9

Existing Notification Services

In this chapter we describe some standards (Sect. 9.1), commercial systems (Sect. 9.2), and research prototypes (Sect. 9.3) that are closely related to event-based systems.

9.1 Standards

In this section we describe standards which are related to event-based systems. This includes the CORBA Event Service, the CORBA Notification Service, the Java Message Service (JMS), and the Data Distribution Service (DDS).

9.1.1 CORBA Event and Notification Service

The *Common Object Request Broker Architecture* (CORBA) [283] is a platform- and language-independent object-oriented middleware architecture fa-

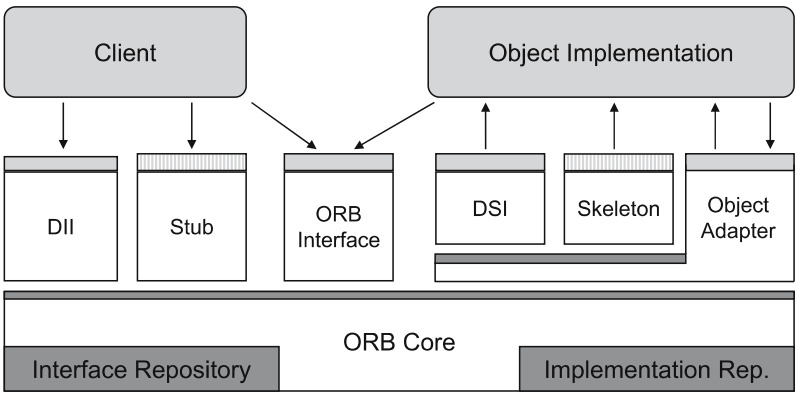

Fig. 9.1. Internal structure of an object request broker (ORB)

cilitating interoperability. CORBA is standardized by the Object Management Group (OMG), and vendors can implement the specification with their products. CORBA is a mature middleware technology that is widely used in financial and telecommunication systems and has inspired many recent middleware initiatives. Some reasons for CORBA's success are its good programming language integration across several mainstream languages, the extensibility of the platform using object services, and its adaptation to heterogeneous distributed systems. The CORBA specification describes the functionality, structure, and the interfaces of the *object request broker (ORB)*. An ORB consists of the following main components (cf. Fig. 9.1):

Object Request Broker (ORB) Core. The ORB core forms the heart of the middleware and handles communication. It resolves object references to locations, performs the marshaling and unmarshaling of method parameters, and sends invocations and results over the network.

Interface Definition Language (IDL). Interfaces of remote objects are defined in IDL, which is purely declarative and is independent of the programming language(s) used for implementations. IDL supports the usual primitive (integers, floats, etc.) and composite data types (e.g., structs). Programming language mappings define how an IDL type is mapped to a type of the programming language that is used.

Static Invocation Interface (SII). An IDL compiler transforms the static interface definitions given in IDL into client-side stub and server-side skeleton source code (in a given programming language). The stubs are called by the client, do the marshaling and unmarshaling of method arguments and method results, and pass the results back to the client. The stubs are also called *Static Invocation Interface (SII)* because their use requires the interface of the called object to be known at compile time. This approach has the advantage that remote method invocations can be statically type-checked by the compiler.

Dynamic Invocation Interface (DII). The DII allows the remote call to be constructed at runtime. This is, for example, useful if the interface of the remote object to be invoked is not known at compile time. Dynamic invocations are usually less efficient than static invocations since they require more code and type-checking must be done at runtime. The server-side complement of the DII is the *Dynamic Service Interface (DSI)*. DII and DSI can together be used for implementing general-purpose gateway, proxy, or browser objects.

Object Adapter (OA). An object adapter is interposed between the ORB and the skeletons. The OA dispatches upcalls received from the ORB to the skeleton of the called object implementation or to the DSI. Other responsibilities of the OA include generation and interpretation of object references, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and reg-

istration of implementations. There can be different types of OAs. The *Portable Object Adapter (POA)* is currently most commonly used.

Interface Repositories. The interface repository contains the IDL definitions of interfaces. The repository can be queried either at compile time or at runtime.

Implementation Repository. The implementation repository contains all implementations of a remote interface at the server side so that remote objects can be located and activated on demand.

Although the need for asynchronicity has been recognized by the OMG, the core design of CORBA is still based on synchronous communication. Before the *Asynchronous Message Invocation (AMI)* was standardized, the only possibility to issue asynchronous two-way calls had been to use deferred synchronous calls, which depend on the tedious dynamic invocation interface. With the static invocation interface, only one-way calls had been possible which provide best-effort method invocations not expecting a return value and thus not requiring blocking. The AMI closes this gap and enables asynchronous two-way calls using the SII. It supports two models, the *polling model* and the *callback model*. In the polling model, the issuer of a call can poll a collocated value-type object to test whether or not the results are now available. In the callback model, the results are delivered to the client by calling a handler method with the results as parameters.

The CORBA platform is extensible by means of object services that address different facets of a distributed computing environment, ranging from transactional support to security. In the next sections, we will take a closer look at the CORBA Event and Notification Services that explicitly deal with anonymous asynchronous communication by providing publish/subscribe functionality.

CORBA Event Service

The OMG acknowledged the need for publish/subscribe communication by introducing the CORBA *Event Service* [277] as a CORBA service in 1994. The current version as of 2005 is 1.2 [285]. With the Event Service, communication among suppliers and consumers can be in *push mode*, in which case a supplier pushes data to a consumer, or in *pull mode*, in which case a consumer requests data from a supplier. Instead of communicating directly with each other, consumers and suppliers are decoupled by an *event channel*. This way it is possible to use push and pull communication at both sides.

