopes are determined; both must be done for all routing states. A distinction of states is at this point only necessary when transmission policies are applied, cf. Sect. 6.7.4. 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 Sect. 6.4.1 that describe visibility in the scope graph.

The above procedures rely on the function `destinations` to determine all eligible destinations in the specified routing table. The naïve matching algorithm, extended with mappings, is given in Fig. 6.26 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., [133, 404], and a more sophisticated handling of notification copies may be applied.

Crossing Scopes

The scoped routing algorithm relies on `cross_scope` to forward a notification between scopes (Fig. 6.27). 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 interscope 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

```
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, RT_B^{S'}) ∈ D
    if get_source(n) ≠ S' then
      set_source(n, S)
      set_scope(n, S')
      set_state(n, s)
      put_in_front(recvQ, (n, RT_B^S))
    fi
  end
end
```

Fig. 6.27. Interscope forwarding

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 interscope 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 interscope 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 interscope traffic of a scope S is handled by a single gateway broker B_i of that scope. Only at this gateway the routing table $\mathsf{RT}_{B_i}^{S}$ contains an entry pointing to sub- and superscopes.
2. Transition anywhere, but only once: Interscope 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 $({}^{o}\hat{I}_{S}^{T}, \mathsf{RT}_{B_1}^{T})$ as part of $\mathsf{RT}_{B_1}^{S}$ if B_1 is the gateway broker of $S \rightsquigarrow T$. All other scope brokers of S register an entry ${}^{o}\hat{I}_{S}^{T}$ that points toward this gateway broker, e.g., $({}^{o}\hat{I}_{S}^{T}, B_2)$ 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 intrascope 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 corresponds 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 toward $\texttt{link}(n)$. 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 destinations. 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.

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 Sect. 6.5, 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.

Scope Multicast

So far, intrascope 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, intrascope 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.

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 build 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.

6.8 Combining Different Implementations

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

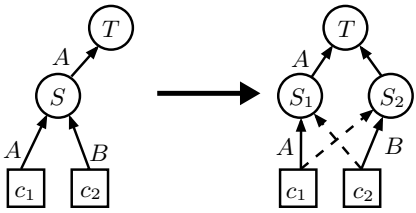


Fig. 6.28. Duplicate scopes to separate QoS requirements

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.

6.8.1 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 (QoS) even within a specific scope. For example, consider noncritical timer information sent in bulk (type A in Fig. 6.28) 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 Fig. 6.28, and each instance is tailored for a different kind of QoS supported. The interfaces are split 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.

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.

6.8.2 Bridging Architectures

Combining different scope architectures requires a gateway between the different implementations of two scopes $S \triangleleft T$. 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. Fig. 6.17). 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. Because of 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 Sect. 6.4.2 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 the 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, while 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 control 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 another's scope architecture. Interface and transmission policies are handled as before—they are always explicitly applied. The discussion about locating cross-scope transitions (cf. Sect. 6.7.4) holds for gateways as well.

6.8.3 Integration With Other Notification Services

The gateway of a scope may not only bridge different scope architectures, but may 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 nonscoped ("legacy") clients are still maintained in separate old-style routing tables RT_{B_i}. In fact, the overlay of all RT_{B_i} constitutes a *default scope* to which every newly created simple component may be assigned. In this way, scoped and nonscoped clients can interact in a controlled way.

6.9 Further Reading

This chapter has introduced the concept of scoping in event-based systems. It offers a module construct; it is an extension point for the integration of

different communication techniques, for handling quality of service, security, and data heterogeneity; and it facilitates the management of event-based systems. Accordingly, related work is very broad and comes from many areas of computer science [135].

The Common Object Request Broker Architecture (CORBA) provides a number of mechanisms to organize and structure distributed systems [283]. It includes the CORBA Notification Service [287]. Event management domains [282] support the federation of multiple notification channels in arbitrary topologies. However, applications have to select their channels and thus move information about application structure into the components—there is no support for an administrator to orchestrate channels and components. A generic solution to avoid static configurations is reflective middleware [96, 223].

The enterprise edition of Java (J2EE, [365]) specifies an execution environment that contains a component model and a number of standardized services, including a notification service (JMS), which is described in Sect. 9.1.3. The standard offers the plain publish/subscribe API (plus transaction support), but not the engineering features of scopes. Many JMS implementations exist, some offer extensions like topic hierarchies, e.g., [97]. In terms of managing application components, Java Management Extensions (JMX) defines a standardized Java way to management and monitoring [363].

If a database management system such as Oracle Streams Advanced Queuing (AQ), cf. Sect. 9.2.3, transports our notifications, we can exploit all the features offered by a database, like transactions, rules, consistency constraints, logging, high availability, authentication, access control, etc., and apply them to the publish/subscribe communication as well [172]. The focus is then more on advanced QoS features than on lean implementation.

Many domain-specific implementations of publish/subscribe run into the engineering issues addressed by scopes. Eder and Panagos [118] pointed out the problems that arise from missing structures in workflow systems. They connected workflow engines from multiple sites with the READY notification service [185]. The service introduced *event zones* to cluster components based on (either) logical, administrative, or geographical boundaries. Boundary brokers connect zones and control the communication between them. A component can belong to only one zone, which limits the structuring capabilities and prohibits composition and mixing of aspects [147, 188].

Wireless sensor networks (WSNs, [95]) also exploit eventing, and the need for structuring mechanisms was identified before, cf. [350, 397].

The field of software architecture is concerned with the overall organization of a software system [165]. Architecture definition languages (ADLs) are employed to describe the high-level conceptual architecture consisting of components, connectors, and specific configurations [256] 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 are among them. Luckham [242] presented the RAPIDE language family. It includes event processing agents to encapsulate event pro-

cessing rules behind input and output interfaces. The architecture definition language can be used to arrange a number of these agents, similar to scope graphs.

Sullivan and Notkin introduced mediators as a design approach that explicitly instantiates and expresses integration relationships and separates them from component function [356]. In a less general approach, Evans and Dickman defined *zones* to support partial system evolution [130]. Barrett et al. [31] proposed an event-based integration (EBI) framework that also covers scope features like transmission policies, mappings, and hierarchical grouping.

As event services are the basis for application integration and evolution, they cannot be expected to run in homogeneous environments. Heterogeneity issues can be handled in traditional request/reply systems, but they are rarely considered in event systems [32]. Database research contributes to the necessary syntactic and semantic data mappings [54, 80].

The field of *coordination theory* investigates techniques for managing the dependencies between a set of active components [246]. It differentiates computation from coordination [295] and localizes interaction in *coordination media* [64, 78]. Scopes event-based communication directly corresponds to this viewpoint.

A key point of scoping is that it does not imply a specific implementation per se. Depending on the intended semantics, adaptability, communication efficiency, etc., alternative implementations are applicable. The system engineer can incorporate existing work on group communication [319]. A wide variety of work exists in this area that supports nested groups [42] and reliable communication [43, 213]. Peer-to-peer systems are another candidate [311, 331, 374].

On lower layers, IP multicast is an obvious implementation candidate. Deering and Cheriton [106] introduce multicast scope control with the help of time-to-live fields (TTL). Administratively scoped IP multicast exploits hierarchical administrative boundaries [259]. Multicast scopes bundle network nodes, but do not support communication between scopes and require static configuration within the IP network routers. Interestingly, such multicast scopes allow us to implement publish/subscribe on IP multicast [26, 291, 357] only within restricted parts of the scope graph.