8 Advanced Topics

In this chapter we provide an overview of several areas of event-based systems that are still the focus of ongoing research. The entire space is too vast to be covered in this book, so we have chosen five topics that are of particular interest to designers of event-based systems instead. In Sect. 8.1 we discuss *security* in a publish/subscribe system and describe a secure publish/subscribe model that can be used as a foundation for access control using events. Section 8.2 investigates the issue of *fault tolerance* in event-based systems. The goal is to build systems that are robust in the face of failure, for instance, by designing self-stabilizing routing algorithms that are guaranteed to reach a correct state after a finite number of steps. The issue of *congestion* in publish/subscribe systems is addresses in Sect. 8.3. Here, we give an example of two congestion control algorithms that are targeted at the asynchronous, decoupled communication in a network of event brokers. Finally, in Sect. 8.4 we focus on *mobility* in event-based systems. The loose coupling of clients in a publish/subscribe system has natural advantages when applied to mobile clients that migrate through the systems, deattaching and reattaching at different points in the network.

8.1 Security

Security has received surprisingly little attention in publish/subscribe systems so far. Unlike composite event detection, it affects many different parts of a publish/subscribe system. In this chapter, we provide an example of a security service [34] for a distributed event system that uses role-based access control to provide three mechanisms: restrictions on the interaction of event clients with the publish/subscribe system, trust levels for event brokers, and the encryption of event data to control information flow in the publish/subscribe system on a fine-grained basis. An advantage of this approach is that it does not require separation of the overlay broker network into distinct trust domains but instead any broker can handle any potentially encrypted event.

The described security service is influenced by the security needs of two applications scenarios discussed in the next section. In Sect. 8.1.2 we define the requirements of a security service, showing how publish/subscribe communication impacts on security. After briefly summarizing existing access control techniques in Sect. 8.1.3, we introduce the secure publish/subscribe model implemented by the service in Sect. 8.1.4. It includes boundary access control using restrictions, different levels of event broker trust, and encryption of event attributes. We finish the overview of related work on security in publish/subscribe systems in Sect. 8.1.5.

8.1.1 Application Scenarios

In this section we look at two application scenarios and examine how they motivate the need for security in a publish/subscribe system. When considering security, we focus on issues of access control to the system and confidentiality of the event data being disseminated in the system.

The Active City

The *Active City* is an extension of theActive Office environment introduced in Sect. 7.1 to a geographically larger system covering an entire city. In an Active City, different city services, such as police and fire departments, ambulances, hospitals, and news agencies, cooperate using a shared event system for information dissemination. Since these city services are under separate management and have individual security implications, the event system must be flexible enough to accommodate a wide range of security policies and mechanisms to enforce them.

An excerpt of a sample event type hierarchy with event attributes that could be employed by cooperating services in an Active City is shown in Fig. 8.1. Information about a road traffic accident reported to the police in an `AccidentEvent` should be visible to the emergency services so that an ambulance can be dispatched if there are any casualties, but only anonymized data should be passed on to a news agency. The challenge is that some information may flow freely through the Active City, whereas other information has to be closely controlled. A simple solution would be for each city service to operate a separate, trusted event-based middleware deployment with controlled gateways between networks, forming an event federation [192]. However, this would result in complex policy management at the gateways, a significant waste of resources due to redundancy, and an increased event notification delay between services. It would also prevent event clients from one domain using the infrastructure of another while roaming. For this application scenario, a more complex solution is required.

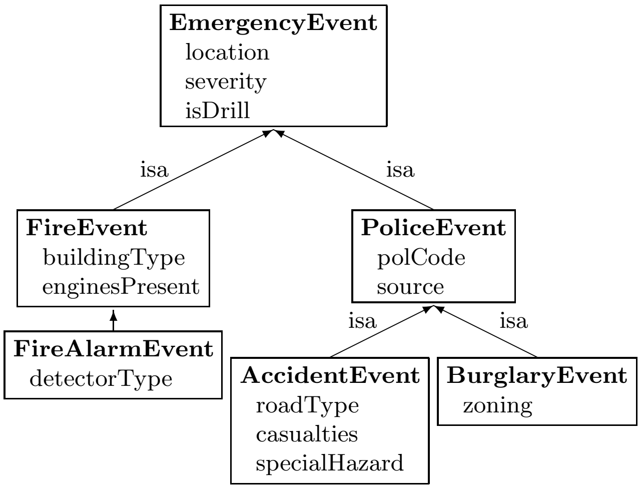


Fig. 8.1. An event type hierarchy for the Active City

News Story Dissemination

In an Internet-wide system for the dissemination of news stories, it is important that customers only receive the service that they are paying for. For example, a customer who has subscribed to a premium service should receive up-to-date news bulletins without delay, as opposed to a standard service subscriber that can only see events relating to older news reports. Moreover, subscribers should only be allowed to subscribe to the news topics that they are entitled to. To ensure this, it is not sufficient to merely rely on subscriptions in the publish/subscribe system because event brokers that perform content-based routing of news events may be under the administration of customers and thus not trusted to honor subscriptions correctly. Using partially trusted event brokers for event dissemination in customer networks is otherwise in the interest of news agencies because it reduces the resource requirements of their middleware deployments. When the service subscription of a customer changes, the event system should quickly adapt to the change in policy.

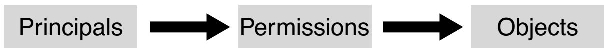
8.1.2 Requirements

Security mechanisms for an event system differ from traditional middleware security because of publish/subscribe communication semantics. Many-to-many interaction in a publish/subscribe system mandates a scalable access control mechanism. The anonymity of the loose coupling between event publishers and subscribers makes it difficult to use standard security techniques, such as access control lists, since principals can often not be identified beforehand. Content-based routing of events conflicts with the encryption of data because an event broker must have access to the content of an event for its routing decision [390]. Any access control mechanism should incur little overhead at publication time because event publications may have a high rate and thus routing should be carried out as quickly as possible.

Since event clients are not trusted, a security service should include perimeter security to control access of event clients to the publish/subscribe system. As seen in the application scenarios, event brokers are trusted to cooperate for the sake of event dissemination, but they may not be allowed to see all event data. Different levels of event broker trust are necessary and must come with mechanisms to remove compromised event brokers. The confidentiality of data stored in event attributes must be preserved even in the light of event matching and content-based routing. At the same time, as much as possible of the overlay broker network should be used for event dissemination so that a single infrastructure for both public and private information exists in order to improve efficiency, administerability, and redundancy in the publish/subscribe system.

8.1.3 Access Control Techniques

In this section, we describe different access control techniques and highlight their applicability to publish/subscribe communication. We assume that the system consists of a set of *objects*, a set of *principals*, and a set of *permissions*. The goal of an access control scheme is to define what principals have what permission to access what objects, as shown below. In a publish/subscribe context, principals correspond to event clients, objects are the event notifications, and permission are the standard operations, such as *publish* and *subscribe*.



Our discussion will focus on *discretionary* access control, where users themselves set access control rights, as opposed to *mandatory* access control, in which rights are set by a centralized authority for the entire system. Discretionary access control is more suitable for large-scale distributed systems because it does not depend on a centralized entity.

Access Control Lists

A simple way to implement discretionary access control are *access control lists (ACLs)*. An ACL is associated with every object and specifies the access permissions of principals to that object. For example, an ACL for file *foo* may state that the file is writable and readable by user *Alice* and only readable by user *Bob*.

A drawback of ACLs is that they create a direct mapping between objects and principals. This is undesirable in a publish/subscribe context, in which the identities of event publishers and event subscribers are not globally known. In addition, events in a publish/subscribe system are short-lived, which makes them bad candidates to manage access permissions.

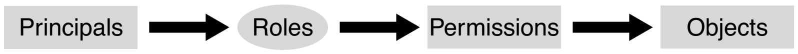
Capabilities

The opposite approach to an ACL is a *capability* that stores access permissions with the principal instead of the object. When a principal wants to access a given object, she must present the capability with the appropriate permissions first. For example, the capability owned by user Alice may state that she can read and write file *foo* and only read file *bar*. Of course, this means that capabilities need to be protected from tampering by principals through digital signatures or secure storage in memory.

Capabilities are more compatible with publish/subscribe communication because event clients manage their own capabilities. However, they are harder to manage because access permissions cannot easily be revoked. In addition, capability-based access control is often not scalable because principals may end up with a large number capabilities when the set of objects in the system changes dynamically. Finally, it also suffers from the problem that both principals and objects need to know about each other, which is not the case in a publish/subscribe system.

Role-Based Access Control

An access control model that extends capabilities and attempts to address some of its short-comings is *role-based access control* model (RBAC) [332]. RBAC simplifies security administration by introducing *roles* as an abstraction between *principals* and *permissions*, as shown below. Roles permit principals and permissions to be grouped intuitively in the system and addresses the anonymity of event clients in an event-based middleware. This grouping increases scalability of the access control mechanism because there are fewer roles than principals and permission in the system. The access control policy for the system focuses on the concept of a role, which is long-lived. To obtain privileges, a principal such as an event publisher or subscriber presents credentials that allow it to acquire a role membership that is associated with the desired permissions. In the rest of this chapter, we describe an access control model for publish/subscribe communication that is based on RBAC.



In our secure publish/subscribe model, we assume the decentralized implementation of a RBAC scheme, such as *Open Architecture for Secure Interworking Services (OASIS)* [22]. OASIS includes an expressive policy language to specify rules for role acquisition. It uses a session-based approach with

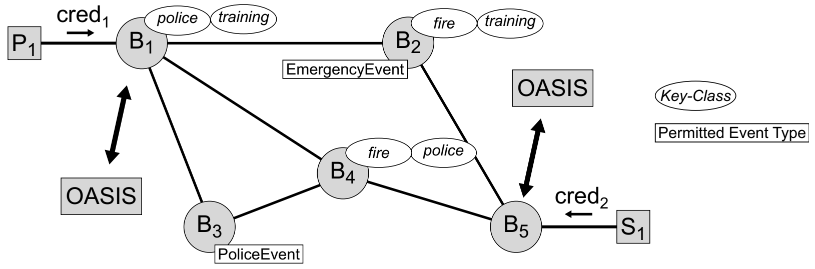


Fig. 8.2. Illustration of the secure publish/subscribe model

event communication to revoke currently active roles of principals in a timely manner after prerequisite credentials were revoked. Credentials in the OASIS implementation are protected with X.509 certificates [207] for authentication and proof of role membership.

8.1.4 Secure Publish/Subscribe Model

In this section we describe a secure publish/subscribe model for an event system. As a general design philosophy, the model couples access control with types of events. If the event space is already structured into event types, it is intuitive to leverage this for the specification of access control policy, but more fine-grained specification in terms of event attributes and content-based subscriptions is also supported by the model.

An example of an distributed publish/subscribe deployment with event brokers that implement the secure publish/subscribe model is given in Fig. 8.2. There are three mechanisms in the model to accomplish access control. First, boundary access control to the middleware, as described in the next section, is achieved by controlling access of event clients to local event brokers with an OASIS policy. Services requested by event clients can either be granted, rejected, or *partially granted* after imposing restrictions. As explained in Sect. 8.1.4, the second mechanism assigns an event broker to a particular trust category that prescribes the types from the event type hierarchy that the event broker is permitted to handle. Finally, confidential event attributes in event notifications are encrypted, limiting access to those attributes. A single event publication can contain both public and private data. Content-based routing decisions on encrypted event attributes can only be carried out by event brokers that possess the necessary decryption key, otherwise events need to be flooded. Event attribute encryption will be introduced in Sect. 8.1.4.

Boundary Access Control

Local event brokers that host event clients are OASIS-aware and perform access control checks for every request made to them. This ensures that only authorized clients have access to the publish/subscribe system in compliance with access control policies. As shown in Fig. 8.2, local event brokers delegate the verification of credentials passed to them by event clients to an OASIS engine. Four types of OASIS policy restrict the actions of event clients. The event client that creates a new event type becomes its *event type owner* and is then responsible for specifying policy.

Connection Policy. This policy states the required credentials for an event client to be permitted hosting by a given event broker. A client can only use the publish/subscribe system if it maintains a connection with at least one local event broker.

Type Management Policy. The creation, modification, and removal of event types in the event type hierarchy is controlled by a type management policy. Usually, credentials certifying that an event client has the role of event type owner for an event type allow it to perform type management. This also avoids conflicts between clients from different applications.

Advertisement Policy. For every event type in the system, an advertisement policy specifies the roles an event publisher must acquire in order to advertise events of this type. This policy is generally specified by the event type owner.

Subscription Policy. Similarly, a subscription policy lists the necessary roles for an event subscriber to subscribe to events of that type. The policy may also prescribe the content-based filter expressions that are permitted and is again defined by the event type owner.

When an event client violates the connection or type management policies, the event broker rejects the operation invoked by the client. For advertisement and subscription policies, certain requests may be partially accepted by imposing a *restriction* on the original event advertisement or subscription. An advertisement restriction limits the advertisement by restricting the events that the event publisher is allowed to publish. Likewise, a subscription restriction transforms the client-requested subscription into a different, less powerful one. The client may or may not be notified by its local event broker that a restriction has been imposed for privacy reasons. The secure publish/subscribe model supports two flavors of restrictions.

Publish/Subscribe Restrictions

This kind of restriction takes the original submitted advertisement or subscription and replaces it by a different, more limited one, as defined by a coverage relation. In the case of an event advertisement, the event type in the advertisement is replaced by a less specific parent type from the event type hierarchy. For event subscriptions, the publish/subscribe restriction specifies an upper bound on the event type and content-based filtering expression that the event subscriber is allowed to submit. If the submitted subscription is covered by the subscription restriction, the subscription is accepted without change, otherwise it is automatically downgraded to the restricted subscription.

Generic Restrictions

A generic restriction is not expressible by the publish/subscribe system since it can include any predicate evaluations permitted by OASIS. Although the original advertisement or subscription submitted by the event client is passed on to the publish/subscribe system, all later events are restricted according to the arbitrary predicate function in the generic restriction. For example, a generic advertisement restriction may reject the publication of events with certain content, and a generic subscription restriction may perform additional filtering of events on the message size of the event notification, which otherwise could not be expressed in an event subscription.