The Event Service specification supports two models: typed and untyped event communication. With the untyped model, which is most common, events are of the CORBA datatype `any` and can thus contain any IDL datatype. Suppliers can call `push` on the `PushConsumer` interface to deliver data and pull-based consumers can call `pull` on the `PullSupplier` interface to get data (Fig. 9.2). Since consumers and suppliers are decoupled by the event channel, they call these methods not on each other but on the event channel's interface

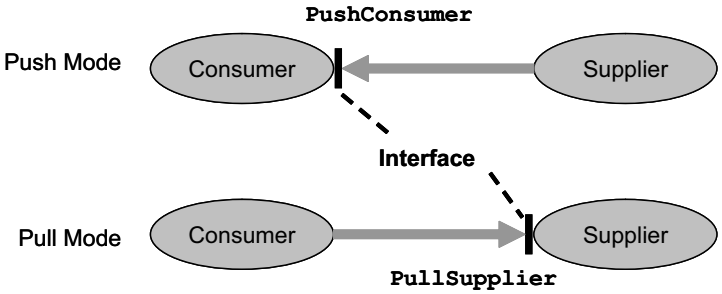

Fig. 9.2. Push mode vs. pull mode (typed event communication)

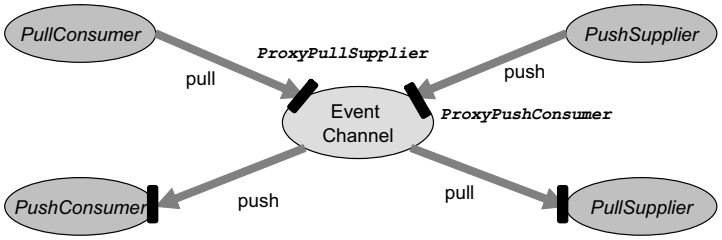

Fig. 9.3. Typed event communication using an event channel

(Fig. 9.3). For typed event communication, which is less common, suppliers and consumers agree on a particular IDL interface and use its methods to exchange information in pull or in push mode.

The Event Service enables CORBA clients to participate in many-to-many communication through an event channel. However, the asynchronous communication is implemented on top of CORBA's synchronous method invocation and thus has the substantial overhead of performing a remote method invocation for every event communication. Moreover, event consumers cannot filter the events they receive from an event channel because no event filtering is supported. In particular, the lack of filtering mechanisms has led to the development of the Notification Service, which can be seen as the successor of the Event Service.

CORBA Notification Service

As the successor of the Event Service, the CORBA *Notification Service* [287] addresses the shortcomings of the Event Service by providing event filtering, quality of service (QoS), and a lightweight form of typed events, called *structured events*. With the Notification Service, suppliers can discover which event types are currently required by all consumers of a channel so that suppliers can produce events on demand, or avoid transmitting events in

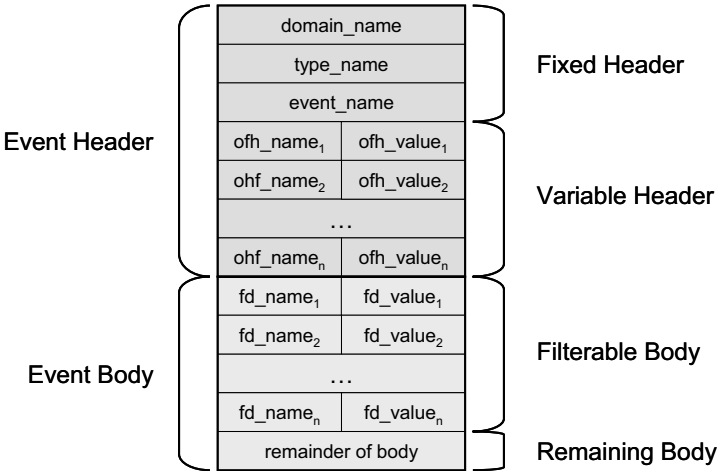

Fig. 9.4. The structure of a structured event (from [287])

which no consumers have interest. Similarly, consumers can discover all event types offered by suppliers so that consumers may subscribe to new event types as they become available. The Notification Service Specification also addresses an optional event type repository that, if present, makes information about the structure of events which may flow through the channel available.

Structured events are divided into a header and a body (Fig. 9.4). The header consists of a fixed and an optional variable header, while the body comprises a filterable and the remaining body. The fixed header contains the domain name, the type, and the unique name of the event. The variable header and the filterable body both contain name/value pairs that hold the data associated with the event. Event consumers can restrict the events that they receive from the event channel by specifying filters over the name/value pairs. The Notification Service Specification requires that an implementation support the *Default Filter Constraint Language*, which is an expressive content-based filtering language that also allows users to filter events based on QoS constraints. Besides the default filter constraint language, an implementation may support any number of additional filter constraint languages.

The Notification Service suffers from the same problems with regard to communication efficiency as the Event Service since both use synchronous two-way calls for event delivery. Moreover, there are still a number of problems inherent to a channel-based solution. Producers and consumers, that is, the application components, have to deal with channels explicitly. They have to select the right ones moving information about the application structure into the components—there is no support for the role of an administrator to arrange channels, producers, and consumers from a system point of view. Using channels also limits system evolution, since the set of channels referenced

by applications is static, a problem which is only recently addressed by reflective middleware [96]. Regarding the structure of a system, CORBA channels cannot reflect any hierarchy because their traffic is completely separated. Although event management domains [282] support the federation of multiple channels in arbitrary topologies, they do not offer any filtering of notifications between coupled channels.

