3

Event-based middleware services

The term *event service* has different definitions. In general, an event service connects producers of information and interested consumers. The service acquires events from producers, it filters them and notifies them to consumers. The event services can be found at different layers of abstraction.

The objective of the chapter is to present and describe event-based middleware services. This chapter describes the characteristics of different event services that are integrated in different layers of a distributed system architecture. We present a brief survey of communication systems whose terms have been used synonymously to the event service notion. The chapter is organized as follows: section 3.1 describes the system layers in which event services can be found. Section 3.2 presents the survey of communication systems and an event service definition. Event-based middlewares are presented in section 3.3. Finally, section 3.3 concludes this chapter.

3.1 Event-based communication layers

Figure 3.1 presents the layers of a distributed system architecture. Event services can be found at these layers, starting with the application layer down to the operating system layer. Note that event-based communication on one layer does not require necessarily an event concept on the next lower layer.

Services at the *application layer* handle application-dependent client subscriptions that are addressed to the system as a whole. The system observes and filters events from external sources. Currently, there are already several independent implementations of event services such as SIFT [Car98], Siena [Car98], OpenCQ [LPT99] and Gryphon [IBM01].

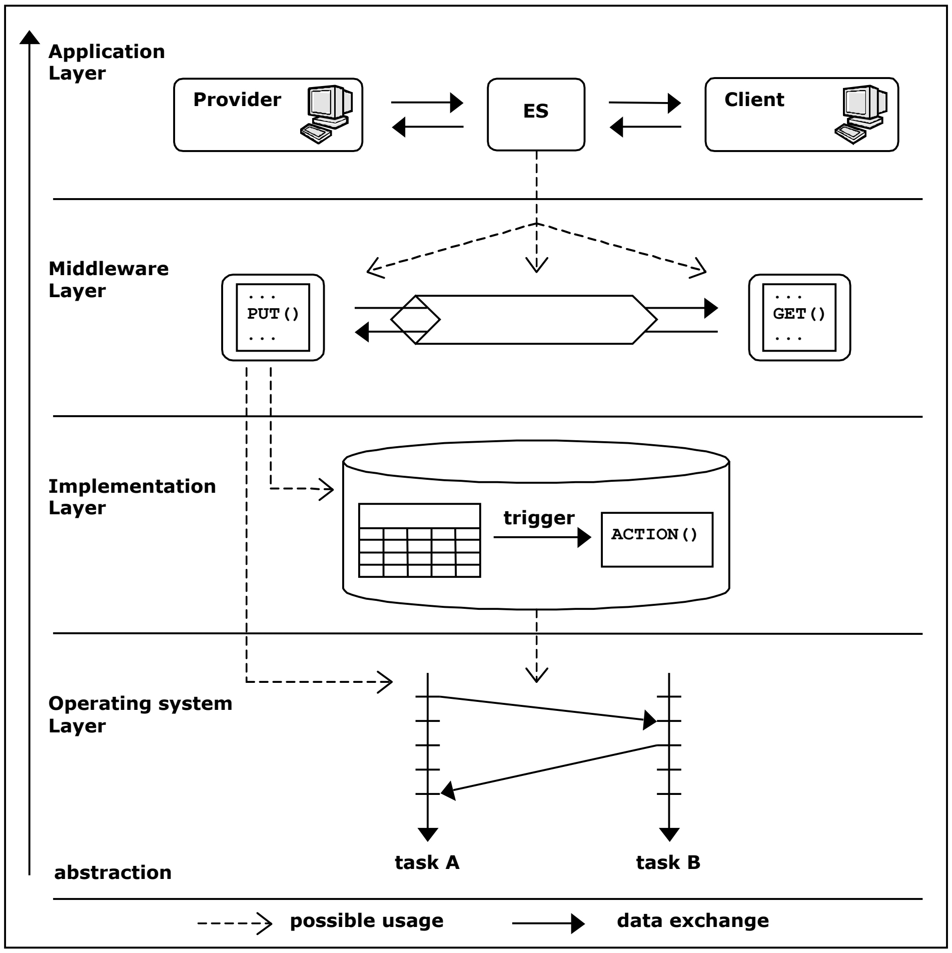

Figure 3.1 Layers of event-based communication

At the *middleware layer*, distributed software components can communicate, e.g., by use of system specific events that may encapsulate and transport events from the higher application layer. Events from the application layer are handled as mere data to be processed and their inner structure and interpretation is not known at this layer. Examples for middleware services are CORBA Notification Service [OMG02a] and Java Message Service [Sun01].

