State of the Art Review of Distributed Event Models

René Meier

Inf. Ing. HTL, M.Sc.,

Department of Computer Science,

University of Dublin, Trinity College,

Dublin, Ireland.

March 2000.

Version: 1.0

CONTENTS

1. INTRODUCTION 1
1.1 Motivation 1
1.2 Roadmap 2
1.3 Event-based Communication 2
1.4 Event Model Features 4
1.4.1 Typed Events 4
1.4.2 Event Filters 5
1.4.3 Quality of Service Constraints 5
1.4.4 Real-Time Constraints 6
1.4.5 Scalability 7

2. OMG CORBA EVENT MODELS 10
2.1 CORBA Event Service 10
2.1.1 The Event Channel Architecture 11
2.1.2 Generic and Typed Event Channels 12
2.2 CORBA Notification Service 12
2.2.1 The Notification Channel Architecture 13
2.2.2 Administrative Capabilities 14
2.2.3 Filtering Capabilities 14
2.2.4 Quality of Service Capabilities 15
2.2.5 Structured Event Communication 16
2.3 Summary 17

3. JAVA EVENT MODELS 19
3.1 Java AWT: Delegation Event Model 19
3.1.1 Delegation Event Model Architecture 19
3.1.2 Event Adapters 21
3.2 Java Distributed Event Model 22
3.2.1 Distributed Event Model Architecture 22
3.2.2 Third Party Objects 23
3.3 Summary 24

4. THE CAMBRIDGE EVENT MODEL 25
4.1 Cambridge Event Model Architecture 25
4.2 Event Filtering 27
4.3 Event Composition 28
4.4 Summary 28

5. ECO 29

5.1 ECO Event Model Architecture 29
5.1.1 Events, Constraints and Objects 29
5.1.2 ECO's Event Application Interface........ 30
5.2 Notify Constrains 31
5.3 Pre- and Post-Constraints 31
5.4 Zones........ 31
5.5 Summary 32

6. JEDI........ 34
6.1 JEDI Event Model Architecture 34
6.2 Event Pattern........ 36
6.3 Reactive Objects and Mobility 37
6.4 Summary 37

7. SIENA........ 39
7.1 The SIENA Event Model Architecture 39
7.1.1 Operational Semantics 40
7.1.2 Server Topologies........ 41
7.2 Event Filter and Pattern........ 43
7.3 Mobility 43
7.4 Summary 44

8. EVENT MODELS BASED ON THE OMG CORBA EVENT SERVICE........ 45
8.1 Real-Time Event Service........ 45
8.1.1 RT Event Service Architecture........ 45
8.1.1.1 QoS Parameter........ 46
8.1.1.2 Event Filter........ 46
8.1.1.3 Periodic Event Processing........ 47
8.1.2 Summary........ 47
8.2 CONCHA 48

9. SYNTHESIS 49
9.1 Event Model Architectures 49
9.2 Event Model Features 54
9.3 Conclusion........ 56

10. REFERENCES........ 58

1. INTRODUCTION

This review deals with an investigation into the architecture and features of distributed communication models that are based on the event paradigm.

A variety of event-based communication models are used in different application domains, including real-time, mobile and large-scale environments. To fulfil the needs of their application domain, event models differ in their architecture and in the features they support. Although several event models are available supporting a variety of features, non of them suffices the requirements of a event-based system running in a large-scale environment that supports application component mobility and real-time event delivery.

1.1 Motivation

In the traditional *client/server* [Maf00] computing model, the means of communication is typically synchronous and logically one-to-one. Clients invoke a method on the remote server and wait for the response to return. This requires that client and server need to have some knowledge[1] of each other. With the use of mobile or large-scale distributed systems, the need for asynchronous, anonymous one-to-many communication pattern arises. *Event Models* are application independent infrastructures that allow event-based communication, where an *Event Supplier* asynchronously communicates *Event Data* to a group of *Event Consumers*, ideally without knowledge of the number and location of the event consumers. Figure 1.1 shows the difference between the traditional client/server computing model and the event-based computing model.

Distributed Event Models are used in a number of application domains, including real-time, mobile and large-scale environments. To fulfil the needs of their application domain, event models differ in their architecture and in the features they provide. In this paper, we review a number of distributed event models and one centralised event model from different application domains. To our knowledge, this has not been presented in this form yet. Thus, each event model is reviewed by presenting background information, such as its history, terminology and application domain, and by describing its architecture and the supported features. By doing this, we outline the approaches taken to provide event-based communication in different application domains. This enables us to reflect on the architecture and the features needed to support event-based communication in our application domain,

[1] The client needs to know the server on which to invoke the method and the server needs to know the client to which to return the response.

which requires application component mobility and real-time event delivery in a large-scale environment.

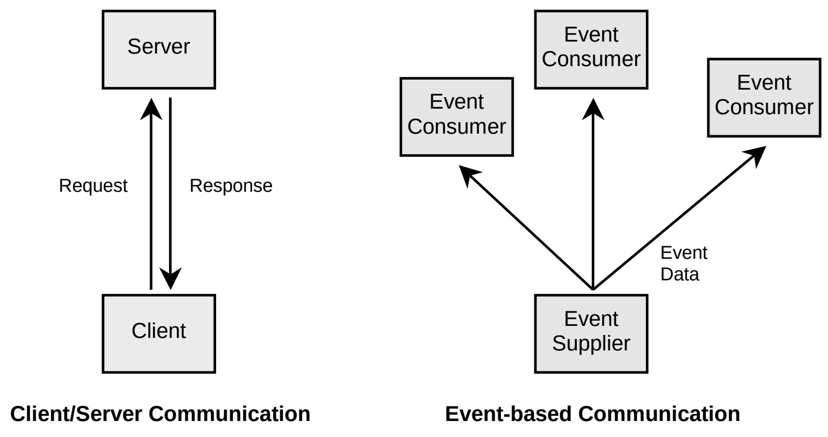


Fig. 1.1. Client/Server and Event-based Computing Models.

1.2 Roadmap

In the remainder of this Section, we introduce the general terminology of event models and the most important event model features, which are supported by several of the reviewed event models.

We then review the chosen event models in the following Sections. Each reviewed model is introduced by presenting general background information, such as the event model's history, the event model's application domain and the event model's specific terminology. The model is described by outlining its architecture and by determining the features supported.

Finally, we conclude this review by summarising and comparing the event models in Section 9.

1.3 Event-based Communication

In order to provide event-based communication pattern, an event model defines two roles for application components. The role of an event supplier which produces event data (events) and the role of an event consumer which processes event data. The event model's infrastructure allows event suppliers[2] to anonymously communicate events to a group of event consumers. To receive events, event consumers have to subscribe to the events they

[2] An event-based application may include several event suppliers, each of which communicating events to a group of event consumers.

are interested in; they are said to register interest in events. As shown in Figure 1.2, once consumers have subscribed to events, they receive all events that are produced within the scope of the event model until they unsubscribe (de-register).

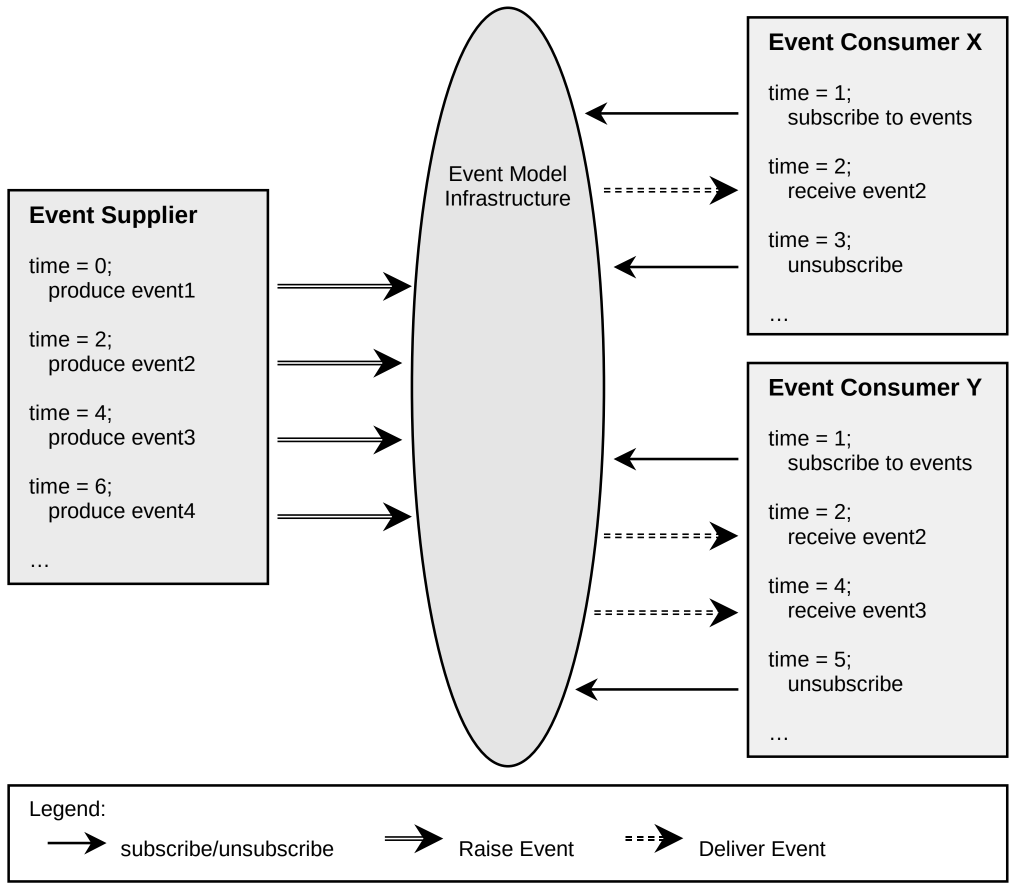


Fig. 1.2. Event-based Communication.

Distributed event models, as opposed to centralised event models where suppliers and consumers are located in the same address space, allow suppliers and consumers to be located on different physical machines. Supplier and consumer are connected via a network, through which event communication takes place.

1.4 Event Model Features

Each of the Sections that reviews an event model includes:

- An introduction to the event model. The history and background to the event model are presented along with a description of the specific terminology of the event model.
- A description of the architecture of the event model. The components of the event model are introduced as well as the relationships among them.
- A presentation of the features of the event model. Special capabilities, such as *Event Filters*, *Real-Time Constraints* and *Quality of Service (QoS) Properties*, featured by the event model are presented.
- An event model summary.

In the remainder of this Section we present a general introduction of the most important event model features and issues. These features represent valuable capabilities of event models and thus, are addressed, although by different means, by many of the reviewed event models.

1.4.1 Typed Events

The main purpose of an event model is to provide a means for delivering event messages (events) from an event producer to one or more event consumers. The structure of the propagated events varies considerably depending on the chosen model and the intended application domain. Generally, events are said to be either *generic or typed*. The information that describes a generic event is a data blob without an expressive structure of the likes of type *any,* e.g. as supported by the Java programming language. Typed events on the other hand, provide a well-defined and expressive data structure into which a wide variety of event types can be mapped.

The structure of typed events ranges from simple to complex, usually consisting of a name (instance name) and additional, user-definable data (parameters). Examples of typed events structures are:

- A single string representing the event name, without any additional parameters
- A set of ordered strings, the first string representing the event name and the remaining strings representing the event parameters
- A single string representing the event name preceding a set of ordered numbers representing the event parameters

- A set of ordered attributes in which each attribute is a triple of name, type and value. A predefined set of types is available. The set of available types may or may not be limited to a subset of the types supported by the programming language used.
- An programming language specific object including a set of attributes a described above and a set of methods

It may be argued that simple forms of typed events are merely generic and not typed. However, we consider events that have some sort of structure recognised by the event model as typed. Significantly, typed events are not only well-defined and expressive, but also enable the utilisation of event filters as introduced in the next Section.

1.4.2 Event Filters

An event-based system may consist of a number of suppliers, all of which produce events that may contain different information. Thus, the number of events propagated in an event based system may be quite large. However, a particular consumer may only be interested in a subset of the events propagated in the system. *Event filters* are a means to control the propagation of events. Filters enable a particular consumer to subscribe to the exact set of events it is interested in receiving[3]. Consumers define filters by specifying the type and parameters of the events they are interested in. Filters are passed to the event model infrastructure during subscription. Before events are propagated, they are matched against the filters and are only delivered to consumers that are interested in them, i.e. the matching produced a positive result. The expressiveness of event filters depends directly on the expressiveness of the event types supported by the event model.

In this Section, the general idea of event filters was introduced. Filtering capability is an important event model feature and is therefore supported by many event models. There are several different approaches for defining filters used in a event system and for passing them to the event model infrastructure. The approaches taken by the reviewed event models will be discussed in the corresponding Sections.

1.4.3 Quality of Service Constraints

A system's *non-functional requirements* include *real-time* and *Quality of Service* (QoS) constraints, the former being a subset of the latter. However, in this document we discuss

[3] An event that is delivered uses network bandwidth and CPU processing power on the consumer side. It is therefore desirable to prevent the delivery of unwanted events.

real-time and QoS constraints separately, thus when referring to QoS we exclude real-time issues.

The requirements of a system include functional or non-functional requirements. Definitions of functional and non-functional requirements are:

Functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do [Som95, p.118].

Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints in the development process, standards and so on. .. Examples are reliability, response time and store occupancy [Som95, p.119].

The QoS characteristics of an event system may be influenced by a variety of QoS constraints. Many QoS properties are directly built into the event system, without allowing the application to influence them. Other QoS properties are configurable by the application, hence allowing the application to control the QoS characteristics of the event system and the event delivery. We say an event model features QoS when applications can access and configure the system's QoS properties according to their needs.

[Som95] identifies reliability, response time and memory management as QoS constraints. Response time is related to an event model's real-time constraints which is discussed below. Examples of an event model's reliability properties are event reliability and connection reliability. Memory management is addressed by properties like maximum number of consumers and producers, queue sizes, order policies and discard policies.

1.4.4 Real-Time Constraints

In order to describe the problem domain, we firstly present a simple *real-time system* description and then state a more definite description of a real-time system as defined in the *Oxford Dictionary of Computing*:

A real-time system is any information processing activity which has to respond to externally generated input stimuli within a finite and specified time [BW96, p.2].

A system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

A real-time system that features event-based communication requires the involved event model itself provide real-time guarantees. As with other non-functional requirements, namely the QoS constraints discussed above, we say an event model features real-time when applications can access and configure the system's real-time properties according to their needs.

Although a distributed event model might not be able to guarantee low latency[4], it may include real-time constraints that enable predictions on the event delivery behaviour and duration, thus allows deterministic event communication. A common real-time feature is a priority assigned to an event that allows a dispatcher to pre-empt the delivery of an event in order to deliver an event that has a higher priority. Other real-time requirements may include delivery deadlines, e.g. earliest and latest delivery time, and delivery timeouts.

1.4.5 Scalability

[CDK94] states that distributed systems operate effectively and efficiently at many different scales, the smallest practicable distributed system consisting of two clients and a server, larger ones consisting of several hundred clients and many servers. Several local-area networks are often interconnected forming wide-area networks that may contain thousands of clients and servers, enabling resources to be shared between them. Hence, distributed systems (of any scale) are typically required to perform in an environment where its scale dynamically changes over time.

