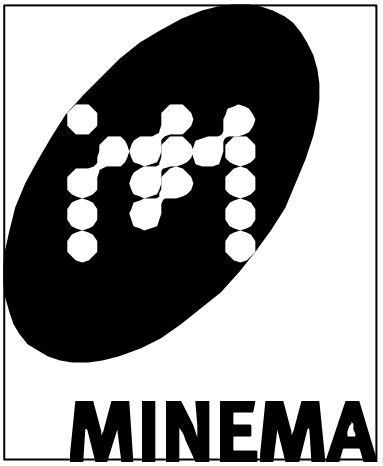


# Taxonomy of Distributed Event-Based Programming Systems[1]

René Meier and Vinny Cahill
Distributed Systems Group, Department of Computer Science
Trinity College Dublin
{rene.meier, vinny.cahill}@cs.tcd.ie

*Abstract*

Event-based middleware is currently being applied for application component integration in a range of application domains. As a result, a variety of event services have been proposed to address different requirements. In order to aid the understanding of the relationships between these systems, this paper presents a taxonomy of distributed event-based programming systems. The taxonomy is structured as a hierarchy of the properties of a distributed event system and may be used as a framework to describe such a system according to its properties. The taxonomy identifies a set of fundamental properties of event systems and categorises them according to the event model supported and the structure of the event service. Event services are further classified according to their organisation and their interaction models, as well as other functional and non-functional features.

| | |
|---|---|
| **Document Identifier** | TR-3 |
| **Document Status** | Published |
| **Created** | 20 October 2005 |
| **Revised** | 14 February 2006 |
| **Distribution** | Public |



[1] A later version of this paper has been published in the Computer Journal [1].

# Contents

# 1 Introduction

The event-based communication model represents a well-established paradigm for asynchronously interconnecting the components that comprise an application in a potentially distributed and heterogeneous environment, and has recently become widely used in a range of application areas including large-scale internet services [2] and mobile computing [3, 4].
Event-based communication is particularly useful in centralised and distributed applications that require one or more application components to react to changes occurring in other application components as it provides a one-to-many or many-to-many communication pattern [5-8]. The asynchronous nature of event-based communication [9, 10] results in a less tightly coupled communication relationship between application components compared to the traditional request/response communication model.
Event-based communication also allows application components to interact anonymously [11] without concern for either the number or the location of the components involved. Anonymous interaction allows application components to establish communication relationships relatively easily, involving modest initialisation effort compared to the request/response communication model. It is therefore well suited for accommodating communities of cooperating distributed components that establish communication relationships dynamically over time in an unpredictable fashion.
Event-based middleware is currently being applied in many application domains including finance, telecommunications, smart environments, multimedia, avionics, health care, and entertainment [2, 3, 12-19]. Moreover, with the widespread deployment and use of wireless technology, where communication relationships amongst heterogeneous application components [9] are established very dynamically during the lifetime of the components, event-based middleware will become even more prevalent as it addresses important application requirements including avoidance of long-lasting and hence potentially expensive connections, hiding of communication latency due to decoupled interaction phases, omission of centralised control, and heterogeneity. The notion of dynamically inaugurating communication relationships among application components without relying on centralised control is central to addressing the needs of a scalable system, representing the ability to accommodate growth in a potentially large-scale distributed environment.
Event-based communication models, or simply event models, are used in applications ranging from small-scale, centralised to large-scale, highly distributed systems [20]. On one hand, they are exploited to interconnect individual components of applications, for example, the components comprising graphical user interfaces [21, 22]. Such graphical components may disseminate user-driven and hence sporadic changes to their state to other components of the application that are required to react to these changes. At the other extreme, publishers of stock trading information may utilise an event service to post the latest trading rates to a group of brokers, potentially located in different cities or even countries [12, 13]. In between these extremes, smart environments often employ event-based communication models to interconnect a large number of application components [16] ranging from light and door actuators and sensors to robotic vehicles moving within and between buildings.
As event-based middleware is used in a large number of applications in a range of domains, a variety of event services have been proposed to address different application requirements. This paper presents a survey of existing event systems structured as a *taxonomy* of *distributed event-based programming systems*. Generally, a taxonomy is a classification that allows different examples of some generic type to be systematically arranged in groups or categorised according to established criteria. The taxonomy presented in this paper is

structured as a hierarchy of the properties of a distributed event system and may be used as a framework to describe an event system according to its properties. Arranging the properties identified by a taxonomy in a hierarchical manner is a common mechanism for presenting and describing systems and their features. For example, Martin et al. [23] describe their taxonomy for distributed computing systems as a hierarchy of questions and answers about the features of such systems.

The ultimate challenge of establishing a taxonomy is to identify the criteria according to which the area of interest is categorised and to arrange them systematically. Our taxonomy identifies a set of fundamental properties of event-based programming systems and categorises them according to the event model supported and the structure of the event service. Event services are further classified according to their organisation and interaction model, as well as other functional features, such as event propagation model and event filtering, and non-functional features, such as ordering semantics and security. These properties are then arranged in a hierarchical manner starting from the root of the taxonomy, which defines the relationship between event system, event service, and event model. Each property is described providing corresponding terminology.

As far as possible, categories have been chosen to be independent but nevertheless, there are some interdependencies between certain categories. These interdependencies are discussed in the relevant sections.

## 1.1 Exploiting the Taxonomy

In addition to providing a means of describing an event system, the taxonomy can be used to broadly summarise event systems and the taxonomy terminology provides a common vocabulary to be used in the general discussion of event systems. Event systems can then be discussed using the same terminology and therefore, can easily be compared with each other or can be matched against system requirements. This can lead to the identification of families of event systems that support a common feature by identifying the set of systems that support a certain set of properties. For example, the properties described in Figure 30 can be used to identify the family of event systems that supports mobility.

The taxonomy may also serve as a basis for identifying the canonical combination of the properties of an event system required by a particular application domain, simply by applying the taxonomy to a number of existing event systems used in that particular application domain and by extracting the common combination of properties. This can be useful for the requirements and design engineering of a novel event system. Moreover, the taxonomy is expected to be utilised to identify novel combinations of the properties of event systems and consequently, may serve as a basis for discovering potential research issues to be addressed in future work. This has already led us to develop STEAM [24], a location-aware event-based middleware for collaborative mobile applications.

## 1.2 Related Work

Our taxonomy presents a set of generic event system properties and hence can be used to classify virtually any distributed event-based programming system regardless of system scale or application domain. The taxonomy identifies a large variety of properties, including quality of service, mobility, and security, and describes these properties as well as possible implementation options in detail.

Existing work on describing event systems has focussed either on providing a high-level reference model or on classifying event systems for a specific application area. Barrett et al.

[25] present a framework for event-based software integration that provides a high-level model for identifying components commonly found at the heart of event-based software integration in large scale systems. This framework identifies the main components of an event system as informers, listeners, registrars, routers, message transformer functions, and delivery constraints. The framework describes the relationships among these components in detail using an object-oriented type model, but does not specify possible patterns of interaction between informers and listeners. Moreover, it does not explicitly identify functional event system features and omits non-functional features altogether.

The work of Rosenblum and Wolf [26] on a design framework for event observation and notification has focussed on supporting the construction of large-scale, event-based systems for the Internet. This framework comprises seven models, namely the object, event, naming, observation, time, notification, and resource models, to capture many of the design dimensions relevant to Internet-scale applications. Even though each of these models is discussed in detail, the overall number of properties according to which an event system may be classified is substantially smaller compared to the taxonomy presented in this paper. This is due to the fact that this framework imposes certain constraints in order to specifically support Internet-scale event observation and notification and because certain issues, such as quality of service, mobility, and security, have not been considered.

Eugster et al. [27] identify the common denominators of variants of the publish/subscribe interaction scheme using three dimensions. These dimensions describe the decoupling between producers and consumers of information in terms of time, space, and synchronisation. This work focuses on implementation issues related to event dissemination, the underlying media, and quality of service aspects, and as such does address other functional and non functional features, such as mobility support, failure mode, and security mechanisms.

## 1.3 Interpreting the Taxonomy

This taxonomy is presented using both figures and corresponding text. The figures outline the relationships among the fundamental properties of event systems and define the terminology to identify them. The text associated with each figure describes the corresponding properties in detail. The figures allow a taxonomy user to easily trace paths through the hierarchy to discover relevant properties. As summarised in Figure 1, the figures consist of nodes representing properties of interest, one of which is the root node and some of which are leaves. Nodes are connected by directed paths. The directed paths are represented by a set of arrows describing the nature of the paths leaving a specific node. A set of dashed arrows leaving a specific node indicates that *exactly one* path has to be chosen when tracing through that node. Solid arrows indicate that *at least one* path has to be chosen, whereas double lined arrows indicate that *all possible paths* need to be followed in parallel. In order to apply the taxonomy to an event system, a taxonomy user traces paths through the hierarchy starting from the root node and selecting the connections that most accurately describe the event system until each selected path reaches a leaf. The terms associated with the nodes along a path describe a property of the event system.

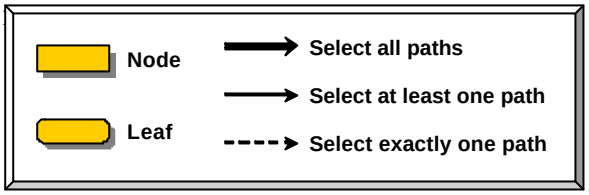


**Figure 1. Taxonomy legend.**

For example, Figure 21 shows that the features of an event service include both functional *and* non-functional features by using double lined arrows to describe the paths between the nodes. Hence, when tracing through the features node, all paths, i.e., both of them, must be selected to describe the corresponding properties of the event system. The solid arrows connecting the nodes in Figure 22 indicate that one kind of event propagation model can be provided by an event service, although some event services may support both the sporadic and the periodic event propagation models. Therefore, either one or both paths may be traced. Figure 4 shows that an event model can be characterised as *either* peer-to-peer, mediator, or implicit. The dashed arrows connecting the nodes, which imply that exactly one path has to be chosen, illustrate this.

# 2 The Taxonomy

The root of the taxonomy, which is depicted in Figure 2, defines the relationship between an event system, an event service and an event model. Every event system has both an event service and an event model, which we define as follows:

- An **event system** is an application that uses an event service to carry out event-based communication.
- An **event service** is middleware that implements an event model, hence providing event-based communication to an event system.
- An **event model** consists of a set of rules describing a communication model that is based on events.

We differentiate between event service and event model in order to capture the facts that an event model defines an application-level view of an event service and that a range of different event services may implement a given event model. Event models essentially reflect the different uses for which they are intended. For example, the objectives of the Java AWT delegation event model [21] differ substantially form those of the CORBA notification service model [28], which leads to major differences in the Application Programming Interfaces (APIs) that they provide. The goal of the CORBA notification service model is to be extremely general-purpose and usable in virtually any domain. Consequently, it supports a wide range of features including typed and untyped event communication, as well as filtering and administrative capabilities. Moreover, a variety of quality of service properties, such as event reliability, connection reliability, event priority, and event delivery order, are supported to control the propagation characteristics of events. This is reflected in a fairly large and complex API. In contrast, the Java AWT delegation event model is intended for small-scale, centralised applications, such as graphical user interfaces, and therefore omits many of the features of the CORBA event model. This results in its API being much simpler than that of the CORBA event model.

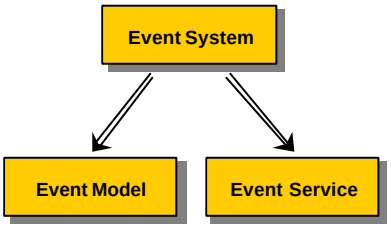


**Figure 2. The root of the taxonomy.**

The CORBA event model also serves as an example of an event model that was specified with the expectation of being implemented by a range of event services, and potentially being exploited in different application domains. The Object Management Group (OMG) leaves open the implementation of their model and therefore, leaves it to different vendors to provide implementations. Consequently, event services supporting the CORBA event model have been implemented and extended by a number of commercial and academic organisations [29], [5], [30].

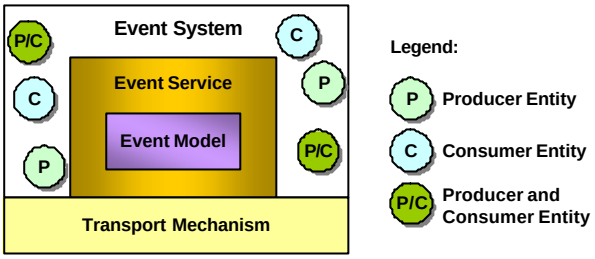


**Figure 3. Event system overview.**

The relationships between event system, event service and event model are summarised from the event system's perspective in Figure 3. Apart from depicting how an event system uses an event service that implements a particular event model, Figure 3 also outlines how event system and service map onto a transport mechanism and how applications use entities as hooks into the event service. Entities are the components of an application that produce and consume events, excluding components of the event service. An entity may play the role of a producer and/or a consumer of events.

There is no generally accepted standard terminology used for the application components that act as consumers or producers of events. As a result, the event systems presented in this paper use a variety of alternative terminology, which is summarised in Table 1, when referring to event producers and consumers. We use the systems outlined in Table 1 later in this paper to illustrate the properties identified by our taxonomy.