Notification Service instances may be federated with the help of ORB domains. However, the necessary bridging between these domains has to be set up manually. The domains are mostly seen as means to model the network and broker infrastructure [318]; they are not targeted at engineering issues of application design. So, in the end one can only assess that the standardized API does not support visibility control and system management sufficiently well, but the CORBA Notification Service may serve as a communication technique to realize a subset of a scope graph.

9.1.2 Jini

The Java programming language is popular for network programming and therefore has some built-in middleware functionality. The*Java Remote Method Invocation* (RMI) specification [367] describes how to synchronously invoke methods of remote objects using request/reply communication between two*Java Virtual Machines (JVMs)* running on separate nodes. The Java RMI compiler generates marshaling code for the proxy object and the server skeleton. Because of the homogeneous environment created by JVMs, in which there is only a single programming language, the burden on the middleware is lower. It is even possible to move executable code between JVMs by using Java's *object serialization* to flatten an object implementation into a byte stream for network transport.

Asynchronous event communication within a single JVM is mainly used in the *abstract window toolkit (AWT)* [358] libraries for graphical user interfaces. The `EventListener` interface can be implemented by a class to become a callback object for asynchronous events, such as mouse or keyboard events. The Jini framework, described below, extends this to provide event communication between different JVMs. Other variants of asynchronous communication in Java are provided by the messaging infrastructure of JMS (Sect. 9.1.3).

The *Jini* specification [360] enables programmers to create network-centric services by defining common functionality for service descriptions to be announced and discovered. For this, it supports distributed events between JVMs. A `RemoteEventListener` interface is capable of receiving remote callbacks of instances of the `RemoteEvent` class. A `RemoteEvent` object contains a reference to the Java object where the event occurred and an `eventID` that identifies the type of event. A `RemoteEventGenerator` accepts registrations from objects and returns instances of the `EventRegistration` class to keep track of registrations. It then sends `RemoteEvent` objects to all interested `RemoteEventListener`s. Event generators and listeners can be decoupled by

third-party agents, for example, to filter events, but the implementation is outside the Jini specification and left to the programmer.

JavaSpaces [366] are a part of the Jini framework; they are similar to Linda tuple spaces. With JavaSpaces, tuples can be inserted, read, and removed from a space which stores each tuple from the time it is inserted to the time it is removed. The corresponding operations are `write`, `read`, and `take`. `read` and `take` take a template and block until a tuple that matches the given template is present in the space. `readIfExists` and `takeIfExists` are the nonblocking versions of `read` and `take`; they return instantaneously if a matching tuple is not in the space. To reveal clients from polling for matching tuples, clients can be notified when a matching tuple is inserted into the tuple space via the `notify` operation. However, since there can be multiple listeners notified, it is not guaranteed that a notified client will actually retrieve a matching tuple. There can be multiple spaces that can reside on different hosts. Transactions are also supported. For example, a tuple which is written within a transaction becomes visible outside the transaction only after the transaction committed. More details on transactions can be found in the Jini specification. Fairness and ordering of operations is not addressed by the JavaSpaces specification. *TSpaces* [402] developed by IBM are similar to JavaSpaces.

Summarizing, as is the case for CORBA, event communication in Jini is built on top of synchronous communication (Java RMI), so the same restrictions that limit scalability and efficiency apply.

9.1.3 Java Message Service (JMS)

The *Java Message Service* (JMS) [364] defines a messaging API for Java. Differently from the CORBA Event or Notification Service, JMS can be used without the enterprise object platform, i.e., J2EE [365], of which it is part. JMS clients can choose any vendor-specific implementation of the JMS specification, called a *JMS provider*. JMS comes with two communication modes: point-to-point and publish/subscribe communication. *Point-to-point communication* follows the one-to-one communication abstraction of message queues. Queues are stored and managed at a JMS server that decouples clients from each other. Direct communication between a sender and a receiver without an intermediate server is not supported. In *publish/subscribe communication*, the JMS server manages a number of *topics*. Clients can publish messages to a topic and subscribe to messages from a topic.

JMS provides a topic-based publish/subscribe service with limited content-based filtering support in the form of *message selectors*. A message selector allows a client to specify the messages it is interested in by specifying a filter that operates on the fields of the message header; body fields cannot be evaluated. The selector syntax is based on a subset of the SQL92 [101] conditional expression syntax.

Like structured CORBA events, a JMS message is divided into a message header and body. The header contains various fields, including the destination

of the message, its delivery mode, a message identifier, the message priority, a type field, and a timestamp. The delivery mode can be set to `PERSISTENT` to enforce exactly-once delivery semantics; otherwise best-effort delivery applies. The type of a message is an optional field that can be used by a JMS provider for type-checking the message. Apart from predefined fields, the header can also contain any number of user-supplied fields. The message body is in one of several formats: a `StreamMessage`, a `TextMessage`, and a `ByteMessage` containing the corresponding Java primitive types. A `MapMessage` is a dictionary of name/value pairs similar to the fields found in the header. Finally, an `ObjectMessage` uses Java's object serialization feature to transmit entire objects between clients.