[4] Latency in distributed communication depends on the available bandwidth and the quality of the underlying network. Both of which depend on the topology and the traffic of the network and may therefore change over time. Thus, "low" latency in a distributed event model is relative.

The term *scalability* is used to describe a distributed system's behaviour when changing its scale. There is no generally accepted scientific definition of scalability, with textbooks tending to rely on examples to explain it and generally providing rather vague definitions. [CDK94] uses the London telephone system that run out of numbers and a file server that becomes a performance bottleneck with increasing number of accesses as examples of lack of scalability. However, the following definition is also provided:

> *The system and application software should not need to change when the scale of the system increases. .. The demand for scalability in distributed systems has led to a design philosophy in which no single resource – hardware or software – is assumed to be in restricted supply. Rather, as the demand for a resource grows, it should be possible to extend the system to meet it [CDK94, p.18].*

[Mul93] describes the evolution of the *Andrew File System* (AFS) to present their design strategy for scalability, but also provides a more specific definition:

> *A scalable distributed system is one that can easily cope with the addition of users and sites, and whose growth involves minimal expense, performance degradation, and administrative complexity [Mul93, p.363].*

From those examples and definitions, it can be observed that the scalability of a systems depends on several factors. With regards to event models, the parameters that may vary are:

- Number of users
- Number of entities (i.e. event consumers and producers)
- Number of mediators (i.e. zero or more)
- Number of nodes (i.e. physical machines)
- Number of activities (i.e. communications)

In principle, these parameters are independent, but in practice they are likely to increase consecutively. For example, increasing the number of entities is likely to amplify the number of activities. As suggested in the definitions above, an event system that scales well must facilitate increasing the number of entities without other entities or mediators becoming a performance bottleneck. Another requirements related to scalability could state that application software on entities should not need to change when the scale of the system increases. Generally, it can be observed that the importance of scalability increases with the complexity of a distributed event system.

2. OMG CORBA EVENT MODELS

The Common Object Request Broker Architecture (CORBA) is an open standard for *object* management specified by the Object Management Group (OMG). The architecture uses Object Request Brokers (ORBs) as the middleware to allow application components, which are objects, to communicate with each other across boundaries such as the network, different operating systems and different programming languages. To extend the ORB core capabilities, the CORBA 2.0 specification [Gro95a] defines a wide range of general-purpose services, one of which is the *CORBA Event Service* [Gro95b]. This service allows application components to communicate with events in addition to the means of communication provided by the bare ORB. In order to extend the Event Service with filtering and quality of service (QoS) capabilities, a *Request For Proposal* [Gro96] to define a CORBA Notification Service was issued by the OMG in 1996. The submitted joint revised submission [Cea98] was accepted by the end of 1998. The event models used in both services include a mediator through which events or notifications respectively are distributed and can be characterised as extremely general, to cover the needs of different business domains, and quite complex due to the large number of interfaces. We review each of these event models in the following sections.

2.1 CORBA Event Service

The *CORBA Event Service* [Gro95b] supports an event model that defines two roles for objects. The role of a *supplier* which produces event data and the role of a *consumer* which processes event data. Suppliers and consumers are collectively addressed as *clients*. There are two approaches to initiate event communication between suppliers and consumers called *push-model* and *pull-model*. They allow either supplier or consumer to initiate communication. The push-model allows the supplier to initiate the transfer of event data to consumers and the pull-model allows a consumer to request the event data from a supplier. In the former model, the supplier initiates event communication and in the latter model the consumer initiates event communication by polling the supplier for event data. A consumer may use either a blocking *(pull)* or a non-blocking *(try_pull)* mechanism for requesting event data. In a simple scenario, where consumers and suppliers invoke directly on each others' interface methods to exchange event data, clients are required to know each others' object references. Therefore, event communication cannot be anonymous. The CORBA Event Service also supports the role of an *event channel* that enables anonymous event communication.

2.1.1 The Event Channel Architecture

The *Event Channel* is an intervening object between suppliers and consumers, that decouples the communication between suppliers and consumers and allows multiple suppliers to communicate with multiple consumers anonymously. An event channel therefore acts as both supplier and consumer of event data. It can communicate with suppliers and consumers using a mix of communication models. E.g. the consumer uses the pull-model to obtain the event data from the event channel, whereas the supplier used the push-model to transmit the event data to the event channel.

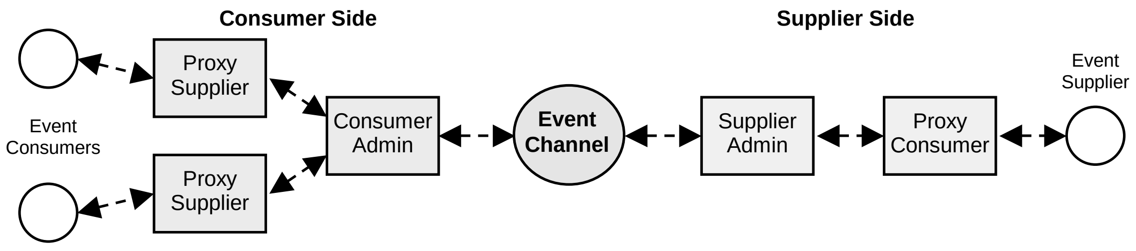


Fig. 2.1. CORBA Event Channel Overview.

An overview of the Event Channel architecture is shown in Figure 2.1. The architecture includes the event channel as well as a group administration and several proxy interfaces on each of the two sides of the channel. The *supplier side* includes all interfaces used by suppliers and the *consumer side* includes all interfaces used by consumers. The group administration interfaces, the *SupplierAdmin* and the *ConsumerAdmin*, act as *factory objects*[5] for adding consumers and suppliers respectively. The operations for adding consumers return *proxy suppliers*, the operations for adding suppliers return *proxy consumers*. The obtained proxy interfaces are then used to connect to the push/pull supplier/consumer interfaces.
The *ConsumerAdmin* interface defines two operations to obtain *proxy suppliers*.

```
Interface ConsumerAdmin {
   ProxyPushSupplier  obtain_push_supplier();
   ProxyPullSupplier  obtain_pull_supplier();
}
```

The *SupplierAdmin* interface defines two operations to obtain *proxy consumers* similar to the *ConsumerAdmin*.

[5] A factory object is an object that instantiates new objects.

```
Interface SupplierAdmin {
  ProxyPushConsumer   obtain_push_consumer();
  ProxyPullConsumer   obtain_pull_consumer();
}
```

An event channel may also include build-in filtering capabilities and quality of service (QoS) capabilities such as event priority or event delivery guaranty. Such features are not defined in the event service specification and depend therefore on the implementation of a particular vendor. To combine the features of different event channels, this architecture supports the composition of event channels. That is, one event channel may consume the events supplied by another. This solution is sufficient, but as a side-effect it increases latency and constrains interoperability. The CORBA event service specification does not include a policy for clients to locate event channels. There are several means for clients to obtain the event channel's object reference, including making use of a naming service or looking it up in a locally stored file or table.

2.1.2 Generic and Typed Event Channels

An event channel can be implemented as either *generic* or *typed* event channel. Both work as described above, but the generic event channel only supports generic event communication, whereas the typed event channel supports both typed and generic event communication. The information that describes a generic event is of data type *any*. Thus, it is flexible enough to cope with the needs of different applications. Typed event data is described in OMG *Interface Definition Language* (IDL). Suppliers call operations on consumers using a mutually agreed IDL interface. A generic event channel can handle events supplied and consumed in any combination of the forms push/pull and generic/typed. Event data supplied in a typed form can be consumed in a generic form and vice versa[6]. The authors of [Cea98] state that many users have found typed event communication difficult to understand and implementers have found it particularly difficult to deal with.

2.2 CORBA Notification Service

In December 1996, the OMG issued a *Request For Proposal* [Gro96] to define a *CORBA Notification Service*. The notification service extends the existing *CORBA Event Service* [Gro95b] described in the last section adding new capabilities such as filtering and quality of

[6] Thus, the event channel must support the conversion of a typed event into a generic event, i.e. IDL into any, and vice versa. Doing this requires an understanding of the generic event channel interfaces and depends on the particular implementation of the event channel.

service (QoS). The finally submitted joint revised submission [Cea98], written by a group of organisations including Borland International, International Business Machines Corporation, Iona Technologies Plc. and NEC Corporation, was accepted by the OMG by the end of 1998.

2.2.1 The Notification Channel Architecture

The main design goal of the *CORBA Notification Service* was to directly extend the existing *CORBA Event Service* enhancing it with important features. This is achieved by inheriting the interfaces of the *Notification Channel* directly from those defined be the *Event Channel* allowing for interoperability between basics event service clients and notification service clients. The notification channel therefore contains all the interfaces and functionality supported by the event channel. Additionally, the notification channel supports multiple instances of the *ConsumerAdmin* and the *SupplierAdmin* interfaces as depicted in Figure 2.2.

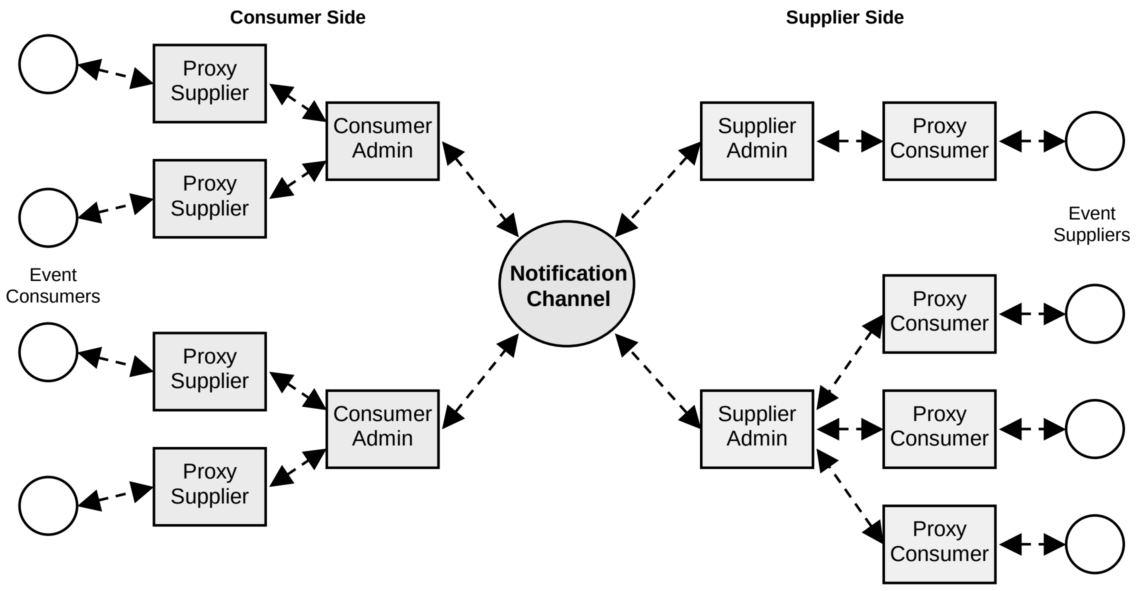


Fig. 2.2. CORBA Notification Channel Overview.

This symmetric architecture[7] of the notification channel supports capabilities such as administration, filtering, quality of service (QoS) and structured event communication, which are described in the following sections. It also includes an optional event type repository that may be used to perform run-time type checking of the event properties or to discover the structure of types of events.

[7] The supplier side and the consumer side of the notification channel are symmetric.

2.2.2 Administrative Capabilities

At creation time, a number of administrative properties can be set on a notification channel to help in managing memory space. Those properties include the maximum number of suppliers and consumers, as well as the maximum number of events that will be queued by the notification channel before the notification channel begins discarding them.

Starting from any object, clients may discover all objects that comprise a notification channel. Whenever a factory object creates a new object, a unique numeric identifier is assigned to it. Factories also support operations to query the identifiers created by it, to convert them into object references and to return the reference of its parent object. These identifiers are unique among all objects created by a particular notification channel, but unlike *Interoperable Object References* (IOR), they are not globally unique. Globally unique identifiers are required by systems that use federated Notification Channels.

The *offer_change* and *subscription_change* operation are available on interfaces supported by notification channels. The former is used by suppliers to indicate changes in the supplied event types, the latter is used by consumers to inform suppliers of the event types they require. Consequently, suppliers know which events are being consumed and which are not. This knowledge can be used to produce notifications on demand and therefore optimise network traffic.

2.2.3 Filtering Capabilities

The most important enhancement introduced by the notification service is the support of filter objects. Filter objects can be assigned to individual proxy objects (proxy supplier and proxy consumer), to admin objects (supplier admin and consumer admin) and to the notification channel itself. This results in hierarchical filtering on each side[8] of the notification channel, where the filter object assigned to an admin object applies to all its proxy objects and therefore to its clients, whereas the filter object assigned to a proxy object applies only to the group of clients connected to itself. Filter objects encapsulate a set of *constraints* which are text strings containing a boolean filtering expression. All filter objects belonging to an event, i.e. defined at proxy, admin and channel level, are evaluated at the proxy level when an event is received, resulting in a decision whether or not to forward an event. The filter object shown below forwards events of domain type “Finance” and type name “ExchangeRateUpdate” or of domain type “Health” and type name “PulseLow” that contain an attribute “office” whose value

[8] As opposed to other event models, filtering can take place on the supplier side as well as on the consumer side.

is equal to seven. The domain type, type name and the attributes of events are further described in section 2.2.5.

```
(($domain_type == “Finance” and $type_name == “ExchangeRateUpdate”)
or ($domain_type == “Health” and $type_name == “PulseLow”))
and office == 7
```

The syntax of the constraint expression conforms to a constraint grammar defined in a *constraint language*. The notification service specifies a default constraint language that is an extension of the Trader Constraint Language [Gro97]. Other constraint languages can be defined and can co-exist in the notification service.

The notification service defines two types of filter objects. The first, called *forwarding filter*, affects the event forwarding decision as described above. The second, called *mapping filter*, influences the delivery policy applied to an event. Mapping filter objects change the characteristics of an event. Constraints can de defined which, when matched, dynamically assign a different priority to an event or set a changed expiration time, i.e. lifeline.

2.2.4 Quality of Service Capabilities

Another extension to the event service introduced by the notification service is the definition of interfaces for influencing the Quality of Service (QoS) characteristics of event delivery. A variety of QoS properties, such as reliability and priority, may be set to control the delivery characteristics of event messages. Operations for setting QoS properties are specified at various levels of scope throughout the notification service architecture. QoS properties can be accessed on:

- The notification channel (per-channel)
- Supplier and consumer group administration (per-admin)
- Proxy suppliers and consumers (per-proxy)
- Individual event messages (per-event)

Table 2.1. shows the supported QoS properties and their level of scope. A detailed discussion of these QoS properties may be found in [Cea98].