**Table 1. Event system terminology.**

| Event System | Producer | Consumer |
|---|---|---|
| CEA [10, 31] | Source object | Client object |
| CONCHA [5] | Multicast supplier | Multicast consumer |
| CORBA [28, 32] | Supplier | Consumer |
| COSMIC [33, 34] | Publisher | Subscriber |
| ECO [6, 35] | Object | Object |
| Elvin [36, 37] | Producer | Consumer |
| Elvin Agents [38, 39] | Producer | Consumer |
| Gryphon [7, 40, 41] | Publisher | Subscriber |
| Hermes [42-44] | Publisher | Subscriber |
| Java AWT [21] | Source | Listener |
| Java Distributed [45] | Generator | Listener |
| JEDI [46] | Active object | Active object |
| Mobile Push [47] | Publisher | Subscriber |
| Obvents [48, 49] | Publisher | Subscriber |
| Rebeca [50, 51] | Producer | Consumer |
| SECO, uSECO, mSECO [6] | Object | Object |
| SIENA [52] | Object of interest | Interested party |

| Event System | Producer | Consumer |
|---|---|---|
| STEAM [24, 53-56] | Producer | Consumer |
| TAO RT CORBA [17, 30] | Supplier | Consumer |
| ToPSS [57-59] | Publisher | Subscriber |

## 2.1 Event Model

The **event model** defines the manner in which an event service is made visible to the application programmer. It specifies the components of an event service to which the application programmer is explicitly exposed and that are used to subscribe to events and to propagate them. In particular, the event model classifies the means by which consumers subscribe to the events in which they are interested and the means by which an application raises and delivers events, as well as the number and location of the components used. As shown in Figure 4, we have identified three distinct categories of event model, which are peer-to-peer, mediator, and implicit.

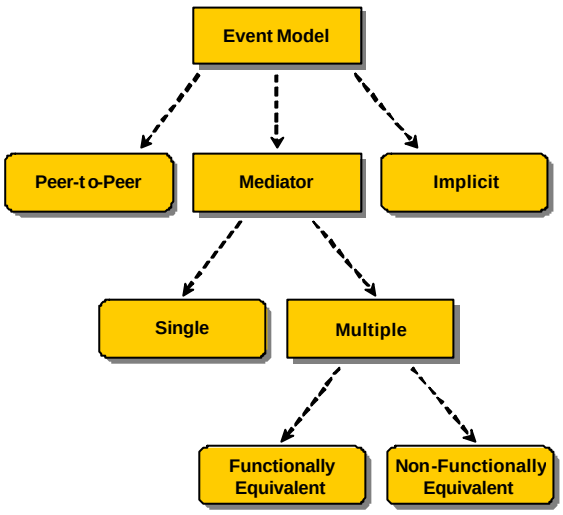


**Figure 4. Event model categories.**

### 2.1.1 Peer to Peer

A **peer to peer** event model allows consumers to subscribe at specific named producers *directly* and producers to deliver events to specific named consumers *directly*. The Java distributed event model is based on a peer-to-peer event model allowing a `RemoteEventListener` to subscribe to events by invoking a `register` method on an explicitly named `EventGenerator`.

```
TheConsumerApplication {//the RemoteEventListener
  //subscribe to an explicit producer
  AnExplicitEventGeneratorRef = retrieveEventGeneratorRef();
  AnExplicitEventGeneratorRef.register(this);

  //delivery handler implementation
  notify(TheRemoteEventInstance) {
    processAnEvent(TheRemoteEventInstance);
  }
}


TheProducerApplication {//the EventGenerator
  //register method implementation
  register(RemoteEventListenerRef) {
    SubscribedRemoteEventListenerRef = RemoteEventListenerRef;
  }

  //raise an event
  AnEventInstance = new Event(someParameters);
  SubscribedRemoteEventListenerRef.notify(AnEventInstance);
}
```

**Figure 5. A producer and a consumer application using the peer-to-peer Java distributed event model.**

The simplified application shown in Figure 5 outlines a subscribing `RemoteEventListener` and an `EventGenerator` invoking the `notify` method on a subscribed `RemoteEventListener` using a `RemoteEventListener` reference to deliver a specific event instance.

### 2.1.2 Mediator

Event models utilising a **mediator** allow consuming entities to subscribe at a *designated* mediator and producing entities to deliver events to the mediator, which then forwards them to the subscribed entities.

The mediator sub-hierarchy explores the number and functionality of mediators in the event model. We differentiate between models utilising a **single** mediator and models exploiting **multiple** mediators. The CORBA event model[2] may use a single mediator (known as an event channel) for propagating all events from producers to consumers. Multiple mediators are further divided into functionally equivalent and non-functionally equivalent mediators. In the former, all mediators are **functionally equivalent**. Thus, entities may subscribe or deliver events to any one of them. Such a mediator is called an event server in the SIENA model. SIENA may use a set of different event server topologies of which all but the centralised topology exploit multiple, functionally equivalent event servers. When mediators are **not functionally equivalent**, entities have to subscribe or deliver events to the correct mediator. For example, an application exploiting the CORBA event model[3] may use multiple event channels each propagating a different type of event.

The simplified application shown in Figure 6 outlines how both CORBA consumers and producers connect to the explicitly-named event channel through which they intend to

[2] The CORBA specification allows its event model to use a single or multiple mediators. For the purpose of this example, we refer to a CORBA event model utilising a *single* mediator.

[3] The CORBA specification allows its event model to use a single or multiple mediators. For the purpose of this example, we refer to a CORBA event model utilising *multiple* mediators.

exchange events. Connected producers may raise events by pushing them to the event channel, which forwards them to all subscribed consumers by invoking their delivery handlers in turn.

```
TheConsumerApplication {
  //connect to an explicit event channel
  ConsumerAdmin = TheEventChannel.forConsumers();
  ProxyPushSupplier = ConsumerAdmin.obtainPushSupplier();
  ProxyPushSupplier.connectPushConsumer(TheConsumer);
}

TheConsumer {
  //delivery handler implementation
  push(TheRemoteEventInstance) {
    processAnEvent(TheRemoteEventInstance);
  }
}


TheProducerApplication {
  //connect to an explicit event channel
  SupplierAdmin = TheEventChannel.forSuppliers();
  ProxyPushConsumer = SupplierAdmin.obtainPushConsumer();
  ProxyPushConsumer.connectPushSupplier(TheSupplier);
}

TheSupplier {
  //raise an event
  AnEventInstance = new Event(someParameters);
  ProxyPushConsumer.push(AnEventInstance);
}
```

**Figure 6. A producer and a consumer application using the mediator-based CORBA event model.**

### 2.1.3 Implicit

An **implicit** event model allows consuming entities subscribe to particular event types rather than at another entity or a mediator. Producers generate events of some type, which are then delivered to the subscribed consumers. The direct approach for CEA source objects to disseminate events to client objects, described by Bacon et al. [31], is based on an implicit event model. Figure 7 shows a simplified version of an active badge application using direct CEA. The consumer subscribes by invoking a register method provided by a local library passing the event type of interest as well as a reference to its delivery handler. The producer declares its event type and subsequently raises events of this type by invoking a signal method provided by a local library. The event service delivers events to all registered consumers by calling their delivery handlers.

```
TheConsumerApplication {
  //subscribe to an event type
  template = Badge_Seen(17, 29);
  EventClient.Register(EventHandler, template);

  //delivery handler implementation
  EventHandler(TheRemoteEventInstance) {
    processAnEvent(TheRemoteEventInstance);
  }
}


TheProducerApplication {
  //specify the event type
  Badge : INTERFACE = Seen : EVENTCLASS [badge : BadgeId; sensor : SensorId];
  END.

  //raise an event
  e = Badge_Seen(17, 29);
  EventSource.Signal(e);
}
```

**Figure 7. A producer and a consumer application using the implicit Direct CEA.**

### 2.1.4 Discussion

An event system exploiting either a peer-to-peer or a mediator-based event model allows its entities to interact by invoking remote methods *directly* on each other or on one or more mediators respectively whereas entities of an event system with an implicit event model interact by subscribing and delivering events locally using event types.
Significantly, these approaches differ in the way the identifiers of the components exposed to the application programmer are obtained and maintained. Peer-to-peer and mediator-based event models require the application programmer to obtain the identifiers of specific producers or mediators respectively, usually by means of a naming service, and to maintain them. Every consumer of an event system utilising a peer-to-peer event model is required to obtain the identifier of each producer in which it is interested, i.e., the application programmer must ensure a consumer subscribes to the correct set of producers, and to maintain the correct set of subscriptions during its lifetime. Similarly, entities of an event system using a mediator-based event model need to acquire the identifiers of the mediators involved, i.e., the application programmer must track the identifiers to the mediators to which a specific entity needs to connect. However, mediator-based event models are likely to obtain and maintain a smaller number of different identifiers compared to peer-to-peer models. There are likely to be significantly fewer mediators in an event system than producers and their number is unlikely to change over time[4], certainly compared to the number of producers as they may be created frequently to provide services for a limited period of time. In contrast, the application programmer in an event system with an implicit event model is not required to acquire the identifiers of producers or mediators at all. The application programmer does not need to explicitly identify the producers with which a consumer needs to communicate as consumers subscribe to producers transparently using event types. This requires a more sophisticated event service as it is responsible for locating peers, maintaining the corresponding identifiers,

[4] An event system may exploit a single mediator whose reference characteristically remains unchanged, assuming the absence of failure, during the lifetime of the system.

mapping event types to identifiers, and for providing a means to define and check the type of events.

Most significantly, the event model exploited by an event system affects one of the main concepts of event-based communications, namely the degree of anonymity among the entities in the system. The means by which consumers subscribe to the events in which they are interested and by which events are propagated and delivered influences the degree of anonymity among them. The peer-to-peer approach permits specific named entities to interact directly with each other. Consequently, entities are not anonymous to each other. Mediator-based event models, where entities register with one or more mediators, provide a degree of anonymity where entities are anonymous to each other but known to the mediator(s). The implicit approach allows entities to interact anonymously. Such entities are anonymous to each other and are only known by the event service that implements the mapping of event types to entities. Nevertheless, entities may choose to identify themselves at the application level regardless of the degree of anonymity provided by the underlying event model. This may be useful for example, in applications that wish to assess the level of trust between producers and consumers.

## 2.2 Event Service

This section deals with the classification of the properties of event service middleware. As Figure 9 shows, we divide the properties of an event service into three distinct categories. The organisation sub tree focuses on the distribution of the producers and consumers as well as the components of the middleware and on the fashion in which the components that comprise an event service cooperate. The interaction model defines the communication path over which event producers and consumers communicate with each other. The feature sub hierarchy addresses the other (functional and non-functional) features provided by an event service.

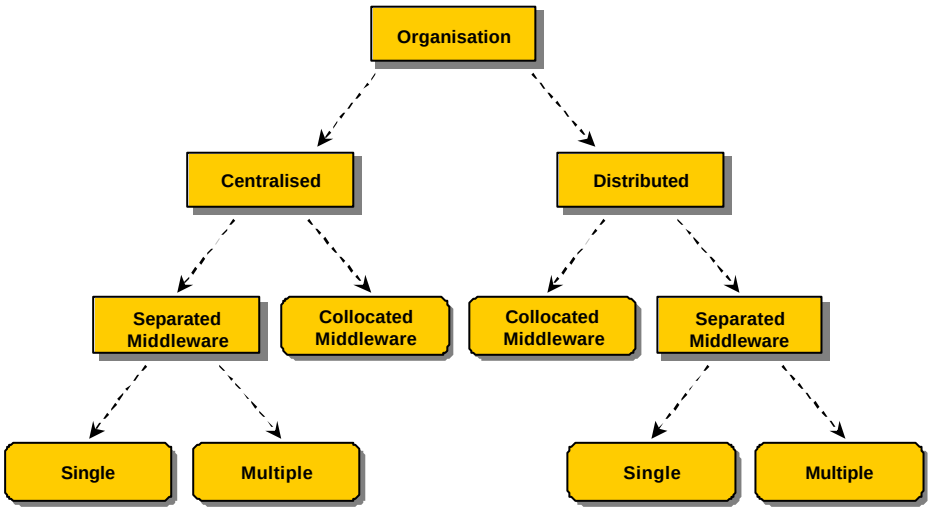


**Figure 8. Event service organisation.**

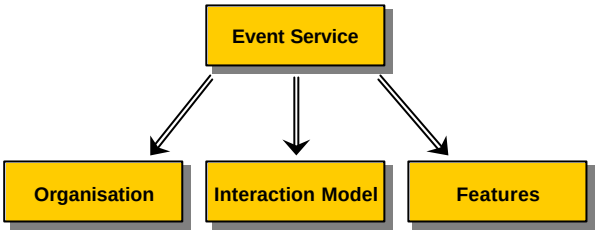


**Figure 9. The event service.**

### 2.2.1 Organisation

As summarised in Figure 8, the **organisation** sub tree classifies an event service as either centralised or distributed according to the location of the event system's entities. These two sub categories are further divided exploring the location of the event service's components.
The entities of an event system are **centralised** if they reside in the same address space on the same physical machine. In contrast, if the entities of an event system are **distributed** they may be located in different address spaces possibly on different physical machines.
Whether the entities of an event system are centralised or distributed, the middleware can be either collocated or separated.

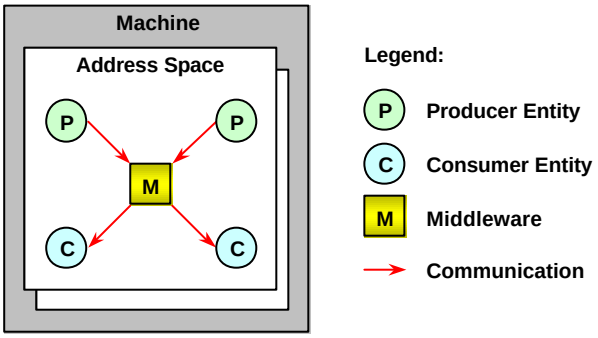


**Figure 10. Centralised event service with collocated middleware.**

**Collocated Middleware.** The event service is collocated with the entities if it resides only in the same address space(s) on the same physical machine(s). As illustrated in Figure 10, the organisation of a centralised event service with collocated middleware results in both the entities and the middleware being located exclusively in the same address space. No part of the event system resides outside the implicit single address space. This organisation may be used for small-scale applications consisting of a relatively small number of entities, such as graphical user interfaces. For example, the Java AWT delegation event model is implemented by the Java Virtual Machine (JVM) to connect the graphical components of an application sharing their address space with the middleware. Another event service that may be used in a similar fashion is provided by the C# programming language [22]. In contrast, the organisation of a distributed event service with collocated middleware results in the middleware being distributed with the entities, each entity using the part of the middleware that is local to it. Figure 11 shows the organisation of a distributed event service with collocated middleware, which may include an arbitrary number of address spaces.