The advantage of publish/subscribe restrictions is that they do not incur an overhead during the dissemination of events. Since the original advertisement or subscription is replaced by a more limited version, the event-based middleware implicitly enforces policy and no events need to be dropped at client-hosting brokers. The same is not true for generic restrictions because their additional expressiveness comes with the price of having to evaluate arbitrary predicates at client-hosting brokers to decide whether an event client can publish or be notified of a given event publication.

Event Broker Trust

The previous mechanism for boundary access control using restrictions assumes that all event brokers are equally trusted to process data, which is not true in practice. When an event broker joins the publish/subscribe system, it authenticates with its credentials and is then believed to participate correctly in the routing of events according to a content-based routing algorithm. It maintains encrypted network connections with its neighbouring event brokers in the overlay broker network. However, an event broker may not be trusted enough to gain access to data in particular event notifications or subscriptions. To make these trust relationships explicit, event brokers are associated with event types from the event type hierarchy that they are permitted to handle. This is illustrated in Fig. 8.2. Event broker B_3 is only permitted to process events of type `PoliceEvent`. Event brokers may be authorized to handle all event types that are more specific or more general than a given type, in other words are sub- or supertypes of an event type.

When routing event advertisements, subscriptions, and notifications in the overlay broker network, an event broker only passes on a message to the next event broker after obtaining proof in the form of a role membership certificate that this event broker is authorized to handle that particular event type. Otherwise, the event broker is forced to make a different routing decision. This can be done by acting as though the untrusted event broker has failed, relying on the fault tolerance properties of event routing in the publish/subscribe system that will ensure a different routing path. Note that event broker trust encompasses the handling of entire event types only, but we relax this restriction by using of event attribute encryption, as described in the next section.

Event Attribute Encryption

The mechanism for event broker trust from the previous section excludes brokers that are not trusted to handle specific event types from routing. As mentioned before, this coarse-grained approach effectively splits the overlay broker network into several trust domains, thus weakening the reliability and efficiency of event routing. A better solution is to prevent an untrusted event broker from accessing confidential data but still enabling it to perform content-based routing on other attributes. We achieve the goal of a single event that can hold private and public information by encrypting event attributes in event publications with different cryptographic keys. Although this introduces a larger runtime overhead due to cryptographic operations during event routing, this is justifiable as it leads to more expressive access control specifications where event types no longer have to be strictly divided into private and public categories. Another advantage of this scheme is that access control policy can also associate event clients, which have access privileges, to event attributes.

In addition to event types, event brokers are also trusted with a number of *key classes*. A key class is a collection of cryptographic keys for encrypting event attributes, that supports key rotation and revocation. Access control to individual event attributes is achieved by signing and encrypting them with a key from a given key class so that only trusted event brokers can decrypt these attributes. An event broker can only read or write an event attribute if it has a role membership that includes access to the appropriate key classes. This also means that an event client can only submit an event subscription or notification that refers to encrypted attributes to its local event broker if it can prove that it possesses credentials for the required key classes. The event broker then performs the cryptographic operations on the client's behalf.

To include event attribute encryption in a typed event model, the event type hierarchy is extended with a description of the key classes that are necessary to access the content of event attributes, as shown in Fig. 8.3. Each event attribute is annotated with its key classes in disjunctive normal form. A conjunction of key classes means that the attribute is encrypted with keys from several key classes in sequence. For example, the `isDrill` attribute in the `EmergencyEvent` type has to be either encrypted under the *police* and *training*, or under the *fire* and *training* key classes. This prevents anyone receiving emergency-related events in the Active City from finding out whether this is an exercise drill unless they are a training instructor with access to the *training* key class. Unencrypted event attributes are denoted with the empty

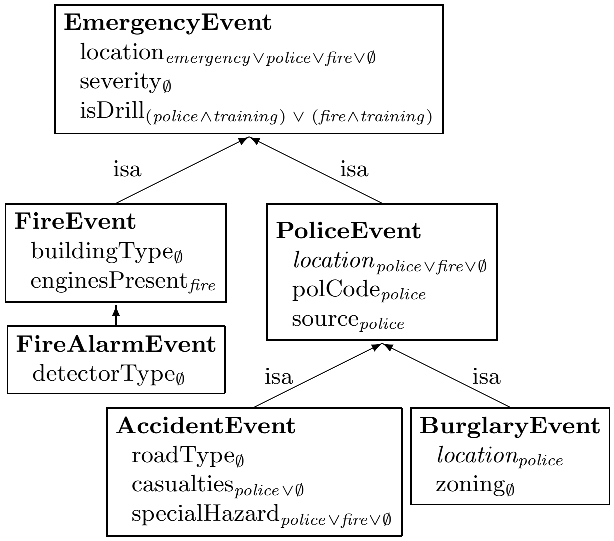

EmergencyEvent
location$_{emergency \vee police \vee fire \vee \emptyset}$
severity$_{\emptyset}$
isDrill$_{(police \wedge training) \vee (fire \wedge training)}$
isa
isa
FireEvent
buildingType$_{\emptyset}$
enginesPresent$_{fire}$
PoliceEvent
location$_{police \vee fire \vee \emptyset}$
polCode$_{police}$
source$_{police}$
FireAlarmEvent
detectorType$_{\emptyset}$
isa
isa
AccidentEvent
roadType$_{\emptyset}$
casualties$_{police \vee \emptyset}$
specialHazard$_{police \vee fire \vee \emptyset}$
BurglaryEvent
location$_{police}$
zoning$_{\emptyset}$

Fig. 8.3. An event type hierarchy with attribute encryption

key class $\emptyset$. In Fig. 8.2 event brokers are annotated with the key classes that they are permitted to use.

Note that the standard subtyping relation between event types must still hold so that a subtype is more specific than its parent type. As a result, key classes can only be removed from inherited event attributes but never added. This is illustrated with the `location` attribute whose access becomes more restrictive as new event types are derived.

Encrypted Attribute Coverage

When an event subscriber submits a content-based subscription for an event type with encrypted attributes, attribute predicates in the subscription must also be encrypted with appropriate key classes for the subscription to match events. The subscriber selects one or more key classes for the encryption of the attribute predicate from all the key classes for which it is authorized. As a consequence, the event model of the publish/subscribe system must be extended to support a coverage relation between event subscriptions and notifications, and among event subscriptions that use attribute encryption. Informally, an encrypted attribute predicate can only be matched by an encrypted event attribute in a notification if it was encrypted with the same key classes. When an attribute predicate should match attributes encrypted under several different key classes, it must be disjunctively encrypted multiple times using these key classes and several copies of the attribute predicate must be included in the subscription. For coverage among subscriptions, an attribute predicate encrypted under particular key classes is covered by another encrypted pred-

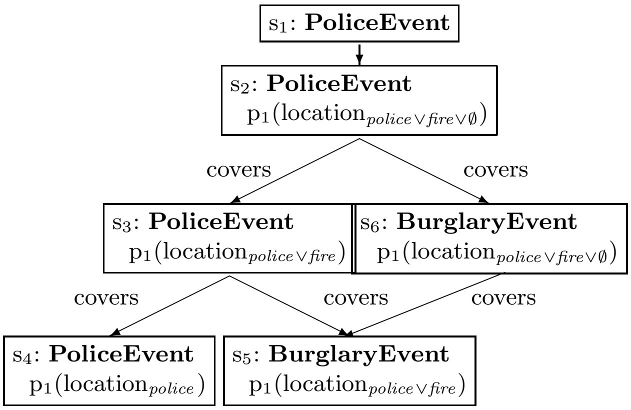

s1: PoliceEvent
s2: PoliceEvent p1(location$_{police \vee fire \vee \emptyset}$)
covers
covers
s3: PoliceEvent p1(location$_{police \vee fire}$)
s6: BurglaryEvent p1(location$_{police \vee fire \vee \emptyset}$)
covers
covers
covers
s4: PoliceEvent p1(location$_{police}$)
s5: BurglaryEvent p1(location$_{police \vee fire}$)

Fig. 8.4. Subscription coverage with attribute encryption

icate if the second predicate covers the first and is encrypted under at least the key classes of the first predicate.

Definition 8.1 (Encrypted Attribute Coverage). *An encrypted event attribute a^K is* covered *by (or* matches*) an encrypted attribute predicate p^L,*

$$a^K \sqsubseteq p^L,$$

iff

$$a \sqsubseteq p \wedge K \subseteq L$$

holds, where K is the set of key classes under which a is conjunctively encrypted and L is the set of a conjunction of key classes under which p is disjunctively encrypted. An encrypted attribute predicate $p_1^{L_1}$ is covered *by another encrypted attribute predicate $p_2^{L_2}$,*

$$p_1^{L_1} \sqsubseteq p_2^{L_2},$$

iff

$$\forall a.\ \ a \sqsubseteq p_1 \Rightarrow a \sqsubseteq p_2 \wedge L_1 \subseteq L_2,$$

holds, where a is an event attribute and L_1 and L_2 are sets of conjunctions of key classes with disjunctive encryption.

We illustrate this extended coverage relation in Fig. 8.4, which shows six example subscriptions with regard to the previous event type hierarchy. Subscription s_1 is the most generic because it does not include any attribute predicates. The attribute predicate in subscription s_3 does not match events with an unencrypted `location` attribute, and therefore s_3 is covered by s_2. Subscription s_4 is most specific because the attribute predicate is only encrypted under the *police* key class.

8.1.5 Further Reading

In this section we provide an overview of previous work in the area of security in publish/subscribe systems. Preliminary work on security issues under publish/subscribe semantics can be found in [390]. It identifies the necessity for ensuring the confidentiality of event publications and subscriptions and suggests accountability for billing purposes; however, no mechanisms are provided. In the work by Miklós [260], upper bound filters on advertisements and subscriptions in Siena are proposed, but the confidentiality of event publications within the publish/subscribe system is not guaranteed.

The *Narada Brokering* project includes a distributed security framework [294, 405] that uses access control lists to control event publishers and subscribers for a topic, limiting the scalability. Cryptographic keys for encrypting publications are centrally managed by a key management center (KMC). An event publisher can choose to use a central topic key from the KMC or the public keys of all event subscribers for encryption, which contradicts decoupled publish/subscribe semantics. Access control can only be provided at the granularity of whole events, and event brokers are implicitly trusted, rather than using different trust levels as supported by our security service.

A publish/subscribe system with scopes (Chap. 6) can be extended to include access control [145]. Scopes, which model visibility in a distributed publish/subscribe system, can be used to split the event system into different trust domains. In effect, this creates multiple distinct overlay networks of event brokers. Secure events only stay within a scope and are therefore never handled by untrusted event clients or brokers. Interactions between trust domains are precisely specified through scope interfaces. Scope interfaces express the access control policies for events crossing trust boundaries. Partitioned overlay networks can use untrusted brokers to create an encrypted tunnel for secure events. This work also proposes an implementation strategy based on aspect-oriented programming [222] to integrate access control with existing publish/subscribe implementations.

8.2 Fault Tolerance

The behavior of a system in the presence of faults is an important property of the system. In Sect. 2.5.2 we described the formal specification of a simple event system. An important goal for such a system was to guarantee *safety* and *liveness* conditions. Recall that a safety condition ensures that nothing bad will happen, whereas liveness stipulates that eventually something good will occur. In other words, Def. 2.5 requires that the system is correct, i.e., exhibits the desired functionality at its interface under all circumstances. To satisfy the specification, all faults occurring in the real world would have to be masked. However, masking all faults is costly if not impossible. Provided that a temporary failure of the system can be accepted, making a system self-stabilizing is an attractive alternative or supplement to fault masking. We will see that self-stabilization comes at cost, namely the weakening of safety conditions. In the following, we describe fault masking and self-stabilization in an event system with an emphasis on the latter.

8.2.1 Fault Masking

Fault masking requires redundancy either in *time* or in *space*. While *time redundancy* repeats actions (e.g., resending a message to cope with message loss), *space redundancy* uses independent copies of the resources that can be affected by faults (e.g., communication channels). Of course, both approaches can also be combined in a single system. However, research about applying fault masking to publish/subscribe systems is still in an early stage.

What are typical scenarios for fault masking in publish/subscribe systems? For example, assume that communication channels fail with in a fail-stop model. Then, we could connect each pair of neighbored brokers with two instead of one communication channel. If one of the communications fails, the brokers can still communicate using the communication channel that is still working. This way, failed communication channels can be masked by the system as long as for any pair of neighbored brokers only one of the two communication channels fails. To be able to mask broker and communication channel failures, we can use two independent broker topologies that do not share physical communication links or computers hosting brokers. In this case, we would have to modify our model such that a client can connect to two remote brokers. We also must take care that no duplicates are delivered and that—if required—the FIFO-producer or causal ordering of messages is ensured. If Byzantine faults can occur, fault masking is much more complicated than in the fail-stop model. This is due to the fact that in the Byzantine model failed links and brokers can behave arbitrarily.

Another possibility for implementing fault masking is to reconfigure the broker network in case of failures such that the failed resources are no longer used in the system. This approach is feasible but not trivial to implement if concurrent faults can occur. While the reconfiguration is in progress, notifications must be buffered at certain brokers. We must ensure that no notifications are lost or duplicated. Extra effort is needed to keep notification ordering guarantees such as publisher-based FIFO or causal ordering, if required [302].

8.2.2 Self-Stabilizing Publish/Subscribe Systems

An alternative (or sensible addition) to fault masking is *self-stabilization*, a concept introduced by Dijkstra [113] in 1974. He defined a system as being self-stabilizing if "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". In contrast to that, a system which is not self-stabilizing may stay in illegitimate states forever, leading to a permanent failure of the system. Self-stabilization models the ability of a system to recover from arbitrary transient faults within a finite time without any intervention from the outside. If the time between consecutive faults is long enough, the system will start to work correctly again. Transient faults include temporary network link failures resulting in message duplication, loss, corruption, or insertion, arbitrary sequences of process crashes and subsequent recoveries, and arbitrary perturbations of the data structures of any fraction of the processes. The program code running at the nodes and inputs from the outside, however, cannot be corrupted. Dolev [116] gives a comprehensive discussion of self-stabilization.