Services on the *implementation layer* notify about internal events, e.g., inserts into a database table. Other examples are event notification in distributed programming, debugging and monitoring of distributed systems. Middleware layer events can be mapped onto these events. Requests on the implementation layer are then translated to operating systems calls. Event-based communication examples are triggers and rules in active database systems.

At the *operating system layer*, events are used for synchronization of system tasks and transaction monitors. This layer also encloses low level event handling in programming languages and GUI frameworks, such as in Java Swing and SWT.

3.2 Communication systems

Many publications give a descriptive introduction of the notion of an event service. Several terms are used synonymously to the term event service and some are closely related. We briefly review the notion of these terms in the following paragraphs.

Personalized system: the term *personalization* encloses the expression of a consumer interest in a subscription. The subscription can be defined either explicitly by the consumer or implicitly by data mining or analysis of client actions. Personalized systems operate on the application level. Several event services support personalized consumer subscriptions, these are personalized systems. Other forms of personalization are adaptation of web pages to certain costumers, personalized web pages, personalized feedback in search engines and automatic personalized filtering and classification of documents.

Alerting service*:* the term *alerting system* or *alerting service* is used as synonym to event system. The term alerting service is now widely used in the context of digital libraries, such as Springer Link Alert [Spr01]. The term alerting also often refers to newsletter services via email. The original applications for alerting services are systems raising alarms in case of hazard, intrusion or engine failure.

Publish/subscribe service: the *publish/subscribe* paradigm is an interaction model that consist both of information providers (publishers, producers) that publish data to the system, and of clients (subscribers, consumers) that subscribe to issues of interest within the system. The role of publish/subscribe systems is to timely send the right information to the right consumers. Publish/subscribe systems are event services that support producers that actively send data to the system. Often, event messages are not distinguished.

Push system: the term *push system* refers to an *Internet-based system* that delivers content to its clients via *topic-based channels*. Push system implementations are often middleware systems that support application layer systems.

Dissemination-based system: in the context of the *Dissemination-based Information System (DBIS)* framework, an event service is an information broker that acquires information from data sources, adds value and distributes the information to clients. The term focuses on the distribution of documents, not the observation and filtering of events.

System for selective dissemination of information: a *system for selective dissemination of information* is similar to an event system. The phrase was introduced in the 60's and it is historically the original term.

Information filtering system: an *information filtering system* is an event service that especially deals with new or changed documents. These services are also referred as *document filter systems*. A first approach to the clear definition of information filtering has been proposed by Belkin and Croft [BC92]. Yan and García-Molina [YGM95] use *information filtering* as synonym for event service, which neglects the aspect of the dissemination and notification. On the other extreme, they use the term *information dissemination* that ignores the aspect of the subscription filtering.

Content-based routing system: the term refers to *information retrieval systems* in which queries are routed to the available servers based on the expected relevance of the server to the query.

(Event) monitoring system: this term refers to systems that monitor certain event sources, filter events and send notifications. In general, the event sources are passively observing sources, such as sensors and measuring heads. The applications range from introspection or supervision of computer programs to communication in distributed systems and web applications.

Event-based infrastructure: event-based infrastructures support asynchronous message exchange between objects in a distributed environment. They define a middleware for interoperability of independent, heterogeneous systems and build an abstraction for program-to-program communication on the system level. Examples are JEDI [CNF01] and the CORBA Event and Notification Service [OMG95, OMG02a]. Event-based infrastructures are considered as event services on middleware layer.

Awareness service: the term *awareness* describes the automatic adjusting of present information of producer and consumer. The consumer is aware of all changes at the producer side, either new objects, deleted objects, or modified objects. The focus of awareness services is the adjustment of repositories, not the management of events. Therefore, awareness services are only related to event services.

Current awareness service: this term is not clearly defined – it is used to describe a variety of service offers. The term is used synonymously to the term alerting service.

Event handling service: the term *event handling service* describes a service defined in the Java Dynamic Management Kit (JDMK) [Sun03c] for the development of software agents.