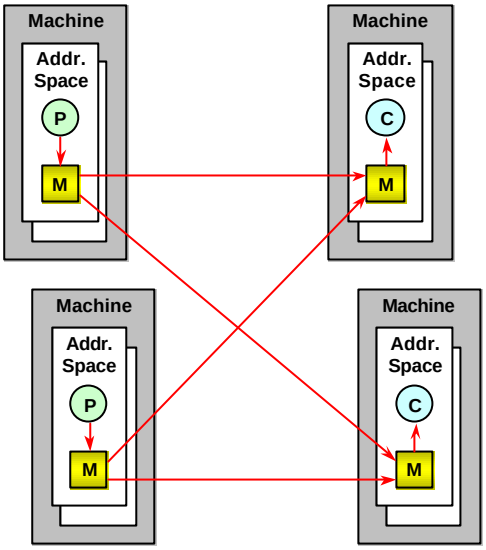


**Figure 11. Distributed event service with collocated middleware.**

This organisation has been adopted by mSECO, an event service implementing the ECO event model. mSECO is implemented as a library that is collocated with each entity. Notably, mSECO is exclusively located in the same address spaces as the entities. Moreover, the address spaces in which the entities reside may or may not be located on different physical machines. Likewise, STEAM adopts this organisation in order to avoid dependence on a service infrastructure other than the machines hosting producers and consumers. This enables STEAM to support the wireless, ad hoc networks for which it has been designed.

**Separated Middleware.** In this case, the event service is at least partially located in one or more separate address spaces possibly on different physical machines. We divide separated middleware into two categories depending on the partitioning of the middleware. Figure 12 depicts an event service with separated **single** middleware, whose entities are centralised and whose middleware is located in a separate address space. This organisation uses exactly two separate address spaces, one including the entities and the other containing the middleware. The two address spaces may reside on the same or on two different physical machines.

Figure 13 illustrates a distributed event service with separated single middleware, whose entities are distributed and whose middleware is located on a single machine. This organisation may involve a large number of address spaces and possibly physical machines, depending on the location of the entities and the middleware. However, all the address spaces may reside on a single physical machine. A CORBA event service providing a single event channel[5] serves as an example of such an organisation. Its entities typically reside in different address spaces distributed over multiple physical machines using an event channel located on another machine. However, the address space in which the event channel resides may be located on the same physical machine as some of the entities' address spaces.

[5] The CORBA event service may utilise one or more event channels. For the purpose of this example, we refer to a CORBA event service utilising a *single* event channel.

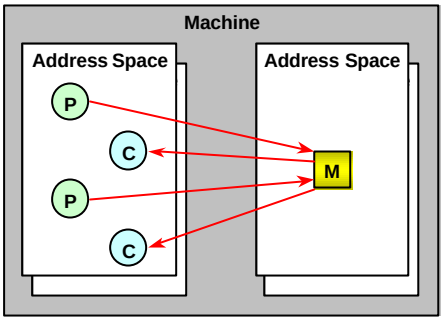


**Figure 12. Centralised event service with separated single middleware.**

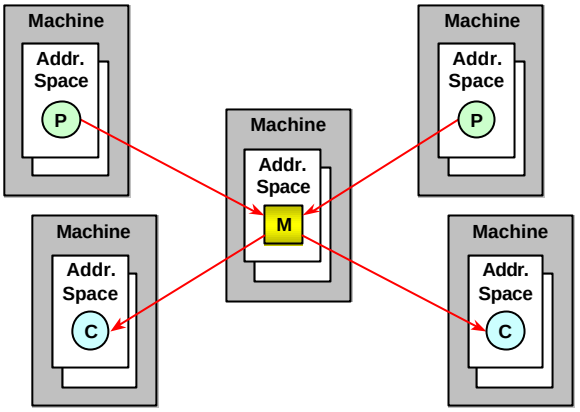


**Figure 13. Distributed event service with separated single middleware.**

Figure 14 and Figure 15 show event services with separated multiple middleware, whose middleware is distributed over a set of cooperating address spaces possibly on different physical machines, for a centralised and a distributed organisation respectively.
Figure 15 also illustrates that some of the middleware's address spaces may be located on the same machines as some of the entities. This also applies for centralised entities with separated multiple middleware. We admit the possibility of an organisation supporting centralised entities with separated multiple middleware although we cannot provide an example for such an organisation. SIENA, which uses an organisation as shown in Figure 15, proposes a set of middleware topologies, called server topologies, of which all but the centralised topology use middleware that is distributed over a set of cooperating machines.

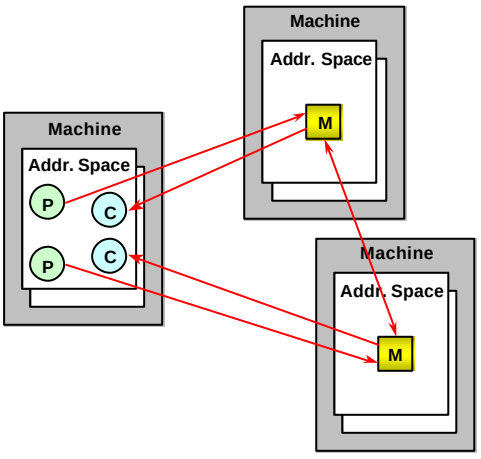


**Figure 14. Centralised event service with separated multiple middleware.**

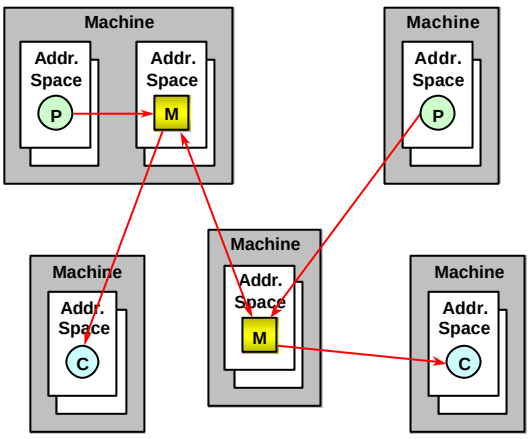


**Figure 15. Distributed event service with separated multiple middleware.**

**Discussion.** The organisation adopted by an event service has a major impact on issues related to the scalability of the system, its behaviour in the presence of failed components, and on the mechanism for communication between entities and the middleware. Conventionally, approaches containing centralised middleware components are more likely to experience performance bottlenecks with increasing scale and tend to suffer more in the presence of

failures than distributed approaches. The use of middleware located in multiple address spaces allows the distribution of the communication load reducing the risk of performance bottlenecks. Instead of having middleware located in a single address space handling all the communication between the entities in an event system, middleware distributed over multiple address spaces may divide the load. Exploiting middleware distributed over multiple address spaces also avoids potential single points of failure in the system. For example, if the middleware in the organisations illustrated in Figure 10, Figure 12 and Figure 13 fails none of the entities in the corresponding systems will be able to communicate. In contrast, a middleware component failing in one of the organisations depicted in Figure 11, Figure 14, or Figure 15 has a less devastating effect on an event system allowing the entities to communicate even in the presence of failure. Significantly, this depends on the middleware being located in multiple address spaces and not on the distribution of the entities in a system.
The organisation of an event service also affects the mechanism through which entities communicate with the middleware. Approaches where entities and middleware reside in different address spaces distributed over different physical machines require a mechanism that supports cross-network communication. A much simpler inter-process communication mechanism may be sufficient for organisations where entities and middleware reside in different address spaces on the same physical machine. Entities and middleware sharing an address space may communicate using a programming-language-specific mechanism, such as procedure call or method invocation.
This taxonomy may serve as a basis for identifying the combinations of event system properties that are well suited as well as the combinations that are less suited or even incompatible. For instance, mediator-based event models map well onto event service organisations with separated middleware. Separated middleware residing in an independent address space may naturally implement a mediator to which producers and consumers may connect. Peer-to-peer and implicit event models are well suited for organisations with collocated middleware. These organisations allow entities to directly connect to each other using interfaces specified by the collocated middleware, which provides a means for mapping events and their types to entities. In addition, the centralised organisation with collocated middleware may map onto mediator-based event models as the collocated middleware may implement a mediator. In contrast, combinations based on separated middleware and peer-to-peer event models, are less suitable as peer-to-peer models imply that entities interact directly.

### 2.2.2 Interaction Model

The interaction sub tree classifies an event service according to the **interaction model** used by the event system. Generally, the interaction model defines the communication path over which event communication between event producers and consumers takes place. It defines the number of intermediate middleware components involved and the manner in which intermediates cooperate to route events from the producers to consumers. Compared to the organisation model, which focuses on the distribution of the entities and the middleware of an event system, i.e., providing a static view of an event service, the interaction model describes the information flow in an event system. Hence, it describes the dynamic aspect of an event service.
As Figure 16 depicts, we divide the interaction model into two main categories, namely intermediate and no intermediate, exploring whether and how many intermediate middleware components an event passes through.

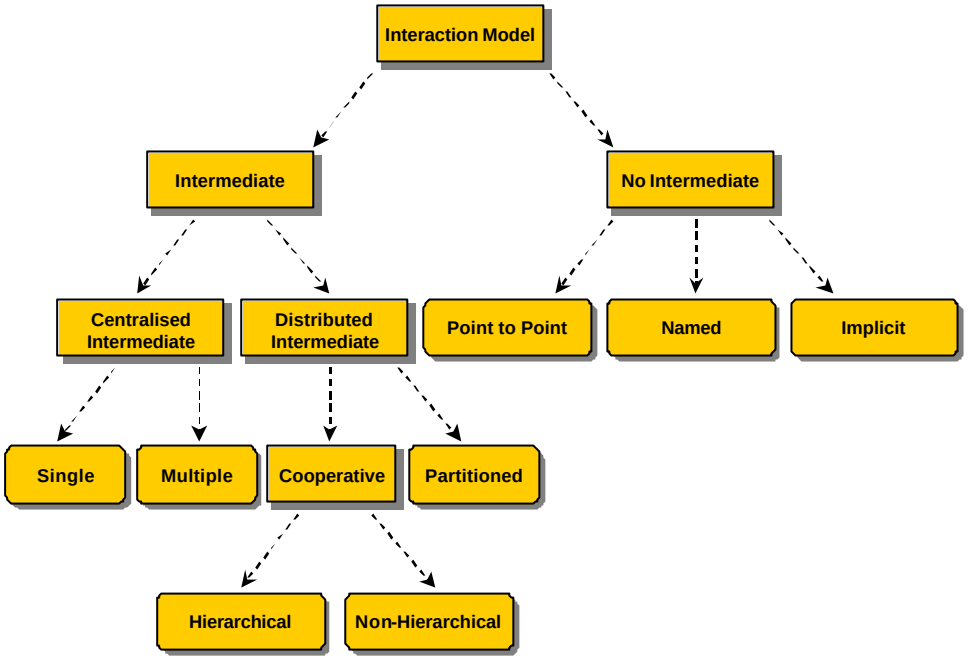


**Figure 16. Event service interaction model.**

**No Intermediate.** The communication path over which event communication between producers and consumers takes place does not include separated intermediate middleware components. Producers and consumers communicate with each other through the middleware collocated with each entity. As Figure 17 illustrates, events that are routed from producers to consumers pass through the collocated middleware, but not through any intermediate middleware component.

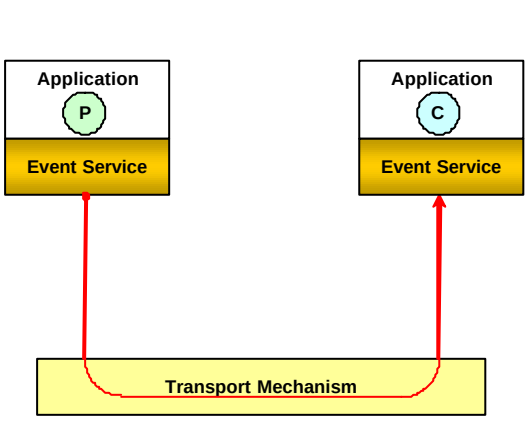


**Figure 17. No intermediate.**

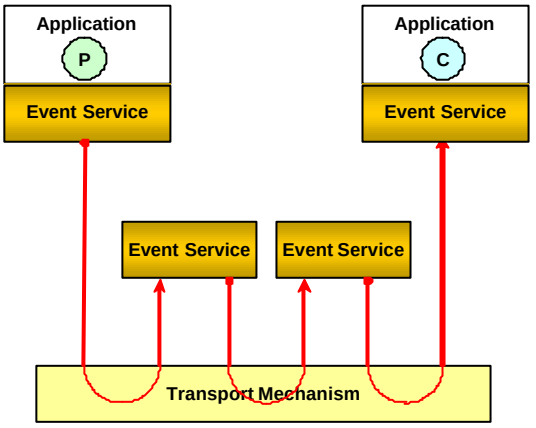


**Figure 18. Distributed intermediate.**

We sub divide this model into three categories according to the means by which entities address each other. These interaction models are called the point-to-point, named, and implicit models.