However, it is, in general, impossible under the fault assumption of self-stabilization to require *any* property that prohibits certain states, i.e., safety properties. For example, the system could deliver a notification n to a client X although X has no active subscription matching n because a fault corrupted the state of the system such that that it “thinks” that X subscribed to n. Therefore, we require that a self-stabilizing publish/subscribe system satisfies the safety property of Def. 2.5 only eventually. This ensures that the system starting from any state will eventually satisfy the actual safety property and continue to do so if no faults occur. The liveness property of Def. 2.5 can be left unchanged. This leads to the following definition of self-stabilizing publish/subscribe systems:

Definition 8.2. *A* self-stabilizing publish/subscribe system *is a publish/subscribe system satisfying the following requirements:*

1. *Eventual Safety Property: Starting from any state, the system eventually satisfies the safety property of Def. 2.5.*
2. *Liveness Property: Starting from any state, the system satisfies the liveness property of Def. 2.5.*

A formal version of this specification can be found in [263].

8.2.3 Self-Stabilizing Content-Based Routing

Under the fault assumption of self-stabilization, the routing configuration can arbitrarily be corrupted by transient faults. Therefore, the applied routing algorithm must ensure (a) that corrupted routing entries are corrected or deleted from the routing table and (b) that missing routing entries are inserted into the routing table.

We assume that each broker stores the information about its neighbors in its ROM. This ensures that this information cannot be corrupted. If it would be stored in RAM or on harddisk, it could also be corrupted by a fault. In this case, we would have to layer self-stabilizing content-based routing on top of a self-stabilizing spanning tree algorithm. Layered composition of self-stabilizing algorithms is a standard technique which is easy to realize when the individual layers have no cyclic state dependencies [116]. In this case, the stabilization time would be bounded by the sum of the stabilization times of the individual layers.

Basic Idea

The basic idea for making content-based routing self-stabilizing is that routing entries are only *leased.* To keep a routing entry, it must be renewed before the *leasing period* π has expired. If a routing entry is not renewed in time, it is removed from the routing table. Interestingly, this approach does not only allow the publish/subscribe system to recover from internal faults but also from certain external faults. For example, if a client crashes, its subscriptions are automatically removed after their leases have expired.

To support leasing of routing table entries, we use a *second chance* algorithm. Routing entries are extended by a *flag* that can only take the two values 1 and 0. Before a routing entry is (re)inserted into the routing table, all existing routing entries whose filter has the same *ID* (as the ID of the filter of the routing entry to be inserted) are removed from the routing table. This is necessary as the IDs of the routing entries can be corrupted, too. We assume that the clock of a broker can only take values between 0 and $\pi - 1$ to ensure that if the clock is corrupted, it can diverge from the correct clock value by at most π. When its clock overruns, a broker deletes all routing entries, whose flags have the value 0 from the routing table and sets the flags of all remaining routing entries to 0 thereafter (new subscriptions have their flags set to 1 initially). Hence, it must be ensured that an entry is renewed once in π to prevent its expiration. On the other hand, it is guaranteed that an entry which is not renewed will be removed from the routing table after at most 2π.

The renewal of routing entries originates at the clients. To maintain its subscriptions without interruption, a client must renew the lease for each of its subscriptions by "resubscribing" to the respective filter once in a *refresh period* ρ. Resubscribing to a filter is done in the same way as subscribing. In general, π must be chosen to be greater than ρ due to varying link delays. The *link delay* δ is the amount of time needed to forward a message over a communication link and to process this message at the receiving broker. In our model, it is considered a fault when δ is not in the range between $\delta_{\min}$ and $\delta_{\max}$. It is important to note that assuming an upper bound for the link delay is a necessary precondition for realizing self-stabilization.

Flooding

The naïve implementation of a self-stabilizing publish/subscribe system is *flooding*: When a broker receives a notification from a local client, the broker forwards the notification to all neighbor brokers. When it receives a notification from a neighbor broker, the notification is forwarded to all other neighbor brokers. Additionally, each processed notification is delivered to all local clients with a matching subscription. Flooding only requires a broker to keep state about the subscriptions of its local clients. Therefore, errors in this state can be corrected locally by forcing clients to renew their subscriptions once in a leasing period. This means that $\rho = \pi$. The main advantage of this scheme is that a coordination among neighboring brokers is not necessary. Hence, no additional network traffic is generated. Additionally, new subscriptions become active immediately. While a corrupted or erroneously inserted subscription survives at most 2π in a routing table and a missing subscription is reinserted after at most π, an erroneously inserted or corrupted notification disappears from the network after at most $d \cdot \delta_{\max}$, where d is the *network diameter*, i.e., the length of the longest path a message can take in the broker network. Hence, for flooding, the *stabilization time* Δ, i.e., the time it takes for the system to reach a legitimate state starting from an arbitrary state, equals $\max\{2\pi, d \cdot \delta_{\max}\}$.

Simple Routing

The solution for flooding can be extended to simple routing. *Simple routing* treats each subscription independently of other subscriptions. A (un)subscription is inserted into (removed from) the routing table and flooded into the broker network. If a broker receives a (un)subscription from a local client, it is forwarded to all neighbor brokers. If it was received from a neighbor broker, it is forwarded to all other neighbor brokers. Thus, simple routing is idempotent to resubscriptions, and a subscription is redistributed through the broker network when it is renewed by the client. Note that here subscriptions become active only gradually.

A critical issue is that the timing assumptions must allow the clients to renew their leases everywhere in the network before they expire. How large must π be with respect to ρ in this case? To answer this question, consider two brokers B and B' connected by the longest path a message can take in the broker network. This situation is illustrated in Fig. 8.5. Assume a local client X of B leases a routing table entry of B at time t_0 and renews this lease at time $t_1 = t_0 + \rho$. X's lease causes other leases to be granted all along the path to broker B'. Considering the best- and worst-cases of the link delay, the first lease reaches B' at time $a_0 = t_0 + d \cdot \delta_{\min}$ in the best case, and the lease renewal reaches B' at time $a_1 = t_1 + d \cdot \delta_{\max}$ in the worst-case. If X refreshes its leases after ρ time and if network delays are unfavorable, two lease renewals will arrive at B' within at most $a_1 - a_0$. Hence, $\pi > a_1 - a_0$ must hold to ensure that the entry is renewed in time. Thus, we get $\pi > \rho + d \cdot (\delta_{\max} - \delta_{\min})$.

The stabilization time Δ depends on the value of π. Since corrupted or erroneously inserted messages can contaminate the network, a delay of $d \cdot \delta_{\max}$ must be assumed before their processing is finished. After at most 2π, their effects will be removed everywhere. Overall, the stabilization time sums up to $\Delta = d \cdot (\delta_{\max} - \delta_{\min}) + 2\pi$. For example, assume that $d = 10$, $\delta_{\max} = 25\,\text{ms}$, and $\delta_{\min} = 5\,\text{ms}$. To guarantee a stabilization time of $\Delta = 30\,\text{s}$, $\pi = 14.9\,\text{s}$ and thus $\rho = 14.7\,\text{s}$ follows. There is a tradeoff between π and ρ. To have low

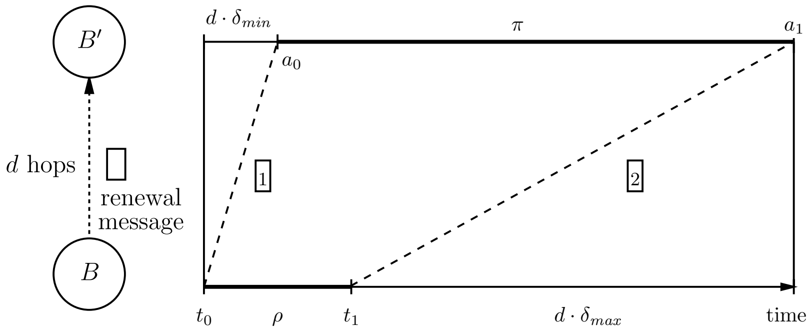


Fig. 8.5. Deriving the minimum leasing time

message overhead, ρ should be as large as possible. However, this implies a large value of π, but π should be as small as possible to facilitate fast recovery.

Advanced Routing Algorithms

The situation is more complicated if advanced content-based routing algorithms such as identity-based, covering-based, or merging-based routing are applied. Contrary to flooding and simple routing these algorithms are—at least the versions presented so far—not idempotent with respect to resubscriptions. However, they can be made idempotent with some minor changes. Note that the maximum stabilization time Δ is not affected by whether an advanced routing algorithm or simple routing is applied because in the worst-case a filter will nevertheless travel all along the longest path in the network.

Consider identity-based routing (for more details we refer to [263]). When a broker B processes a new or canceled subscription F from destination D, it counts the number d of destinations $D' \neq D$ for which a subscription matching the same set of notifications exists in T_B. Depending on the value of d, F is forwarded differently. If $d = 0$, F is forwarded to all neighbors if $D \in L_B$ and to all neighbors except D if $D \in N_B$. If $d = 1$ and $D' \in N_B$, F is forwarded only to D'. If $d = 1$ and $D' \in L_B$ or if $d \geq 2$, F is not forwarded at all. This scheme is not idempotent to resubscriptions because if $d \geq 2$ and one of the identical subscriptions is renewed at B, none of those subscriptions will be forwarded. This can be circumvented if B takes only those subscriptions into account when calculating d whose flag is 1. In this case, in each leasing period that subscription of the identical subscriptions which is renewed first after the broker has run the second chance algorithm is forwarded, ensuring correct forwarding.

Covering-based routing can also be made self-stabilizing. In this case, only routing entries with flag 1 are taken into account when looking for identical subscriptions. However, when looking for subscriptions that really cover a given subscription (i.e., match a real superset of notifications), additionally also those routing entries with flag 0 are considered. This is to avoid sending covered subscriptions unnecessarily to neighbors because they are refreshed before a covering subscription is refreshed. To make merging-based routing self-stabilizing, the refreshing of merged filters must additionally be ensured.

Discussion

The values of π and ρ depend on the delay of the links in the network. So far, we assumed that these values are fixed and equal for every broker in the system. In many scenarios, link delays vary a lot such that it could be advantageous to incorporate this property into the algorithm. We assume that the value of link delay stored at every adjacent broker cannot be corrupted (i.e., it is stored in ROM). The values of π and ρ have then to be calculated individually for every subscription, depending on where the publishers are. Additionally, π and ρ have to be refreshed the same way as described previously for subscriptions. Advertisements that are sent periodically by the publishers could be used for this purpose. Taking this approach, the broker algorithm can take advantage of faster links and stabilize subtrees of the broker topology faster if the links allow for this. The application of leasing is a common way to keep soft states. This technique is used in many protocols and algorithms such as the Routing Information Protocol (RIP, RFC2453) and Directed Diffusion [205].

Simulation

We carried out a discrete event simulation to compare self-stabilizing content-based routing to flooding with respect to their message complexity. Before we discuss the results, we describe the setup of the experiments.

Setup

We consider a broker hierarchy being a completely filled 3-ary tree with five levels. Hence, the hierarchy consists of 121 brokers of which 81 are leaf brokers. Since we use a tree for routing, this implies a total number of 120 communication links. We use hierarchical routing, but similar results can be obtained for peer-to-peer routing, too. With hierarchical routing, subscriptions are only propagated from the broker to which the subscribing client is connected toward the root broker. This suffices because every notification is routed through the root broker. Hence, control messages travel over at most four links. We use identity-based routing and consider 1000 different filter classes (e.g., stocks) to which clients can subscribe.

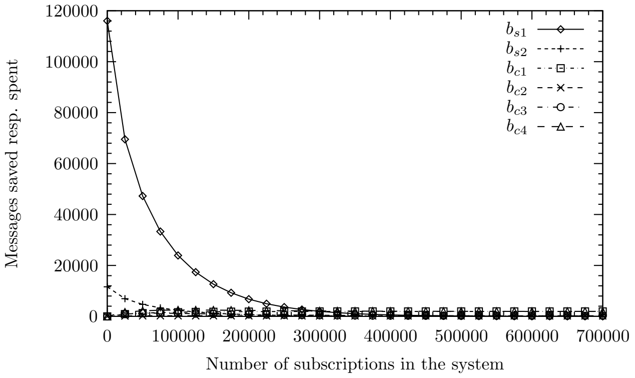
Subscribers only attach to leaf brokers. Results for scenarios where clients can attach to every broker in the hierarchy can be derived similarly. Instead of dealing with clients directly, we assume independent arrivals of new subscriptions with exponentially distributed interarrival times and an expected time of λ^{-1} between consecutive arrivals. When a new subscription arrives, it is assigned randomly to one of the leaf brokers and one of the filter classes is randomly chosen. The lifetime of individual subscriptions is exponentially distributed with an expected lifetime of μ^{-1}. Each notification is published at a randomly chosen leaf broker. Hence, notifications travel over at most eight links. The corresponding filter class is also chosen randomly. The interarrival times between consecutive publications are exponentially distributed with an expected delay of ω^{-1}. We assume a constant delay in the overlay network of $\delta = 25\,\mathrm{ms}$, including the communication and the processing delay caused by the receiving broker.

To illustrate the effects of changing the parameters, we considered two possible values for some of the system parameters: For each of the 1000 filter classes, a publication is expected every 1 s (10 s), i.e., $\omega_1 = 1000\,\mathrm{s}^{-1}$ ($\omega_2 = 100\,\mathrm{s}^{-1}$). The expected subscription lifetime is 600 s (60 s), i.e., $\mu_1 = (600\,\mathrm{s})^{-1}$ ($\mu_2 = (60\,\mathrm{s})^{-1}$). Each client refreshes its subscriptions once in 60 s (600 s), i.e., a refresh period of $\rho_1 = 60\,\mathrm{s}$ ($\rho_2 = 600\,\mathrm{s}$). Since $d = 8$ in our scenario, the leasing period is $\pi_1 = 60.2\,\mathrm{s}$ ($\pi_2 = 600.2\,\mathrm{s}$) for ρ_1 (ρ_2). Hence, a subscription will on average be refreshed 10 (100) times before it is canceled by the subscribing client if $\mu = (600\,\mathrm{s})^{-1}$. The resulting stabilization time is $\Delta_1 = 120.6\,\mathrm{s}$ ($\Delta_2 = 1200.6\,\mathrm{s}$).