Then, after having reviewed these terms, we define an **event service** as an event-based infrastructure that supports *event-based applications* and therefore, is a communication broker between event producers and consumers. The infrastructure becomes aware of events by means of event messages reporting the events. The event messages are processed either by the producers (push) or by the service actively observing the producers (pull). Consumers define their interest in certain events by means of *event subscriptions*. The information about observed events is filtered according to these subscriptions and notifications are sent to subscribed consumers.

3.3 Event-based middleware

The concept of a middleware was introduced to facilitate communication between entities in a heterogeneous distributed computing environment. A middleware is an additional software layer between the operating system and the distributed application on every node of a distributed system that uses operating system functions and attempts to provide a homogeneous view of the world to the applications. It is widely used and has proved to be a successful abstraction that helps with the design and implementation of complex distributed systems.

One of the main tasks of a middleware is to manage the communication of components in the distributed system. Therefore, middlewares can be classified according to their communication model.

3.3.1 Synchronous request/reply middleware

Traditional middleware is based on the idea of *request/reply communication* between two entities in the system. A client request information from a server and then waits until the server has responded with a reply. In the request/reply communication the client object must know the identity of the server and can only interact with a single server at a time, which limits scalability.

In RPC, a function calls another function that is located on a different node in the system. On object-oriented systems, a method of a client object invokes a method of a remote server object that runs on a different node, resulting a client/server relation. The call is synchronous because the client is blocked until the remote method returns with a result value from the server.

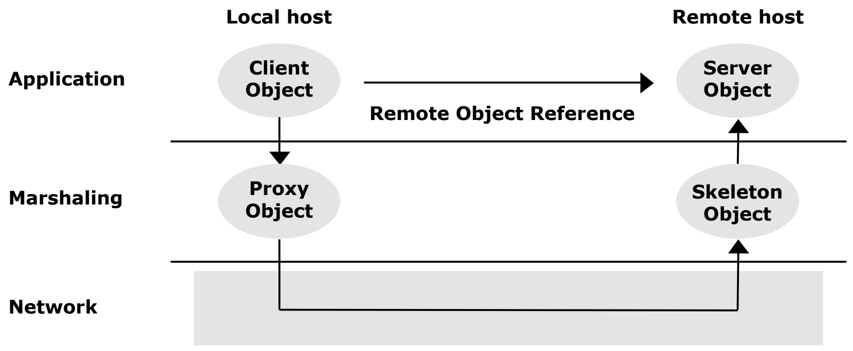

Figure 3.2 Components of a synchronous request/reply middleware

Figure 3.2 illustrates the components of a synchronous object-oriented *request/reply middleware*. An object reference can either point to a local or a remote object. A method invocation to a remote object is intercepted locally and the associated method parameters are marshaled into a representation that can be transferred over the network. The marshaling code is usually statically auto-generated by the middleware. The parameters are unmarshaled again after the transmission, and the method of the remote object is invoked at the server side. The process is repeated to return the result value from the server method back to the client.

The distribution transparency provided by a request/reply middleware creates the illusion that local and remote method calls can be treated uniformly. As a consequence, a tight coupling between clients and servers is introduced. Although this works well in a LAN, where network communication is reliable and inexpensive in terms of latency and bandwidth, the paradigm breaks down in WANs, where failure is common and the latency of remote calls can be several orders of magnitude higher than that of local ones.

Another problem when building a large-scale distributed system with a request/reply middleware is that all communication is one-to-one between exactly two entities, the client and the server. By contrast, in a large-scale system, a single component may need to communicate with several other components in multicast style, potentially without even having knowledge of all its communication partners.

3.3.2 Asynchronous message-oriented middleware

Message-oriented middleware is based on the model of message-passing between a sender and a receiver in a distributed system, which is shown in figure 3.3. A sender sends a message to a receiver by first inserting it into a local message queue. Then, the message-oriented middleware has the responsibility to deliver the message through the network to the receiver's message queue, which gets an asynchronous callback once the message has arrived.

The sender and the receiver are loosely-coupled during their communication because they only communicate via the indirection of message queues and thus do not have to be running at the same time. Message queues can provide strong reliability guarantees in case of failure by storing persistent messages on disk.