Producer and consumer entities may communicate directly with each other in a **point-to-point** fashion, using explicit entity addresses, which are provided by the application. The

middleware uses explicit entity addresses and a unicast communication pattern when routing events from producing to consuming entities. The Java distributed event model allows producers to route events to the subscribed consumers using the explicit consumer addresses provided by the application.

Producer and consumer entities may communicate directly with each other using a **name** service to map event descriptions, such as event types, to entity addresses provided by the application. The middleware uses either a unicast or a multicast communication pattern to route events from a producer to the interested consumers. uSECO uses a name service, called the Application Instance Repository (AIR), to resolve the addresses of the entities that are interested in a specific event type and a unicast communication pattern to route events.

Producers and consumers may communicate directly with each other using an **implicit** means to map event descriptions to entity addresses provided by the application. The middleware uses a multicast communication pattern when routing events from producers to consumers. mSECO, a multicast version of the uSECO event service, does not rely on an AIR since it uses an implicit means, based on generating addresses from event descriptions, to map events to the multicast addresses representing the interested consumers.

**Intermediate.** The communication path over which event communication between producers and consumers takes place includes at least one separated intermediate middleware component. Thus, events that are routed from producers to consumers pass through one or more intermediate middleware components.

The intermediate interaction model is divided into two sub categories according to the number of intermediate middleware components in the communication path. In the **centralised intermediate** model, the communication path includes a single intermediate middleware component. In contrast, the **distributed intermediate** model involves two or more intermediates through which events are routed. Figure 18 depicts the distributed intermediate interaction model with a communication path that includes two distributed intermediates.

Both centralised and distributed intermediates can be divided further. We classify centralised intermediates according to their number as an event service may exploit one or multiple centralised intermediates.

All communication paths between producing and consuming entities may include the same **single** centralised intermediate. An event system using this interaction model includes exactly one centralised intermediate. In contrast, an event system may exploit **multiple** centralised intermediates. In this case, producers and consumers are divided into groups and all communication paths between the producers and consumers within each group include a centralised intermediate that is specific to that group. This results in an event system that uses several centralised intermediates, the number of which corresponds to the number of groups. Multiple centralised intermediates may be used to support groups of entities that share a common interest. The common interest of an individual group may be expressed by a specific type of event that is handled exclusively by a particular centralised intermediate. For example, the CORBA event service may utilise multiple centralised intermediates implemented as event channels. Each channel may handle a specific type of event exclusively. Producers and consumers intending to communicate using a specific event type connect to the corresponding event channel, therefore defining the communication path over which event communication takes place. Alternatively, the CORBA event service may use a single centralised intermediate implemented as a single event channel through which all events are routed. Figure 19 and Figure 20 illustrate the single centralised intermediate and multiple centralised intermediate interaction models respectively. Figure 20 shows two groups of entities, each comprising of a producer and a consumer using a single centralised intermediate through

which events are routed. The communication path associated with one group is outlined with solid arrows and the communication path associated with the other is depicted using dashed arrows.

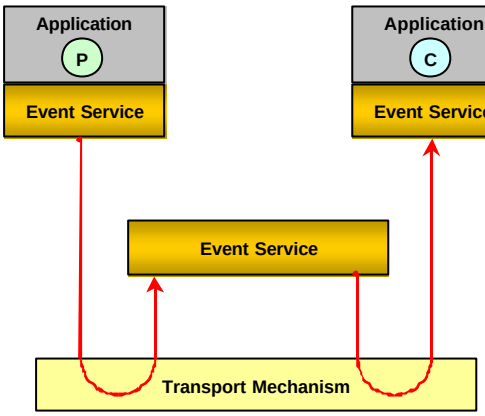


**Figure 19. Single centralised intermediate.**

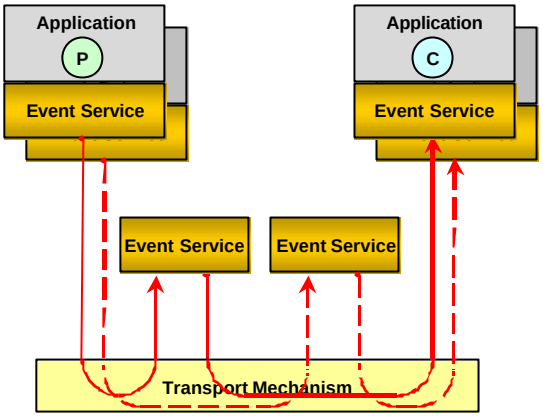


**Figure 20. Multiple centralised intermediate.**

We classify distributed intermediates as partitioned or cooperative according to the fashion in which intermediates cooperate to route events from event producers to consumers.

Generally, the distributed intermediate interaction model includes two or more intermediate middleware components in the communication path between consumers and producers. An event service implementing the **partitioned distributed intermediate** interaction model consists of a number of independent groups of intermediates, each group handling only a specific type of event. Entities sharing a common interest need to connect to the group that handles the type of event that corresponds to their common interest. For example, the CORBA event model specification proposes to chain different implementations of event channels, acting as a group of partitioned distributed intermediates, in order to combine non-functional features supported by individual event channels.

In contrast, **cooperative distributed intermediates** do not form independent groups, all intermediates cooperate to route events from consumers to producers. Entities connect to the most convenient, e.g., physically closest, intermediate. Each intermediate manages the events for the entities that are physically connected to it and cooperates with other intermediates to route them to remote entities. Cooperative distributed intermediates cooperate with each other either in a **hierarchical** or in a **non-hierarchical** manner.

JEDI proposes a hierarchical structure of cooperative distributed intermediates, called dispatching servers. Dispatching servers are interconnected in a tree topology through which events are routed. Entities may connect to any dispatching server, each of which forwards the events it receives from the producers connected to it to its parent and to its descendants to route them to all interested consumers. SIENA describes four different topologies of cooperative distributed intermediates. One of them serves as an additional example of hierarchical cooperative distributed intermediates, another two, namely the acyclic and the so-called peer-to-peer topologies, illustrate non-hierarchical cooperative distributed intermediates.

Burcea et al. [58] use a tree-based topology of cooperative distributed intermediates in a simulation of a network of ToPSS event brokers servicing an urban area. Producers may be co-hosted with and consumers may connect to any of these intermediate brokers. Brokers

route events from producers to subscribers and are capable of storing state for deferred transfer to temporarily unavailable subscribers.
Gryphon assumes a network of non-hierarchical brokers to which producers and consumers can connect at their convenience. Gryphon organises these brokers into a logical tree structure, called the spanning tree, that allows for efficient matching of events to subscribers, i.e., to efficiently determine the set of consumers interested in a specific event. Hermes introduces the notion of an overlay routing network for organising a network of nodes into a non-hierarchical application-level network of event brokers. Producers and consumers connect to the broker network and individual brokers subsequently route events through the overlay network.
**Discussion.** Mediator-based event models map naturally onto interaction models that include intermediate middleware components. For example, interaction models using either multiple centralised or partitioned distributed intermediates may implement event models that include multiple non-functionally equivalent mediators. These event models expose mediating application components to the application, which must ensure entities subscribe to the correct intermediate middleware component. Cooperative distributed intermediates may implement multiple functionally equivalent mediators whereas a single centralised intermediate may implement an event model based on a single mediator. Both the named and the implicit interaction model are appropriate for implicit event models, since neither of them relies on intermediates and because implicit event models do not prohibit the use of middleware components providing naming services. The peer-to-peer event model exposes entities explicitly to the application. It is therefore best implemented by a point-to-point based interaction model using these entity addresses to route events from producers to consumers.
There are numerous possible combinations of interaction and organisation models as many organisations are appropriate for different interaction models. For example, both centralised and distributed organisations with separated middleware are suitable for interaction models whose communication paths between producers and consumers involve intermediate middleware components. Distributed organisations with collocated middleware may be combined with interaction models that do not rely on intermediates. Centralised organisations with collocated middleware may possibly be combined with every interaction model. Although centralised collocated organisations may be best suited for the single intermediate interaction model as its middleware component maps naturally onto a single intermediate, it is also appropriate for the implicit interaction model with its middleware component implementing a means to map event types to entity addresses.

### 2.2.3 Features

The **features** supported by an event service can be classified as either **functional** or **non-functional** as shown in Figure 21.
These functional and non-functional features address requirements regarding the functional and non-functional behaviour of a system. Functional requirements are statements of services a system should provide, how a system should react to particular inputs and how a system should behave in particular situations. In some cases, the functional requirements may also explicitly state what a system should not do [60, p.118]. Non-functional requirements are constraints on the services or functions offered by a system. They include timing constraints, constraints in the development process, standards to be adopted and so on [60, p.119].
Based on these definitions, we classify the functions made available by an event service as functional features and consider constraints on (or properties of other) system attributes as non-functional features. For example, we classify mobility support as a functional feature

because it describes services to enable event-based communication for mobile entities but threat security as a non-functional feature since it describes techniques that address how event-based communication is secured.

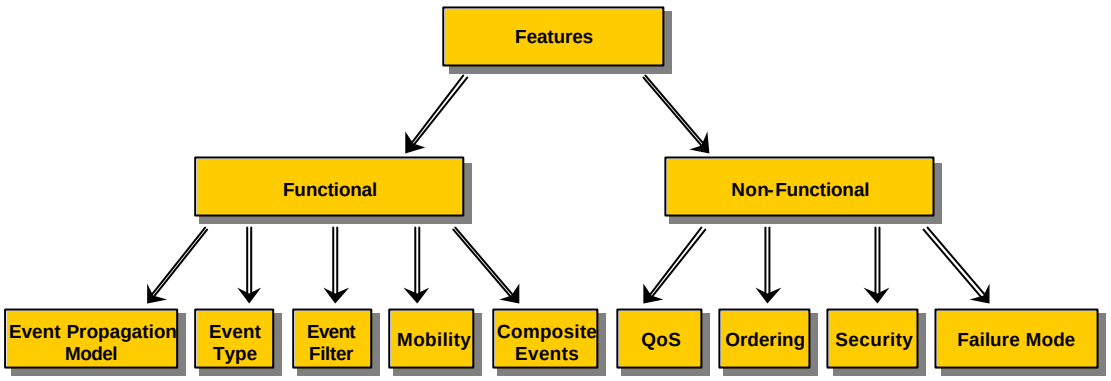


**Figure 21. Event service features.**

### 2.2.3.1 Functional Features

**Event Propagation Model.** Events are delivered by an event service according to an event propagation model. Figure 22 depicts the event propagation model sub hierarchy and shows how the event propagation model is divided into two categories describing sporadic and periodic event propagation. **Sporadic** event propagation models propagate events only if the relevant state of the producer has changed. **Periodic** event propagation models propagate events periodically, even if no state change has occurred since the last event.

Both sporadic and periodic event propagation can be based either on the push or the pull model. The **sporadic push** model is considered the traditional event propagation model and is therefore most likely to be supported by an event service. However, an event service may support several of the propagation models shown in Figure 22.

Event propagation based on the **sporadic push** model is producer-driven and producers propagate events as they are generated. The sporadic push model is supported by many event models including the Java AWT delegation event model, CORBA-based event models, Mobile Push, Obvents, ToPSS, and STEAM.

Event propagation based on the **sporadic pull** model is also known as event polling. Event propagation is consumer-driven as consumers poll producers for available events. Event producers propagate events in response to requests from consumers. Among others, this propagation model is supported by the CORBA notification service, by Obvents, and by ToPSS.

Event propagation based on the **periodic push** model is well suited for “heartbeat” or “watchdog” mechanisms as well as for disseminating events according to a predefined schedule. Event propagation is producer-driven and producers propagate events periodically. Both the COSMIC and TAO RT CORBA event services use the periodic push propagation model as a means to statically schedule event propagation while reserving the required resources for events that have hard real-time delivery deadlines.

Event propagation based on the **periodic pull** model represents traditional polling. Event propagation is consumer-driven as consumers poll producers periodically. Producers propagate events in response to requests from consumers.

Periodic event propagation models imply that events with identical content may be propagated as the state of the producer may not have changed since the previous event was propagated. We argue that periodic events still conform to our definition of events when considering the passage of time as a change to a producer's state even though periodic events may not contain an explicit description of time.

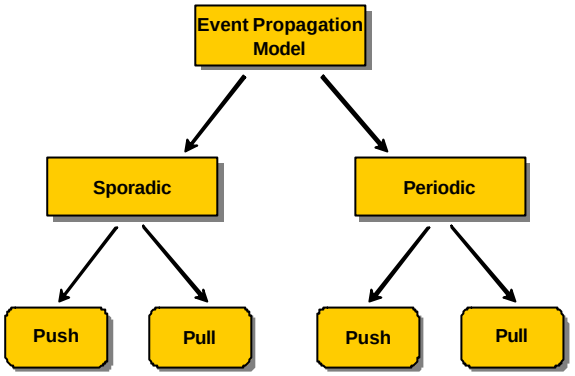


**Figure 22. Event propagation model.**

**Event Type.** Events propagated by an event service can be classified according to their structure and hence are said to be of a specific event type. As outlined in Figure 23, we differentiate between generic and typed events.

The information that constitutes a **generic** event, which is also known as an un-typed event, is a data blob with an *implicit structure.* The structure is neither recognised nor interpreted by the event service. The CORBA event service is one of the few event services that supports propagation of generic events.

In contrast, the information that describes **typed** events includes an *explicit and expressive structure* that may be recognised and interpreted by the event service. Typed events enable the use of event filters.

Event types are represented by a structure with varying **expressive power**. The expressive power of an event type describes the variety of information that they can be included in an event of that type. The expressive power of the structures outlined in Figure 23 increases from left to right.