We are interested in how the system behaves in equilibrium for different numbers of active subscriptions N. In equilibrium, $\mathrm{d}N/\mathrm{d}t = 0$ where $\mathrm{d}N/\mathrm{d}t = \lambda - \mu \cdot N(t)$, implying $N = \lambda/\mu$. Thus, if N and μ is given, λ can be determined. If the system was started with no active subscriptions, we would have to wait until the system approximately reached equilibrium before we begin the measurements. However, in our scenario it is possible to start the system right in the equilibrium. At time 0, we create N subscriptions. For each of these subscriptions, we determine how long it will live, for which filter class it is, and at which leaf broker it is allocated. Since we use an exponential distribution for the lifetime, this approach is feasible because the exponential distribution is memoryless.

Results

The results of our simulation are depicted in Fig. 8.6. Note that the right plot is a magnification of the most interesting part of the left plot. In Fig. 8.6, $b_{s1/2}$ is the notification bandwidth saved if filtering is applied instead of flooding. The figure shows b_{s1} and b_{s2}, which correspond to the publication rate ω_1 and ω_2, respectively. Because b_s linearly depends on ω, a decrease of ω by a factor of 10 leads to a use of one tenth as much notification bandwidth. If there are no subscriptions in the system, $b_{s1} = 116,000\,s^{-1}$ and $b_{s2} = 11,600\,s^{-1}$, respectively. These numbers are $4000\,s^{-1}$ and $400\,s^{-1}$ less than the overall number of notifications published per second. This is because with hierarchical routing, a notification is always propagated to the root broker. The control traffic b_c is caused by subscribing, refreshing, and unsubscribing clients. It only



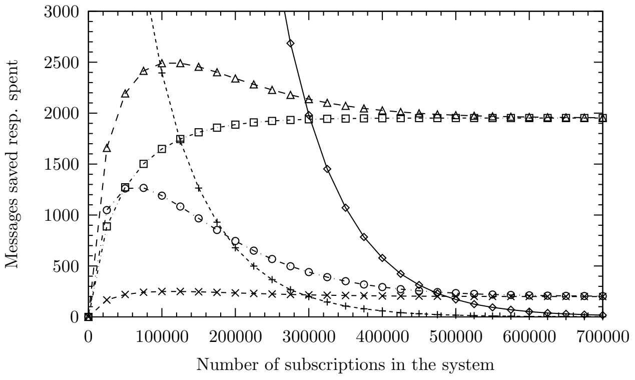


Fig. 8.6. Notification bandwidth saved by doing filtering instead of flooding ($b_{s1}: \omega_1 = 1000\,s^{-1}, b_{s2}: \omega_2 = 100\,s^{-1}$) and control traffic caused by filtering and leasing ($b_{c1}, b_{c4}: \rho_1 = 60\,s$, $b_{c2}, b_{c3}: \rho_2 = 600\,s$, $b_{c1}, b_{c2}: \mu_1 = (600\,s)^{-1}$, $b_{c3}, b_{c4}: \mu_2 = (60\,s)^{-1}$). The *lower figure* magnifies the most interesting part of the *upper figure*

arises if filtering is used. The figure shows b_{c1}, b_{c2}, b_{c3}, and b_{c4}, which result from the different combinations of μ and ρ. The value to which b_c converges for large numbers of subscriptions mainly depends on the refresh period ρ. Thus, b_{c1} and b_{c3} converge to $120,000/\rho_1 = 2000s^{-1}$, while b_{c2} and b_{c4} converge to $120,000/\rho_2 = 200s^{-1}$. The evolution of b_c for numbers of subscriptions in the range between 0 and $200,000$ is largely influenced by the value of μ. A small μ such as μ_2 leads to a hump (cf. b_{c3} and b_{c4} in Fig. 8.6). Filtering saves bandwidth compared to flooding if b_s exceeds b_c. The points where the curve of the respective variants of b_s and b_c intersect are important: If the number of subscriptions is smaller than at the intersection point, filtering is superior, while for larger numbers flooding is better. For example, the curves of b_{s1} and b_{c1} intersect for about $300,000$ subscriptions. Thus, filtering is superior for less than $300,000$ subscriptions, while flooding is superior for more than $300,000$

subscriptions. Since we consider eight scenarios, we have eight intersection points in Fig. 8.6.

The results gained through the simulation show that applying self-stabilizing filtering makes sense if the average number of subscriptions in the system does not grow beyond a certain point. However, it is important to note that all assumptions taken for the simulation depict worst-case scenarios. For example, the equal distribution of subscriptions to leaf brokers is disadvantageous for filtering. If there was locality in the interests of the clients, filtering would always save a portion of the notification traffic, regardless of how large the number of subscriptions grows [263], and the control traffic would also be smaller. In such scenarios, filtering can be superior to flooding for all numbers of subscriptions. Recently, Jaeger and Mühl [212] have published analytical results that come to the same results as those presented here.

8.2.4 Generic Self-Stabilization Through Periodic Rebuild

In a self-stabilizing system, arbitrary transient faults can occur. The only parts that cannot be corrupted are the program code and the data stored in ROM. In general, we cannot reason about how a routing algorithm (that works correctly in a fault-free system) behaves when it receives corrupted messages or when it is applied to perturbed routing tables. What can merely be assumed is that it will eventually work correctly again when it is restarted from a legitimate initial routing configuration.

In this section, we present a generic wrapper algorithm $\mathcal{A}$ that makes a publish/subscribe system self-stabilizing, regardless of which correct routing algorithm $\mathcal{R}$ it wraps. The only assumptions are that (a) $\mathcal{R}$ has no private state but draws its decision solely on the basis of the respective routing table, that (b) $\mathcal{R}$ terminates after finite time when called, and that (c) each client refreshes its subscriptions once in a refresh period ρ. The wrapper algorithm periodically rebuilds the routing tables starting from an initial routing configuration that is stored in the ROM of each broker. Note that most routing algorithms use an empty initial routing configuration [263]. Our algorithm can be seen as a periodic precautionary distributed reset [16].

Basic Idea

Each broker B maintains two routing tables T_B^0 and T_B^1, which are alternately rebuilt on a periodic basis, and a flag $a_B \in \{0, 1\}$ that determines which of both routing tables is currently rebuilt.[1] However, notification routing always uses both routing tables to determine the target destinations of a notification. A notification is forwarded to a destination if it matches a routing entry for

[1] An optimized solution can be implemented with only one table and two flags for every entry indicating to which routing table(s) the entry belongs.

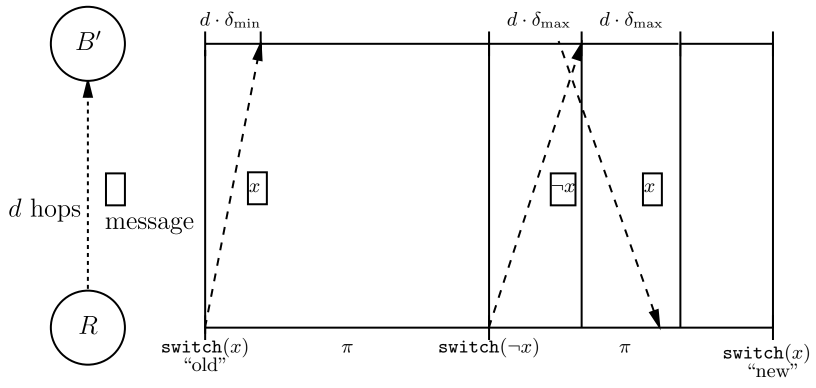


Fig. 8.7. Choosing π such that "old" and "new" update messages do not interleave

this destination in any of the two routing tables. If the routing tables are in a correct state, this does no harm.

Since $\mathcal{A}$ wraps $\mathcal{R}$, every call to $\mathcal{R}$ is intercepted by $\mathcal{A}$. This way, $\mathcal{A}$ determines which routing table the next call of $\mathcal{R}$ operates on in the following way: For every (un)subscription from a local client of B, $T_B^{a_B}$ will be used. If update messages are generated by $\mathcal{R}$ in reaction to the (un)subscription, they will be tagged with a_B. Accordingly, when a broker B' receives an update message tagged with x from a neighbor broker, then $T_{B'}^{x}$ will be used by $\mathcal{R}$ for this call.

The periodical rebuild is triggered by a modulo clock on the root broker R every π. The rebuild sets $a_R \leftarrow \neg a_R$. Then, it initializes $T_R^{a_R}$ with the initial routing configuration stored in ROM and propagates a `switch`(a_R) message to all of its neighbors. Similarly, when a broker B' receives a `switch`(x) message from a neighbor, it sets $a_{B'} \leftarrow x$, initializes $T_{B'}^{a_{B'}}$, and forwards a `switch`(x) message to all other neighbors. If a (un)subscription is issued twice by a client between two consecutive `switch` messages without an intervening unsubscription (subscription), this could raise a problem because $\mathcal{R}$ might not tolerate resubscriptions. To avoid this potential problem, a (un)subscription from a local client will be discarded by the wrapper algorithm $\mathcal{A}$ if it is redundant with respect to the contents of the currently active routing table.

Correctness

Before we show the correctness of our scheme, we prove a preparatory lemma.

Lemma 8.1. *In a correct system, if $\pi > 2d \cdot \delta_{\max}$, no "old" update messages tagged with x can arrive at any broker after the root broker issued the next "new"* `switch`(x) *message.*

Proof. Old update messages tagged with x disappear at most $d \cdot \delta_{\max}$ after the last broker has received the `switch`($\neg x$) message. This means that at most

$2d \cdot \delta_{\max}$ after the root broker has sent the `switch`($\neg x$) message no old update messages tagged with x can arrive. Since π is greater than this value, only new update messages tagged with x can arrive at any broker after the next new `switch`(x) message is issued by the root broker (Fig. 8.7). □

Theorem 8.1. *When the wrapper algorithm is applied and* $\pi \geq \rho + 2 \cdot d \cdot \delta_{\max}$ *holds, the publish/subscribe system is self-stabilizing and the stabilization time* Δ *is bounded by* $2 \cdot \pi + d \cdot \delta_{\max}$.

Proof. For the correctness, we have to show that (a) the system stays in a correct state if it currently is in a correct state and that (b) the system will eventually enter a correct state if it is currently in an incorrect state.

(a) For the system to stay in a correct state, we have to ensure that (a1) at each broker the rebuild process of the routing table which is currently rebuilt is completed before the next `switch` message is received, that (a2) the rebuild is based only on new update messages, and that (a3) all new updates messages are received after the respective `switch` message.

(a1) This means that at each broker the time between two consecutive `switch` messages must be large enough to ensure that all necessary update messages are received in time. The time difference at which two brokers receive the same `switch` message cannot be greater than $d \cdot \delta_{\max}$. At all brokers, the clients need at most ρ to reissue all their subscriptions after the broker has received the `switch` message. The resulting update messages need at most $d \cdot \delta_{\max}$ to travel through the broker network. Therefore, $\pi \geq \rho + 2 \cdot d \cdot \delta_{\max}$ must hold to guarantee that at each broker the rebuild is complete before the next `switch` message is received.

(a2) By Lemma 8.1 and the fact that $\pi \geq \rho + 2 \cdot d \cdot \delta_{\max}$.

(a3) Due to the FIFO property of the communication channels and the fact that the topology is acyclic, a broker B' can only receive update messages and (un)subscriptions of local clients tagged with x after B' received the corresponding `switch`(x) message.

(b) Starting from an arbitrary state, every broker receives the next `switch` message after at most $\pi + d \cdot \delta_{\max}$. This message causes the receiving broker to reinitialize one of its two routing tables. As a result of (a) it is guaranteed that this routing table will be completely rebuilt before the subsequent `switch` message is received. This second `switch` message is received by all brokers at most $2\pi + d \cdot \delta_{\max}$ from the beginning. It causes the other routing table to be reinitialized. After all brokers have received and processed the second `switch` message, the system is guaranteed to be in a correct state again. This is because at all brokers the one routing table is completely rebuilt, while the other is reinitialized. Therefore, the stabilization time Δ is $2 \cdot \pi + d \cdot \delta_{\max}$ in the worst-case (Fig. 8.8). □

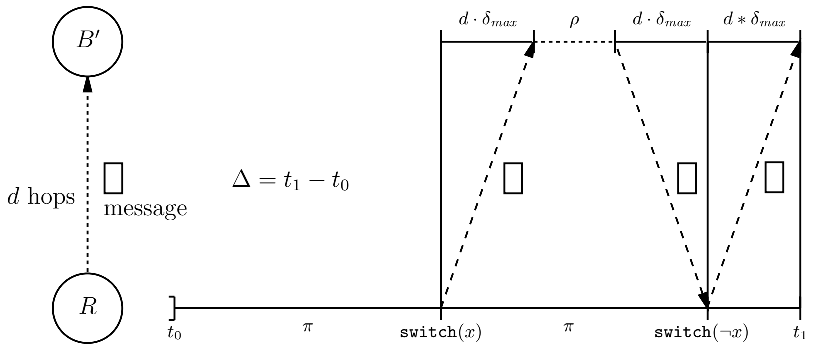


Fig. 8.8. Derivation of the maximum stabilization time

8.2.5 Further Reading

Many self-stabilizing algorithms have been proposed for various kinds of scenarios, while there are only a few contributions that cover publish/subscribe systems. Recently, Shen and Tirthapura [343] presented an alternative approach for self-stabilizing content-based routing. In their approach, all pairs of neighboring brokers periodically exchange sketches of those parts of their routing tables concerning their other neighbors to detect corruption. The sketches that are exchanged are lossy because they are based on bloom filters (which are a generalization of hash functions). However, because of the information loss, it is not guaranteed that an existing corruption is detected deterministically. Hence, the algorithm is not self-stabilizing in the usual sense. Moreover, although generally all data structures can be corrupted arbitrarily, the authors' algorithm computes the bloom filters incrementally. Thus, once a bloom filter is corrupted, it may never be corrected. Furthermore, in their algorithm, clients do not renew their subscriptions. Without this, corrupted routing entries regarding local clients are never corrected. Finally, their algorithm is restricted to simple routing in its current form.

8.3 Congestion Control

Many existing research prototypes of publish/subscribe systems make the assumption that event publication messages are negligible in size and therefore cannot saturate the available network bandwidth or processing power. However, this is not true in practice, and a publish/subscribe system can suffer from congestion leading to a degradation of service to clients. In this section we discuss the connection problem in the context of a publish/subscribe systems. We describe a scalable congestion control mechanism [313] that prevents the occurrence of congestion in a reliable publish/subscribe system. It consists of two algorithms, PDCC and SDCC, that are used in combination to address different aspects of congestion control in the event-based middleware. To motivate the need for congestion control in an event-based middleware, we begin with an overview of the congestion control problem and the requirements for a mechanism to handle congestion. The main part of this section is the description of the two congestion control algorithms as an example of how to perform congestion control in a publish/subscribe system.