QoS property	Per-Event	Per-Proxy	Per-Admin	Per-Channel
EventReliability	✓			✓
ConnectionReliability		✓	✓	✓
Priority	✓	✓	✓	✓
StartTime	✓			
StopTime	✓			
Timeout	✓	✓	✓	✓
StartTimeSupported		✓	✓	✓
StopTimeSupported		✓	✓	✓
MaxEventPerConsumer		✓	✓	✓
OrderPolicy		✓	✓	✓
DiscardPolicy		✓	✓	✓
MaximumBatchSize		✓	✓	✓
PacingInterval		✓	✓	✓

Tab. 2.1. Notification Service QoS properties.

The list of supported QoS properties, along with the levels of scope where settings can be made, provides a very flexible QoS configuration of a notification channel. However, meaningless QoS settings are not prevented. An event message is transmitted through three conceptual points; the supplier side, the consumer side and the notification channel. End-to-end QoS requirements can only be guaranteed with the co-operation of all three parties. Although event delivery can be assured by setting persistent reliability and assign high priority and long lifetime to a message, no predictions can be made regarding delivery latency.

2.2.5 Structured Event Communication

To provide an easy-to-use but strongly typed event communication an new event message style, the structured event message, is introduced by the notification service. Structured events provide a well-defined data structure into which a wide variety of event types can be mapped. As depicted in Figure 2.3, structured events compose of a header and a body.

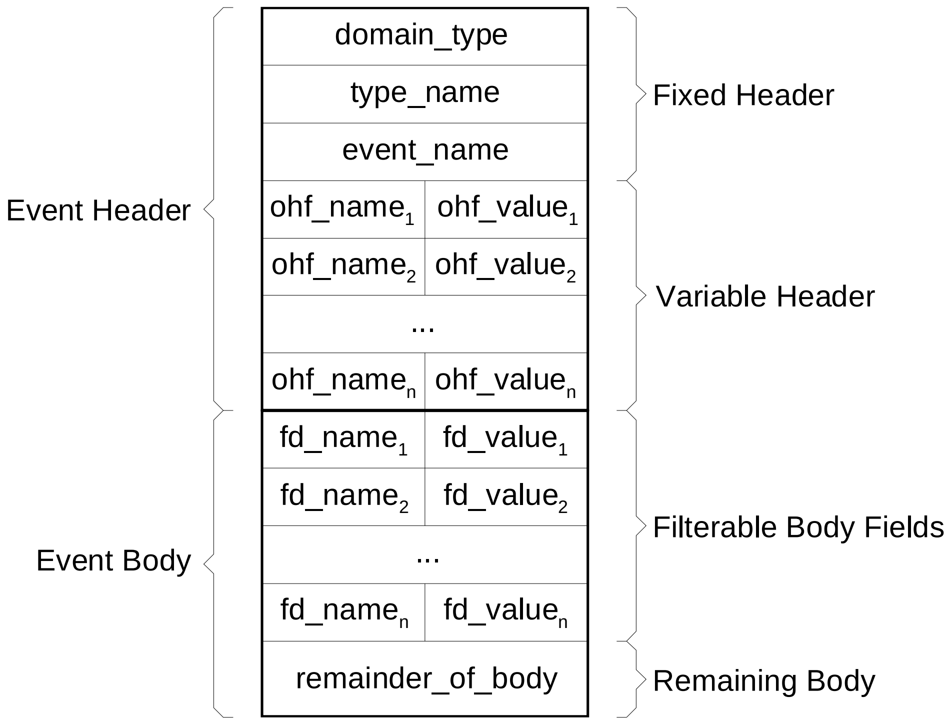


Fig. 2.3. The General Structure of a Structured Event.

The fixed event header contains event type and instance identifier and the variable event header may contain QoS requirement name-value pairs, e.g. message priority. The event body contains the filterable event data also name-value pairs and a remainder that may be used to transmit additional large data blobs, e.g. files.

2.3 Summary

Both, the *CORBA Event Service* and the *CORBA Notification Service* specify an event model that defines the role of *event suppliers*, *event consumers* and an *event channel* or a *notification channel* respectively. Event communication may either be initiated directly between suppliers and consumers or may be delivered via a channel. The channel acts as a *mediator* between event suppliers and consumers, thus enables anonymous event commination. Both CORBA event services feature the *push-model* and the *pull-model* for delivering event data, whereas the other event models reviewed in this document support the push-model only. Event data is propagated in the form of generic events or typed events and, with the notification service, in the form of structured events. The notification service directly extends the event service by enhancing it with *Filtering Capabilities*, *QoS Properties* and *Administrative Features*.

Both event models can be characterised as extremely general, covering the needs of a variety of business domains, including telecommunications, finance, and medical and consequently complex due to the large number of interfaces and properties. Neither of the event models addresses *federated event communication*. To overcome this, the OMG issued a *Request For Proposals* for a service that provides the ability to configure, manage, and control a group of event channels connected together in a topology of arbitrary complexity [Gro98]. The request for proposal was answered by a consortium that submitted a specification for management of event domains in December 1999, the vote for the suggested architecture was completed in March 2000[9].

[9] The decision was not available for publication yet.

3. JAVA EVENT MODELS

Java is an *object-oriented programming language* developed by Sun, that was formally announced at a major conference in May 1995. It is suitable for use for the same type of development tasks as C and C++, but without the difficulties and source of bugs common to those languages. Java generated immediate interest in the business community and became increasingly popular because of Internet-related development, such as the World Wide Web.

The Java architecture includes a *Delegation Event Model* [SM97] and a *Distributed Event Model* [SM98]. The delegation event model is used for event communication within a single *Java Virtual Machine* (JVM) for small centralised applications such as *Graphical User Interfaces* (GUIs), whereas the distributed event model enables event communication between objects located in different JVM's which may be distributed across virtual, and even physical, machines. Both event models are reviewed in the following Sections.

3.1 Java AWT: Delegation Event Model

As part of the *Java Foundation Classes* (JFC), a class library, the *Abstract Window Toolkit* (AWT) is the standard *Application Program Interface* (API) for providing *Graphical User Interfaces* (GUIs) for Java applications. The *Delegation Event Model* [SM97] was introduced in AWT by *Java Development Kit* (JDK) 1.1. to catch and process GUI events. It replaces the event processing model in version 1.0 of the AWT that is based on inheritance. The delegation event model provides a more robust framework to support more complex Java applications than the inheritance-based event model and supports filtering of events. A number of other Java components also make use of the delegation event model. It has been adopted by the *JavaBeans* component architecture for general event processing and has been extended by the *Swing* component set, a new GUI toolkit. Furthermore, it is supported by the *EmbeddedJava* and *PersonalJava* application environments. The former is used for building embedded applications with dedicated functionality and strictly limited memory. The latter is used for building network-connected applications for consumer devices for home, office and mobile use.

3.1.1 Delegation Event Model Architecture

The delegation event model is the only centralised event model we review. Event suppliers are called *Event Sources* and event consumers are called *Event Listeners*. An event source, typically a GUI component, is said to *fire* events, when propagating an event of a specific

Event Type. Event source, event listener and event types are encapsulated in two JDK 1.1 interfaces;

```
Java.util.EventObject
Java.util.EventListener
```

Java applications can implement simple event-based communication by implementing methods defined in the two JDK interfaces in the manner described in [SM97]. In order to establish event-based communication, listeners register with the specific event type they are interested in by invoking the set<EventType>Listener or add<EventType>Listener methods on the source and passing the interface reference of their event handler, as shown in Figure 3.1. The source provides a set<EventType>Listener and an add<EventType>Listener per supported event type. To register a single listener, the *single-cast* set<EventType>Listener method is invoked. The *multi-cast* add<EventType>Listener method is invoked to register several listeners that are interested in the same event type on a source. To fire an event, the event source object invokes the handler method on the listener object and passes the instance of an event type.

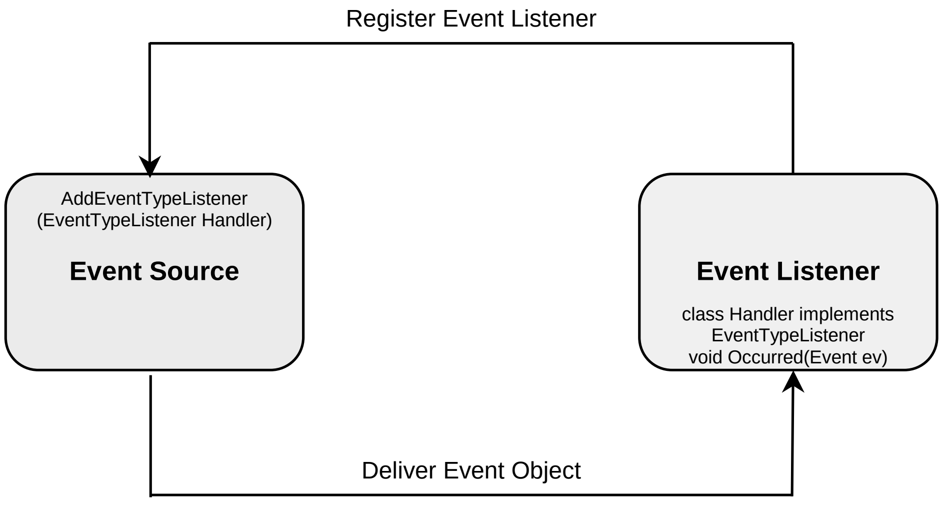


Fig. 3.1. Java Delegation Event Model Overview.

Events are always delivered synchronously, meaning that the listener's event handler is actually executed by the source thread. Hence, a multi-cast source that fires an event must deliver the event to the listeners in sequence.[10]

[10] No guarantees are made about the order in which the events are delivered to a set of registered listeners for a given event on a given source.

The delegation event model couples event sources and event listeners very tightly together, thus there is no possibility for anonymity. The only way to partially de-couple sources and listener is by using so-called Event Adapters.

3.1.2 Event Adapters

A Java application may use an *Event Adapter* component to interpose between event source and event listener. An event adapter can be inserted between source and listener to partially de-couple the event communication between them as shown in Figure 3.2. As opposed to other event models, e.g. CORBA Event/Notification Service, the presented event adapter is asymmetric, i.e. it hides the listener form the source, but not vice versa. Thus, an event adapter introduces some notion of anonymity, i.e. the listener is anonymous but the source is not.

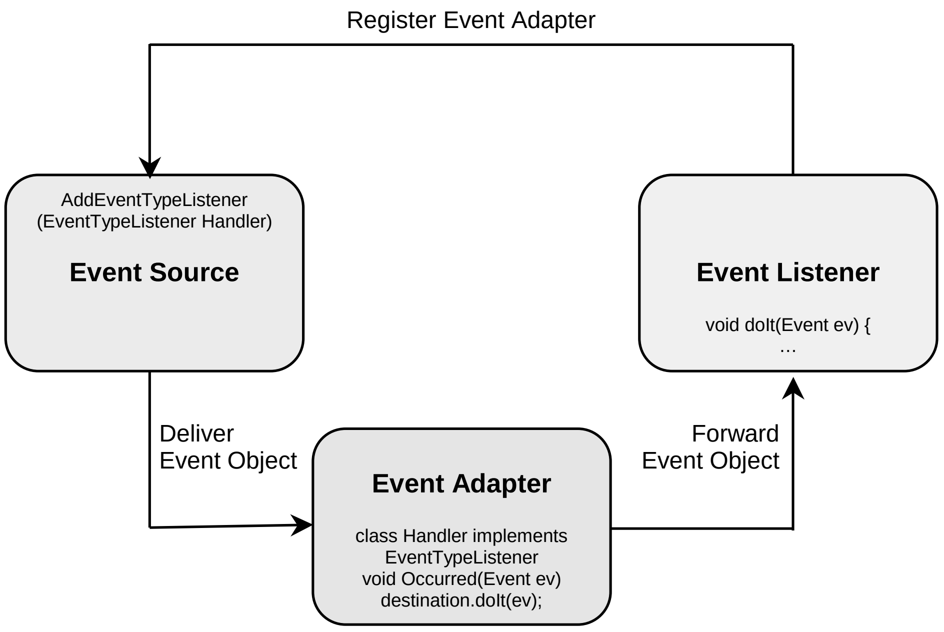


Fig. 3.2. Java Delegation Event Model with an Event Adapter.

The event adapter is an extremely important component of the delegation event model. Besides de-coupling the source from the listener, an event adapter may also introduce additional behaviour on event delivery, including event queuing, event filtering and event demultiplexing. The demultiplexing adapter technique may be used in applications where a given event listener object only implements a given event listener interface once. Thus if the listener registers itself with multiple event sources for the same event, the listener has to determine for itself which source actually emitted a particular event. An adapter may be used to allow the events fired by different event sources to invoke on different methods on the listener object. For example, this technique may be used to have two buttons invoke on two different methods in the same listener.

3.2 Java Distributed Event Model

Java's *Distributed Event Model* [SM98] relies on Java *Remote Method Invocation* (RMI) that enables objects in one JVM to seamlessly invoke methods on objects in a remote JVM. Therefore, it allows an object in one JVM to register interest in the occurrence of some event occurrence in an object in some other JVM. This is the event model adopted by Jini [SM99], a Java technology that provides a simple mechanism which enables devices to plug together to form an communication community without any planning or installation.

3.2.1 Distributed Event Model Architecture

The architecture of Java's distributed event model is similar to its delegation event model. It specifies the interface that is used to send an event and the information that an event must contain and allows various degrees of delivery assurance, different policies of event scheduling and an interposing object that may collect, hold, filter and forward events. Although it provides an example of an interest registration interface, it does not specify such an interface. This is to allow a wide variety of *kinds* of events. Thus, the way these events register interest may vary from object to object.

The entities involved are:

- The Event Generator is the event supplier that generates events and sends them to registered listeners.
- The *Remote Event Listener* is the event consumer that registers interest in some kind of events in some other objects.
- The *Remote Event* is the event object that is passed from generator to listener. It contains information about the occurred event kind, a reference to the generator object, a sequence number to identify the particular event instance and a *handback* object supplied by the listener.

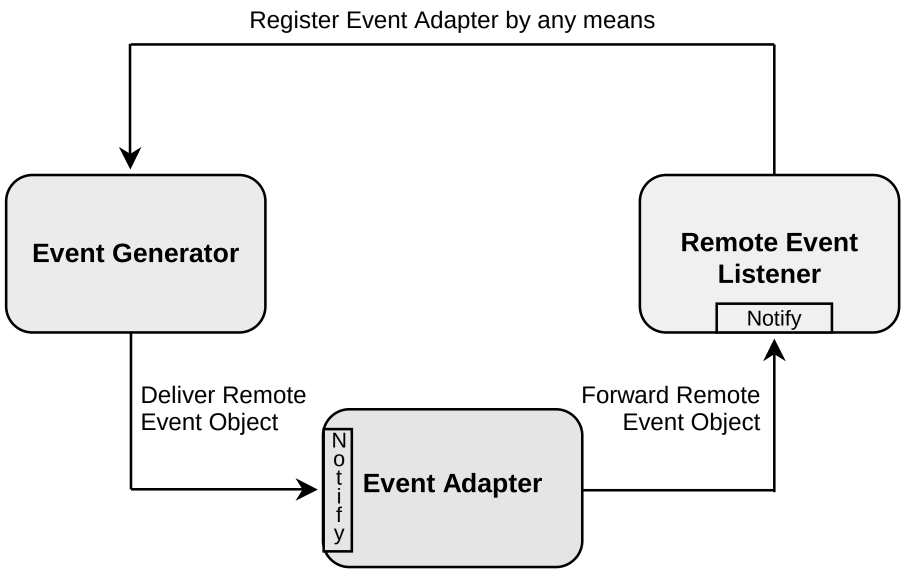
The event listener registers with the event generator that generates the events it is interested in. Such a registration is limited to a given duration using the notation of a *lease*. This is discussed in more details in Java's *Distributed Leasing Specification*. The event generator sends events to the registered listener by calling the listeners *notify* method. As shown here,

```
public interface RemoteEventListener
                     extends Remote, java.util.EventListener {
   void notify(RemoteEvent theEvent
                     throws UnknownEventException, RemoteException;
}
```

the notify method has a single parameter, the event. To know if the call was successful a call to the notify method is synchronous. The sequence number included in the event acts as a hint to the number of event occurrences and is guaranteed to be strictly increasing.