Messages can be consumed synchronously or asynchronously, i.e., either pull or push can be used to transfer messages to the respective consumer. There exist two ways of message acknowledgment: messages can either be acknowledged automatically or specifically by the client. Moreover, messages can be persistent or volatile. *Persistent messages* are delivered exactly once to a consumer. They also do not get lost if the provider fails; they usually are logged to stable storage. However, this comes at the cost of a much higher overhead. *Volatile messages* are delivered at most once; they may get lost if the provider fails.

With JMS, subscriptions can either be durable or not. With *durable subscriptions*, notifications are retained while the subscriber is disconnected from the provider until they have been delivered or expired. To the contrary, with a nondurable subscription, notifications that are published while the subscriber is disconnected may get lost.

Sessions can be transactional or nontransactional. *Transactional sessions* allow clients to group the publication and the consumption of several messages into an atomic unit of work. On the producer side, produced messages are retained until commit and if transaction aborts messages are discarded. On the consumer side, all consumed messages are kept until commit and are automatically acknowledged on commit. If the transaction aborts, the messages are redelivered. Hence, messages are actually sent and received when the transaction commits. Since the production and the consumption of the same message cannot be part of the same transaction, only *local* transactions are possible. Another consequence is that transacted sessions cannot be used to implement request/reply interaction. Moreover, point-to-point operations and publish/subscribe operations cannot be mixed inside a single transaction.

Although, at first sight, JMS appears to be a strong contestant for a large-scale middleware, it suffers from several shortfalls: First, the entire model is centralized with respect to JMS servers. As a result, JMS servers are heavyweight middleware components and can become bottlenecks because the JMS specification does not address the routing of JMS messages across multiple servers or the distribution of servers to achieve load balancing. Second, content-based filtering of messages in JMS only considers the message header but not the message body. This seriously reduces the usefulness of message

filtering. Finally, JMS is tightly integrated with the Java language. This has the advantage that object instances can be published in a message, but comes with the price of only supporting Java clients, which is not feasible in a large-scale, heterogeneous distributed system.

Another main problem is that aspects that will be important for any JMS implementation are not addressed by the JMS specification. This includes, for example, exception handling, load balancing, fault tolerance, end-to-end security, administration, and message type repositories. For example, the specification leaves open how to define topics or how they are interrelated. Many of these aspects are nevertheless addressed and implemented differently by individual vendors. Hence, applications using these products are incompatible if they use these implementation-specific features.

9.1.4 Data Distribution for Real-Time Systems (DDS)

The Data Distribution Service for Real-Time Systems (DDS) [286, 299] was standardized by the OMG in 2004. DDS follows a "data-centric" approach: it creates the illusion of a *global data space* populated by data objects that applications in distributed nodes can access via read and write operations [298]. Related industrial products, e.g., Splice DDS from Thales (US) [375, 384] and NDDS [324] from Real-Time Innovations (US) are available. The specification describes two layers of interfaces:

- The mandatory *data-centric publish/subscribe (DCPS)* level is targeted toward the efficient delivery of information to interested recipients. It allows for content-based publish/subscribe communication between publishers and subscribers and lays an emphasis on quality of service (QoS).
- The optional higher *data local reconstruction layer (DLRL)* level allows for a simple integration of the service into the application layer. The DLRL automatically reconstructs the state of cached objects locally from updates and allows applications to access objects as if they were local.

Since real-time systems are the application domain of the DDS, special care must be taken to design the interfaces such that real-time requirements can be met by the implementation. The service implementation must be able to preallocate resources reducing dynamic resource allocation to a minimum. For example, copying data should be minimized for efficiency reasons and resource usage should be predictable and bounded. Also due to efficiency reasons, typed events with interfaces are used such that type-safety can be ensured at compile time. Here, typed means that for each datatype, specific classes are generated. Generation tools translate event descriptions into the proper interfaces bridging the gap between typed interfaces and the generic service implementation. The specification pays attention to separate producers from consumers such that they can be implemented independently to facilitate extensibility. QoS is an important issue for the DDS. QoS is supported through several QoS policies that declaratively specify which QoS should be provided

instead of how this QoS should be realized. Publishers offers a maximum level for each QoS policy, while subscribers request a minimum level for each QoS policy. For example, a subscriber can request that it wants to receive an update at least once in a given time interval. The next two sections describe the DCPS and the DLRL in more detail.

Data-Centric Publish/Subscribe (DCPS)

The Data-Centric Publish/Subscribe (DCPS) layer is responsible for getting data from publishers to interested subscribers. In the following we describe the main components of DCPS (Fig. 9.5).

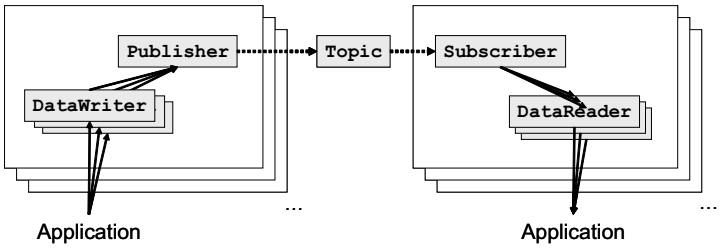

Fig. 9.5. Conceptual overview of data-centric publish/subscribe (DCPS)