8.3.1 The Congestion Problem

We argue that it is necessary to provide congestion control for overlay networks, such as the one established by a distributed publish/subscribe system. *Congestion* occurs when there are not enough resources to sustain the rate at which event publishers send publication messages in an event-based middleware. We distinguish between two kinds of congestion,

1. *network congestion*, where the network bandwidth between event brokers is the limiting resource
2. *event broker congestion*, when the processing of messages at an event broker cannot cope with the data rate

Both kinds of congestion may lead to the loss of messages at event brokers. Message loss is especially undesirable under guaranteed delivery semantics because the resulting retransmission of messages worsens the level of congestion in the system. An event-based middleware suffers from *congestion collapse* when the message loss dominates its operation and prevents event clients from receiving any useful service.

Usually there are two reasons for congestion in an event-based middleware. In many cases, congestion is caused by the underprovisioning of the deployed middleware in terms of network bandwidth or processing power of event brokers so that the middleware cannot handle resource requirements of event dissemination during normal or peak operation. A second, more subtle cause for congestion is the temporary need for more resources as a result of recovery after a failure under guaranteed delivery semantics.

Note that even though connections between event brokers use TCP congestion control, this is not sufficient to prevent congestion in the overlay broker network because of application-level queuing at event brokers. Both network and event broker congestion manifest themselves as the buildup of buffer queues at event brokers. To deal with congestion, current middleware deployments are often vastly overprovisioned, which is a waste of resources. Instead, a congestion control mechanism can address this problem directly.

8.3.2 Requirements

A congestion control mechanism in a publish/subscribe context differs from traditional congestion control found in other networking systems. This is due to the many-to-many communication semantics supported by the publish/ subscribe model and the content-based filtering of messages at application-level event brokers during event dissemination. Not all event subscribers receive the same set of publication messages sent by event publishers, as opposed to the case in application-level multicast, for example. Reliable event dissemination semantics leads to the selective retransmission of publication messages to a subset of recovering event subscribers, which further complicates congestion control. To guide the design of our congestion control mechanism, we formulate six requirements for congestion control in an event-based middleware:

Burstiness. The processing of publication messages at event brokers is bursty because of application-level scheduling and the variable processing cost of content-based filtering of event publications. This means that a congestion condition can arise quickly, requiring early detection by the congestion control mechanism.

Queue Sizes. Due to the burstiness of event routing and the need to cache event streams for retransmission, buffer sizes at event brokers are much higher compared to standard networking components. Buffer overflow only occurs when significant congestion already exists in the system. As a consequence, message loss cannot be used as an indicator for congestion in event-based middleware.

Recovery Control. The congestion control mechanism must ensure that event brokers that are recovering event publications that were previously lost will eventually complete recovery successfully. At the same time, recovering event brokers must be prevented from contributing to congestion. Although negative acknowledgment (NACK) messages are small and themselves cause little congestion, they potentially trigger the retransmission of large event publication messages.

Robustness. It is important that the congestion control mechanism is robust and can protect itself against malicious event clients. A possible design choice is to provide congestion control in the overlay broker network only, ensuring that the publication rate of messages by publisher-hosting brokers can be supported by all interested subscriber-hosting brokers. Flow control between client-hosting brokers and event clients is handled by a separate mechanism that can disconnect malicious clients.

Architecture Independence. The congestion control mechanism should not be tightly coupled to internal implementation details of an event broker. Instead, as a higher-level middleware service, it should support the evolution of the event broker implementation. For example, the detection of congestion should not depend on a particular buffer implementation or queuing discipline used by event brokers.

Fairness. When congestion requires the reduction of publication rates, fair throttling of event publishers must be ensured. The available resources at publisher-hosting brokers should be split equally among all hosted event publishers.

8.3.3 Congestion Control Algorithms

Typically a congestion control mechanism first detects congestion in the system and then adapts system parameters to remove its cause. In this section we describe two such algorithms that provide congestion control for a publish/subscribe system in accordance with the requirements stated in the previous section. The algorithms involve publisher-hosting brokers (PHB) and subscriber-hosting brokers (SHB).

1. A *PHB-driven congestion control algorithm* ensures that publisher-hosting brokers cannot cause congestion because of too high a publication rate. This is achieved by a feedback loop between publishers and subscribers to monitor congestion in the overlay broker network and control the event publication rate at the publishers.
2. An *SHB-driven congestion control algorithm* manages the recovery of subscribers after failure. It limits the rate of NACK messages that cause the retransmission of event publications from publisher-hosting brokers depending on congestion.

These two congestion control algorithms are independent of each other but should be used in conjunction to prevent congestion during both regular operation and recovery. Both algorithms need to distinguish between recovering and nonrecovering event brokers in order to ensure that subscribers can recover successfully. For a simpler presentation of the algorithms, we assume that only event brokers are internal nodes in event dissemination trees with client-hosting brokers constituting the root or leaf nodes. Next we will describe the two algorithms in turn.

PHB-Driven Congestion Control

The *PHB-driven congestion control algorithm* (PDCC) controls the rate at which new publication messages are published by a *publication endpoint* (pubend), such as a set of event publishers. The publication rate is adjusted depending on a*congestion metric*. We use the observed rate of publication messages at subscriber-hosting brokers as our congestion metric, which is similar to the throughput-based metric of TCP Vegas [50]. The rationale behind this is that a decrease in the message rate at a subscriber-hosting broker with an unchanged publication rate at the pubend is an indication of more queuing in the overlay broker network. This queue buildup is considered to be caused by network or event broker congestion in the system. Subscriber-hosting brokers calculate their own congestion metric and notify the publishers upstream whenever they believe that they are suffering from congestion. Congestion indications are aggregated at intermediate brokers so that the pubend is only informed of the worst congestion point. Two types of control messages are used to exchange congestion information between event brokers in an aggregated fashion.

Downstream Congestion Query (DCQ) Messages

The PDCC mechanism is triggered by DCQ messages sent by a publisher-hosting broker down the event dissemination tree to all subscriber-hosting brokers. Since congestion control is performed per tree, a DCQ message carries a tree identifier (`treeID`). A monotonically increasing `sequenceNo` is used for aggregation and the `mPos` field stores the current position in the event stream, which is, for example, the latest assigned event timestamp.

pubendID	sequenceNo	mPubend

Upstream Congestion Alert (UCA) Messages

UCA messages are sent by subscriber-hosting brokers to inform about congestion. They flow upwards in the event dissemination tree and are aggregated at intermediate brokers so that a publisher-hosting broker only receives a single UCA message in response to a DCQ message. Apart from the tree identifier and the sequence number of the triggering DCQ message, a UCA message contains the minimum throughput rates observed at recovering (`minRecSHBRate`) and nonrecovering (`minNonRecSHBRate`) subscriber-hosting brokers.

pubendID	sequenceNo	minRecSHBRate	minNonRecSHBRate

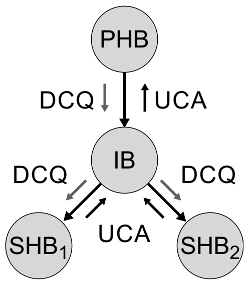


Fig. 8.9. Flow of DCQ and UCA messages

Figure 8.9 summarizes the propagation of DCQ and UCA messages through an overlay broker network in the PHB-driven congestion control algorithm. For the PDCC scheme to be efficient, DCQ and UCA messages must not suffer from congestion and should maintain low delays and loss rates. Next we describe the behavior of the three types of event brokers when processing DCQ and UCA messages in the PDCC algorithm.

Publisher-Hosting Broker (PHB)

A publisher-hosting broker triggers the PDCC mechanism by periodically sending DCQ messages with an incremented sequence number. The interval t_{dcq} at which DCQ messages are dispatched determines the time between UCA responses in a congested system. The higher the rate of responses, the quicker the system adapts to congestion.

When the PHB has not received any UCA messages for a period of time t_{nouca}, it assumes that the system is currently not congested. Therefore, it increases the publication rate if the rate is throttled and the pubend could publish at a higher rate. To increase the publication rate, we use a hybrid scheme with additive and multiplicative increase. The new rate r_{new} is calculated from the old rate r_{old} according to

$$r_{\text{new}} = \max\ \left[\ r_{\text{old}} + r_{\text{min}},\ r_{\text{old}} + f_{\text{incr}} \cdot (r_{\text{old}} - r_{\text{decr}})\ \right], \tag{8.1}$$

where r_{decr} is the publication rate after the last decrease, f_{incr} is a multiplicative increment factor, and r_{min} is the minimum possible increase. The multiplicative use of f_{incr} allows the publication rate to grow faster than a fixed additive increase. However, when the publication rate is already close to the optimal operation point before congestion occurs, it is necessary to limit the increase. This is done by recording the publication rate r_{decr} at which the increase started and using it to restrict the multiplicative increase. This scheme results in the publication rate probing whether the congestion condition has disappeared and, if not, oscillating around the optimal operation point.

When the PHB receives a UCA message, a decision is made about a reduction of the current publication rate. The rate is kept constant if the sequence number in the received UCA message is smaller than the sequence number of the DCQ message that was sent after the last decrease. The reason for this is that the system did not have enough time to adapt to the last change in rate and therefore more time should pass before another adjustment. The rate is also not reduced if the congestion metric in the UCA message is larger than the value in the previous message. This means that the congestion situation in the system is improving, and further reduction of the rate is unnecessary. Otherwise, the publication rate is decreased according to

$$r_{\text{new}} = \max\ \left[\ f_{\text{decr}_1} \cdot r_{\text{old}},\ r_{\text{decr}} + f_{\text{decr}_2} \cdot (r_{\text{old}} - r_{\text{decr}})\ \right] \quad \text{iff} \quad r_{\text{decr}} \neq r_{\text{old}} \tag{8.2}$$

$$r_{\text{new}} = f_{\text{decr}_1} \cdot r_{\text{old}} \quad \text{otherwise}, \tag{8.3}$$

where f_{decr_1} and f_{decr_2} are multiplicative decrement factors. The first term in Eq. (8.2) multiplicatively decreases the rate by a factor f_{decr_1}, whereas the second term reduces the rate relative to the previous decrement r_{decr}. Similar to Eq. (8.1), the second term prevents an aggressive rate reduction when congestion is encountered for the first time after an increase. Since the PDCC mechanism constantly attempts to increase the publication rate in order to achieve a higher throughput, it will eventually cause SHBs to send UCA messages if there is resource shortage in the system, but this should not result in a strong reduction of the publication rate. Taking the maximum of the two decrement values ensures that the publication rate stays close to the optimal operating point. If the congestion situation does not improve after one reduction, the publication rate is reduced again. This time a strong multiplicative decrease according to Eq. (8.3) is performed because the condition $r_{\text{decr}} = r_{\text{old}}$ holds.

Intermediate Broker (IB)

To avoid the problem of feedback implosion [100], aggregation logic for UCA messages at intermediate brokers (IB)must consolidate multiple messages from different SHBs such that the minimum observed rate at any SHB is passed upstream in a UCA message. This enables the publisher-hosting broker to adjust its publication rate to provide for the most congested SHB in the system. Another requirement is that UCA messages that occur for the first time are immediately sent upstream, allowing the publisher-hosting broker to respond as quickly as possible to new congestion in the system.

In Fig. 8.10 the algorithm for processing DCQ and UCA messages at an intermediate broker is given. An IB stores the maximum sequence number `seqNo` and the minimum throughput values for nonrecovering (`minNonRecSHBRate`) and recovering (`minRecSHBRate`) SHBs from the UCA messages that it has processed. After the initialization of these variables (line 1), the function `processDCQ` handles DCQ messages by relaying them down the event dissemination tree in line 6. When a UCA message arrives, the function `processUCAMsg` is called, which first updates the throughput minima (lines 10–11). A new UCA message is only sent upstream if the sequence number of the received message is greater than the maximum sequence number stored at the IB (line 12). This ensures that UCA messages with the same sequence number coming from different SHBs are aggregated before propagation. The first UCA message with a new sequence number immediately triggers a UCA message so that the pubend is quickly informed about new congestion. Subsequent UCA messages from other SHBs that have the same sequence number will be aggregated and contribute toward the throughput minima in the next UCA message. After a UCA message has been sent in line 13, `seqNo` is updated (line 14) and both throughput minima are reset in line 15.

```
initialization:
 seqNo ← 0
 minNonRecSHBRate ← ∞
 minRecSHBRate ← ∞

processDCQ(dcqMsg):
 sendDownstream(dcqMsg)

processUCA(ucaMsg):
 minNonRecSHBRate ←
   MIN(minNonRecSHBRate, ucaMsg.minNonRecSHBRate)
 minRecSHBRate ←
   MIN(minRecSHBRate, ucaMsg.minRecSHBRate)
 IF ucaMsg.seqNo > seqNo THEN
  sendUpstream(ucaMsg.seqNo, minNonRecSHBRate,
      minRecSHBRate)
  seqNo ← ucaMsg.seqNo
  minNonRecSHBRate ← ∞
  minRecSHBRate ← ∞
```

Fig. 8.10. Processing of DCQ and UCA messages at IBs

The example in Fig. 8.11 demonstrates the operation of the aggregation logic at IBs. The topology of six event brokers has two congested event brokers, SHB_1 and SHB_2, and three intermediate brokers $IB_{1,2,3}$ that aggregate UCA messages. Congestion in the system is first detected by SHB_1, and its UCA message with a congestion metric of 0.8 is directly propagated to the PHB. When SHB_2 notices congestion, its UCA message is consolidated at IB_2, which updates its throughput minimum to 0.4. Eventually a UCA message with the congestion value of SHB_2 will propagate up the event dissemination tree in response to a new DCQ message because SHB_2 is more congested than SHB_1.