3.2.2 Third Party Objects

As in the Java delegation event model [SM97], the distributed event model may be enhanced with third party objects, or agents, so called *Distributed Event Adapters*. As shown in Figure 3.3, an event adapter interposes (mediates) between event generator and event listener and


Register Event Adapter by any means
Event Generator
Remote Event Listener
Notify
Deliver Remote Event Object
Forward Remote Event Object
Notify
Event Adapter

must support the notify method. Thus it allows enhancements to functionality without changing the basic interfaces.

Fig. 3.3. Java Distributed Event Model with an Event Adapter.

It may act as filter or mailbox and may introduce policies of reliability to the event model. The use of event adapters also introduces a notation of anonymity, but increases event delivery latency. Since event adapter functionality is not specified, it can be used to provide application specific features for a listener or a group of listener while co-existing and co-operating in a system with other listeners and adapters without affecting them.

3.3 Summary

The Java programming language architecture introduces a delegation event model used for processing events in small centralised applications such as GUI's and a distributed event model used for event communication between object located in different JVM's. Both event models are adapted by a number of Java components and environments.

The delegation and the distributed event model have a similar architecture in that they allow the use of adapters. Adapters are objects that interpose between event generator and listener, thus enhance the system with application specific functionality, such as filers or QoS, without loosing compatibility. The delegation event model can easily be included in applications. This is achieved by implementing the specified event listener interface and by registering it with an event source. The distributed event model can be described as a thin event model specification that does not specify an event registration method, but therefore allows event generators, listeners and adapters from different vendors to co-exist and co-operate within a system.

4. THE CAMBRIDGE EVENT MODEL

Jean Bacon, John Bates, Richard Hayton and Ken Moody developed a *Composite Event Model* at the University of Cambridge Computer Laboratory. As described in [BBHM96], the Cambridge event model has a simple architecture, but has a feature that is not commonly supported by event models, that of *event composition*. The approach to event communication taken in the Cambridge event model is described below using an example.

4.1 Cambridge Event Model Architecture

The Cambridge event model, based on the client-server computing paradigm [Maf00], defines the role of an event service[11] that supplies event objects (events) and the role of an event client that receives events. To include the Cambridge event model architecture, applications must extend their clients and services to incorporate:

- Event specification; by services
- Registration; by clients at services
- Notification; by services to clients

The model includes an Interface Definition Language (IDL[12]) for events, which enables services to specify (declare) the events they can notify. The IDL also allows clients to see the event declarations of the services and to select those of interest, i.e. register with them. The for the event model selected IDL is a, in Cambridge developed, high-performance PRC system. A pre-processor is used to translate the IDL code. The pre-processor not only generates client and service stubs for marshalling and un-marshalling of method invocations but also event specific stubs required by event objects.

The examples below show the use of the Cambridge event model on the service side for an active badge system. The IDL declaration of an event class, the instantiation of an event object and the notifying of an event object are shown.

[11] An event service is an event producing entity located on a physical machine that may host one or more event services. Hence, an event service corresponds to the server of the client-server computing paradigm.

[12] The IDL of the event model is different from the IDL specified by the OMG, but has similar functionality.

- IDL declaration of an event class:

```
Badge : INTERFACE =
   Seen : EVENTCLASS [badge  : BadgeId;
                      sensor : SensorId];
END.
```

- Instantiation of an event object that represents badge 17 being detected by sensor 29:

```
e = Badge_Seen(17, 29);
```

- Sending an event object:

```
EventServer.Signal(e);
```

The examples below show the use of the Cambridge event model on the client side for an active badge system. The instantiation of a template, the registration and de-registration of interest with an event and the evaluation of an event object are shown.

- Register interest with an event:

```
template = Badge_Seen(P,R);
EventClient.Register(clientenv, EventHandler, template);
```

In order to receive the events an event client is interested in, it creates an event template and registers with the relevant event service. To do this, an event client invokes on the register method and passes a client specific parameter list. The parameter list includes a template for event filtering and an event handler which is the method within the client that is invoked on event notification. Furthermore, a parameter called clientenv for client specific purpose is passed.

- Evaluate a received event object:

```
void EventHandler(Opaque *clientenv, Event *event);
```

Event clients support an event handler method which is invoked when an event is delivered. Hence, the actual event data is passed to the client and the clientenv field is returned to it. This field can be decoded by the client to retrieve instance specific information in addition to the event data.

- De-register interest with an event:

```
EventClient.DeRegister(template);
```

If a client no longer requires a registration in an event, it cancels its interest by invoking the deregistration method.

4.2 Event Filtering

Event Filtering is supported by the event model through the use of *templates*. A client specifies a template that describes which events it is interested in. The template is passed to the service as a parameter when the client registers with the service. An application programmer uses a pre-processor to generate extra code for each event object. This code is invoked at run-time to match the template against the actual event. Templates may use variables in place of parameters to indicate ‘wild cards’ which match any value in raised events. They are of the general form:

template = EventTypeName(par_1, par_2, .., par_n);

Examples of filter template object instantiations are:

- Notify of every sensor where badge 17 is seen:
  ```
  templateWhere = Seen(17,R);
  ```
- Notify of every badge seen at sensor 29:
  ```
  templateWho = Seen(P,29);
  ```
- Notify of every badge seen at any sensor:
  ```
  templateGod = Seen(P,R);
  ```

As discussed in [Haa98], the template approach for event filtering is very limited. Because events and templates have the form:

EventTypeName(arg_1, arg_2, .., arg_n)
EventTypeName(par_1, par_2, .., par_n)

they only allow expressions to be matched one-by-one against actual event parameters, i.e. arg_1 against par_1, arg_2 against par_2, etc. It is impossible for parameters to be compared against each other. Furthermore, a logical conjunction is always implied between template expressions.

4.3 Event Composition

Clients may require events from multiple services and may want to detect a specific pattern of event occurrences from these different sources. Such a combination of event occurrences, where a client is interested in a sequence of event occurrences but not in any of the event alone, is called *event composition*. To address this requirement, the Cambridge event model supports the combination of event templates in the general form of:

```
EventTypeNameA(par1, par2, .., parn); EventTypeNameB(par1, par2,
.., parn); ..
```

A composite event scenario example for an active badge system is to monitor for everyone who is in a building when the fire alarm is sounded. The event template sequence for this scenario could be:

```
FireAlarm(7); Seen(7,P,R);
```

This event template sequence traces every badge (worn by a person) seen by any sensor in building 7 after the fire alarm went off in that building.

Such an event template sequence is being checked by a *monitor* which is busy until it has finished that detection. A composite event specification language may be used to define a monitor machine that is able to control more complex sequences of composite event templates.

4.4 Summary

The architecture of the Cambridge event model can be characterised as simple and easy to understand. Although, it is less flexible that other event models and does not feature QoS or real-time capabilities, it does support event templates that are used for event filtering and allows event composition. The approach of using event templates for event filtering is very limited. But may be compensated for by using composite events. Composite events are currently not commonly supported by event models, but may become more common due to their usefulness in constructing complex event filters.

The model includes an IDL for events, that enables services to specify the events they can notify. A pre-processor is used to translate the IDL code into client and service stubs for marshalling and un-marshalling of method invocations and event specific stubs required by event objects. Using a pre-processor is elegant, but, depending on the actual implementation, may not scale well in a distributed application.

5. ECO

The ECO event model, whose architecture and features are described in the following Sections, was originally designed to provide the means for event-based communication in the VOID shell [CCK+95]. The VOID shell is a system for distributed virtual world support, developed at Trinity College Dublin, as a part of the Moonlight [CCK+95] project. The version of the ECO event model implemented as a central part of the VOID shell is called ECOlib [OCC+95]. ECOlib features event filters called notify constraints and another constraint type called pre- and post-constraints.

So far, the ECO event model was also implemented in [ODC+96] as DECO (Distributed ECO) and in [Haa98] as SECO (Scalable ECO)[13]. Both, DECO and SECO, allow event-based communication across a distributed system. The features supported by DECO include precompiled notify constraints which are not dynamically linkable and an extension to the ECO event model called *zones*, which we will discuss further below. DECO relies on the ISIS framework for group communication, whereas SECO is implemented using Kanga [Bur96] as the means for synchronous communication across a distributed system. In order to scale well, SECO features notify constraints that are dynamically linkable.

5.1 ECO Event Model Architecture

The abbreviation ECO stands for Events, Constraints and Objects, which are the three central concepts used in the ECO event model. [SCT95a] and [SCT95b] describe the rationale of the ECO event model, the three central concepts and ECO's event application interface. In the following Sections, we first explain ECO's three central concepts, then introduce ECO's event applications interface with its three operations, and finally we present ECO's features, namely constraints and zones.

5.1.1 Events, Constraints and Objects

ECO's basic abstractions are objects that represent entities, events that provide the means for entities to interact and constraints that allow the specification of synchronisation and notification requirements.

Objects, also called entities, are instances of classes and have attributes and methods. They are encapsulated, thus cannot directly access each other's attributes or invoke on each

[13] SECO has been implemented as uSECO based on unicast communication and as mSECO based on multicast communication. Both versions are described in [HMN+00].

other's methods, but may communicate by *announcing and processing events*. An object that announces events acts as an event supplier and an object that processes event acts as an event consumer. An event that is propagated between objects is of a particular type, which determines the number and the type of the event's parameters. Constraints are used to specify a condition that controls the propagation of events. ECO specifies different types of constraints that may be used for various purposes. Notify constraints act as event filters, whereas pre- and post-constraints may be used to implement synchronisation or concurrency.

5.1.2 ECO's Event Application Interface

The event API provided by ECO includes the three fundamental operations needed in an event based system, which are subscribe, unsubscribe and announce. An event processor registers interest in a particular event type by invoking on the subscribe method. To perform the opposite of a subscription, i.e. to cancel interest in a particular event type, an event processor invokes on the unsubscribe method. Objects that produce events use the announce method to communicate them to event processors.

The subscribe operation has the form:

```
Subscribe(eventType, eventHandler, notifyConstraint, preConstraint,
postConstraint)
```

The *eventType* is the type of the event the event processor is interested in and the *eventHandler* is a callback that is invoked on by the event announcer when events are to be delivered. The event announcer evaluates the *notifyConstraint* to determine the propagation of an event. *PreConstraints* and *postConstraints* are executed locally on the event processor side to control the delivery of an event as described below.

The unsubscribe operation has the form:

```
Unsubscribe(eventType, eventHandler)
```

The eventType and the eventHandler are used to identify the subscription previously made by the event processor that has to be cancelled.

The announce operation has the form:

```
Announce(eventType, eventParameters)
```

The eventType identifies the type of the event to be propagated and the *eventParameters* is the actual event data to be distributed.

5.2 Notify Constrains

Notify constraints are ECO's means to support event filtering and are therefore evaluated on the event announcing side. The ECO event model does not specify how notify constraints be defined or implemented, but does define typed events that may include a large number of parameters with various types. Thus, it is up to the particular implementation to support notify constrains that are sufficient for event filtering on typed events.

5.3 Pre- and Post-Constraints

Pre- and post-constraints behave as event processor method wrappers providing a means to control the delivery of events. These constraints are evaluated local to the event processor and are used to implement:

- Synchronisation within the event processor
- Control of the concurrency level within a method or within the event processor
- Timing control, e.g. start time for earliest delivery and end time for latest delivery
- Method pre- and post-conditions

In addition to this, pre-constraints can be implemented to request that events be discarded, enqueued or processed before they are delivered.

5.4 Zones

Event filters, i.e. notify constraints, are supported by many event models to provide the means of minimising event propagation. However, the ECO event model supports another way to further reduce event propagation. Events in ECO may also be *scoped*. This ensures that events are not delivered outside their scope even with a matching notify constraint.

ECO organises its entities into *zones*. An entity associated with a particular zone is said to be a member of that zone. [ODC+96] and [O'C97] state that members of an ECO zone may change dynamically and that zones may overlap allowing entities to become members of several zones. This is useful in a scenario where a robot in a smart building is interested in events announced within the smart building. The robot subscribes to doorOpenClose and generalAlarm events. To scope these events, the robot also joins the zone that consists of the floor he is currently moving on[14] and the zone that consists of all alarm sensors within the

[14] This scenario also shows the dynamic group membership [Bir96] aspect of zones. The robot has to dynamically change its group membership when moving from one floor to

building. The former type of zone is said to be *geographical*, the latter is said to be *functional*. Without the zones, the robot would receive doorOpenClose events from the whole building, although only the ones close to its position, i.e. on the current floor, are of interest to it. As for generalAlarm events, the robot is obviously interested in alarms that are announced anywhere in the building, such as a fire alarm event on the ground floor or a power failure event on another floor.

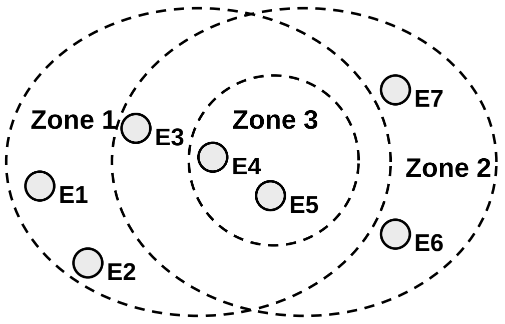


Fig. 5.1. Overlapped and Nested Zones.

[O'C97] describes several different ECO zone concepts. As already mentioned, overlapping zones allow an entity to become a member of several zones simultaneously. Whereas nested zones allow large zones that contain many entities to be subdivided. In a scenario where the announced event is limited to a particular zone, of which the announcing entity is not a member, the zone is said to be targeted. Finally, zones may be created and deleted dynamically, and entities may change their zone memberships at run time. These zones concepts may also be combined (Figure 5.1 shows zones that are overlapped and nested).

5.5 Summary

The ECO event model specifies the three basic operations needed to provide event-based communication, namely subscribe, unsubscribe and announce. Its architecture does not depend on a centralised component for event propagation, such as CORBA's event channel. This removes a single point of failure. ECO's main features include event filters, called notify constraints, pre-constraints and post-constraints that act as wrappers on the event processor side and the notion of zones that limit the scope of event propagation.

another. Before joining the new zone, the robot might have to create the new zone and after leaving the old zone, the robot might have to delete the old zone respectively.