The structure that represents an event type is either **fixed** or **application-specific**. The former is predefined by the event service whereas the latter may be defined by the application.

Both fixed and application-specific structures can be sub divided. Fixed structures consist either of a name, a name and some numeric parameters, or a name and some string parameters. A **name** usually consists of a single string. The **name and string parameters** structure therefore consists of a set of strings. The first string representing the event name and the remaining strings representing the event parameters. JEDI uses an event structure consisting of a name and a set of string parameters. The **name and numeric parameters** structure consists of a single string and a set of numbers: the string representing the event name and the numbers representing the event parameters. The version of CEA described by Bacon et al. [31] supports typed events that consist of a structure consisting of a name and a set of number parameters. Application-specific structures consist of either attributes or an object. The **attributes** structure consists of a set of attributes in which each attribute is a triple of name, type, and value. The CORBA notification service supports a general event structure consisting of attributes. The **object** structure consists of a programming-language-specific

object including a set of attributes. One of the key properties of both ECO and Obvents is their support of events in the form of specific application defined objects.
Event types may be organised into **type hierarchies**. Such event type hierarchies are similar to class hierarchies in object-oriented programming languages like Java or C++ in that event types can be derived from each other. Specialised event types can be derived from more general event types using inheritance. Event filters that match events of a certain general type will also match events of sub-types derived from that general type. Hermes is an event service that centres around the notion of event types and supports event type hierarchies.

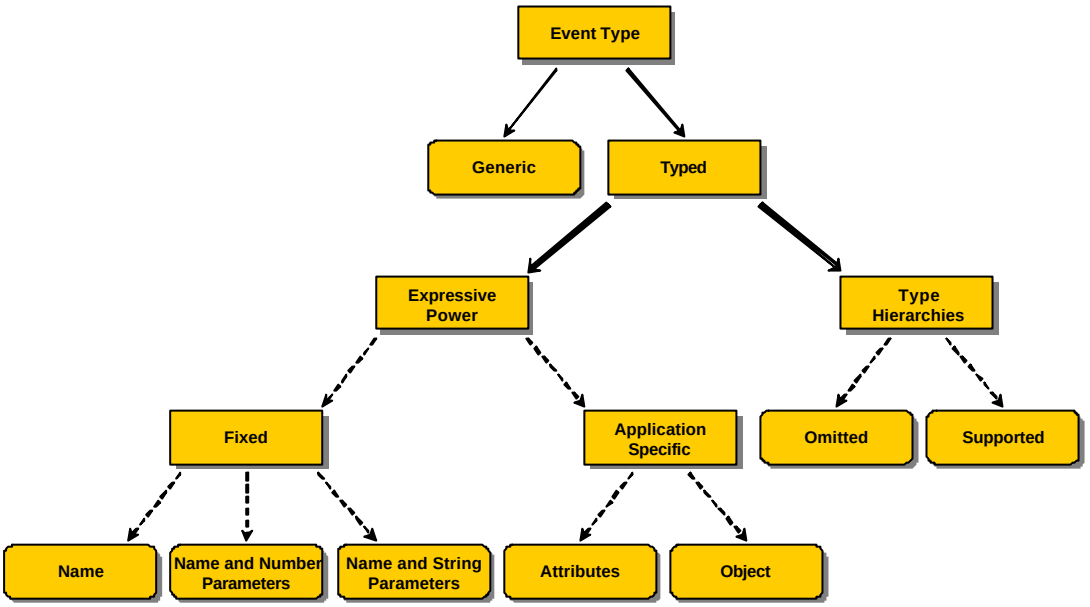


**Figure 23. Event type.**

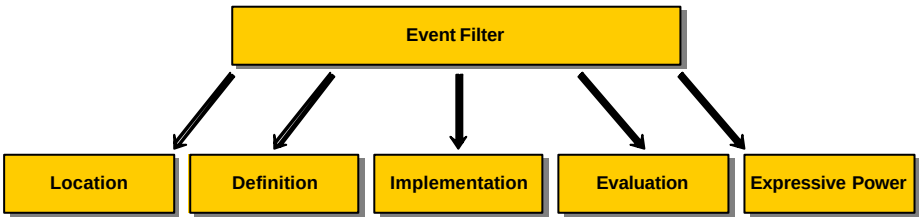


**Figure 24. Event filter.**

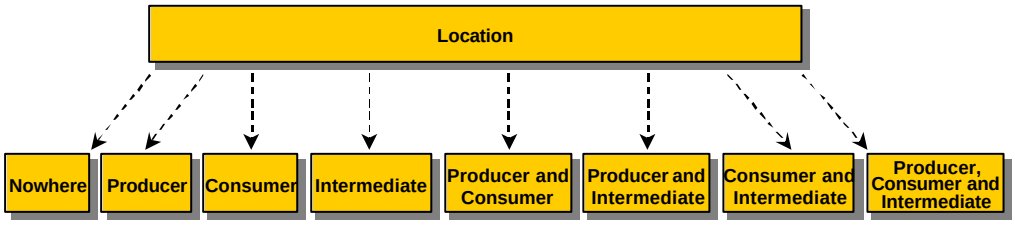


**Figure 25. Event filter location.**

**Event Filter.** Event filters control the propagation of events by allowing consumers to subscribe to the exact subset of the events in which they are interested. Events are matched against filters and are only delivered if the match produced a positive result. Figure 24 shows the properties according to which we classify event filters.

Event filters must be evaluated at a particular **location**. If supported, event filters may be evaluated at the consumer side, the producer side or at the intermediate. Furthermore, a set of event filters may be evaluated sequentially at more than one location, thus they may be applied at any combination of consumer, producer, and intermediate. Figure 25 summarises all possible combinations of event filter locations.

Filters are **not supported** and events are consequently propagated to all subscribers. The CORBA event service is an example of an event service that does not support event filters.

Filters are evaluated at the **producer** side. This minimises the use of network bandwidth and consumer processing overhead as events are filtered as close to the producer as possible. SECO serves as an example of an event service that supports producer-side filtering.

Filters are evaluated at the **consumer** side. This allows an implementation of a precise matching algorithm as the required set of events is typically well-known at the consumer side. The Java distributed event model allows filters to be applied at the remote event listener.

Filters are evaluated at the **intermediate**. This is a natural location for service-wide filters (as well as quality of service properties) since all events are propagated through the intermediate.

Filters are evaluated at the **producer and the consumer** side. ECO supports filters in the form of pre- and post constraints, which may be applied at the producer and the consumer side respectively.

Filters are evaluated at the **producer** side **and** at the **intermediate** thereby combining the characteristics of producer-side and intermediate filter evaluation.

Filters are evaluated at the **consumer** side **and** at the **intermediate**. In addition to allowing filtering at the remote event listener, the Java distributed event model supports optional event adapters at which filters may be applied as well.

Filters are evaluated at the **producer and the consumer** side, as well as at the **intermediate**. The CORBA notification service supports filtering in a hierarchical manner that allows filters to be evaluated at the producer and the consumer side, as well as at intermediates.

As shown in Figure 26, event filters can be defined by the application by using a **constraint language** that is specified as part of the event service or by using the features of an application **programming language**. The CORBA notification service specifies a constraint language that allows applications to use constraint expressions to define event filters. When using a programming language to define event filters, applications may use a **subset** of the types, operators, and combinators supported by the programming language or may be permitted to use all types, operators, and combinators supported by the **language**. SIENA limits applications to using a specific subset of the types, operators, and combinators available whereas SECO allows them to use all available types, operators, and combinators.

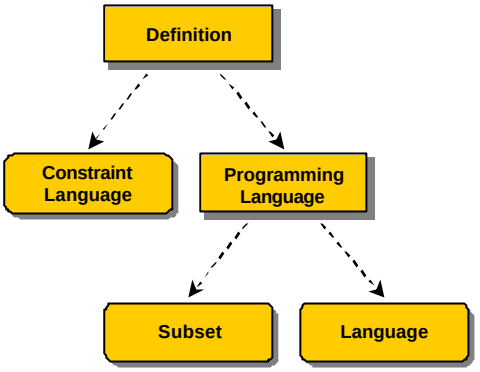


**Figure 26. Event filter definition.**

Figure 27 summarises possible **implementations** techniques for event filters. An event filter can be implemented using either a character string, a function, or an object. Character **strings** can provide a textual representation of filter expressions that are typically parsed by the event service applying them. Filters that are implemented as **functions** are applied by executing these functions. **Object** filters must be instantiated before they can be applied by invoking a method of the object. Both the CORBA notification service and SIENA implement event filters as strings that are parsed at run time whereas SECO filters are implemented as objects providing an `evaluate()` operation.

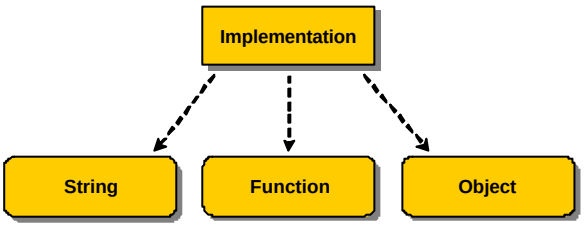


**Figure 27. Event filter implementation.**

Event filters are **evaluated** by the event service to determine the list of interested subscribers. As shown in Figure 28, event filters are evaluated at a particular **time** using a specific **mechanism** to match events against filters.

The evaluation mechanism is divided into two sub categories depending on whether filter specifications are **interpreted** or **compiled**. The former are characteristically evaluated using an event model specific interpretation mechanism while the latter can be evaluated using operations provided by the programming language. Both interpretable and executable filters are either generated by a **pre-processor** or are **implicitly** provided by the application. The CORBA notification service specifies a constraint language that allows applications to implicitly provide filter expressions that are interpreted by the evaluation mechanism. STEAM on the other hand, allows applications to implicitly define and then to compile their filters.

Event filters are evaluated either at **subscription** time or at event **propagation** time. Evaluating filters at subscription time may be useful when matching parameters describing the current context of the subscriber that are only relevant at that point in time or when matching pre-constraint filters. Such pre-constraint filters may assess the availability of resources, authenticate a connection, or process admission control. However, event services, including the CORBA notification service, SIENA, STEAM, Elvin, and COSMIC,

traditionally evaluate event filters at event propagation time when the actual list of interested subscribers can be determined.

Figure 29 summaries issues related to the **expressive power** of event filters. Event filters may be defined using an expressive structure that is described using a set of types, operators, and combinators.

The structure enclosed in an event filter may contain a set of types with varying expressive power. These sets are either **implicit** or **predefined** by the event service and their expressive power generally increases with the number of types they comprise. While both implicit and predefined sets can contain one or more types, predefined sets are typically larger and hence more expressive than implicit sets.

JEDI and CEA [31] are examples of event models supporting implicit types. Both of them support string types while CEA provides a second implicit type, namely number. In contrast, event models such as SIENA and STEAM provide predefined sets comprising a larger number of types.

An event filter may contain a set of **operators** with varying expressive power. From left to right, the sets outlined in Figure 29 are supersets of each other and hence increase in their expressive power. The filter may support **equality** and inequality operators, less than and greater than **magnitude** operators that may be combined with equality operators, or magnitude operators that can be combined to form **range** operators. JEDI and CEA only support equality operators whereas SIENA and STEAM support equality, magnitude, and range operators.

An event filter may employ a set of **combinators** with varying expressive power that may be used to combine terms including types and operators. The expressive power of the set of combinators outlined in Figure 29 increases from left to right. The structure may **not** contain any combinator or may contain either a single **implicit** combinator or an **arbitrary** number of combinators. CEA supports an implicit conjunctive combinator that requires *all* terms defined by a specific filter to match individually for the filter to return a positive result while SIENA and STEAM provide a range of arbitrarily applicable combinators. STEAM filters are defined as a collection of either conjunctive or disjunctive terms. These filter terms are matched against the relevant parameters of an event either in a conjunctive or a disjunctive manner, thus defining whether *all* or *at least one* of the terms that comprise a filter must be true for the filter to match.

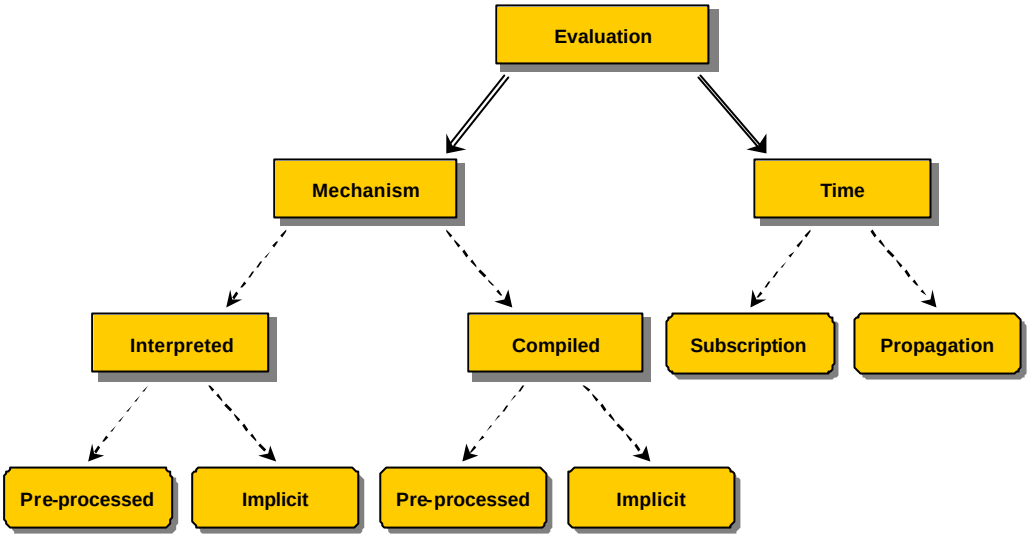


**Figure 28. Event filter evaluation.**

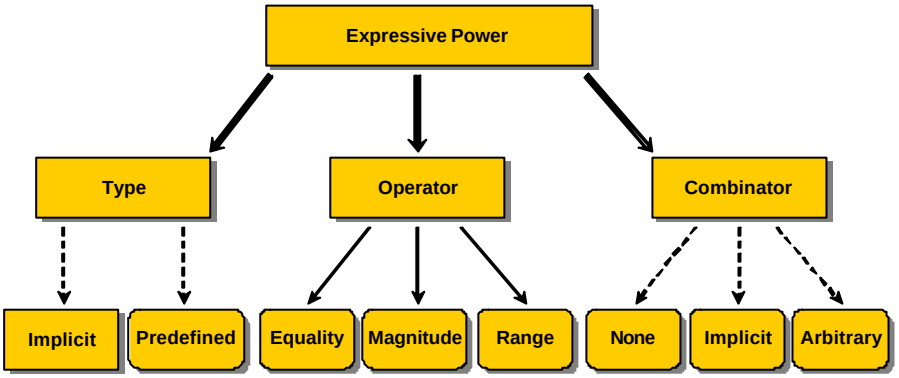


**Figure 29. Event filter expressive power.**

**Mobility.** Another functional event service property, which is becoming increasingly important with the emergence of wireless communication, is support for entity mobility. Figure 30 summaries the degree of mobility that may be provided by an event service.