A `Publisher` is an object responsible for data distribution. A `DataWriter` is a typed facade that provides access to a publisher. It is bound to exactly one `Topic`, `Publisher`, and application datatype. An application uses a `DataWriter` to communicate with a `Publisher` to the let it know the existence and the value of data objects of a given type. A `Subscriber` is an object responsible for receiving published data. A `DataReader` provides typed access to a subscriber, i.e., to the received data. It is bound to exactly one `Topic`, `Subscriber`, and application datatype. The application associates a `DataReader` to a `Subscriber` to receive the datatype described by the `DataReader`. The QoS experienced by a subscriber is affected by a number of issues. In addition to the `Topic` QoS, the QoS of the `DataWriter`, and the QoS of the `Publisher` affect the QoS on the publisher's side. On the subscriber's side, the QoS is affected by the `Topic` QoS, the `DataReader` QoS, and the `Subscriber` QoS. A `Topic` is conceptually located between publishers and subscribers. It has a name that is unique in the domain and a QoS policy. DCPS differs from other notification services by the fix binding of a `Topic` to a datatype. `ContentFilteredTopic` and `MultiTopic` derive from `Topic`; they can only be used by a `Subscriber`. A `ContentFilteredTopic` provides means for content-based filtering that is similar to the `WHERE` clause of an SQL query. The optional `MultiTopic` class allows users to get data from multiple

topics and to combine, filter, and rearrange this data. The data will then be filtered and possibly rearranged using aggregation and projection.

`Topic`, `Publisher`, and `Subscriber` objects are created using the respective `create_` operation of `DomainParticipant`. A `DomainParticipant` acts as an entry point for an application to the service, serves as a factory for many of the classes, and acts as container for the other objects that make up the service. It represents the local membership of an application in a domain which is a distributed concept, allowing all applications of this domain to communicate with each other.

Datatypes represent information that is sent and received atomically. Instances of a datatype are identified by a *key*. Data with the same key are treated as successive values of the same instance, while data with different keys are treated as referring to different instances. By default, data modifications are disseminated individually, independently, and uncorrelated from other modifications. It is, however, possible that an application requests several modifications to be sent and also received atomically.

To publish data an application first creates a `DomainParticipant` using `DomainParticipantFactory`. If the respective `Topic` does not exist, the applications creates it using the `DomainParticipant`. Then, the application creates a `Publisher` using the `DomainParticipant` and uses the `Publisher` to create a `DataWriter`. If the application decides to publish data, it calls the `write` on the corresponding `DataWriter`.

To subscribe to data, an application uses a `DomainParticipant` to find the `Topic` of interest. Then, it uses the `DomainParticipant` to create a `Subscriber` and uses the `Subscriber` to create a `DataReader`. To receive data an application can either use a `Listener` or a `WaitSet` object. These represent the two basic ways of receiving data and are called *notification-based* and *wait-based*, respectively.

The notification-based interaction style uses listeners. Applications register handlers that are invoked by the middleware to notify the applications about asynchronous events such as the arrival of new data or a QoS violation. From the `Listener` interface, more specific listeners such as `DataReaderListener` derive; they add methods depending on the concrete `Listener`.

The wait-based interaction style uses `WaitSet` objects that allow an application to wait until one or more of the attached `Condition` objects are triggered or else until a timeout expires. `Condition` is subclassed by `GuardCondition`, `StatusCondition`, and `ReadCondition`. A `GuardCondition` is under the control of an application and can be used by the application to manually wake up the `WaitSet`. A `StatusCondition` is attached to any entity; it provides information about the communication status of the respective entity such as the arrival of new data. A `ReadCondition` allows an application to specify the data samples, in which it is interested. This allows the middleware to enable the condition only when suitable information is available. If data are available, the application can either call `read` or `take` on the respective

`DataReader`. While `read` allows the data to be read again later, `take` removes the data from the `DataReader`.

Data Local Reconstruction Layer (DLRL)

The *Data Local Reconstruction Layer (DLRL)* is an optional layer that may be built on top of DCPS. DLRL allows for a simple integration of the service into the applications by offering an interface on a higher level than DCPS does. DLRL defines an object cache that allows the application to access objects "as if" it were locally available by automatically reconstructing the state of the cached objects from the updates received. To achieve this, object modifications are propagated using DCPS to all parties having a copy of the respective object in their cache and the copies are accordingly updated.

With the DLRL an application can describe DLRL objects with methods, attributes, and relations. Attributes can be either local or shared. As their name suggests, only shared attributes take part in dissemination. To ensure their dissemination, shared attributes are attached to DCPS entities. A DLRL object has at least one shared attribute. DLRL objects can be manipulated using the native language constructs, which in turn triggers changes to the corresponding DCPS entities in the background. Single inheritance of DLRL objects is supported and different kinds of associations can be used to relate DLRL objects to each other. The associations can be used to navigate among the DLRL objects. With the DLRL, the application model is given in OMG IDL. In addition, for example, the mapping from application types to topics and which attributes should be shared are defined using XML. The required classes are then generated automatically by an IDL compiler.

To achieve the dissemination of object modifications, the DLRL specification defines several mappings between the DCPS and the DLRL layer:

1. The *structural mapping* defines the relation between DLRL objects and DCPS data. It is very similar to an object to relation mapping known from database management. Each DLRL object is mapped to a DCPS data sample. Topics correspond to database tables, and data samples correspond to tuples.
2. The *operational mapping* defines the relation between DLRL objects and DCPS entities (e.g., `Topic`). For example, each DLRL class is mapped to several DCPS topics. The use of the DCPS entities is totally transparent to the application using DLRL.
3. The *functional mapping* defines the relation between DLRL functions (mainly access to the DLRL objects) and the DCPS functions.