Of ECO's features, pre-constraints, post-constraints and zones are unique amongst event models. Pre- and post-constraints may be used to implement additional subscription-specific functionality on the event processor side such as synchronisation, timing control and concurrency control. Furthermore, pre-constraints may implement event delivery strategies such as enqueuing, pre-processing or discarding. Since pre- and post-constraints are subscription specific, they provide a very flexible means of controlling event delivery. Whether such a degree of flexibility is efficient[15], manageable[16] and feasible[17] in a large-scale distributed system depends on the application domain.

The notion of geographical and functional zones is used in ECO to limit the scope of the propagation of events. Several different concepts of zones, including overlapping, nested, targeted and dynamic zones as well as combinations of them, are described in [O'C97]. Limiting the scope of event propagation, regardless of matching notify constraints, provides a powerful means of minimising network traffic and CPU processing time, particularly in a large-scale system.

[15] Most event-based systems are likely to have similar delivery strategies on the event processor side. Therefore, its is more efficient to have the delivery strategy build in, instead of having every event processor defining similar pre and post constraints.

[16] Although synchronisation delivery strategies might be chosen based on a local decision, a system that implements a wide range of different strategies may become unmanageable.

[17] A wide range of different delivery strategies may interfere with other system features such as event ordering or timely delivery.

6. JEDI

The *JEDI* (Java Event-based Distributed Infrastructure) is an object-oriented infrastructure implemented in the Java programming language that supports the development of event-based applications. The JEDI architecture, which is described in [CDNF98a] and [CDNF98b], has been developed by CEFRIEL – Politecnico di Milano. It has been used to implement a workflow management system called OPSS[18] and a process support system called PROSYT[19].

6.1 JEDI Event Model Architecture

The architecture of the JEDI event model, which is described in Figure 6.1, is based on the notion of *active objects* (AOs) and *event dispatchers* (EDs). An AO is an autonomous entity that performs the role of an event producer or an event consumer, thus generating or notifying events. The ordered delivery of the events is the responsibility of the ED. The ED supports an event subscribe and an event unsubscribe operation. These are invoked by the AOs to register, or cancel, with the particular event.

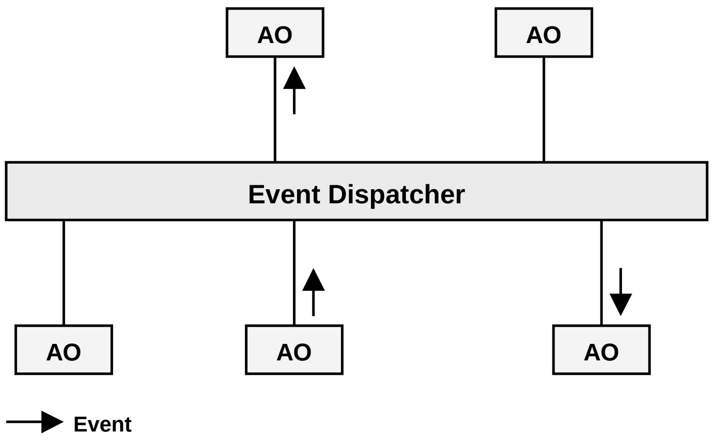


Fig. 6.1. A Logical View of the JEDI Architecture.

In JEDI, events are defined as a set of ordered strings. The first string representing the event name and the remaining strings representing the event parameters. This allows the

[18] OPSS (ORCHESTRA Process Support System) is introduced in [CDNF98a] and [CDNF98b].

[19] PROSYT is introduced in [CG98]. It uses mobile agents in its design and implementation and thus makes use of JEDI's mobility feature, which is introduced in this Section.

definition of an event using a notation similar to function calls in traditional programming. An example of an event in JEDI might be

```
print(MyDocument, OurLaserPrinter)
```

where *print* is the event centralised component name and *MyDocument* and *OurLaserPrinter* are the event parameters.

As shown in Figure 6.1, the ED is the logically centralsed component of the JEDI architecture that must have global knowlegde of the generated events and the subscriptions. However, [CDNF98b] statess that a centralised implementation of the ED can become a critical bottleneck for a distributed system. Thus, JEDI provides two implementations of the ED; a centralised and a distributed version. The centralised version covers the requirements of simple, small-scale applications exchanging a limited number of events. Whereas the distributed version addresses the needs of large-scale applications, interconnecting several AOs running on different nodes of the network.

The distributed version of the ED consists of a set of *dispatching servers* (DSs) interconnected in a tree structure. As depicted in Figure 6.2, each DS is located on a different node and is connected to one parent DS, unless it is the root DS, and to zero or more descendants. The AOs are connected to the distributed ED via a DS.

In the distributed version of the ED a *hierarchical strategy* is employed for the distribution of event, subscription and unsubscription messages amongst the DSs. Each DS that receives a subscription or unsubscription request from an AO (or another DS) updates its entry table and passes the request on to its parent DS. Hence, all subscription and unsubscription requests are propagated upwards the tree until they reach the root. On an AO producing an event, it is passed to the local DS where it is forwarded up the tree. Each DS that receives an event checks its descendants, passes the event onto any descendant that has requested the event and then forwards the event to its parent. Thus, events are also propagated upwards the tree until they reach the root. This strategy ensures that all the relevant nodes receive all the subscription, unsubscription and event messages.

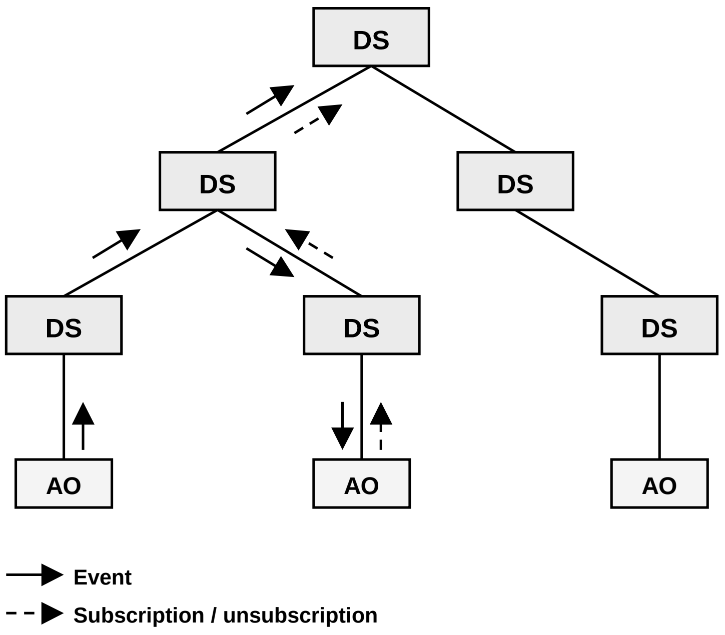


Fig. 6.2. The Structure of Dispatching Servers.

6.2 Event Pattern

JEDI allows a very simple form of event filtering through the use of *event pattern*. When subscribing, AOs register interest in a specific event or in an event pattern. Like events, event patterns are defined as a set of ordered strings. The first string representing the pattern name and the remaining strings representing the pattern parameters. [CDNF98b] states that each string of a pattern may end with an asterisk and that an event *e* matches a pattern *p* if the following conditions hold:

- The name of *e* is equal to the name of *p*, if *p* does not contain the asterisk or both names start with the same sequence of characters and the name of *p* ends with an asterisks;
- *e* and *p* have the same number of parameters; and
- Each parameter of *e* is equal to the parameter of *p* having the same position or both start with the same sequence of characters and the parameter of *p* ends with an asterisks.

6.3 Reactive Objects and Mobility

The JEDI architecture supports the mobility of objects through the use of *reactive objects*. Reactive objects are a particular type of active object defining an abstract method called *processMessage*. The application programmer has to implement this abstract method which is automatically invoked each time an event is delivered to the reactive object. This enables an implementation of a reactive object to autonomously move across the nodes of a network by invoking JEDI's *move* operation. Invoking on the move operation causes the following actions to occur:

- The reactive object is disconnected from the ED and the thread of execution controlling it is stopped;
- The reactive object is serialised using the standard Java facilities;
- The reactive object is moved via the network to its new destination, where it is reconnected to the ED; and
- During the migration, events to be received by the reactive object are stored by the ED until the reactive object has successfully moved location and is ready to receive the stored events.

6.4 Summary

The JEDI infrastructure provides a simple, easy to understand architecture for event-based communication. Its architecture is based on two components, active objects (AOs) and event dispatchers (EDs). In its centralised version, the event dispatcher may become a critical bottleneck and is a single point of failure. To overcome this, JEDI also features a distributed version of the event dispatcher, which consists of a set of dispatching servers (DSs) interconnected in a hierarchical structure. The hierarchical topology improves the robustness and scalability of the JEDI architecture. However, in case of a failing DS the JEDI network has to deal with segment separation. Furthermore, the fact that all messages are being forwarded via several dispatching servers to the root of the tree might cause the overloading of the higher-level servers.

Events in JEDI are defined as a set of ordered strings, limiting the parameter that can be defined for an event. Event filtering is supported in a very simple and limited manner through the use of event patterns, which are essentially events defined as a set of ordered strings.

An interesting feature of JEDI is the ability to move objects across the nodes of the network. Mobility is supported through the use of a particular type of active objects called reactive objects and through the use of a move operation. Mobility is desirable to increase the flexibility and the effectiveness[20] of an application and to support event-based communication among mobile devices such as Personal Digital Assistants (PDAs).

[20] Mobility can be used to implement load balancing and to reduce the network traffic by moving applications close to the resources they need.

7. SIENA

Like JEDI [CDNF98a][CDNF98b], SIENA (Scalable Internet Event Notification Architecture) has been developed in Politecnico di Milano and features an architecture that is similar to but more advanced than JEDI's. SIENA has been designed to support event based communication in wide-area networks such as the Internet and features code mobility. Its architecture is introduced in [CRW98] and a detailed description can be found in the Ph.D. thesis [Car98].

7.1 The SIENA Event Model Architecture

The SIENA infrastructure implements a scalable general-purpose event model that is based on a distributed architecture of event servers. In SIENA's terminology, events are produced by *objects of interest* and consumed by *interested parties*. The propagation of events is regulated by mechanisms called *advertisement*, *subscription* and *publication*.

The high-level view of the SIENA architecture, shown in Figure 7.1, includes these event propagation mechanisms. While the subscription and the publication mechanism are common to most event models, the advertisement mechanism is specific to SIENA. The subscribe and unsubscribe operations are used by an interested party to register and cancel interest in a certain event type and the publish operation is invoked by an object of interest to propagate an event. Also called by an object of interest, the advertise operation indicates an objects intention to produce events of a certain type. The unadvertise operation indicates that an object of interest no longer wishes to produce such an event. The unadvertise operation has the opposite effect to the advertise operation, hence cancelling an advertisement. The advertise mechanism provides additional information to the event service, enabling the event servers to route subscriptions and publications more effectively. The interfaces of these three mechanisms are:

```
Subscribe(interested party, pattern)
Unsubscribe(interested party, pattern)
Publish(event)
Advertise(object of interest, filter)
Unadvertise(object of interest, filter)
```

To uniquely identify interested parties, objects of interest and event servers within SIENA, and for them to communicate with each other, SIENA uses a naming scheme, referred to as a *generic URI naming scheme*. This means that every party, object and server has an URI of the form *mailto:john@cs.edu*[21] associated with it.

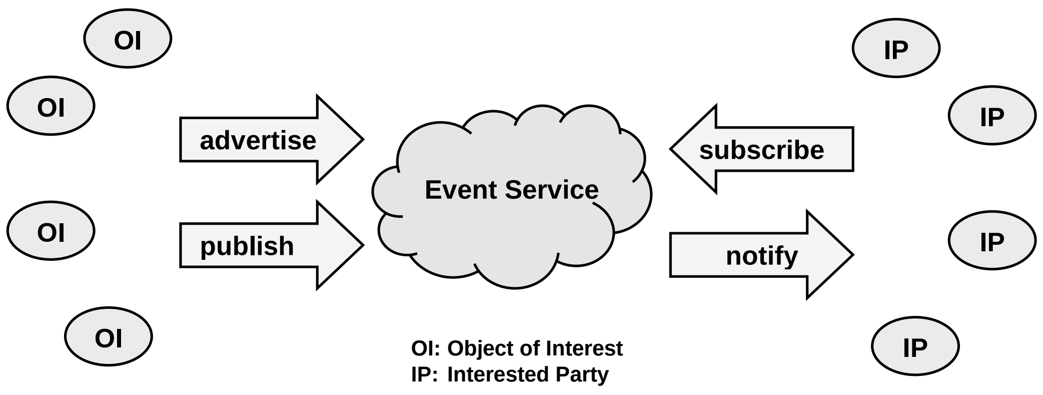


Fig. 7.1. The High-level View of the SIENA Architecture.

SIENA supports events in the form of a set of attributes in which each attribute is a triple of name, type and value. Each attribute is uniquely identified by its name. A predefined set of types is available to define events of the form:

```
string   event   = account/debit
time     date    = 15.01.2000
int      number  = 12345
float    amount  = 215.31
```

7.1.1 Operational Semantics

The SIENA infrastructure includes two different behaviours for the event service in response to advertisement and subscription. [CRW98] argues that the reason for supporting both is to find the most appropriate solution for a flexible and scalable event service depending on the requirements of the chosen application domain.

In the *subscription-based* version of SIENA, only subscriptions determine the semantics of the event service. This behaviour is similar to the semantics of other subscription-based event models. Advertisements are not required, but can be used to optimise the routing of

[21] The URI that identifies and object is both the unique name and the communication method of that object. SIENA implements the most common URI schemas, namely *mailto* and *http*.

subscriptions. This implies that events will be delivered to all interested parties that have subscribed to them[22].

In the *advertisement-based* version of SIENA, both advertisements and subscriptions determine the semantics of the event service. The event service will only guarantee the delivery of an event if an object of interest advertises an event of a particular type and an interested party subscribes to the same type of event[23].

7.1.2 Server Topologies

The propagation of events is the responsibility of a logically centralised component to which objects of interest and interested parties are connected. This component is implemented as a set of *event servers* co-operating with each other to provide a network-wide event service. [Car98] describes four different event server topology implementations, namely centralised, hierarchical, acyclic peer-to-peer and generic peer-to-peer, as shown in Figure 7.2. [Car98] states that each topology was tested for the flexibility and the scalability of the service and found that the distributed topologies outperformed the centralised approach, when the number of objects of interest and interested parties increased.

Among the in Figure 7.2 described topologies, the centralised version is the simplest, implementing only a client-server protocol for the co-operation between the event server and the event clients, i.e. interested parties and objects of interest. It is important to note that this is also the case for the hierarchical topology. An event server does not distinguish between other event servers and its clients, hence treating servers as clients. In the two peer-to-peer topologies, event servers communicate with each other as peers, thus allowing bi-directional flow of subscriptions, advertisements and events.

[22] Events of a particular type are delivered to all interested parties that have subscribed to the type if the pattern matches.

[23] Events are only delivered to an interested party if the event service has received an advertisement and a subscription for the particular event type and the event filter and the event pattern match.