Many event services do not support mobility; all entities in such an event system are assumed to have a **static** location. However, an event system may contain entities that may move location from one host machine to another thereby assuming the address of the current host machine. The **mobile code** category refers to event services that support entities that can move from one computer to another and subsequently execute at their destination. JEDI supports this feature through its concept of reactive objects. Loke et al. [38, 39] propose an extension to Elvin that enables mobile code, referred to as mobile agents, to migrate from one host to another in order to perform computations on behalf of mobile multi-agent applications. The **mobile device** category refers to event services that support portable computing devices, such as notebook computers and handheld devices, which may move location while keeping their addresses, thereby moving the entities they host. Mobile devices may host nomadic and collaborative entities and may be capable of wireless networking. **Nomadic entities** interact through either a fixed network infrastructure or a mobile computing environment to which they connect via nodes acting as access gateways. Characteristically, they may suffer periods

of disconnection while moving between points of connectivity. For example, SIENA's mobility support service allows nomadic entities to connect to proxy components using wireless connections based on General Packet Radio Services (GPRS) [61] technology. These proxy components run on event servers that act as access points and transparently manage (and synchronise) subscriptions and events on behalf of a moving entity. Mobile Push and ToPSS propose a similar approach to supporting nomadic application components in which entities disconnect from the event service infrastructure while moving. ToPSS supports application scenarios in which nomadic entities disconnect for substantial periods of time as well as those where disconnection periods are very short. The former scenario reflects the behaviour of subscribers accessing the event service at distinct locations with considerable commuting times from one area of connectivity to another whereas the latter characterises subscribers employing handover mechanisms when roaming between overlapping connectivity areas.

Nomadic entities may access the event service infrastructure either through fixed or wireless connections. In contrast, **collaborative entities** use a wireless network to interact with other mobile entities that have come together at some common location. Collaborative entities may use ad hoc networks to support communication without the need for a separate infrastructure, thus allowing loosely-coupled entities to communicate and collaborate in a spontaneous manner. STEAM exploits geographical scopes, called proximities [62], in order to accommodate collaborative entities. It allows entities residing in the same proximity to dynamically establish wireless ad hoc connections to one another and subsequently to deliver events at the location of the proximity.

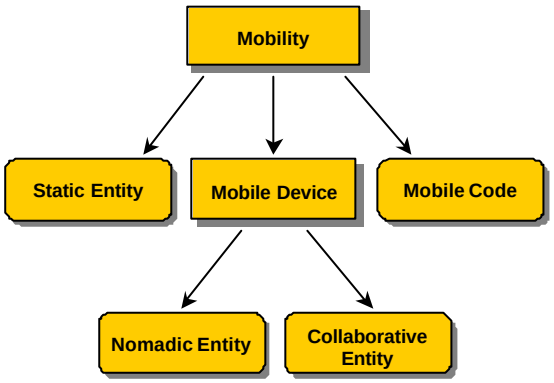


**Figure 30. Mobility support.**

**Composite Events.** Subscribers may require an event service to recognise the occurrence of a specific pattern of two or more particular events possibly propagated by different producers. Services inform subscribers of such a combination of event occurrences by means of a notification called a composite event. Subscribers express their interest in composite events by defining what can be termed composite event filters, which specify the sequences of event occurrences of interest, typically using an application-level language.

Composite event filters can be applied analogously to ordinary event filters. However, the location at which composite event filters may be evaluated depends on the locations of the set of producers potentially propagating relevant events. Composite event filters must be evaluated at a location included on the propagation paths of all events of interest. For example, composite event filters for recognising event patterns composed of events propagated by several distributed producers generally cannot be evaluated at the producer

side. Such composite event filters must be evaluated at an intermediate, at the consumer side, or at a combination of consumer and intermediate. Furthermore, when intermediates are distributed, composite event filters must be evaluated at an intermediate located on all event propagation paths.

As depicted in Figure 31, an event service may **omit** composite events. However, when **supported**, the occurrence of composite events causes the service to notify subscribers accordingly. Subscribers may specify the **number** of the events involved, their logical **relationship**, and the **time** window in which the events involved must occur. Exactly **two** or **three or more** events may be defined in a pattern that describes their sequence of occurrence along with a time window that may be defined **implicitly** by the event service or **explicitly** by the subscriber application. This window defines the time interval during which a certain number of events must occur in a given pattern for composite events to be detected.

As part of their work on CEA, Bacon et al. [63] have defined an application-level language for specifying sequences of event occurrences of interest. Monitors then use a combination of event filters to detect composite events that conform to these sequences. Pietzuch et al. [64, 65] propose a general composite event detection framework that is similar to the CEA approach in that it also introduces a high-level specification language for event occurrences of interest. However, this framework has been designed independently of specific event system and as such, can accompany a range of existing event-based middleware architectures. The language for composite event specification can be used to express patterns including sequence ($event_1$ *followed by* $event_2$), alteration ($event_1$ *or* $event_2$), and parallelisation ($event_1$ *and* $event_2$). The interval timestamp model [66] has been adopted for handling the clock uncertainties that are intrinsic to distributed systems.

Other specification languages for the detection of composite events have been proposed by Mansouri-Samani and Sloman [67] as well as by Chakravarthy and Mishra [68]. GEM [67] is a generalised event monitoring language that is based on rules. It proposes a tree-based approach for composite event detection and supports temporal constraints. Snoop [68] is an expressive event specification language designed to accommodate the requirements of a wide range of applications. It is event-model independent and focuses on supporting powerful temporal constraints.

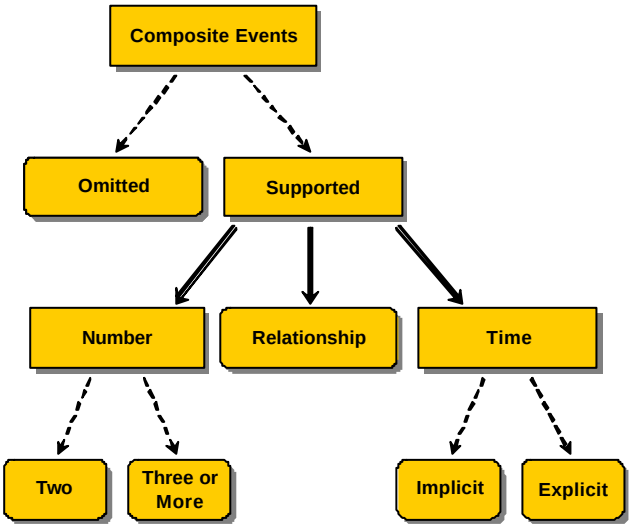


**Figure 31. Composite events.**

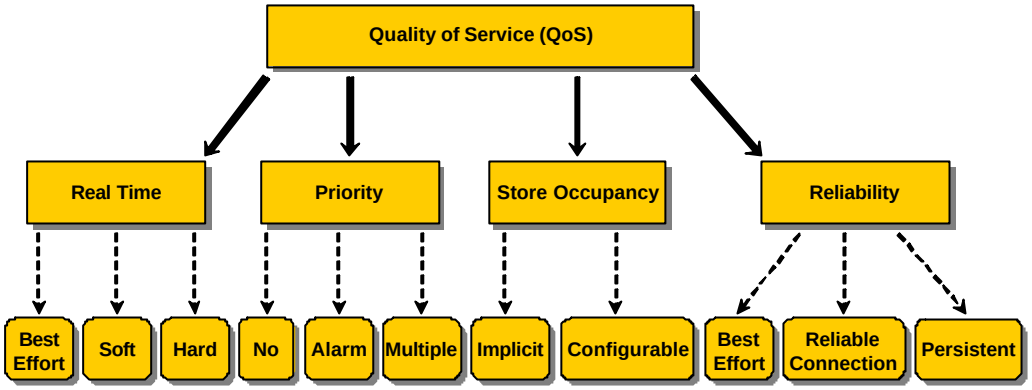


**Figure 32. Quality of service.**

### 2.2.3.2 Non-functional Features

**Quality of Service.** The QoS of an event service may be configured according to the requirements of a particular application. Figure 32 shows that we divide the QoS supported by an event service into four categories describing the behaviour of an event service when propagating and delivering events.

The **real-time** category explores the guarantees provided by an event service regarding the timely delivery of events. Real-time guarantees can be either best-effort, soft or hard. In the **best-effort** case, no deadlines can be associated with events. An event service supporting **soft real-time** provides guarantees with a probability that is sufficient to be used for soft real-time deadlines and a **hard real-time** service provides guarantees with a probability that is sufficiently high to be used for hard real-time deadlines. Hard real-time guarantees must meet their temporal specification in all anticipated load and fault scenarios [69]. The CORBA notification service allows deadlines defining earliest and latest delivery time to be assigned to events that are enforced with a probability that is sufficient to be used for soft real-time deadlines. Generally, hard real-time guarantees are difficult to provide as they require a predictable communication pattern, usually only available in a small-scale environment. This is particularly true for distributed event systems. Distributed event systems are traditionally based on anonymous one-to-many communication patterns that tend to be unpredictable and are likely used in systems consisting of a large number of loosely-coupled entities. However, the TAO RT event service, an extension to the CORBA event service that was developed for avionics applications, supports hard real-time guarantees. COSMIC uses event channels as an abstraction for network resources and allows applications to assign timeliness properties to channels. It supports best-effort guarantees in the form of non real-time event channels as well as soft and hard real-time guarantees through soft real-time channels and hard real-time channels respectively.

In order to influence the sequence in which events are delivered, a **priority** may be assigned to an individual event. Usually, **no** priority can be assigned and therefore all events have identical priority. An event service that supports **alarm** events allows a single priority to be assigned to certain events. The CORBA notification service provides **multiple** priorities.

**Store occupancy** describes the maximum size of memory required by an event service to operate at any point during its lifetime. This size can be either **implicit** or it may be **configurable** according to the requirements of a particular application. Implicit store occupancy either imposes a fixed maximum memory size or allocates the required memory dynamically whereas configurable store occupancy typically depends on a number of

parameters. These parameters may describe the maximum size of the queues that buffer events as well as the maximum number of producers, consumers, and mediators that may be supported by an event service.

The **reliability** category investigates the guarantees provided by an event service regarding the delivery of events in the presence of failure. An event service is said to provide **best-effort** reliability if no specific delivery guarantees are made. Events may or may not be delivered to subscribers in the presence of failure. An event service that supports **reliable connections** guarantees events being delivered to all correctly functioning subscribers. Upon restart from a failure, connections between producers and subscribers are re-established without re-subscription and event delivery resumes. A **persistent** event service guarantees events being delivered to all subscribers. Upon restart from a failure, connections between producers and subscribers are re-established without re-subscription and persistently buffered events are retransmitted. The CORBA notification service may support any of these three delivery policies.

**Ordering.** An event service delivers events according to a certain ordering semantic. Figure 33 shows that an event service may deliver events in a certain order in a **subset** of the system or **system wide**, i.e., throughout the system. Event services with a system wide ordering strategy employ exactly one delivery order whereas event systems with subset orders associate different ordering strategies with various parts of the system.

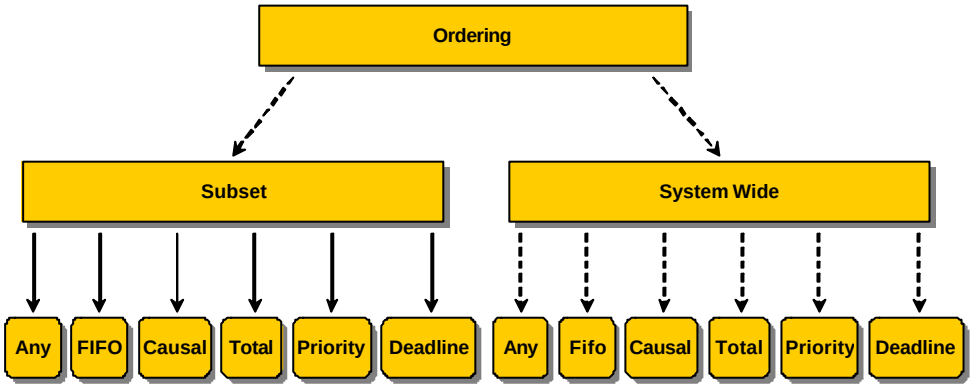


**Figure 33. Ordering.**

Events may be delivered in **any** order. Such unordered events may be received by any subscriber in any order. **FIFO** order refers to a strategy where two events that are raised by the same producer are delivered by consumers with matching subscriptions in the order in which they were raised. **Causally**-ordered events are delivered in the order they were published as determined by the well-known *happens-before* relationship [70] while **totally**-ordered events are delivered in the same order by all subscribers but not necessarily in the order they were raised [71]. Mechanisms for providing unordered and FIFO order semantics are generally relatively straightforward since they do not require distributed coordination. In contrast, enforcing causal and total order semantics requires cooperation between all producers and consumers involved.