Several classes are used by an application to access DLRL objects at runtime. A `Cache` contains a set of objects that are locally available and that are managed consistently. Its contents are updated transparently when updates arrive. A `Cache` is created using a `CacheFactory`. At creation time its mode is set to read-only, write-only, or read/write. A `Cache` comprises one or more

`CacheAccess` objects that isolate a set of objects in a given access mode. A `CacheAccess` allows users to globally manipulate DLRL objects in isolation.

9.1.5 WS Eventing and WS Notification

Most early Web services were based on synchronous request/reply interaction. After Web services had been on the market for some years, the need for asynchronous push capabilities was recognized. These capabilities are needed for services such as stock quoting services if they should not be based on resource-intensive polling. Pushing information to a service requires that the service can be contacted using a communication endpoint. *Web Services Addressing (WS-Addressing)* introduces service endpoint references for Web services. These endpoints can be passed as message parameters, for example, to register a subscription, and to subsequently deliver messages to the registered service. The different parts of WS-Addressing are currently being standardized by the World Wide Web Consortium (W3C). On top of WS-Addressing, *Web Services Eventing (WS-Eventing)* [201, 387] resides. It lets a Web service, called *event sink*, register at another Web service, called *event source*, such that the former can receive notification messages from the latter. A subscription is only valid until an expiration time, which is passed by the event source to the event sink as part of the subscription reply message. The event sink can request the notifications to be filtered by an *event filter*, which is a Boolean expression that is by default given as an XML XPath expression.

Web Services Notification (WSN) [200, 386] is an alternative to WS-Eventing. WSN is currently being standardized by the OASIS (Organization for the Advancement of Structured Information Standards). It consists of *Web Services Base Notification (WS-BaseNotification)* [274], *Web Services Brokered Notification (WS-BrokeredNotification)* [275], and *Web Services Topics (WS-Topics)* [276]. WSN also builds upon WS-Addressing. WS-BaseNotification defines the `NotificationConsumer` interface and the `NotificationProducer` interface used for *direct notification*, and specifies messages and message exchanges to be implemented by services that wish to act in these roles along with operational requirements expected of them. Consumers register their subscriptions directly at the producers. Content-based filtering is supported by *selector expressions*. A producer sends a notification directly to the consumers that registered a matching subscription. WS-BrokeredNotification defines interfaces, messages, and message exchanges needed for *brokered notification*, which uses notification brokers as intermediaries to decouple producers from consumers. WS-Topics define the concepts centered around topic-based publish/subscribe such as topics, topic spaces, topic trees, and topic expressions.

9.1.6 The High-Level Architecture (HLA)

The *High-Level Architecture(HLA)* [98] originated at the U.S. Department of Defense in 1996 and was later standardized by the IEEE (Standard 1516)

and by the OMG. The corresponding standard of the OMG is described in the *Distributed Simulation Systems (DSS)* specification [284]. The HLA is mainly used to deploy distributed simulations. It provides the specification of a common technical architecture for use across all classes of simulations in the US Department of Defense serving as a structural basis for simulation interoperability. With the HLA a simulation is carried out by a set of *federates*. Each federate manages a set of *objects* (e.g., tanks), which move in a *routing space*. Inside the HLA, *Data Distribution Management (DDM)* services support the routing of data among federates during the course of a federation execution. Especially, DDM allows for content-based subscriptions based on object attributes. However, content-based filtering is usually done on the client side. Federates express their interest to receive updates by subscribing to all updates that occur in a rectangular *region* of the routing space. Besides these *subscription regions*, there are *update regions*. Regions may change, for example, when an object moves in the routing space.

For distributing the updates, region-based and grid-based approaches are used [48]. With the *region-based* approach usually one multicast group is used for every update region and the subscribing federates join those groups that overlap with their subscription regions. With the *grid-based* approach, the routing space is divided into *cells* and for each cell a multicast group is used. Publishing federates publish updates to those multicast groups the update belongs to and subscribing federates join all groups that overlap with their subscription regions.

9.2 Commercial Systems

We discuss IBM WebSphere MQ in Sect. 9.2.1, TIBCO Rendezvous in Sect. 9.2.2, and Oracle Advanced Queuing in Sect. 9.2.3. Instead of describing all features of these commercial systems, we put an emphasis on those features which are related to publish/subscribe.

9.2.1 IBM WebSphere MQ

IBM *WebSphere MQ (MQ)* [198, 199] (formerly known as IBM MQSeries) is a messaging platform that is part of IBM's WebSphere suite. MQ is a powerful middleware, whose strength lies in the simple integration of legacy applications through loosely coupled queues. A particular strength of WebSphere MQ is its availability for many platforms including Windows, Linux, Solaris, and many others. Its main focus is on point-to-point messaging using queues, especially request/reply on communication. A *queue manager* is a process that manages a set of queues and offers the queuing services to applications via an API. Several programming language bindings of the API to send and receive messages to and from queues exist. WebSphere MQ comes with advanced messaging features, such as transactions, clustered queue managers for load

balancing and availability, and built-in security mechanisms. Additionally, a queue manager provides functions to administrators so that they can create new queues, alter the properties of existing queues, and control the operation of the queue manager. For a program to use the services of a queue manager, it must establish a connection to that queue manager.