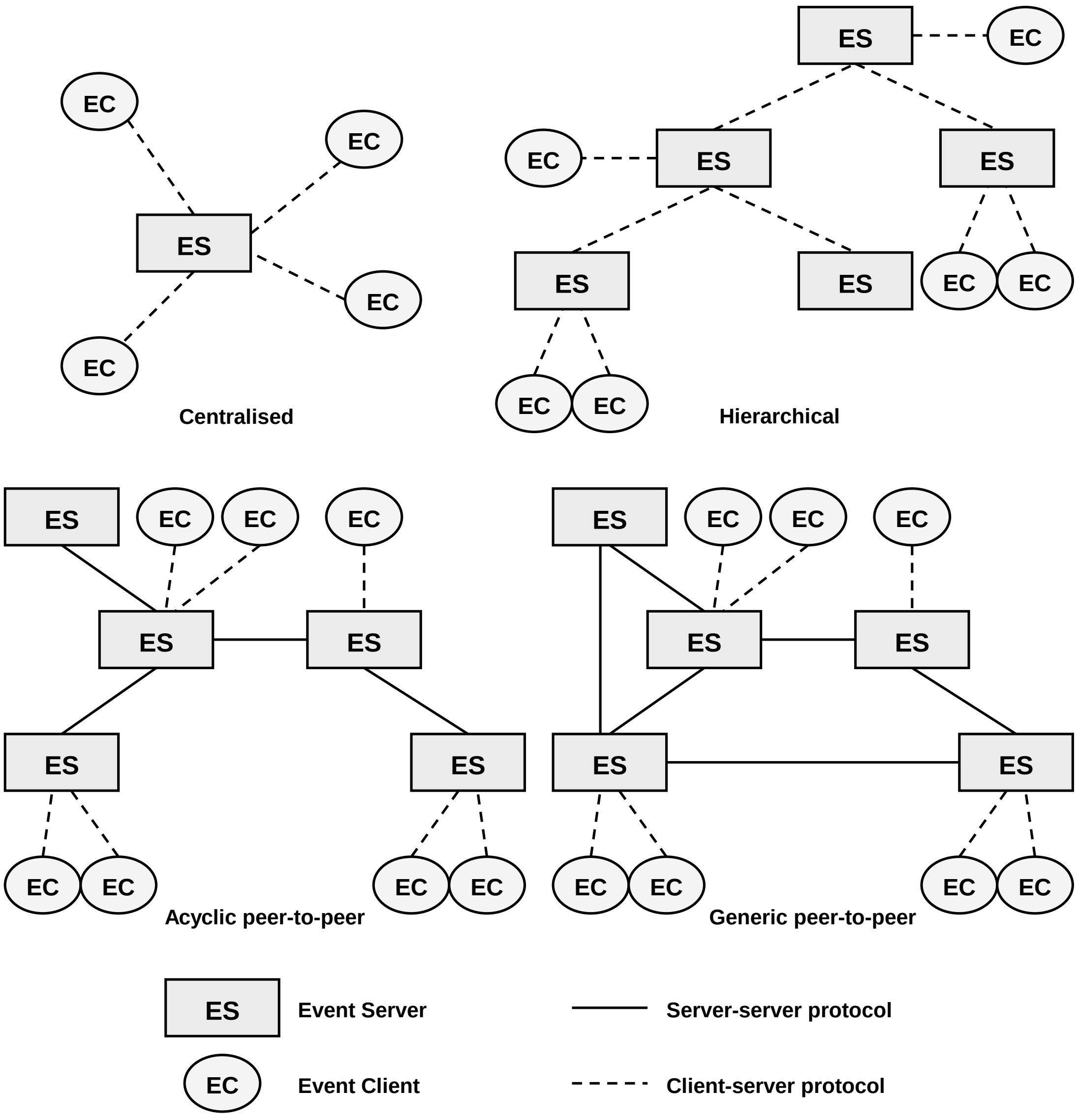


Fig. 7.2. Event Server Topologies.

[CRW98] presents and discusses two main optimisation strategies for communication and computation resources:

- Applying event filters and event pattern physically as close as possible to publishers, i.e. objects of interests; and
- Replicating events, by means of multicast, physically as close as possible to subscribers, i.e. interested parties.

SIENA uses *IP multicast* as the underlying transport mechanism.

7.2 Event Filter and Pattern

An *event filter* is specified by a set of attribute names and types and some constraints on their values, i.e. an operator and a value per constraint. A fixed set of operators are available to define event filters of the form:

```
string   event  == account/*
time     date   >= 01.01.2000
float    amount >  100.00
```

This filter selects all account transfers in this millennium that are transferring an amount larger than a certain minimum limit.

An *event pattern* is specified by combining a set of event filters using filter combinators. This allows a pattern to select a combination of events, i.e. several events that together match an algebraic combination of filters. The following example shows an event pattern that monitors an account and selects it if a large amount of money is transferred more that once, e.g. such an account has to become a premium account:

```
string   event  == account/*
int      number == 12345
float    amount >= 100000.00

and then

string   event  == account/*
int      number == 56789
float    amount <  250.50
```

A description of all the attribute types, filter operators and pattern combinators supported by SIENA can be found in [Car98].

7.3 Mobility

SIENA does not support code mobility directly, but supports the integration of mobile objects into its infrastructure. [Car98] identifies three different approaches to support mobility, called *transparent*, *native* and *external*. The native approach, where mobility is supported through the use of a move operation, is adopted by JEDI [CDNF98a][BBHM96] but is not supported by SIENA. However, the SIENA architecture supports the remaining two approaches. The transparent approach uses network-level mechanisms to transparently manage mobility of objects. This is feasible since objects are addressed by URI's, hence hiding their location. The external approach relies on an extension layer that is added externally between the event service and the mobile objects. This layer manages the movement of objects by providing a move operation and by handling (buffering, forwarding, synchronising, etc.) all the subscriptions, advertisements and events.

7.4 Summary

The SIENA infrastructure implements a general-purpose event service that is based on a distributed architecture of event servers. It is designed to scale well in wide-area networks such as the internet. Although implemented as a set of event servers, the logically centralised component responsible for event propagation suffers from overloading of event servers and network separation due to event server failure.

A mechanism introduced by SIENA, besides the quite common subscription and publish operations, is called advertisement. Advertisement optimises the routing of subscriptions and publications, but requires a logically centralised component, i.e. the event servers, to manage advertisements.

SIENA supports events in the form of a set of attributes in which each attribute is a triple of name, type and value. This allows the definition of an event using a notation similar to objects[24] in object-oriented programming. The definition of event is limited by the set of attribute types available. This may be seen as a strict limitation, but we believe that this approach usually suffices to cover the needs of applications and that new attribute types can be added as needed. The propagation of events is controlled through the use of event filters and combinations of filters called event patterns.

SIENA does not support code mobility directly, but supports the integration of mobile objects into its infrastructure through a transparent and a external approach. Code mobility is becoming more and more popular in distributed systems and in wide-area networks such as the internet. Although SIENA addresses mobility, an integrated support of mobility is desirable and thus should be subject to further investigation in a future version.

[24] Event are defined as objects that contain attributes but do not contain any methods.

8. EVENT MODELS BASED ON THE OMG CORBA EVENT SERVICE

In the following Sections, we present two event architectures that are based on the OMG CORBA event service. Both identify important features required by some applications but lacked by CORBA's event service. The first of the presented event architectures extends the CORBA event service to satisfy the quality of service (QoS) needs of real-time applications. The second proposes a reliable multicast extension to provide reliable and total ordered event delivery.

8.1 Real-Time Event Service

As part of the TAO project[25] at Washington University, an extension to the CORBA event service [Gro95b] was developed called real-time (RT) event service. As described in [HLS97], the real-time event service addresses the QoS requirements of real-time applications and was designed for an avionics mission control application.

The CORBA event service provides a flexible model for event-based communication in systems based on ORB middleware. However, it lacks important features required by real-time applications. To address this, the RT event service supports *real-time event dispatching and scheduling*, *source based and type based filtering*, *event correlation* and *periodic event processing*. Since we have already reviewed the CORBA event service earlier in this document, the following Sections present the extensions proposed by the RT event service only.

8.1.1 RT Event Service Architecture

The architecture of the RT event service and the CORBA event service are identical with regards to the role of event subscriber, event consumer and their mediator, the event channel. However, the event channel of the RT event service has been adapted to support the added features. The three main features included, namely the support of QoS parameters, filtering on typed events and periodic event processing, are described in the remainder of this Section. The high-level architecture of the RT event service, including the main event channel modules, is depicted in Figure 8.1.

[25] TAO is a real-time ORB end-system that provides end-to-end quality QoS guarantees to applications. A more detailed description of TAO is presented in [HLS97].

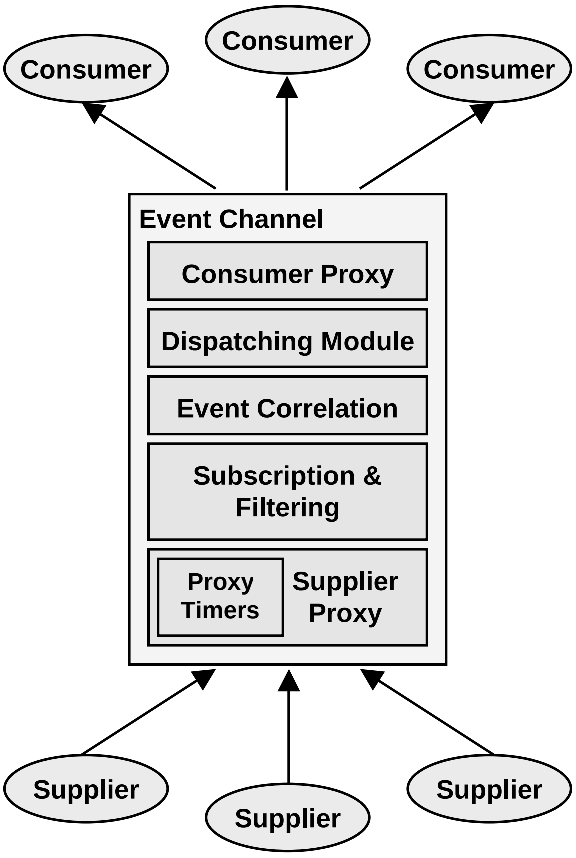


Fig. 8.1. RT Event Channel Overview.

8.1.1.1 QoS Parameter

The event channel's standard proxy interfaces have been extended to allow consumers and suppliers to register their execution requirements and characteristics with the event channel, using QoS parameter. These parameters are used by the event channel's dispatching mechanism to determine event dispatch ordering and pre-emption strategies. The dispatching module implements priority-based event dispatching and pre-emption using priority queues. [HLS97] describes the three different pre-emption strategies supported by the event channel. The strategies supported are; *real-time upcall (RTU) dispatching*, *real-time pre-emptive thread dispatching* and *single-threaded priority-based dispatching*.

8.1.1.2 Event Filter

In an event-based infrastructure implemented using the CORBA event service, event channels featuring different event dispatching strategies may be chained to create event filters. However, this increases the number of hops an event must travel when propagated from supplier to consumer, hence increasing overhead and latency. To overcome this, the RT event service supports event filtering and event correlation mechanisms. The event filtering mechanism allows suppliers to specify the type of events they generate. It also allows consumers to register interest in events generated by certain suppliers or to register interest in events of a particular type. The former being called supplier-based filtering and the latter type-based filtering. Any combination of supplier-based and type-based filtering is supported as well. The event correlation mechanism allows consumers to specify logical OR and AND dependencies among events. The former semantics lets the channel notify the consumer when *any* of the specified events dependencies are satisfied. The latter semantics lets the channel notify the consumer when *all* the specified events dependencies are satisfied.

8.1.1.3 Periodic Event Processing

The supplier proxy module allows consumers to specify event dependency timeouts. *Priority Timers* manage those timeouts and notify consumers, i.e. dispatch timeout events, even if their dependencies are not satisfied within some time period. This mechanism is well-suited for periodic event processing, i.e. a real-time “watchdog”.

8.1.2 Summary

[HLS97] identifies important event service features required by real-time applications but lacked by the OMG CORBA event service. The presented RT event service addresses these requirements by extending CORBA’s event service with QoS parameters, typed event filters and periodic event processing.

The QoS parameters are used by the event channels dispatcher module to determine event dispatch ordering and pre-emption. Such a dispatching mechanism is essential for predictable end-to-end QoS, required by real-time applications. As we have argued earlier, filters are useful to limit the propagation of events, minimising network traffic and CPU processing time. The RT event service also supports the combination of events providing a powerful means to define dependencies on a group of events using OR and AND semantics. Although these features are essential for real-time applications, the major drawback of this RT event service is the centralised implementation of the event channel. The event channel is the single point of failure of the architecture and may suffer from overloading with increasing numbers of event suppliers and consumers. However, it provides the global knowledge necessary for “hard” real-time and suffices in small-scale applications like the avionics mission control it was designed for.

8.2 CONCHA

In this Section, we present extensions to the OMG CORBA event service that address multicasting, reliability and total ordering. These proposals were developed in a prototype environment called CONCHA. CONCHA [OFB99] is a conference system developed at University of Coimbra, Portugal and its abbreviation stands for CONference system based on java and corba event service CHAnnels.

CONCHA extends CORBA's event service with reliable multicast communications based on the use of the light-weight reliable multicast protocol (LRMP)[26], which also features total ordered package delivery. Figure 8.2 shows an overview of CONCHA's event channel with the integrated multicast support.

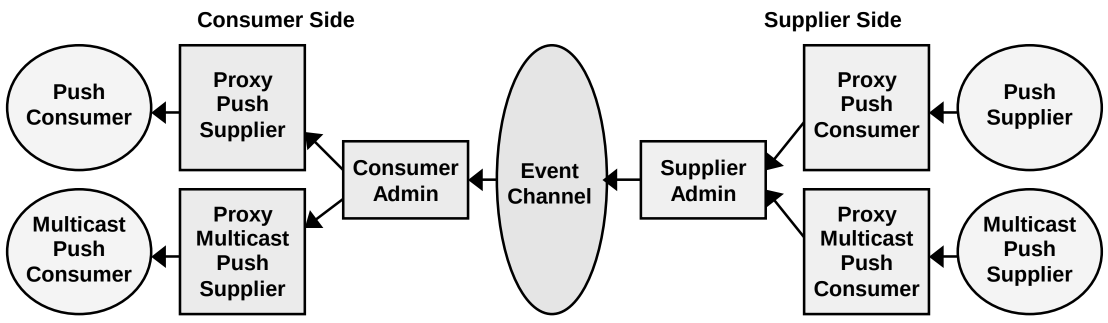


Fig. 8.2. CONCHA Event Channel Overview.

The multicast extension is implemented by providing a single multicast proxy that deals with all the multicast push suppliers and another one that deals with all the multicast push consumers[27]. Hence, this architecture provides an alternative mechanism to propagate events using reliable multicast[28], besides the standard mechanism that uses IIOP. The propagation of events is not limited to either the standard or the multicast mechanism, but allows the combination of the two, allowing a multicast push supplier to propagate events to both multicast and standard consumer. This approach does not require multicast group management since all events are delivered to all consumers and thus only one multicast group is used.

[26] LRMP is a reliable general-purpose transport protocol based on unreliable underlying network transport protocols such as UDP/IP. It features loss repair, ordered package delivery, flow control and ensures reliability using a NACK semantics.