Alternatively, events may be delivered according to an associated **priority** or **deadline**. These semantics imply that the delivery of some event can be pre-empted in order to deliver an event that has a higher priority or to deliver an event that has a deadline that is close to expiring. Ordering in real-time systems may also be determined by deadlines.

The CORBA notification service supports various semantics for defining event delivery order for a specific event channel, including any, FIFO, priority, and deadline order. This approach allows applications with a single event channel to define a system wide order and applications comprising multiple channels to associate a specific order with each channel. CONCHA and TAO RT are other CORBA-based event services that support delivery order semantics. CONCHA features totally-ordered event delivery and TAO RT CORBA provides a dispatching mechanism for priority-based event delivery.

**Security.** As discussed below, event services can support a number of mechanisms to alleviate the security concerns that may arise in applications that disseminate events among a population of distributed producers and consumers. However, the event model that is exploited for such applications can have an impact on security concerns as some models are more secure than others. Peer to peer models, in which explicitly named entities interact directly, can be considered more secure than mediator-based models where interaction requires a trusted mediator (or group of mediators) or indeed implicit models where the middleware as a whole must be trusted.

Event services may **omit** mechanisms that address security concerns or may support security properties by providing techniques for event message **confidentiality** and for **authentication**.

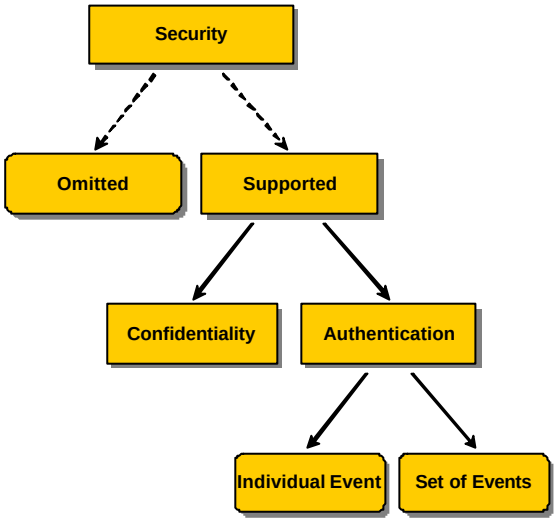


**Figure 34. Security.**

Event messages that contain sensitive content may be transmitted over a network in an encrypted and therefore confidential form rather than as plain text. This enables producers and consumers to keep event messages secret from third parties. For example, Elvin supports a security framework that exploits the Secure Socket Layer (SSL) protocol for managing the security of its message transmissions over the Internet.

Essentially, authentication establishes the identity of specific events and serves as the basis for a mechanism that polices access to certain operations. Such an access control mechanism may regulate access privileges for event dissemination, forwarding, and delivery. Access may be granted to an **individual event** or to a **set of events**. Such a set of events may be defined by various means. Access may be granted to events of a specific type, to the events disseminated by a specific producer or a group of producers, to the events described by a subscription or by the subscriptions issued by a certain consumer, or to the events handled by a particular mediator. For example, Elvin's security framework enables servers to authorise

access to events using keys, which may be associated with either a connection to a specific entity or an individual event.

Wang et al. [72] outline security issues in event services without attempting to present an actual security model. Their work specially focuses on Internet-scale event systems and discusses security paradoxes, such as anonymity vs. authentication, that arise due to the nature of event systems.

**Failure Mode.** The failure mode describes the behaviour of an event service in the presence of a single component failing silently. A fail-silent component is a self-checking component that either functions correctly or stops functioning after an internal failure is detected [73]. As outlined in Figure 35, the failure mode category explores support for the failed component being an entity, a middleware component, or a part of the network.

A failed **entity** may be either a **consumer** or a **producer**. A failed consumer does not cause the remainder of the system to suffer. A failed producer causes a partial or a total system failure. A **partial system failure** affects the communication related to some event types that may result in fewer events being propagated. No event communication can take place in case of a **total system failure**. A system consisting of a single producer and a number of consumers fails totally if the sole producer fails silently.

A **middleware** component failing silently causes a partial or a total system failure similar to the effect of a failed producer. A **partial system failure** affects either a **geographical** or a **functional** part of the system. The former disconnects a part of the system from the rest of the system. Event communication may take place within the partitions, but no event communication takes place between the partitions. A geographical partial system failure may be caused by a failing SIENA event server that is part of a hierarchical or an acyclic non-hierarchical server topology. The latter stops communication related to a particular event type throughout the system. However, communication related to other event types does not suffer. A functional partial system failure may be caused by a failed event channel in a CORBA event service utilising multiple channels, each managing a specific event type. A failing centralised JEDI event dispatcher causes a total system failure.

A part of the **network** failing silently may be redundant or may cause partial or total system failure. A **redundant** part of the network failing in SIENA utilising a general non-hierarchical server topology may not cause the remainder of the system to suffer. Similarly, Hermes' overlay routing layer enables a system to overcome failures in redundant parts of the network by using an adaptive routing strategy.

A **partial system failure** disconnects a part of the system from the rest of the system. Event communication may take place within the partitions, but no event communication takes place between the partitions. SIENA utilising an acyclic non-hierarchical server topology and Rebeca, which assumes an acyclic non-hierarchical network topology, behave in this manner. A system in which all producers are connected through a single network is susceptible to **total system failure** where no event communication can take place.

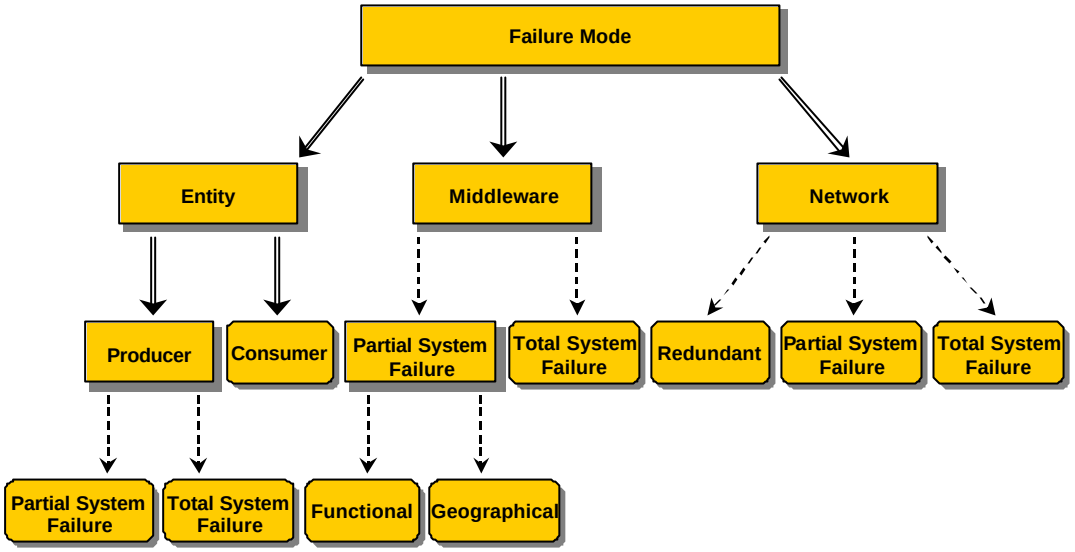


**Figure 35. Failure mode.**

# 3 Classification of Event Systems

Table 2 illustrates how a taxonomy user may apply the taxonomy to existing event systems. It presents a number of selected event services that have been summarised using the terminology of the taxonomy presented in this paper. These event services have been selected to cover various properties and because sufficiently detailed documentation is available to describe them. The CORBA notification service has been chosen due to its widespread use and due to its support of a wide range of non-functional features. SIENA, SECO, and Hermes, which have been designed in academia, have been chosen because of their organisational and interaction models as well as their exploitation (or lack) of event server topologies.
Table 2 demonstrates that using a common vocabulary for describing event services facilitates comparison of service properties. For example, Table 2 shows that both, the CORBA notification service and SIENA are based on an event model that includes either a single event server or a topology of multiple event servers and that the SECO event model excludes the use of such mediators altogether. It also shows that Hermes' fault tolerance mechanisms alleviate the effects of failed middleware components once its overlay routing layer has adapted. The implementation of Hermes' client-side programming model is application-specific since Hermes defines the set of Extensible Markup Language (XML) [74] messages to be exchanged across brokers and clients but not the bindings between client programming language and these XML messages. The programming model properties shown are based upon the Java version of a client implementation proposed in [44]. Moreover, Pietzuch [44] proposes a set of higher-level middleware services for composite event detection, security, and congestion control that can be built on top of Hermes. However, these services are not intrinsic to Hermes and as a result, were not considered in Table 2.

**Table 2. Categorisation of event systems.**

| | **CORBA Notification Service** | **SIENA** | **SECO** | **Hermes** |
|---|---|---|---|---|
| **Event Model** | Single mediator or multiple, non-functionally equivalent mediators | Single or multiple mediators | Implicit | Multiple mediators |
| **Event Service Organisation** | Single or multiple distributed, separated middleware | Single or multiple distributed, separated middleware | Distributed, collocated middleware | Multiple distributed, separated middleware |
| **Event Service Interaction Model** | Centralised intermediate or partitioned, distributed intermediate | Centralised intermediate or cooperative, distributed intermediate | No Intermediate, named (uSECO) or implicit (mSECO) | Cooperative, distributed intermediate |
| **Functional Event Service Features** | | | | |
| **Event Propagation Model** | Sporadic push and pull | Sporadic push | Sporadic push | Sporadic push |
| **Event Type** | Typed | Typed | Typed | Typed |
| **Expressive Power** | Application specific attributes | Application specific attributes | Application specific object | Application specific object |
| **Type Hierarchies** | Omitted | Omitted | Omitted | Supported |
| **Event Filter** | | | | |

| | CORBA Notification Service | SIENA | SECO | Hermes |
|---|---|---|---|---|
| **Location** | Producer, consumer, and intermediate | Intermediate | Producer and consumer | Intermediat e |
| **Definition** | Constraint language | Constraint language | Programming language | Programming language |
| **Implementation** | String | String | Object | Object |
| **Evaluation** | | | | |
| **Mechanism** | Implicit interpreted | Implicit interpreted | Implicit compiled | Implicit interpreted |
| **Time** | Propagation | Propagation | Propagation | Propagation |
| **Expressive Power** | | | | |
| **Type** | Predefined | Predefined | Predefined | Predefined |
| **Operator** | Range | Range | Range | Range |
| **Combinator** | Arbitrary | Arbitrary | Arbitrary | Arbitrary |
| **Mobility** | Static | Static and nomadic entity | Static | Static |
| **Composite Events** | Omitted | Omitted | Omitted | Omitted |
| **Non-Functional Event Service Features** | | | | |
| **Quality of Service** | | | | |
| **Real-time** | Soft | Best effort | Best effort | Best effort |
| **Priority** | Multiple | No | No | No |
| **Store Occupancy** | Configurable | Implicit | Implicit | Implicit |
| **Reliability** | Best effort, reliable connection or persistent | Best effort | Best effort (uSECO) or reliable connection (mSECO) | Reliable connection (temporarily) and then best effort |
| **Ordering** | Any, FIFO, priority or deadline | Any | Any | Any |
| **Security** | Omitted | Omitted | Omitted | Omitted |
| **Failure Mode** | | | | |
| **Entity** | Partial system failure | Partial system failure | Partial system failure | Partial system failure |
| **Middleware** | Functional partial system failure or total system failure | Geographical partial system failure or total system failure | Results in failed entity | Geographical or functional partial system failure (temporarily) |
| **Network** | Partial system failure | Redundant or partial system failure | Partial system failure | Redundant or partial system failure |

# 4 Conclusion

This paper presented a taxonomy of distributed event-based programming systems. The taxonomy identifies a set of fundamental properties of event-based programming systems and categorises them according to their event model and the structure of their event service. The event service is further classified according to its organisation and interaction model, as well as other functional and non-functional features. These properties are then arranged in a hierarchical manner starting from the root of the taxonomy, which defines the relationships between an event system, an event service and an event model. Each of these properties is described in detail and a range of event systems are used as examples.

We have demonstrated how a taxonomy user may apply the taxonomy to existing event systems by categorising a number of selected event services, which have been chosen to cover various properties, according to the taxonomy.

Our taxonomy differs from related work in that it identifies an extensive set of generic event system properties describing various systems dimensions in detail. The taxonomy considers functional and non-functional properties, including mobility, security, and quality of service, and describes the possible options for these properties. As a result, it can be used to classify virtually any distributed event-based programming system regardless of system scale or application domain whereas existing work focuses on providing a framework designed for a specific application area or based on a particular high-level model.

Event systems may evolve together with future advancements in the information technology industry. Such next-generation event systems may support additional, novel properties in order to accommodate new application requirements that may result from these advances. For example, a means for consumers to electronically pay producers for the information they disseminate may arise as an important feature in future event-based systems. Consequently, the taxonomy may need to be extended to support such novel properties. The hierarchical structure on which our taxonomy is based may easily cope with such potential enhancements. Adding novel properties or refining existing properties is straightforward as such changes affect a specific part of the taxonomy only and do not require a reorganisation of the existing hierarchy.