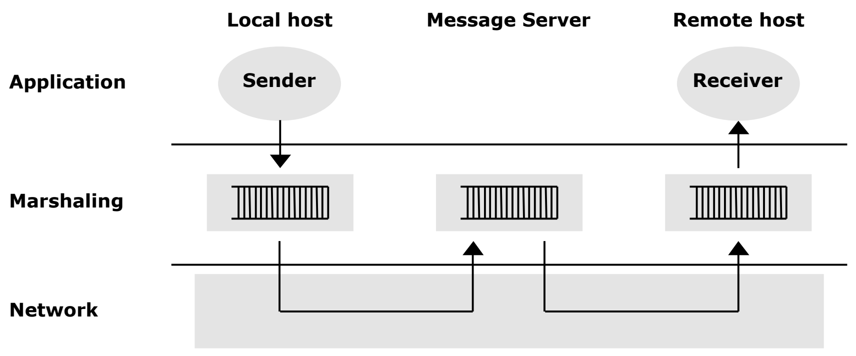

Figure 3.3 Components of an asynchronous messaging middleware

In a standard message-oriented middleware, communication between clients is still one-to-one but the model is easily extensible to have many-to-many. To increase scalability for large-scale systems, message queues can be stored on dedicated message servers that may transform messages in transit and support multi-hop message routing.

A disadvantage of the message-passing model is that it is harder to integrate with a programming language because interaction with the middleware happens through external application programming interface (API) calls and is not part of the language itself, as is the case for remote method invocation. This means that messages cannot be statically type-checked by the compiler, but some messaging middlewares support dynamic type-checking of message content.

3.3.3 Publish/subscribe middleware

Publish/subscribe communication is a more scalable paradigm that addresses the shortcomings of request/reply communication by supporting many-to-many interaction among entities. It is an efficient way to disseminate data to a large number of clients depending on their interest. It ensures that information is delivered to the right place at the right time.

In the publish/subscribe model, there are two different types of clients: (i) information producers that publish data in form of events, and (ii) information consumers that receive them. Consumers describe the kind of events that they want to receive through an event subscription. Events coming from producers will be delivered to all interested consumers with matching interests.

A *publish/subscribe system* (figure 3.4) implements the publish/subscribe model and provides an event service to applications by storing and managing event subscriptions and asynchronously disseminating events.

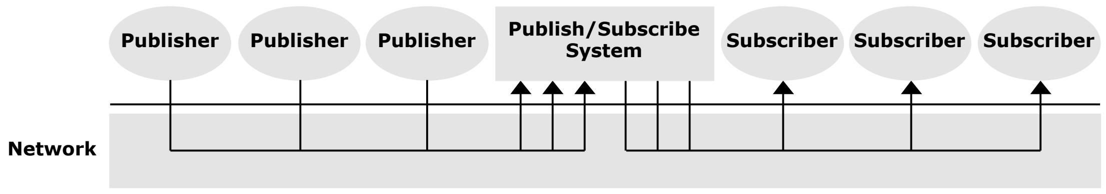

Figure 3.4 Components of a publish/subscribe system

The *event subscription mechanisms* that a publish/subscribe system implements can be classified in four main groups:

Channel-based subscription

The *channel-based subscription* is the most simple subscription mechanism. It is based on the use of uniquely identified logical channels. An event channel is useful as intermediary among the event service and one or more producers or consumers. A producer explicitly signals events to the event service through one or more channels. The event service recognizes the events through the identificator of the channel which notifies them. Consumers subscribe to specified channels to receive events which interest them. The event service delivers the events to all consumers who have subscribed to this channel.

Events are delivered efficiently since no filtering is done by the service, then, from a functional point of view, the event services that implements channel-based subscription are equivalent to reliable diffusion mechanisms.

The lack of flexibility and expressiveness of this mechanism leads to overlay broad subscriptions, resulting in high network traffic and requiring additional client-side filtering. This subscription mechanism is used by the CORBA Event Service [OMG95], Enterprise JavaBeans and the Publisher/listener pattern in Java AWT.

Topic-based subscription

The *topic-based subscription*, also known as the *subject-based subscription*, is a refinement over the channel-based subscription. Consumer subscriptions can express interest about a set of topics through an expression. Expressions are character chain lists that can be filtered. Consequently, a subscription can concern one or more topics and a notification can correspond to several subscriptions. Filters are described with regular expressions. The event service evaluates the expressions matching the subscriptions to the received events (with respect to topic) and filtering them.