[27] The multicast extension has only been implemented for the push model, but has not been adapted by the pull model.

[28] When using the multicast mechanism, event propagation will also benefit from other LRMP features such as ordered package delivery.

9. SYNTHESIS

This document has reviewed a variety of infrastructures that provide event-based communication in distributed systems. The described event models were selected to cover a range of architectures and features required by different application domains in order to present the state-of-the-art in event-based technology. Although we are aware of other event models, i.e. Yeast [KR95], GEM [MSS97] and COBEA [CB98] to name some of them, they were not presented in this document because none of them elucidate new features. Based on this review, we describe the rationale of a proposed event model that answers the requirements of a system providing a means for reliable event-based communication in a large-scale environment that supports application component mobility.

9.1 Event Model Architectures

The architecture of event models can be categorised according to one of their features, that of application component mobility, and according to the presence of a mediator component.

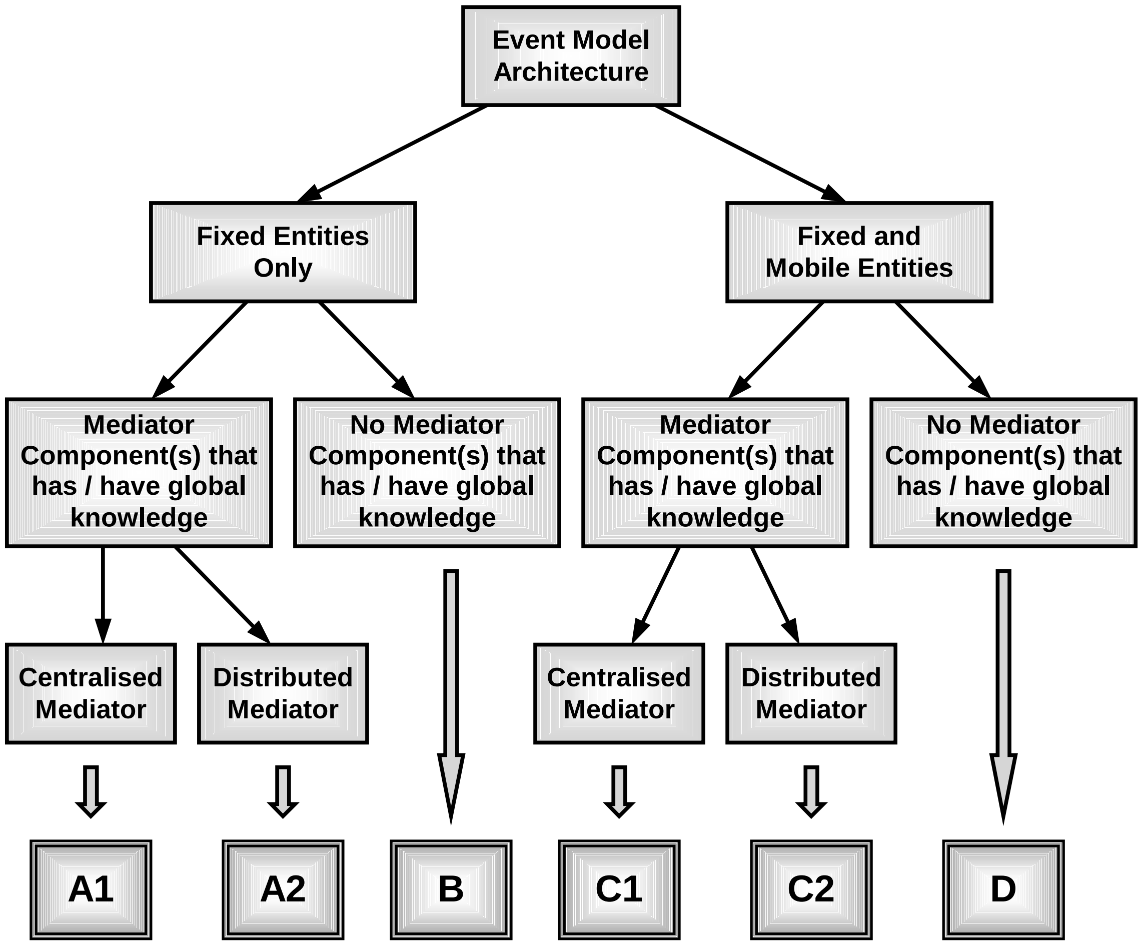

Event Model Architecture
Fixed Entities Only
Fixed and Mobile Entities
Mediator Component(s) that has / have global knowledge
No Mediator Component(s) that has / have global knowledge
Mediator Component(s) that has / have global knowledge
No Mediator Component(s) that has / have global knowledge
Centralised Mediator
Distributed Mediator
Centralised Mediator
Distributed Mediator
A1
A2
B
C1
C2
D

Fig. 9.1. Taxonomy for Event Model Architectures.

Our taxonomy for event model architectures, as depicted in Figure 9.1, distinguishes between mediator components that are implemented either in a centralised or in a distributed manner. We call the distributed version “logically” centralised because, alike the physically centralised version, the mediator component maintains *global knowledge* of the event system and it’s entities. The identified types A to D of event model architectures are described in more detail in Figure 9.2 to Figure 9.7.

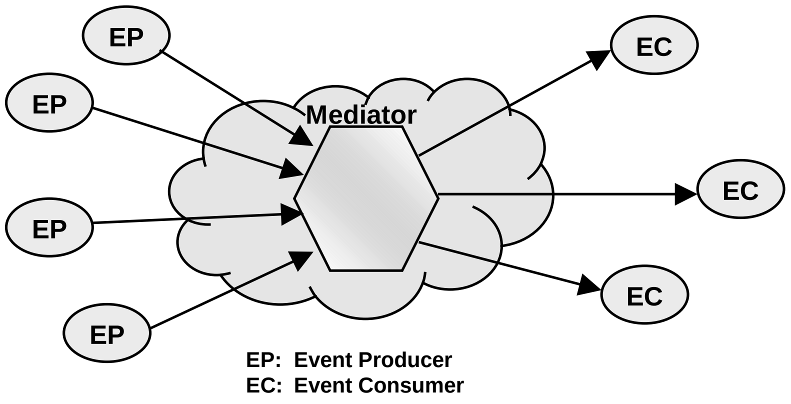


Fig. 9.2. Event Model Architecture Taxonomy Type A1.

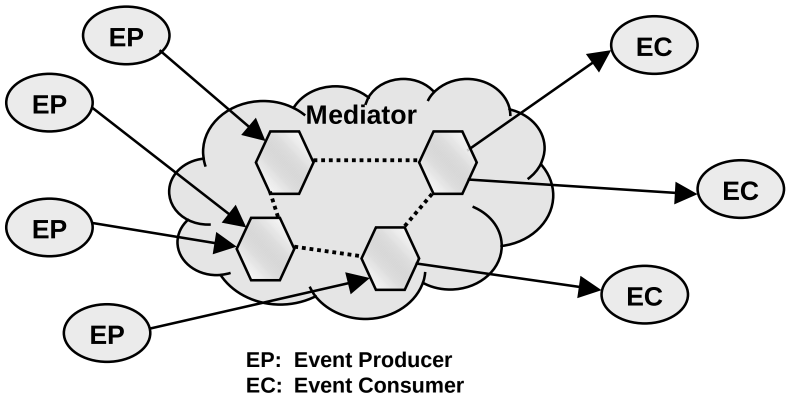


Fig. 9.3. Event Model Architecture Taxonomy Type A2.

Architecture types A1, A2 and B, as shown in Figure 9.2, Figure 9.3 and Figure 9.4, do not support mobile application components (entities). A1 features a centralised implementation of the mediator component, whereas A2 supports a distributed version. Such a mediator component is responsible for propagating event messages among entities and hence must have global knowledge of all entities, i.e. all event producers and event consumers, and their subscriptions. Mediator components allow a centralised management of events and their subscriptions, thus a relatively simple implementation of non-functional requirements such as reliability and QoS. However, a physically centralised mediator, i.e. architecture type A1, represents a single point of failure, may become a critical performance bottleneck and hence will not scale well in a distributed environment. The approach of a distributed mediator topology, i.e. architecture type A2, improves the robustness and scalability of a event model's architecture. However, such an architecture has still to deal with network segment separation, possible bottlenecks and scalability in large-scale distributed systems.

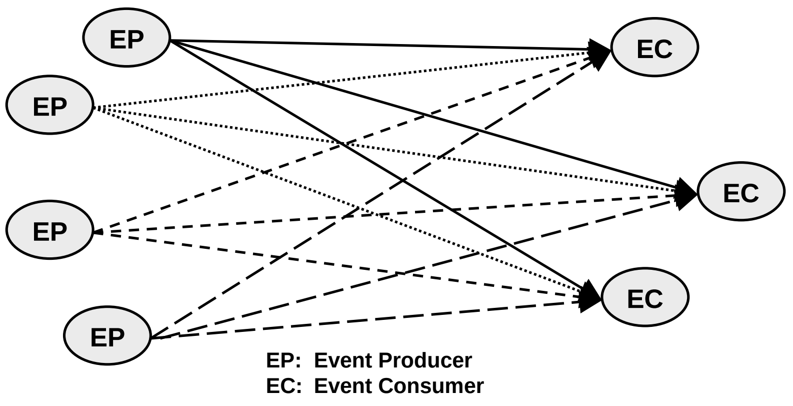


Fig. 9.4. Event Model Architecture Taxonomy Type B.

An architecture of type B does not depend on a mediator component. Its entities communicate directly with each other in a one-to-many manner, thus improving overall robustness and scalability of the system. [Haa98] determines that this architecture type scales well even in large-scale distributed systems. However, because of the lack of a component that has global knowledge, the system's entities are required to manage and propagate event messages based on a local decision and the implementation of non-functional features will become more complex.

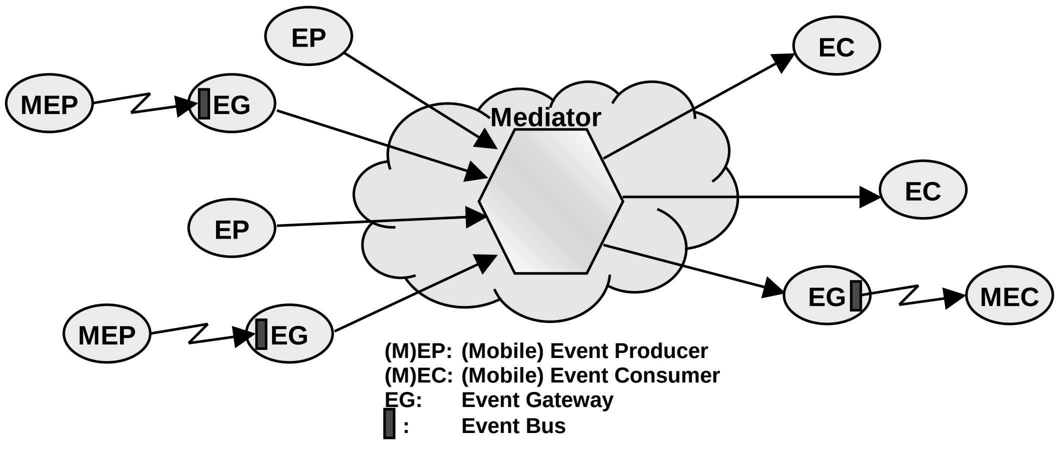


Fig. 9.5. Event Model Architecture Taxonomy Type C1.

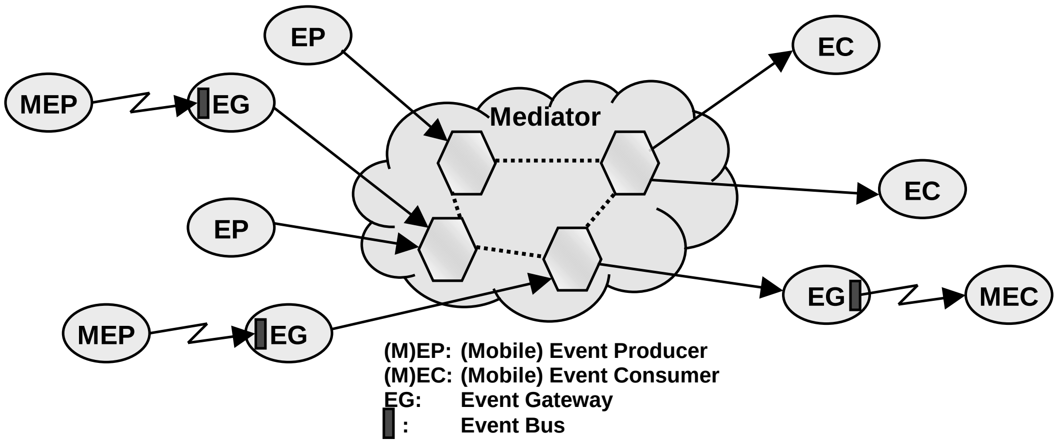


Fig. 9.6. Event Model Architecture Taxonomy Type C2.

Architecture types C1, C2 and D, as shown in Figure 9.5, Figure 9.6 and Figure 9.7, do support mobile entities. With regards to the mediator component, they can be characterised as described above analogous the types A1, A2 and B respectively. For the event system to communicate with mobile entities, a new entity type, called event gateway, is introduced. Event gateways act as proxies between fixed and mobile entities and are responsible for managing the propagation of subscriptions and events from and to mobile entities. Generally, it can be said that supporting mobile entities increases the complexity of the event model implementation significantly due to the management (synchronisation, buffering, forwarding, etc.) of entities, subscriptions and events.

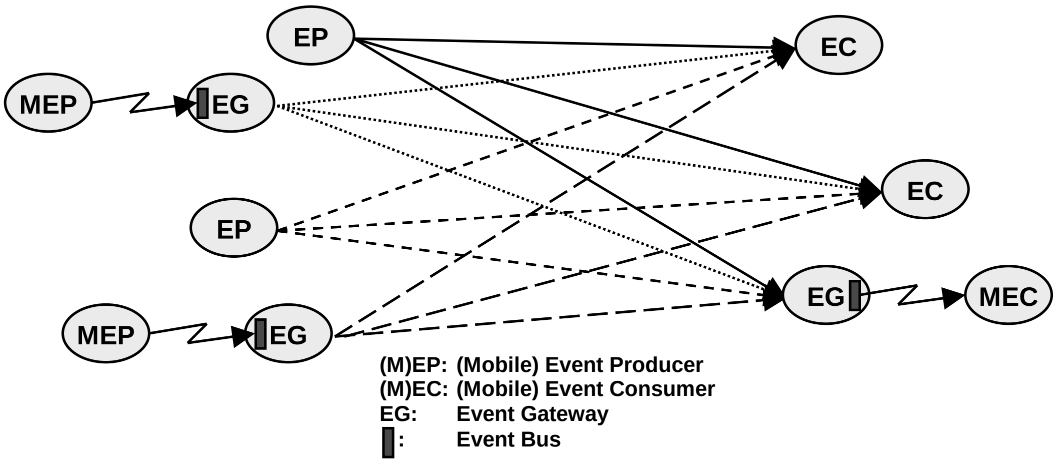

EP
EC
MEP
EG
EC
EP
EG
MEC
MEP
EG
(M)EP: (Mobile) Event Producer
(M)EC: (Mobile) Event Consumer
EG: Event Gateway
▮ : Event Bus

Fig. 9.7. Event Model Architecture Taxonomy Type D.