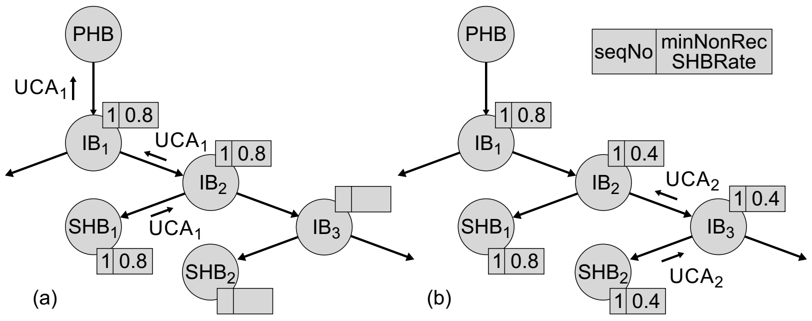


Fig. 8.11. Consolidation of UCA messages at IBs

Subscriber-Hosting Broker (SHB)

The congestion metric used by subscriber-hosting brokers depends on their observed throughput of publication messages and is independent of the actual publication rate of the pubend. An SHB monitors the ratio of PHB and SHB message rate,

$$t = \frac{r_{\text{pubend}}}{r_{\text{SHB}}}, \tag{8.4}$$

and uses this to decide when to send UCA messages with congestion alerts. To allow for burstiness in the throughput due to application-level routing as mentioned previously, t is passed through a standard first-order low-pass filter,

$$\bar{t} = (1 - \alpha)\,\bar{t} + \alpha\, t, \tag{8.5}$$

to obtain a smoothed congestion metric $\bar{t}$ with an empirical value of $\alpha = 0.1$. An SHB has to apply a different strategy for sending UCA messages depending on whether it is recovering event publications or not. We assume that an SHB can determine whether it is a recovering or a nonrecovering event broker. A suitable criterion to detect recovery would be, for example, that the SHB is ignoring new event publications because its event stream is saturated with old events caused by NACK messages.

Nonrecovering SHB. A nonrecovering SHB should receive publication messages at the same rate at which they are sent by the pubend. Therefore, if the smoothed throughput ratio $\bar{t}$ drops below unity by a threshold Δt_{nonrec},

$$\bar{t} < 1 - \Delta t_{\text{nonrec}}, \tag{8.6}$$

the SHB assumes that it has started falling behind in the event stream because of congestion. In rare cases, an SHB could be falling behind slowly because $\bar{t}$ stays below 1 but above $1 - \Delta t_{\text{nonrec}}$ for a long time. Unless there is already significant congestion in the system, this will not cause a queue overflow if buffer sizes are large. An SHB can detect this situation by periodically comparing its current position in its event stream m_{SHB} to the event stream position m_{tree} from the last received DCQ message. If the difference is larger than Δt_s,

$$m_{\text{SHB}} < m_{\text{tree}} + \Delta t_s, \tag{8.7}$$

a UCA message is triggered, even though the congestion metric $\bar{t}$ is above its threshold value.

Recovering SHB. A recovering SHB must receive publication messages at a higher rate than the publication rate, or it will never manage to successfully catch up and recover all lost publication messages. In some applications there is an additional requirement to maintain a minimum recovery rate $1 + \Delta t_{\text{rec}}$

in order to put a bound on recovery time. Thus, a recovering SHB sends a UCA message if

$$\bar{t} < 1 + \Delta t_{\mathrm{rec}}. \tag{8.8}$$

The threshold value Δt_{rec} influences how much of the congested resource will be used for recovery messages as opposed to new publication messages and hence controls the duration of recovery.

SHB-Driven Congestion Control

The *SHB-driven congestion control algorithm (SDCC)* manages the rate at which an SHB requests missed event publications by sending NACK messages upstream to the corresponding PHB. An SHB maintains a*NACK window* to decide which parts of the event stream to request. To control the rate of NACK messages being sent, the NACK window is open and closed additively by the SDCC algorithm depending on the level of congestion in the system. As for the PDCC mechanism, the change in recovery rate throughput is used as a metric for detecting congestion.

At the start of recovery, an SHB uses a small initial NACK window size $nwnd_0$. The NACK window is adjusted during recovery when the recovery rate r_{SHB} changes. The recovery rate r_{SHB} is defined as the ratio between the current NACK window size *nwnd* and the estimate of the round trip time *RTT*, which it takes to retrieve a lost event publication from the pubend,

$$r_{\mathrm{SHB}} = \frac{nwnd}{RTT}. \tag{8.9}$$

The NACK window size is changed in a similar fashion to TCP Vegas. When the recovery rate r_{SHB} increases by at least a factor α_{nack}, the NACK window is opened by one additional NACK message per round trip time. When r_{SHB} decreases by at least a factor β_{nack}, the NACK window is reduced by one NACK message,

$$nwnd_{\mathrm{new}} = nwnd_{\mathrm{old}} \pm size_{\mathrm{nack}}. \tag{8.10}$$

This is sufficient to ensure that resent event publications triggered by NACK messages from recovering event brokers do not overload the publish/subscribe system.

8.3.4 Further Reading

A large body of work exists in the area of congestion control in networks, although these solutions do not address the special requirements for congestion control in an publish/subscribe system. In this section we provide a brief overview of applicable work, contrasting it with our approach for congestion control.

Transmission Control Protocol (TCP)

The TCP protocol comes with a point-to-point, end-to-end congestion control algorithm with a congestion window that uses*additive increase, multiplicative decrease (AIMD)* [211]. *Slow start* helps open the congestion window more quickly. Packet loss is the only indicator for congestion in the system, and *fast retransmit* enables the receiver to signal packet loss by ACK repetition to avoid timeouts. TCP Vegas [50] attempts to detect congestion before packet loss occurs by using a throughput-based congestion metric, which is similar to the congestion metric used in the PDCC and SDCC algorithms.

Reliable Multicast

Reliable multicast protocols are similar to reliable publish/subscribe systems due to their one-to-many communication semantics, but typically they have no filtering at intermediate nodes and do not guarantee that all leaves in the multicast tree will eventually catch up with the sender. In general, multicast congestion control schemes can be divided into two categories [407], namely:

1. *sender-based*schemes, in which all receivers support the same message rate
2. *receiver-based*schemes with different message rates by means of transcoded versions of data

Since we can make few assumptions about the content of event publications, a receiver-based approach is not feasible. Congestion control for multicast is often implemented at the transport level relying on router support. It must adhere to existing standards to ensure fairness and compatibility with TCP [149, 179]. Since there are many receivers in the multicast tree, scalable feedback processing of congestion information is important. Unlike *feedback suppression* [107], our approach does not discard information because it consolidates feedback in a scalable way.

The *PGMCC* congestion control protocol [328] forms a feedback loop between the sender and the most congested receiver. The sender chooses this receiver depending on receiver reports in NACK messages. The congestion control protocol for *SRM* [344] is similar except that the feedback agent can give positive and negative feedback, and a receiver locally decides whether to send a congestion notification upstream to compete for becoming the new feedback agent. An approach that does not rely on network support, except minimal congestion feedback in NACK messages, is *LE-SBCC* [376]. Here a cascaded filter model transforms the NACK messages from the multicast tree to appear like unicast NACKs before feeding them into an AIMD module. However, no consolidation of NACK messages can be performed. All these schemes have in common that they use a loss-based congestion metric, which is not a good indicator for congestion in an application-level overlay network.

Multicast Available Bit Rate (ABR) ATM

The ATM Forum Traffic Management Specification [334] includes an available bit rate (ABR) category for traffic though an ATM network. At connection setup, *forward and backward resource management* (FRM/BRM) cells are exchanged between the sender and receiver to create a resource reservation, which is modified at intermediate ATM switches. All involved parties agree on an acceptable cell rate depending on the congestion in the system. In our case, it is difficult to determine an acceptable message rate for an IB since the cost of processing event publications varies depending on size, content, and event subscriptions.

Multicast ABR requires flow control for one-to-many communication. An FRM cell is sent by the source and all receivers in the multicast tree respond with BRM cells, which are consolidated at ATM switches [329]. Different ways of consolidating feedback cells have been proposed [134]. These algorithms have a trade-off between timely response to congestion and the introduction of *consolidation noise* when new BRM cells do not include feedback from all downstream branches. Our consolidation logic at intermediate brokers tries to balance this trade-off by aggregating UCA messages with the same sequence number, but also short-cutting new UCA messages. The scalable flow control protocol in [409] follows a soft synchronization approach, where BRM cells triggered by different FRM cells can be consolidated at a branch point.

Overlay Networks

Congestion control for application-level overlay networks is sparse, mainly because application-level routing is a novel research focus. A hybrid system for application-level reliable multicast in heterogeneous networks that addresses congestion control is *RMX* [76]. It uses a receiver-based scheme with the transcoding of application data. Global flow control in an overlay network can be viewed as a dynamic optimization problem [13], in which a cost-benefit approach helps find an optimal solution.

8.4 Mobility

The emergence of mobile computing has opened up a whole new field of services provided for the benefit of the mobile user. Many such services can exploit the fact that the mobile device is aware of its current location. For example, car navigation systems use knowledge about current and past locations to aid drivers in finding their way through unknown cities. Location information can even be combined with other sources of data, e.g., the weather report, information on traffic jams, or free parking spaces. In such cases, the system can propose routes that avoid places where traffic is high or weather conditions are unpleasant, or can direct the driver to the nearest free parking space. All these are examples for *location-based services.*

A convenient way to construct location-based services is to build them using event infrastructures, such as those provided by publish/subscribe systems. Here, producers and consumers are enabled to exchange information based on message type or content rather than particular destination identifiers or addresses. This *loose coupling* of producers and consumers is the premier advantage of publish/subscribe systems, which facilitates mobile communication. Producers are relieved from managing interested consumers, and vice versa. In the following we study how these advantages can be exploited and what extensions are eligible in the context of mobile services.

We argue that support for mobility should be an issue of the publish/subscribe middleware itself and not be delegated to the application layer. Three kinds of application scenarios have to be supported: (i) existing applications in a static environment, (ii) existing applications in a mobile environment, and (iii) mobility-aware applications. Since publish/subscribe systems and applications have been deployed very successfully, extending existing systems and models is preferred to creating new "mobile" middleware from scratch in order to facilitate the integration of the first two scenarios. As a consequence, the middleware must transparently handle some of the new mobility issues. This allows existing event-based applications to directly interact with and even to be deployed as mobile applications. On the other hand, mobility-aware applications (the third scenario) require the middleware to support a semiautomated handling of location changes. If no such support is available, mobility is actually controlled by the application and not by the movement of the client.

We differentiate among support for two different and orthogonal types of mobility. The first type of mobility is called *physical* mobility, where clients may temporarily disconnect from the publish/subscribe system (due to power-saving requirements or the network characteristics). This means that applications are not necessarily aware of the fact that the client is moving, allowing existing applications to be transferred to mobile environments. The second type of mobility is called *logical mobility*, where clients remain attached to the their broker and have an application-level notion of location, which is described by *location-dependent subscriptions.* As an example, consider a car looking for a free parking space in the street it is currently driving along. In this situation it may subscribe to "New free parking space on Rebeca Drive". However, if Rebeca Drive is a very long street, the same driver will also receive notifications about free parking spaces very far down the road (or behind him), which are impossible to reach in good time. What the user would like to do is to specify a subscription such that he receives all notifications about "vacancies in the vicinity of his current location". We call these subscriptions *location-dependent.*

In this section we analyze and discuss the basic issues involved when adding mobility support to a publish/subscribe infrastructure. We identify and define two orthogonal forms of mobility (physical and logical mobility) and discuss the requirements of a system supporting both types of mobility.

8.4.1 Mobility Issues in Publish/Subscribe Middleware

Mobile clients have many characteristics, among them the need to disconnect from the network for different reasons. Be it for geographical, administrative, or power saving reasons, being connected to the same broker all the time is no longer possible. Hence, we have to take into account that clients will disconnect from their border broker once in a while. The middleware has to deal with moving clients and the possibility that a disconnected client reconnects at the same or a different broker later.

A first step toward mobility is to enhance existing publish/subscribe middleware to allow for roaming clients so that existing applications can be used in mobile environments. This means that the existing interface operations for accessing the middleware and the applications on top are not required to change. More important, the quality of service offered by the middleware must not degrade substantially. The resulting location transparency is necessary to make existing applications mobile, e.g., stock quote monitoring seamlessly transferred from PCs to PDAs.

On the other hand, future applications do not want complete transparency, but rely on *mobility awareness*. More specifically, mobility support should blend out unwanted phenomena, like disconnectedness, and enforce wanted behavior, like the location awareness in location-based services. Consequently, extending the interface of the publish/subscribe middleware to facilitate location awareness is a promising open issue, since most existing work concentrated on the transparency only.

When roaming, clients change (at least some portion of) the context they are operating in, and they might want to react to these changes, e.g., to adapt their subscriptions. However, an appropriate infrastructure support has to relieve the application from having to react “manually” to all changes. The middleware should rather offer an automated adaptation to context changes, i.e., facilitating location dependency. This leads to different notions of mobility and we distinguish:

- *Physical mobility*: A client that is physically mobile disconnects for certain periods of time and has different border brokers along its itinerary through the infrastructure. The main concern of physical mobility is *location transparency.*
- *Logical mobility*: A client that is logically mobile is aware of its location changes. In order to relieve the client from adapting *manually* to new locations, the main concern of logical mobility is *automated* location awareness within the publish/subscribe middleware.

Physical and logical mobility are two orthogonal aspects of mobility. Since the physical layout of a publish/subscribe system does usually not correspond to geographical realities, it seems reasonable to separate the two notions of mobility. In the following, we assume logical mobility to be a refinement of physical mobility in that a client remains connected to the same broker when roaming logically. The two notions have different quality of service requirements and therefore different solutions are developed to match both.

8.4.2 Physical Mobility

Physical mobility is similar to what in the area of mobile computing is called *terminal mobility* or *roaming*. A client accesses the system through a certain number of *access points* (GSM base stations, WLAN access points, or border brokers). When moving physically, the client may get out of reach of one access point and move into the reach of a second access point which are not necessarily overlapping. In general we cannot expect to have seamless access to the broker network but more a sequence of phases of connectedness, e.g., on the daily route between home and office. In this setting we analyze the quality of service requirements from the viewpoint of roaming clients:

- **Interface.** Obviously, the existing interface to the publish/subscribe system must not change as legacy applications are not aware of mobility.
- **Completeness.** Despite intermittent disconnects, the liveness condition of Def. 2.5 must be satisfied, i.e., a finite time after subscribing, the delivery of notifications that are published after this time and match the subscription is guaranteed.
- **Ordering.** In Sect. 2.5.3 FIFO-producer and causal ordering were discussed; they are eligible features in the mobile case, too.
- **Responsiveness.** The delay of relocating a roaming client should be minimal to maximize the responsiveness of the system. This has to be taken into account when designing a relocation protocol.