# Acknowledgments

The work described in this paper was partly supported by the Irish Higher Education Authority's Programme for Research in Third Level Institutions cycle 0 (1998-2001) and by the FET programme of the Commission of the European Union under research contract IST-2000-26031 (CORTEX).

# References

[1] R. Meier and V. Cahill, "Taxonomy of Distributed Event-Based Programming Systems," *The Computer Journal*, vol. 48, pp. 602-626, 2005.

[2] D. Chambers, G. Lyons, and J. Duggan, "Design of Virtual Store using Distributed Object Technology," in *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE/ICSE 2000)*. Limerick, Ireland: IEEE Computer Society, 2000, pp. 66-75.

[3] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper, "Implementing a Sentient Computing System," *IEEE Computer*, vol. 34, pp. 50-56, 2001.

[4] H. Muller and C. Randell, "An Event-Driven Sensor Architecture for Low Power Wearables," in *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC/ICSE2000)*. Limerick, Ireland: IEEE Computer Society, 2000, pp. 39-41.

[5] 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 Conference on Enterprise Distributed Object Computing (EDOC'99)*. Mannheim, Germany: IEEE Publishing, 1999, pp. 255-261.

[6] 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 IEEE International Symposium on Software Engineering for Parallel and Distributed Systems (ICSE/PDSE 2000)*. Limerick, Ireland: IEEE Computer Society, 2000, pp. 83-95.

[7] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems," in *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*. Austin, TX, USA: Springer-Verlag, 1999, pp. 262-272.

[8] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman, "Exploiting IP Multicast in Content-Based Publish-Subscribe Systems," in *Proceedings of IFIP/ACM International Conference on Distributed Processing (Middleware 2000)*. New York, USA: Springer-Verlag, 2000, pp. 185-207.

[9] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems, Concepts and Design*, Third ed. Harlow, Essex, England: Pearson Education Limited, 2001.

[10] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic Support for Distributed Applications," *IEEE Computer*, vol. 33, pp. 68-76, 2000.

[11] R. Meier, "Event-Based Middleware for Collaborative Ad Hoc Applications," Department of Computer Science, University of Dublin, Trinity College, Ireland, Ph.D. Thesis September 2003.

[12] M. Erzberger and M. Altherr, "Every Dad Needs a Mom - Message-Oriented Middleware," SoftWired AG, Zurich, Switzerland, White Paper 1999.

[13] S. Maffeis, "Developing Publish/Subscribe Applications with iBus," SoftWired AG, Zurich, Switzerland, White Paper 1999.

[14] C. Ma and J. Bacon, "COBEA: A CORBA-Based Event Architecture," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. Santa Fe, New Mexico, USA: USENIX Association, 1998, pp. 117-131.

[15] A. Hopper, A. Harter, and T. Blackie, "The Active Badge System," in *Proceedings of the Conference on Human Factors in Computing Systems (INTERCHI'93)*. Amsterdam, The Netherlands: ACM Press, 1993.

[16] S. J. Kang, S. H. Park, and J. H. Park, "ROOM-BRIDGE: A Vertically Configurable Network Architecture and Real-Time Middleware for Interoperability between Ubiquitous Consumer Devices in Home," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*. Heidelberg, Germany: Springer-Verlag, 2001, pp. 232-251.

[17] 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'97)*. Atlanta, Georgia, USA: ACM Press, 1997, pp. 184-200.

[18] J. Bacon, K. Moody, and W. Yao, "Access Control and Trust in the use of Widely Distributed Services," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware2001)*. Heidelberg, Germany: Springer-Verlag, 2001, pp. 295-310.

[19] 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*. University of Iowa, Iowa City, USA, 1995, pp. 172-177.

[20] R. Meier and V. Cahill, "Taxonomy of Distributed Event-Based Programming Systems," in *Proceedings of the International Workshop on Distributed Event-Based Systems (IEEE ICDCS/DEBS'02)*. Vienna, Austria: IEEE Computer Society, 2002, pp. 585-588.

[21] Sun Microsystems Inc., *Java AWT: Delegation Event Model*: Sun Microsystems Inc., 1997.

[22] Microsoft Corporation, *C# Language Specification, Version 0.28*: Microsoft Corporation, 2001.

[23] B. E. Martin, C. H. Pedersen, and J. Bedford-Roberts, "An Object-Based Taxonomy for Distributed Computing Systems," *IEEE Computer*, vol. 24, pp. 17-27, 1991.

[24] R. Meier and V. Cahill, "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications," in *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03), LNCS 2893*. Paris, France: Springer-Verlag, 2003, pp. 285-296.

[25] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise, "A Framework for Event-based Software Integration," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, pp. 378 - 421, 1996.

[26] D. S. Rosenblum and A. L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," in *Proceedings of the The Fifth Symposium on the Foundations of Software Engineering (FSE5) and The Sixth European Software Engineering Conference (ACM SIGSOFT ESEC97)*. Zurich, Switzerland, 1997, pp. 344-360.

[27] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, pp. 114-131, 2003.

[28] Object Management Group, *CORBAservices: Common Object Services Specification - Notification Service Specification, Version 1.0*: Object Management Group, 2000.

[29] Iona Technologies, "Orbix 6.1 Technical Overview," Iona Technologies, Dublin, Ireland, White Paper December 2003.

[30] D. C. Schmidt, "Real-Time CORBA with TAO (The ACE ORB)," http://www.cs.wustl.edu/~schmidt/TAO.html, White Paper 2004.

[31] J. Bacon, J. Bates, R. Hayton, and K. Moody, "Using Events to Build Distributed Applications," in *Proceedings of the Second International Workshop on Services in Distributed and Networked Environments (SDNE'95)*. Whistler, British Columbia, Canada: IEEE Computer Society, 1995, pp. 148-155.

[32] Object Management Group, *CORBAservices: Common Object Services Specification - Event Service Specification*: Object Management Group, 1995.

[33] J. Kaiser, C. Brudna, C. Mitidieri, and C. Pereira, "COSMIC: A Middleware for Event-Based Interaction on CAN," in *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003)*, vol. 2. Lisbon, Portugal: IEEE Computer Society, 2003, pp. 669-676.

[34] J. Kaiser, C. Brudna, and C. Mitidieri, "A Real-Time Event Channel Model for the CAN Bus," in *Proceedings of the Eleventh International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2003)*. Nice, France: IEEE Computer Society, 2003, pp. 120.2.

[35] 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*. Connemara, Ireland: ACM Press, 1996, pp. 17-24.

[36] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, "Content Based Routing with Elvin4," in *Proceedings of AUUG2K*. Canberra, Australia, 2000.

[37] P. Sutton, R. Arkins, and B. Segall, "Supporting Disconnectedness – Transparent Information Delivery for Mobile and Invisible Computing," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*. Brisbane, Australia: IEEE CS Press, 2001, pp. 277-285.

[38] A. Padovitz, S. W. Loke, and A. B. Zaslavsky, "Using the Publish-Subscribe Communication Genre for Mobile Agents," in *Proceedings of the First German Conference on Multiagent System Technologies (MATES'03), LNCS 2831*. Erfurt, Germany: Springer-Verlag Heidelberg, Germany, 2003, pp. 180-191.

[39] S. W. Loke, A. Padovitz, and A. B. Zaslavsky:, "Context-Based Addressing: The Concept and an Implementation for Large-Scale Mobile Agent Systems," in *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03), LNCS 2893*. Paris, France: Springer-Verlag Heidelberg, Germany, 2003, pp. 274-284.

[40] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra, "Matching Events in a Content-based Subscription System," in *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*. Atlanta, GA, USA, 1999, pp. 53-61.

[41] S. Bhola, R. E. Strom, S. Bagchi, Y. Zhao, and J. S. Auerbach, "Exactly-once Delivery in a Content-based Publish-Subscribe System," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002)*. Bethesda, MD, USA: IEEE Computer Society, 2002, pp. 7-16.

[42] P. R. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria: IEEE Computer Society, 2002, pp. 611-618.

[43] P. R. Pietzuch and J. Bacon, "Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware," in *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (ACM SIGMOD/PODS/DEBS'03)*. San Diego, California, USA: ACM Press, 2003, pp. 1-8.

[44] P. R. Pietzuch, "Hermes: A Scalable Event-Based Middleware," Queens' College, University of Cambridge, UK, Ph.D. Thesis February 2004.

[45] Sun Microsystems Inc., *Java Distributed Event Specification*: Sun Microsystems Inc., 1998.

[46] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, pp. 827-850, 2001.

[47] I. Podnar, M. Hauswirth, and M. Jazayeri, "Mobile Push: Delivering Content to Mobile Users," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria: IEEE Computer Society, 2002, pp. 563-570.

[48] P. T. Eugster, R. Guerraoui, and C. H. Damm, "On Objects and Events," in *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*. Tampa, Florida, USA: ACM Press, 2001, pp. 131-146.

[49] P. Eugster, "Type-Based Publish/Subscribe," Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, PhD Thesis December 2001.

[50] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann, "Engineering Event-Based Systems with Scopes," in *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*. Málaga, Spain: Springer-Verlag, 2002, pp. 309-333.

[51] L. Fiege, F. C. Gartner, O. Kasten, and A. Zeidler, "Supporting Mobility in Content-Based Publish/Subscribe Middleware," in *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*. Rio de Janeiro, Brazil, 2003, pp. 103-122.

[52] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 283 - 331, 2001.

[53] R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks," in *Proceedings of the International Workshop on Distributed Event-Based Systems (IEEE ICDCS/DEBS'02)*. Vienna, Austria: IEEE Computer Society, 2002, pp. 639-644.

[54] R. Meier and V. Cahill, "Location-Aware Event-Based Middleware: A paradigm for Collaborative Mobile Applications?," presented at the 8th CaberNet Radicals Workshop, Ajaccio, Corsica, France, 2003.

[55] R. Meier, B. Hughes, R. Cunningham, and V. Cahill, "Towards Real-Time Middleware for Applications of Vehicular Ad Hoc Networks," in *Proceedings of the 5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'05), LNCS 3543*. Athens, Greece: Springer-Verlag, 2005, pp. 1-13.

[56] R. Meier, V. Cahill, A. Nedos, and S. Clarke, "Proximity-Based Service Discovery in Mobile Ad Hoc Networks," in *Proceedings of the 5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'05), LNCS 3543*. Athens, Greece: Springer-Verlag, 2005, pp. 115-129.

[57] G. Cugola and H.-A. Jacobsen, "Using Publish/Subscribe Middleware for Mobile Systems," *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R)*, vol. 6, pp. 25-33, 2002.

[58] I. Burcea, H.-A. Jacobsen, E. d. Lara, V. Muthusamy, and M. Petrovic, "Disconnected Operation in Publish/Subscribe Middleware," in *Proceedings of the IEEE International Conference on Mobile Data Management (MDM 2004)*. Berkeley, California, USA: IEEE Computer Society, 2004, pp. 39-50.

[59] Z. Xu and H.-A. Jacobsen, "Efficient Constraint Processing for Location-aware Computing," in *Proceedings of the 6th International Conference on Mobile Data Management (MDM 2005)*. Ayia Napa, Cyprus: ACM Press, 2005, pp. 3-12.

[60] I. Sommerville, *Software Engineering*. Boston, MA, USA: Addison Wesley, 1995.

[61] C. Bettstetter, H.-J. Vögel, and J. Eberspächer, "GSM Phase 2+ General Packet Radio Service GPRS: Architecture, Protocols, and Air Interface," *IEEE Communications Surveys and Tutorials*, vol. 2, pp. 2-14, 1999.

[62] R. Meier, "Communication Paradigms for Mobile Computing," *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R)*, vol. 6, pp. 56-58, 2002.

[63] J. Bacon, J. Bates, R. Hayton, and K. Moody, "Using Events to Build Distributed Applications," in *Proceedings of the Seventh ACM SIGOPS European Workshop*. Connemara, Ireland: ACM Press, 1996, pp. 9-16.

[64] P. R. Pietzuch, B. Shand, and J. Bacon, "A Framework for Event Composition in Distributed Systems," in *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Middleware (Middleware 2003)*. Rio de Janeiro, Brazil: Springer, 2003, pp. 62-82.

[65] P. R. Pietzuch, B. Shand, and J. Bacon, "Composite Event Detection as a Generic Middleware Extension," *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, pp. 44-55, 2004.

[66] C. Liebig, M. Cilia, and A. Buchmann, "Event Composition in Time-Dependent Distributed Systems," in *Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*. Edinburgh, Scotland: IEEE Computer Society, 1999, pp. 70-78.

[67] M. Mansouri-Samani and M. Sloman, "GEM: A Generalized Event Monitoring Language for Distributed Systems," *IEE/IOP/BCS Distributed Systems Engineering Journal*, vol. 4, pp. 96-108, 1997.

[68] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language For Active Databases," *Data & Knowledge Engineering*, vol. 14, pp. 1-26, 1994.

[69] H. Kopetz, *Real-Time Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[70] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558-565, 1978.

[71] K. Birman, *Building Secure and Reliable Network Applications*. Greenwich, CT, USA: Manning Publishing Co., 1996.

[72] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf, "Security Issues and Requirements for Internet-scale Publish-Subscribe Systems," in *Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS)*. Big Island, Hawaii, USA: IEEE Computer Society, 2002, pp. 303.

[73] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, vol. 34, pp. 56-78, 1991.

[74] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, "Extensible Markup Language (XML) 1.0 (Third Edition)," http://www.w3.org/TR/2004/REC-xml-20040204, W3C Recommendations February 2004.