An overview of the architecture types of the reviewed event models is shown in Table 9.1. Additionally, the table includes a brief description of the event models' application domains and a classification according to their use.

Event Model	**Application Domain**	**Dis-tribution**	**Architecture Type**	**Academic or Commercial**
OMG CORBA: Event Service	General-purpose, middleware applications	✓	A1, i.e. event channel	Commercial
OMG CORBA: Notification Service	General-purpose, middleware applications	✓	A1, i.e. notification channel	Commercial
Java AWT: Delegation Event Model	Small-scale, centralised GUI applications	✗	B	Commercial
Java: Distributed Event Model	Ad-hoc, wide-area applications	✓	B	Commercial
Cambridge Event Model	General-purpose applications	✓	B	Academic
ECO	General-purpose, large-scale applications	✓	B	Academic
JEDI	General-purpose, mobile applications	✓	C1 & C2, i.e. event dispatcher & dispatching servers	Academic
SIENA	Mobile, wide-area applications	✓	C1 & C2, i.e. event server(s)	Academic
RT Event Service (TAO) (CORBA ES extension)	Small-scale, real-time middleware applications	✓	A1, i.e. event channel	Academic & commercial
CONCHA (CORBA ES extension)	Reliable, general-purpose, middleware applications	✓	A1, i.e. event channel	Academic

Tab. 9.1. An Overview of the Reviewed Event Model Architectures.

CORBA based event models, namely the Event Service [Gro95b], the Notification Service [Cea98], the RT Event Service [HLS97] and CONCHA [OFB99], are examples of architectures including a centralised mediator, thus are of architecture type A1. The limitations due to the use of a centralised mediator may be improved by adapting mediator *federation* as suggested in [Gro96], which results in an architecture of type A2. Furthermore, CORBA based event systems entities may invoke directly on each others event interfaces, as proposed in [Gro95b], therefore being of architecture type B. However, this approach requires the application to maintain information on the location of the event system's entities that are active at any given time and the event consumers to subscribe to every event producer separately. This approach is not sufficient in systems where entities dynamically join and leave.

Event models that do not rely on a mediator component, i.e. of architecture type B, are the ECO [Haa98], the Cambridge [BBHM96] and the distributed Java [SM98] event model.

An event model architecture of type C1 or C2, i.e. that supports entity mobility and implements either a centralised or a distributed mediator topology, is investigated and adapted in JEDI [CDNF98a] [CDNF98b] and SIENA [CRW98] [Car98].

Trinity College Dublin is currently investigating the design and the implementation of an event model of type D. However, no publications are available yet by the time this document was written.

9.2 Event Model Features

Besides its architecture, an event model is characterised by the features it supports. A summary of the features supported by the reviewed event models can be found in Table 9.2.

Event Model	**Typed Events**	**Event Filter**	**Real-Time**	**QoS**	**Mobility**	**Other Features & Remarks**
OMG CORBA: Event Service	✓					
OMG CORBA: Notification Service	✓	✓	✓	✓		
Java AWT: Delegation Event Model	✓	See remarks	See remarks	See remarks		User definable adapters may feature filters, real-time and QoS
Java: Distributed Event Model	✓	See remarks	See remarks	See remarks		User definable adapters may feature filters, real-time and QoS
Cambridge Event Model	✓	Templates				Event composition
ECO	✓	✓				Pre-constraints, post- constraints, zones
JEDI	✓	Pattern			✓	Event are defined as a set of ordered strings, based on IP multicast
SIENA	✓	✓			✓	Advertisement, event combination, based on IP multicast
RT Event Service (TAO) (CORBA ES extension)	✓	✓	✓	✓		Event correlation, periodic event processing
CONCHA (CORBA ES extension)	✓			See remarks		Total ordering, based on reliable IP multicast

Tab. 9.2. A Summary of the Reviewed Event Model Features.

Among event model features, the ability to control the propagation of event messages, i.e. to filter them, is the most important one. [Haa98] shows that filters are a powerful means to significantly reduce the number of event messages propagated in a system. Fewer copies of each specific event message are propagated, preventing event storming and hence, minimising network traffic and CPU processing time. This results in improved overall system stability and scalability, which is particularly important in large-scale environments.

The implementation of event filters depends on the implementation of event messages. Well-structured, i.e. typed, event messages are required for the implementation of expressive and thus powerful event filters. Although the CORBA Event Service [Gro95b] supports generic and typed events, filters are not supported due to the fact that generic events, implemented as of type *any*, are hard to filter. As an improvement on this, JEDI [CDNF98a] features event as a set of ordered strings on which filters, so called patterns, can be matched; the Cambridge event model [BBHM96] supports filters in the form of templates. Both approaches allow a simple form of filtering but are limited in their expressive power. In order to support flexible and expressive filters, events are defines as a set of attributes in several event models including CORBA Notification [Cea98], ECO [Haa98], SIENA [Car98] and RT [HLS97]. Other means to control the propagation of event messages are event composition, also called combination or correlation, pre- and post-constraints and zones. We consider event composition a useful feature in some applications. However, its implementation depends on event monitors that are difficult to manage and resource intensive. ECO [Haa98] suggest the use of pre-constraints and post-constraints as a very flexible mechanism to control event delivery. Whether such a degree of flexibility is efficient, manageable and feasible in a large-scale distributed system depends on the application domain. ECO also proposes the use of zones to limit the scope of the event propagation. An addition to filters, this feature can be very useful, especially in a large-scale system, to further improve scalability, limit network traffic and reduce CPU processing time.

Many application domains require timed delivery of event messages. These real-time requirements are addressed by the CORBA Notification Service [Cea98] and by the RT Event Service [HLS97] by assigning priorities to event messages and using a dispatching mechanism as the means to control event delivery. Both solutions depend on a centralised mediator that implements the dispatching mechanism. From an applications point of view, it is preferable to rather assign a delivery deadline than a delivery priority to an event message. Hence, to have a prediction on the event delivery time. This is partly addressed by the CORBA Notification Service, but unfortunately depends on the centralised mediator. Ideally, a real-time event delivery mechanism should support the denotation of delivery deadlines, provide delivery predictions and should not depend on a centralised mediator.

Quality of Service (QoS) requirements, other than real-time, include the reliability of events and connections, the delivery order of events and the memory management of the event model, the latter being addressed by queue sizes and event discard policies. As for real-time requirements, QoS requirements are desired by many application domains, but are often omitted since they are difficult to support, especially without a centralised mediator.

Application component mobility is becoming more and more popular in distributed systems and in wide-area networks. The integration of mobile application components, e.g. of Personal Digital Assistants (PDAs), in event-based systems is supported by JEDI [CDNF98a] [CDNF98b] and SIENA [Car98]. Mobility is desirable not only to integrate addition functionality, but also to increase the flexibility and the effectiveness of a system. However, integrating mobile components into a system where entities are added and removed dynamically, is by definition hard. Combining mobility with real-time or QoS features is even harder due to the changing connection characteristic, e.g. bandwidth and reliability, among mobile and fixed components.

An advertisement mechanism and the use of multicast-based communication are supported by some event models to further optimise event propagation, thus further limiting network traffic and reducing CPU processing time. Advertisement is not only a clever way to optimise the routing of subscriptions and event messages, but may also sustain the management of mobile application components. However, to coordinate themselves, advertisements may depend on a logically centralised component, i.e. the event models mediator. The use of a multicast-based communication provides an means of one-to-many event messages propagation. Thus, the propagation of an event message by means of distributing a copy of that message to each receiver is optimised by distributing a single multicast message instead. However, commonly used multicast mechanisms, such as IP multicast, are connectionless best-effort (unreliable) services. Furthermore, the management of multicast groups requires global knowledge of the system. Thus, a logically centralised means must be available.

9.3 Conclusion

In conclusions, in can be said that the architecture and the features of an event model strongly depend on its application domain. However, scalability is a main issue in every distributed system and hence in an event model and thus, becomes more important with the increasing scale of the system. Surprisingly, many event model architectures depend on the use of a, at least logically, centralised mediator component. Such a mediator may be a single point of failure, can become a critical performance bottleneck, and does not scale well and should therefore be avoided. Scalability is commonly address by event models through the use of event filters as a means of controlling and thus limiting the propagation of events. Although other mechanisms, such as zones and pre-constraints, were proposed, filters are the most powerful means used in event models to improve scalability. Other event models features include the support of real-time, QoS and application component mobility, but as already mentioned, whether or not they are required depends strongly on the event model's application domain.

10. REFERENCES

[BBHM96] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 9-16, Connemara, Ireland, September 1996.

[Bir96] K. Birman. *Building Secure and Reliable Network Applications*, chapter 13.9 and 13.10. Manning Publishing Co., 1996.

[Bur96] G. Burke. Kanga: a framework for building application specific communication protocols. Master's thesis, Dept. of Computer Science, Trinity College Dublin, Ireland, September 1996.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley Longman Limited, second edition, 1996.

[Car98] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Italy, December 1998.

[CB98] M. Chaoying and J. Bacon. COBEA: A CORBA-based event architecture. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 117-131, Santa Fe, New Mexico, USA, April 1998.

[CCK+95] V. Cahill, A. Condon, D. Kelly, S. McGerty, K. O'Connell, G. Starovic, and B. Tangney. MOONLIGHT: VOID shell specification. Technical report, Dept. of Computer Science, Trinity College Dublin, Ireland, May 1995. Technical report TCD-CS-95-15.

[CDK94] G. Coulouris, J. Dollimore, and T Kindberg. *Distributed Systems, Concepts and Design*. Addison Wesley, second edition, 1994.

[CDNF98a] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 98)*, Kyoto, Japan, April 1998.

[CDNF98b] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. Technical report, CEFRIEL - Politecnico di Milano, Italy, August 1998. To appear in IEEE Transaction of Software Engineering (TSE).

[Cea98] NEC Corporation and Iona Technologies PLC et al. CORBAservices: Notification service specification - joint revised submission. Technical Report OMG Document telecom/98-11-01, Object Management Group, November 1998. http://www.omg.org/techprocess/meetings/schedule/Notification_Service_RTF.html.

[CG98] G. Cugola and C. Ghezzi. The design and implementation of PROSYT: an experience in developing an event-based, mobile application. In Proceeding of the IEEE 8th International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (IEEE WET ICE '99), Stanford University, Stanford, California, USA, June 1999.

[CRW98] A. Carzaniga, D. Rosenblum, and A. Wolf. Design of a scalable event notification service: Interface and architecture. Technical report, Dept. of Computer Science, University of Colorado, USA, August 1998. Technical report CU-CS-863-98.

[Gro95a] Object Management Group. *The Common Object Request Broker: Architecture and Specification, revision 2.2*. Object Management Group, 1995. http://www.omg.org/library/c2indx.html.

[Gro95b] Object Management Group. *CORBAservices: Common Object Services Specification - chapter 4, Event Service Specification*. Object Management Group, March 1995. http://www.omg.org/library/csindx.html.

[Gro96] Object Management Group. *CORBAservices: Notification Service Specification - Request For Proposal*. Object Management Group, December 1996. http://www.omg.org/techprocess/meetings/schedule/Notification_Service_RFP.html.

[Gro97] Object Management Group. *CORBAservices: Common Object Services Specification - chapter 16, Trading Object Service Specification*. Object Management Group, March 1997. http://www.omg.org/library/csindx.html.

[Gro98] Object Management Group*. CORBAservices: Management of Event Networks - Request For Proposal*. Object Management Group, September 1998. http://www.omg.org/techprocess/meetings/schedule/Mgmt._of_Event_Networks_RFP.html.

[Haa98] M. Haahr. Implementation and evaluation of scalability techniques in the ECO model. Master's thesis, Dept. of Computer Science, Trinity College Dublin, Ireland, August 1998. Also technical report TCD-CS-1999-42.

[HLS97] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 184-200, Atlanta, Georgia, USA, October 1997. ACM, New York, USA. Also technical report #WUCS-97-31, Dept. of Computer Science, Washington University, St.Louis, USA.

[HMN+00] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and scalability in the ECO distributed event model. In *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000)*, Limerick, Ireland, June 2000. Also technical report TCD-CS-00-??, Dept. of Computer Science, Trinity College Dublin, Ireland.

[KR95] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845-857, October 1995.

[Maf00] S. Maffeis. Client/server term definition. In A. Ralston, D. Hemmendinger, and E. Reilly, editors, *Encyclopedia of Computer Science, 4th edition.* International Thomson Computer Publishing, March 2000.

[MSS97] M. Mansouri-Samani and M. Sloman. GEM: A generalized event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96-108, June 1997.

[Mul93] S. Mullender. *Distributed Systems*. Addison Wesley, second edition, 1993.

[O'C97] K. O'Connell. *System Support for Distributed Multi-User Virtual Worlds*. PhD thesis, Dept. of Computer Science, Trinity College Dublin, Ireland, October 1997.

[OCC+95] K. O'Connell, V. Cahill, A. Condon, S. McGerty, G. Starovic, and B. Tangney. The VOID shell: A toolkit for the development of distributed video games and virtual worlds. In *Proceedings of the Workshop on Simulation and Interaction in Virtual Environments*, pages 172-177, University of Iowa, Iowa City, USA, July 1995. Also technical report TCD-CS-95-27, Dept. of Computer Science, Trinity College Dublin, Ireland.

[ODC+96] K. O'Connell, T. Dinneen, S. Collins, B. Tangney, N. Harris, and V. Cahill. Techniques for handling scale and distribution in virtual worlds. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 17-24, Connemara, Ireland, September 1996. Also technical report TCD-CS-96-14, Dept. of Computer Science, Trinity College Dublin, Ireland.

[OFB99] J. Orvalho, L. Figueiredo, and F. Boavida. Evaluating light-weight reliable multicast protocol extensions to the CORBA event service. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*, University of Mannheim, Germany, September 1999.

[SCT95a] G. Starovic, V. Cahill, and B. Tangney. The ECO model: Events + constraints + objects. Technical report, Dept. of Computer Science, Trinity College Dublin, Ireland, February 1995. Technical report TCD-CS-95-05.

[SCT95b] G. Starovic, V. Cahill, and B. Tangney. An event based object model for distributed programming. In *Proceedings of the 1995 International Conference on Object Oriented Information System*, pages 72-86, London, UK, December 1995. Springer-Verlag. Also technical report TCD-CS-95-28, Dept. of Computer Science, Trinity College Dublin, Ireland.

[SM97] Inc. Sun Microsystems. Java AWT: Delegation event model. http://java.sun.com/products/jdk/1.1/docs/guide/awt/designspec/events.html, January 1997.

[SM98] Inc. Sun Microsystems. *Java Distributed Event Specification*. Sun Microsystems, Inc., July 1998. http://www.javasoft.com/products/javaspaces/specs.

[SM99] Inc. Sun Microsystems. *Jini: Distributed Event Specification*. Sun Microsystems, Inc., January 1999. http://www.sun.com/jini/specs.

[Som95] I. Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1995.