Possible Solutions

One solution would be to rely on Mobile IP [306] for connecting clients to border brokers, hiding physical mobility in the network layer. The drawback, however, is that the communication is also hidden from the publish/subscribe middleware, which is then not able to draw from any notification delivery localities or routing optimizations, thereby possibly violating the requirement of responsiveness. Such an approach might only be feasible if the physical and logical layout of a given system is completely orthogonal.

A different, naïve solution to implement physical mobility would be to use sequences of *sub-unsub-sub* calls to register a client at a new broker. When a client moves from border broker B_1 to B_2, it simply unsubscribes at B_1 and resubscribes at B_2, without any support in the middleware. But a client may not detect leaving the range of a broker and is in this case not able to unsubscribe at its old location. Even more severely, during its

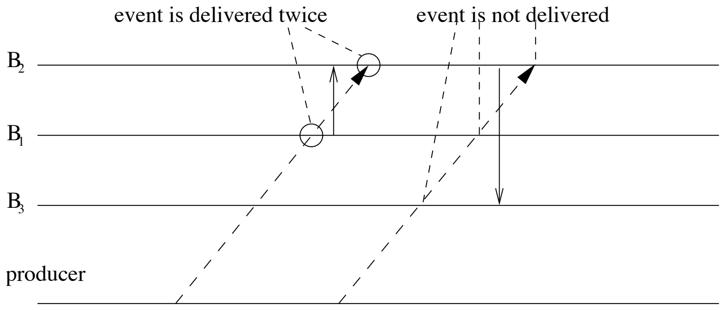


Fig. 8.12. Missing notifications in a flooding scenario

time of disconnectedness, the client might miss several notifications or or get duplicates, even if notifications are flooded in the network and the location change is instantaneous. This problem is depicted in Fig. 8.12. Hence, this solution is not complete and we outline an algorithm in Sect. 8.4.2 that takes into account all requirements stated above. The complete algorithm is detailed by Zeidler and Fiege [408].

Notification Delivery with Roaming Clients

In this section we introduce an algorithm for extending standard brokers (cf. Chap. 4) to cope with mobile clients, maintaining their subscriptions as well as guaranteeing the required quality of service as described in the previous section. Apart from guaranteeing complete notification delivery, our algorithm also ensures that the old border broker will eventually receive an equivalent to an explicit *sign-off* from the client, even if an explicit unsubscribe was not possible.

Our mechanism uses a natural way of distributed caching, which seems in general preferable to a potentially problematic central caching proxy.

Prerequisites

The solution sketched below can be used in every environment that meets the following requirements:

1. Border brokers have to install and maintain a buffer for all notifications that are not yet delivered in order to deal with disconnects.
2. The underlying routing infrastructure uses advertisements. Although not strictly necessary, the relocation effort is reduced substantially in that they guide the search for the old delivery path. Simple routing is assumed as routing strategy for now, and more advanced routing algorithms are discussed later.
3. Border brokers or clients must have some means of detecting the new configuration that a client has entered the range of the broker. Some form of beacon or heartbeat is presupposed; we do not go into the details here.

4. For now, we assume that only subscribers are mobile and that clients acting as producers remain stationary.

Algorithm Outline

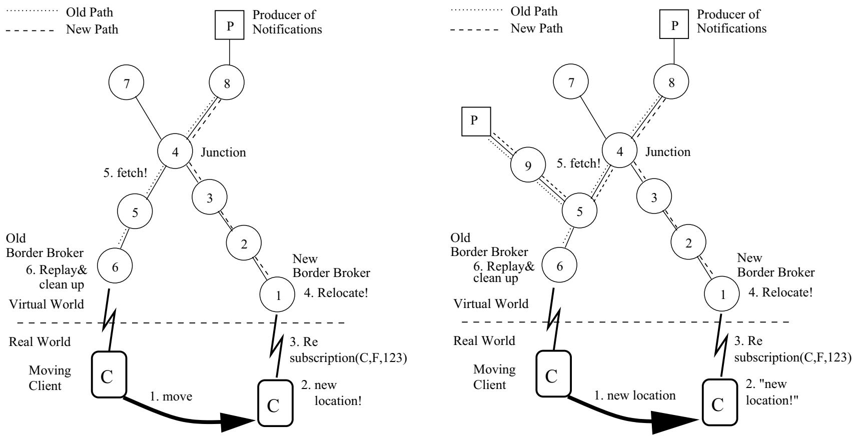
We use a stepwise refinement of traditional subscription forwarding, as discusses in Chap. 4, to devise the algorithm:

1. When reconnecting to a broker, subscriptions are automatically reissued so that clients do not need to resubscribe manually.
2. The broker network configuration is updated to accommodate to client relocation rather than handling an independent new (re)subscription from a new location.
3. Notifications forwarded to the old location have to be replayed to the new one in order to bridge disconnectedness.
4. Delivery of new notifications has to be postponed until the replay is finished. In this way, moving does not influence the FIFO-producer order of notifications, fulfilling the ordering requirement.

Consider the scenario of Fig. 8.13(a) with a single consumer. Client C is moving from broker B_6 to broker B_1 (step 1 in the figure). The local broker, which resides on the client, e.g., in the form of libraries, is informed by the new border broker (i.e., B_6) about its relocation, according to the prerequisites. The local broker then reissues the active subscriptions, which were previously forwarded through and recorded in the local broker anyway. By avoiding manual resubscriptions of the client application, the first requirement stated at the beginning of this section is achieved, i.e., the interface to the middleware is not changed.

In the second step, we enable the publish/subscribe middleware to relocate the client. The goal of the relocation process is to update the routing configuration by redirecting the delivery paths currently leading to the old destination of C to the new destination. During this process, reissued subscriptions are propagated as usual, e.g., in the direction of any received advertisement if advertisements are used, through B_2 and B_3 to broker B_4, setting up their routing tables. At B_4 the old and new path from producer P to client C meet (dotted and dashed line, respectively). Broker B_4 is aware of the junction because an entry of the old path of this subscription/ client is already in its routing table.[2] When the routing table in the junction is updated, new published notifications will be delivered to the relocated client. Without assuming any knowledge about the old location of the moving client, the system is able to draw from localities in that only a portion of the delivery path is changed. Changes are limited to the smallest subgraph necessary for diverting routing paths, facilitating the timeliness/efficiency requirement, which is only available with inherent middleware support.

[2] Subscriptions can be identified if simple routing is used.



(a) Single Producer (b) Multiple Producer

Fig. 8.13. Moving client scenarios with one and multiple producers

The third step ensures completeness over phases of disconnectedness during movement. The junction broker B_4 sends a fetch request along the old path to B_6 following the routing table entries for the given subscription. All brokers along this path update their routing tables such that they are pointing into the direction the fetch originates from, i.e., B_4. Border broker B_6 as last recipient replays all buffered notifications. If delivered notifications are annotated with sequence numbers by the border broker, reissued subscriptions can in turn carry the last received number to qualify the replay. Note that replays are forwarded only in the direction of a specific subscription and do not mingle with other clients' data. After replaying, the path from the old broker to the junction broker can be shut down by deleting the subscription's routing table entries as long as advertisement and routing entry point into the same direction; thereby excluding and stopping at the junction. In this way the notifications that passed the junction broker before its update are collected and sent toward the new location, ensuring the required completeness.

The last step finally reorders the notifications so that the sender FIFO condition remains valid after relocation. The new border broker has to block and cache all incoming notifications that are to be delivered to the given client (not impeding communication of other clients) until the replay is finished. As with all buffering, consistency can always only be guaranteed for a predefined, finite amount of time or space.

Figure 8.13(b) shows a scenario with multiple producers. In this case, several junctions exist which all lie on the path from the first junction to the old border broker of the client. For the two producers, the junctions are at brokers B_4 and B_5, respectively.

Extensions

Mobile Producers. So far we have assumed that only consumers can be mobile. When a producer is mobile, the notifications it publishes while it is disconnected from the system are not forwarded but are queued by its local broker. When the producer reconnects to a new border broker, the local broker reissues the advertisements currently active, while still queuing newly published notifications. The forwarding of these advertisements will in turn lead to overlapping subscriptions being forwarded to the new location of the producer. When this process has finished, the queued notifications are forwarded if a matching subscription exists or they are discarded, otherwise. Then, the normal handling of published notifications starts again. Delivery paths that lead to the old location of the producer and which are no longer needed are similarly dropped as described above.

Covering-Based Routing. If covering-based routing instead of simple routing is used, the fetch phase of the algorithm has to be extended. Now, the junction is reached if an entry with a covering subscription $F' \supset F$ is already registered. At this point the delivery path to the new location is correctly built up, but we do not know whether the old location lies in the direction of F' or in the direction of the advertisements. The fetch phase is extended in that fetch requests are sent toward all advertisements and all covering subscriptions; it is a kind of flooding in the overlay network of matching producers and consumers of similar interests. Only one of the fetch requests will not get dropped and reach the old border broker. The replay has to be flooded in the same overlay network if no tunneling mechanisms, internal or external, are used.

Merging-Based Routing. The extension for covering stated above can also cope with a broker network applying merging. Only the number of potential covers increases, and hence the size of those parts of the overlay network that are flooded. Both covering and merging promise to increase routing efficiency, but, on the other hand, aggravate relocation management.

Movement Speed. For simplicity reasons we assume that the client's movement speed is not too fast for the relocation process to terminate before the client moves again, i.e., the process always terminates at the correct broker. However, if resubscriptions of the local broker are annotated with a relocation counter, which is reset after a successful replay, concurrent relocation processes can be identified and controlled in the middleware, avoiding the speed limit.

Cache Management. Even if storage constraints in the border brokers are not of concern, mobile clients may be disconnected for a long period of time in which more missed notifications are cached than the client can handle during replay. The possibly limited resources of mobile clients must be taken into account when designing cache sizes or limiting the replay by semantic filtering [195].

Discussion

The above algorithm shows how relocation and adaptation of the delivery paths is performed in a fully distributed fashion. Many optimizations exist for this algorithm (e.g., [59, 92]). They typically reduce the number of necessary messages, but they also impose further constraints on network layout or require additional information about client movement. The approach presented here is a generalization that is robust and simple. Its central features are:

- **No explicit `moveOut`**. The algorithm ensures by design that the new broker can identify a relocated client and handle this appropriately. Moreover, the algorithm ensures that the broker at the old location eventually receives an equivalent to a `moveOut` for proper garbage collection.
- **No central caching proxy**. The algorithm is fully distributed and buffers information wherever necessary, thereby drawing optimally from localities.
- **No information loss**. By buffering information appropriately, the algorithm ensures that no information is lost due to relocation. As with all buffering schemes, this is only true modulo space and/or time constrains.
- **No "out-of-band" communication**. All messages sent related to a relocation process are sent *explicitly* within the broker network and not leaving the paradigm of publish/subscribe. Therefore, we do not need globally unique sequence numbers and can guarantee FIFO-producer ordering as well as not sending duplicates.
- **Optimal use of localities**. The algorithm draws optimally from localities and ensures that only the least necessary subgraph is reconfigured.

8.4.3 Logical Mobility

While physical mobility is a rather technical issue invisible to the application, logical mobility involves location awareness. An example for logical mobility is when clients move around a house or building that is served by only one border broker. In this case, the user might be interested to receive just those notifications that refer to the room in which he is currently located. Note that a client can be both logically and physically mobile at the same time.

A logically mobile client moving from one location to another, e.g., from one room to the other in a company building, will expect a frictionless change of location explicitly without a notable setup time after having changed from its own office to the conference room next door. The adaptation of some location-dependent subscription should take place "instantaneously". Intuitively, we would like to experience the notion of being subscribed to "everything, everywhere, all the time" and increase the reactivity of the system to moving clients.

Location-Dependent Filters

A publish/subscribe system offering *location-dependent filters* has the same interface as a regular publish/subscribe system (i.e., it offers the *pub*, *sub*, *unsub*, *notify* primitives). However, in specifying subscription filters for name/value pairs referring to "*location*", it supports a new primitive to specify things like "all notifications where the attribute *location* equals my current location". More precisely, we postulate a specific marker *myloc* that can be used in a subscription. The marker stands for a specific set of locations that depend on the current location of the client. For example, a client could issue a subscription for all free parking spaces in the vicinity of his current location as follows: $(\textit{service} = \text{"parking"}), (\textit{location} \in \textit{myloc}), (\textit{car-type} \geq \text{"compact"})$.

The set of locations associated with the marker is taken from a particular range L of locations. This set is application dependent and can, for instance, contain all the different rooms of a building, all the streets of a town, or all the geographical coordinates given by a GPS system up to a certain granularity. Given a notification with the attribute *location*, the subscription $(\textit{location} \in \textit{myloc})$ will evaluate to true for a particular client at location y iff $x \in \textit{myloc}(y)$, where $\textit{myloc}(y)$ is the specific set of locations associated with y. Then, we say that the notification matches the location-dependent filter.

The simplest form of $\textit{myloc}(y)$ is simply the set $\{y\}$. In this case a notification matches the subscription if $x = y$. But in the car example, the car driver looking for a parking space might want to specify:

$$(\textit{location} = \text{"at most two blocks away from } \textit{myloc}\text{"})$$

Then, *myloc* corresponds to all elements of L that satisfy this requirement.

A Tentative but Incomplete Solution for Logical Mobility

While location-dependent filters are not directly supported by current publish/subscribe middleware, one might argue that it is not very difficult to emulate them on top of currently available systems in this case. The idea would be to build a wrapper around an existing system that follows the location changes of the users and transparently unsubscribes to the old location and subscribes to the new one when the user moves. However, depending on the internal routing strategy of the event system, it may lead to unexpected results. The routing strategies deployed in many existing content-based event systems such as SIENA [71], Elvin [341], and REBECA [136] lead to *blackout periods* where no notifications are delivered. The problem is that it usually takes a significant time delay to process a new subscription. After subscribing to a filter, it takes some time t_d until the subscription is propagated to a potential source. Then it takes at least another t_d time until a notification reaches the subscriber. This phenomenon is depicted in Fig. 8.14. (Note that the delay t_d may be different for different notification sources and may change over time.) If the client remains at any new location less than $2t_d$ time, then the subscriber will "starve", i.e., it will receive few or no notifications.