However, in large-based applications event consumers benefit from a more precise specification of interest than just topic names. Subdividing the event space into topics has the disadvantage that it is not flexible and may lead to consumer having to filter events coming from general topics. Some topic-based event services structure the topic space by hierarchical topics. Wildcards in topics names are often supported to allow subscription to several topics simultaneously. Most commercial subscription mechanism implementations are topic-based, such as Java JMS [Sun01].

Content-based subscription

In the *content-based subscription* mechanism, the structure of an event subscription can be any function over the content of an event – it is not restricted. It extends the filter domain to all the event content, a content-based subscription usually depends on the structure of the events. A content-based subscription is often expressed in a subscription language that specifies a filter expression over events. This subscription language is provided by the event service. This subscription mechanism introduces a trade-off between scalability and expressiveness.

Type-based subscription

The work on *type-based subscription* recognizes the need for better integration of event-based communication with mainstream, object oriented programming languages. Events are programming language objects with fields and methods. Subscriptions specify the programming language type of objects that a consumer is interested in, while observing the subtyping relation.

The advantages of type-based subscription mechanism are that the object-oriented principle of encapsulation is not violated because fields in an object are only accessed via its methods, and no separate subscription language is necessary to express interest in events.

An important feature of the publish/subscribe model is that clients are decoupled. A producer does not need to know all consumers that receive data and, consumers do not know the identity of producers that send data to them. Thus, this loose coupling removes dependencies between clients so that the publish/subscribe model is a scalable communication paradigm for large-scale systems.

The *publish/subscribe middleware* complements the scalable and efficient implementation of the publish/subscribe model with additional functionality.

3.3.4 Tuple spaces

A *tuple space* is a shared collection of ordered data tuples that supports three operations: `read` to read a tuple from the tuple space that matches a template, `out` to read and remove a tuple, and `in` to insert a new tuple into the space.

The tuple space concept was originally proposed as part of the Linda coordination language. A tuple space provides a persistent and shared memory (or space), accessed through an API that allows distributed processes to read, write and remove information represented as tuples (type, attribute, value pairs). In the Linda system, tuples can be concurrently read or removed from the space by different processes. In this programming paradigm, concurrency and interoperability mechanisms can be easily implemented, as well as more advanced communication and coordination mechanisms such as distributed queues and locks. Queries on the tuple space can also be defined, allowing interested processes (subscribers) to retrieve existing tuples or to block until tuples matching this query are added to the space. Those queries are type-based (also known as templates or anti-tuples). A template matches a tuple if both have an equal number of fields and each template field matches the corresponding tuple field. Those two mechanisms combined provide a powerful publish/subscribe semantics to this model.

Current examples of systems that implement this model are IBM TSpaces and JavaSpaces from Sun. IBM TSpaces, for example, combines the traditional Linda API with DBMS features such as transactional semantics, database indexing, dynamically modified behavior (download and installation of new data types to tuples and new operators); transactional semantics allowing, for example roll-back of operations, access control, and event notification (applications can register to be notified whenever the tuple space is changed.

Tuple spaces fill the gap between message-oriented middleware and database systems. For not adhering to a fixed database schema, it is more flexible, since it does not restrict the format of the tuples stored or the types of data the tuple space contains. At the same time, they provide all the asynchronicity and anonymity of publish/subscribe middleware, working as an inter-process communication and the basis for parallel programming and artificial intelligence techniques. For such characteristic, it is becoming more and more popular in mobile and ubiquitous computing applications.

After having reviewed middlewares and the communication models that they implement, we define in this work an **event-based middleware** as a middleware for large-scale distributed systems that implements the publish/subscribe communication between components, providing a scalable and efficient event service. It addresses traditional middleware requirements, such as usability, manageability, interoperability and extensibility.

3.4 Conclusion

This chapter presented a briefly review of the terms used synonymously to the event service term which allowed us to give an event service definition in general terms. Also, this chapter presented event-based middlewares services in order to analyze its advantages and disadvantages with respect to the implemented communication model. The study of event-based middleware services allowed us to give an event-based middleware definition and to determine the features for an event-based framework defined as an event-based middleware framework.