WebSphere MQ Publish/Subscribe (MQPS)

WebSphere MQ Publish/Subscribe (MQPS) allows MQ applications to communicate using publish/subscribe communication. MQPS was originally a supplement for MQSeries but was later incorporated into WebSphere MQ. It offers topic-based publish/subscribe communication; no content-based subscriptions are supported. In topic-based subscriptions, two wildcards can be used: while a `?` can be replaced by any single character, an `*` can be replaced by any sequence of characters. It is suggested to use the `/` to organize the topics into a hierarchy. The publisher specifies the topic of a publication when it publishes the information, and the subscriber specifies the topics on which it wants to receive publications. The routing of messages from producers to subscribers is carried out by a *broker* that uses standard MQ functionality to achieve this. Hence, an application using MQPS can use all the features available to existing MQ applications. Publishers can optionally register their intention to publish information on a certain topic at the broker. Publishers and subscribers do not have to be on the same machine as a broker. They can reside anywhere in the network, provided there is a route from their queue manager to the broker.

Related topics can be grouped together to form a *stream*. Streams separate the information flow of the grouped topics from topics in other streams. At each broker that supports a stream, there is a queue with name of the stream. There is a default stream. Streams can also be used to restrict the types of publication a broker has to deal with. This can, for example, be used for load balancing. Access control is also done based on streams.

Brokers can be connected to each other to form a hierarchy. Subscriptions flow to all nodes in the network that support the respective stream. A broker consolidates all the subscriptions that are registered with it, whether from applications directly or from other brokers. In turn, it registers subscriptions for these topics with its neighbors, unless a subscription already exists. Hence, forwarding of identical subscriptions is avoided. When an application publishes information, the receiving broker forwards it (possibly through one or more other brokers) to any applications that have valid subscriptions for it, including applications registered at other brokers supporting this stream.

MQPS allows publications to be retained such that they can be delivered to subsequent subscribers. This way, new subscribers can gather information without having to wait until it (or an updated version) is published again.

WebSphere Business Integration Event Broker (BIEB)

The *WebSphere Business Integration Event Broker (BIEB)* is a complement to WebSphere MQ. BIEB provides high-performance nonpersistent publish/subscribe functionality to clients that can then use content-based subscriptions in addition to topic-based subscriptions. Brokers can be connected to form a hierarchy. Brokers can also be grouped together to form fully connected *collectives*; in this case, the collectives are then connected to form a hierarchy. Brokers can also be *cloned* to improve the availability of the publish/subscribe system. Subscriptions are propagated through the broker network. However, only the topic filter is propagated and not the content filter. Hence, a broker might receive publications in which none of its subscribers is interested. Additionally, it is possible to use IP multicast to distribute subscriptions and publications in LANs.

Message *flows* can be defined that describe operations to be performed on an incoming message, and the sequence in which they are carried out. A flow consists of a number of *flow nodes*, each of which corresponds to a processing step. The *flow connections*, which connect flow nodes, define which processing steps are carried out, in which order, and under which conditions. A flow node can also contain a *subflow* which allows message flows to be composed. Message flows run in a container called *message flow project* which are deployed at a broker. *Subscription points* can be used to make information associated with a particular topic available in a number of different formats. For example, stock prices might be published with a default currency of dollars, but might be required by subscribers expressed in other currencies. Subscription nodes are implicitly connected to *publication nodes* of message flows.

9.2.2 TIBCO Rendezvous

TIBCO Software (US) is a major player in the publish/subscribe middleware market. Its publish/subscribe middleware product TIBCO Rendezvous has been available for many years and has been applied by major customers, especially in the area of financial services. For example, the NASDAQ has implemented its trading floor using TIB Rendezvous. According to TIBCO, the trading floor infrastructure handles 1.8 billion real-time messages per day and 25 thousand trades per second. The current version of TIBCO Rendezvous, as of January 2006, is Version 7.4 [381]. TIBCO Rendezvous is available for many platforms including Linux, Windows, Solaris, and FreeBSD, and APIs are available for many programming languages including Java, C, C++, and Perl 5. TIBCO Rendezvous originally was called TIBCO's Information Bus (TIB) and was renamed later. It is based on ideas presented by Oki et al. [289], who proposed a distributed implementation of a subject-based publish/subscribe system called the *Information Bus*.

TIBCO Rendezvous uses patented subject-based addressing [345]. Subscriptions select subjects from a subject hierarchy. A single subject is selected by its dotted name (e.g., `stocks.technology.fooInc`), where the

parts of the name that are separated by dots are called *elements*. An application can use *wildcards* to select more than one subject. The wildcard `*` can be replaced by any element, while the wildcard `>` can be replaced by any dot-separated sequence of elements. Hence, `stocks.technology.*` matches `stocks.technology.foo` but not `stocks.technology.software.bar`, while `stocks.technology.>` matches both. The mapping from subjects to underlying transport protocols, in particular to specific IP multicast addresses, has to be done manually, and it is statically encoded in every producer and consumer. Although the inherent communication efficiency of IP multicast is appealing, it comes at the cost of a rather static configuration, which not only complicates maintenance, but also restricts configurability and integration, and thus the range of possible application domains [382].

A program, which wants to participate in a distributed system in which hosts communicate by the means of TIBCO Rendezvous, uses a TIBCO Rendezvous API library matching the used platform and programming language. In such a system each participating host runs a *rendezvous daemon* (`rvd`), which runs as a separate process. Each message published by a program is handed out to the local daemon via the API library and is then multicast to all daemons in this network. Programs attempt to connect to a local daemon. If a local daemon process is not yet running, the program starts one automatically and connects to it. The daemons hide many details from the programs such as data transport, packet ordering, receipt acknowledgment, and retransmission requests.