An intuitive but inefficient solution

Another basic solution that can immediately be built using existing technology is again based on flooding. The local broker can then decide to deliver a

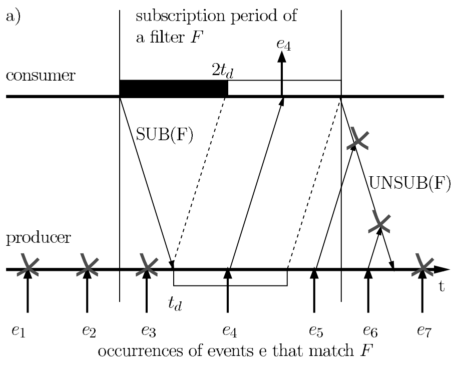


Fig. 8.14. Blackout period after subscribing with simple routing

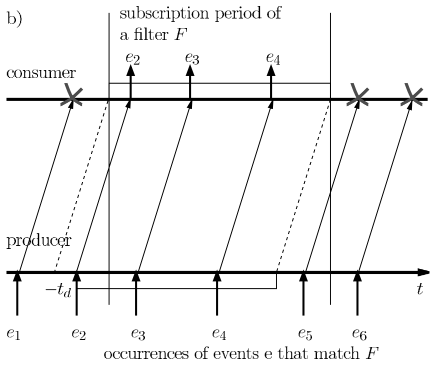


Fig. 8.15. Blackout period with flooding and client-side filtering

notification to a client depending on the client's current location (Fig. 8.15). Obviously, flooding prevents the blackout periods, which were present in the previous solution, but it should be equally clear that flooding is a very expensive routing strategy, especially for large pub/sub systems [267].

Quality of Service of Logical Mobility

Interestingly, while flooding is very expensive and therefore not desirable, it comes very close to the quality of service that we would like to achieve for logical mobility, namely to the notion of being subscribed to "everything, everywhere, all the time". The problem is that it is hard to precisely define the behavior of flooding without reverting to some unpleasantly theoretical constructions of operational semantics.

With logical mobility there is, however, no danger of receiving a notification twice because the consumer remains attached to the same "delivery path". The quality of service we require for logical mobility therefore is simply stated as follows: On change of location from x to y, all notifications should be delivered to the consumer "as if" flooding were used as underlying

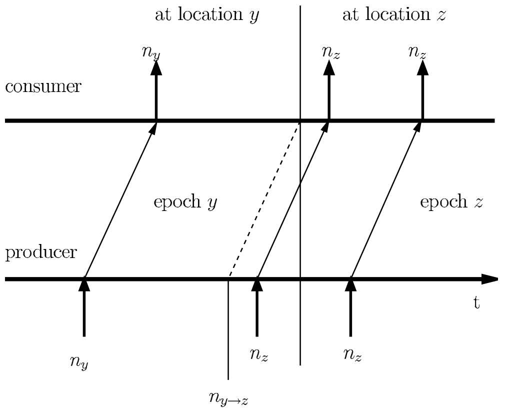


Fig. 8.16. Defining the quality of service for logical mobility using virtual notifications $n_{y\to z}$ that arrives at the consumer just at the time of the location change from y to z

routing strategy. This statement is made a little more concrete in Fig. 8.16, where the sequence of notifications generated by any consumer is divided into epochs that correspond to when the notification actually arrives at the consumer (the epoch borders between locations y and z are drawn as a virtual notification $n_{y\to z}$). We require that all notifications matching the current location-dependent subscription from every such epoch must be delivered. Intuitively, the epochs define the semantics of flooding.

Location-Dependent Filters for Logical Mobility

We now describe the algorithmic solution to the scenario where clients are only logically mobile, i.e., they remain attached to a single border broker.

Main Idea

Consider an arbitrary routing path between a producer and a consumer. This path consists of a sequence of brokers $B_1, B_2, \ldots, B_{k-1}, B_k$, where B_1 is the local broker of the consumer and B_k is the local broker of the producer (Fig. 8.17 shows the setup for $k = 3$). Assume the consumer has issued a location-dependent subscription F. Using the "usual" content-based routing algorithms, the current value $\tilde{F}$ of F, which instantiates the marker variable with the current location, would permeate the network in such a way that the filters along the routing path allow a matching subscription published by the producer to reach the consumer. Formally, the filters $F_1, F_2, \ldots, F_k$ along the links between the brokers should maintain a set-inclusion property (cf. Sect. 4.3.2))

$$F_k \supseteq F_{k-1} \supseteq \ldots \supseteq F_2 \supseteq F_1 \supseteq F_0 = \tilde{F}.$$

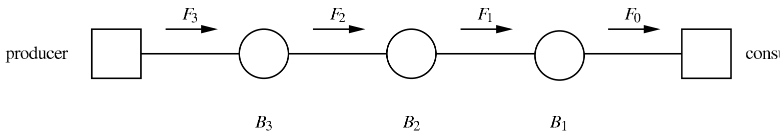


Fig. 8.17. Network setting for the example

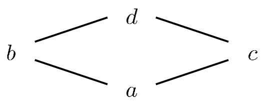


Fig. 8.18. Movement graph defining movement restrictions of a consumer

Obviously, if for any new value $\tilde{F}$ of F a new subscription must flow through the network toward the producers, notifications published in the meantime might go unnoticed. The idea of the proposed scheme is to always have the local broker of the consumer do perfect client-side filtering (i.e., set $F_0 = \tilde{F}$), but to let possible future notifications reach brokers that are nearer to the consumer so that their delay to reach the consumer is lower once the consumer switches to a new location.

Let T denote the set of time values, which for simplicity we will assume to be the set of natural numbers $\mathbb{N}$. Let L denote the set of all consumer locations. Then we define a function $loc : T \to L$ that describes the movement of the consumer over time. For example, for a location set $L = \{a, b, c, d\}$ a possible value of loc is $\{(1, a), (2, b), (3, d), \ldots\}$, meaning that at time 1, the consumer's location is a, at time 2 it is b, and so on.

We assume that loc is subject to some movement restrictions, which in effect define a maximum speed of movement for the consumer. We assume that such a restriction is given by a *movement graph* such as the one depicted in Fig. 8.18. The graph formalizes which locations can be reached from which locations in one movement step of the consumer. One movement step has some application-defined correspondence to one time step.

Given the function loc and a movement graph, it is possible to define a function $ploc : L \times \mathbb{N} \to 2^L$ of possible (future) locations (the notation 2^L denotes the powerset of L, i.e., the set of all subsets of L). The function takes a current location x and a number of consumer steps $q \geq 0$ and returns the set of possible locations, which the consumer could be in starting from x after q steps in the movement graph.

Since a possible move of the consumer always is to remain at the same location, for all locations $x \in L$ and all $q \in \mathbb{N}$ we should require that

$$ploc(x, q) \subseteq ploc(x, q + 1). \tag{8.11}$$

Taking the example values from above, possible values for $ploc$ are as follows:

$$ploc(a, 0) = \{a\} \qquad ploc(a, 1) = \{a, b, c\} \qquad ploc(a, 2) = \{a, b, c, d\}$$

Now, if the consumer is at location a, for example, every broker B_i along the path toward a producer should subscribe for $ploc(a, q)$ for some q, which is an increasing sequence of natural numbers depending on i and the network characteristics. If the time it takes for a broker to process a new subscription is on the order of the time a client remains at one particular location, then the individual filters F_i along the sample network setting in Fig. 8.17 should be set as $F_i = ploc(a, i)$, e.g., $F_0 = ploc(a, 0) = \{a\}$, $F_1 = ploc(a, 1) = \{a, b, c\}$, and so on. This requirement should be maintained throughout location changes by the consumer. For example, whenever a consumer moves from an old location x to a new location y, this will cause B_1 to change the location-dependent part of filter F_0 for client-side filtering from the old to the new location. Broker B_1 updates its routing table appropriately.

In general, broker B_i sends a message with the new location to B_{i+1} instructing it to change F_i from $ploc(x, i)$ to $ploc(y, i)$ and consequently to update the routing table by removing certain locations and adding new locations. Removing and adding new locations corresponds to unsubscribing and subscribing to the corresponding filters. The normal administration messages can be used to do this. Note that Eq. (8.11) guarantees the subset relationship, which should always hold on every path between a producer and a consumer.

Example

As an example, consider the value of *loc* where at time 1 the client is in location a, at time 2 at b, and at time 3 at d in the movement graph depicted in Fig. 8.18. Table 8.1 gives the values of *ploc* for all locations and the first four time instances. For $t = 0$ the value of *ploc* is equal to the current location. For $t = 1$ it returns all locations reachable in one time step in the movement graph, etc.

Table 8.1. Values of $ploc(x, t)$ for the example setting

t	$x = a$	$x = b$	$x = c$	$x = d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, c, d\}$	$\{b, c, d\}$
2	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$
3	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$

Now assume again the setting depicted in Fig. 8.17. The values of Table 8.1 directly determine the filter settings for $F_0, \ldots, F_3$ as shown in Table 8.2. At time $t = 1$ the client moves to location b. This means that F_0 changes from $\{a\}$ to $\{b\}$ and that F_1 must unsubscribe to c and subscribe to d, yielding $F_1 = \{a, b, d\}$. At time $t = 2$ the client moves to d, causing F_0 to change to $\{d\}$ and F_1 to unsubscribe to a and subscribe to c. All other filters remain unchanged.

Table 8.2. Values of filters in example setting

time t	F_3	F_2	F_1	F_0
0	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c\}$	$\{a\}$
1	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,d\}$	$\{b\}$
2	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{b,c,d\}$	$\{d\}$

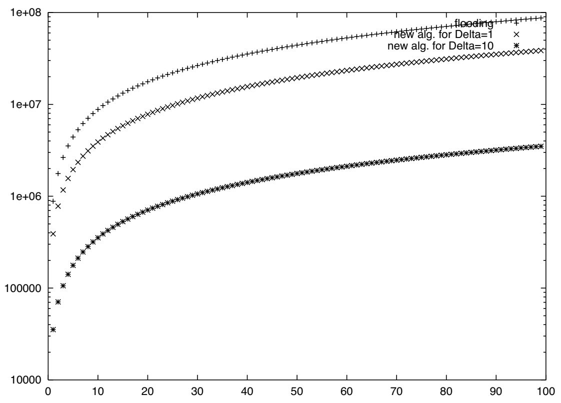


Fig. 8.19. Total number of messages generated for flooding and two scenarios of the new algorithm ($\Delta = 1s$ and $\Delta = 10s$). Note that the y-axis has a logarithmic scale. The x-axis denotes time in seconds

The example nicely shows that the method does some sort of "restricted flooding", i.e., all notifications reach broker B_2, but from there the uncertainty is restricted and so is the flow of notifications forwarded by B_2. In fact, the method described above using the *ploc* function can be regarded as an abstraction of both "trivial" implementations discussed (i.e., both implementations are instantiations of our scheme).

We have informally analyzed the total number of messages (notifications and administrative messages) generated by our new algorithm for an arguably realistic network setting, exactly one consumer and two different speeds of consumer movement: fast movement (Δ = 1s) and slow (Δ = 10s). We compare the results of these calculations with the total number of messages generated by flooding in Fig. 8.19 (see [143] for a detailed description of the system assumptions and the derivation of these numbers). It is interesting to see that although our algorithm generates administrative messages on all network links for every location change of the consumer, the fraction of messages saved is still considerable. We also note that many of the assumptions made in calculating these figures have been very conservative. For example, we assume that there is only one consumer in the network and that notifications are generated by the producers according to a uniform distribution over set of locations. Both assumptions prevent routing strategy optimizations to play to their strengths.

Concluding Mobility

The presented approach to support mobility in publish/subscribe middleware can only be seen as a first start for generic mobility support. We have analyzed the problem of mobility from the viewpoint of the event-based paradigm and have identified two separate flavors of mobility. While physical mobility is tied to the notion of rebinding a client to different brokers and can be implemented transparently, logical mobility refers to a certain form of location awareness offering a client a fine-grained control over notification delivery in the form of location-dependent filters.

Many other interesting problems concerning the combination of mobility and publish/subscribe infrastructures remain. For example, location-dependent filters may be generalized to "dynamic filters" that depend on a function of the local state of the client (not only its current location), like a client interested in receiving notifications for sales that he still can afford.

8.4.4 Further Reading

Further details on the movement algorithms can be found in [141, 142, 408]. Work on middleware for mobile computing usually concentrated on classical synchronous middleware like CORBA. Only recently, position papers have stated that publish/subscribe systems have an enormous potential to better accommodate the needs of large mobile communities [89, 208]. Research in publish/subscribe systems has mainly focused on *static* systems, where clients do not move and the publish/subscribe infrastructure remains relatively stable throughout the system's lifetime, e.g., Elvin [341], Gryphon [197], REBECA [144], and SIENA [71]. If present at all, mobility support is a concern of the application layer. Applications detect the need to change a subscription and have to react explicitly and manually to this detection.

Huang and Garcia-Molina [195, 196] provide a good overview of possible options for supporting mobility in publish/subscribe systems. They describe algorithms for a "new" middleware system tailored and optimized to mobile and ad hoc networks, not so much an extension of an existing system. Cambridge Event Architecture (CEA) [20] and JEDI [92] also address problems of mobility. JEDI uses explicit moveIn and moveOut operations to relocate clients. Hence, mobility is controlled by the application, which is not transparent and even is unrealistic since clients usually only can react *after* having been moved. The mobility extensions of SIENA [59] are very similar. Explicit sign-offs are required and interim notifications stored during disconnectedness are directly forwarded to a new location upon request. Cugola et al. [89] proposes a leader election and group management protocol for dynamic dispatching trees to dynamically adapt the internals of the JEDI event system, their implementation model is based on multicast and it groups identical subscribers. An extension for Elvin allows for disconnectedness using a central caching proxy [368], which is a potential performance bottleneck. Jacobsen [208] presents some very interesting ideas on location-based services and the possible expressiveness of subscription languages. STEAM [257] is an event service designed for wireless ad hoc networks. Subscribers consume only events produced by geographically close-by publishers. It relies on proximity-based group communication.