With TIBCO Rendezvous *messages* are the entities that travel among programs. A message comprises *data fields*, a subject indicating its destination, and an optional reply-to subject. Each field contains one data item which can be identified by either by its name or by its numerical identifier. Programs do not have to know the wire format of messages; conversions to and from the wire format are transparent to the application. The wire format contains, besides the data itself, also metainformation about the data contained such that the data is "self-describing" in the sense that the receiver is able to interpret and use the data properly.

To register interest in a set of *event occurrences*, a program creates an *event object* whose parameters specify that set. The programmer can specify in which event queue an occurred event is inserted and which callback function is invoked when the event is dispatched. Dispatching can be done in several ways. Queues can be prioritized and grouped to have a fine-grained control of dispatching. Discarding policies can be chosen that specify which event (e.g., the first in a queue) is discarded when the queue size exceeds a given limit. Besides *message events*, which signal the arrival of a message, *timer events* and *I/O events* are supported. The *event driver* recognizes the occurrence of events and places them in the appropriate event queues for dispatch. To receive messages, programs create *listener events*, which specify that messages which match a subject name (that may contain wildcards) are of interest, define callback functions to process the inbound messages, and dispatch events in

a loop. A *transport* defines the delivery scope of messages. While *network transports* deliver messages across a network, *intraprocess transports* deliver messages only between program threads within a single process. The creation of a transport takes a `service` parameter. Messages do not travel among transports having different `service` parameters. Together with all listener events bound to a transport, a transport defines the actual set of receivers of a published message.

TIBCO Rendezvous supports two levels of message reliability. With *reliable delivery* the middleware tries to do its best to ensure that a message reaches all participants. However, certain faults, such as daemon crashes, can lead to applications not getting all messages they would have gotten without this fault. The advantage of this scheme is its good performance. With *certified delivery*, the delivery of messages is guaranteed. Messages additionally carry the sender's name, a subject-independent message ID, and an expiration time. This information is used by daemons and routers to request retransmissions of missing messages and to discard expired messages. Despite retransmissions, the order in which messages are delivered satisfies a FIFO-sender policy. To ensure that messages can be delivered even in case of daemon crashes and subsequent restarts, messages are stored persistently. However, reliability comes at a cost: certified delivery greatly degrades the performance of the system.

Independent networks of TIBCO Rendezvous instances can be connected with *information routers*. They forward messages between distinct networks so that subscribers can transparently listen for subject names and receive messages from other networks. Administrators managing the routers have control over the subject names (and associated messages) that are relayed and flow in or out of a network. These routers offer a basic means of structuring.

TIBCO Rendezvous has proven to be scalable to large-scale systems. However, if the subject-based filtering is not expressive enough, extra filtering of events is left to the subscribers. In these cases, scalability can become a problem and the network might be overwhelmed by too many event broadcasts. A JMS implementation is also available from TIBCO (cf. Sect. 9.1.3).

9.2.3 Oracle Streams Advanced Queuing

Oracle Streams Advanced Queuing (AQ) was the first database-integrated messaging system in the industry. This approach is contrary to products such as TIBCO Rendezvous (cf. Sect. 9.2.2), which are not bundled with a database. With the release of Oracle 10, AQ was renamed to *Oracle Streams Advanced Queuing*. AQ offers a JMS implementation (cf. Sect. 9.1.3) called *Oracle JMS*, which is compliant to JMS 1.1 and a proprietary API for queues. Oracle recommends using the standardized JMS API instead of the proprietary AQ API, if Java is used as programming language. As a result of the database integration of AQ, all the functionality offered by the Oracle 10 database can be applied to messaging. This includes query support, indexing,

transactions, triggers, consistency constraints, logging, replication, authentication, access control, backup, recovery, data export, and data import.

The basic abstraction of AQ, which decouples producers of messages from consumers of messages, are *queues*. Due to the tight database integration of AQ, queues are normal database tables and messages are normal rows in database tables. Hence, messages can be accessed (i.e., queried) using standard SQL. SQL can be used to access the message properties and the payload. Message histories are available and indexes can be used to optimize access.

Messages can be *enqueued* into or *dequeued* from a queue. Multiple producers can enqueue messages into a queue, and multiple consumers can dequeue messages from a queue. AQ distinguishes among *single-consumer* and *multi-consumer* queues. While single-consumer queues are used for point-to-point messaging, multiconsumer queues can be used for different kinds of point-to-multipoint messaging, including publish/subscribe communication. To allow multiple consumers to dequeue the *same* message from a queue, AQ supports *message recipients* and *queue subscriber*. If a message should be consumed by multiple consumers, it remains in the queue until it is consumed by all its intended consumers. While message recipients are specified by the producer of a message, applications or other queues must subscribe to a queue to become a queue subscriber. Subscriptions can be *rule based*. In this case, not all messages that are enqueued can be dequeued by a queue subscriber, but only those that match the subscription, which is specified in a syntax similar to a `WHERE` clause of SQL. A subscriber can specify a callback that is invoked to notify it asynchronously about the availability of a new matching message.

There are a number of enqueue and dequeue options available, such as an earliest dequeue time for a message and a message expiration time. Messages are not necessarily dequeued in the order in which they are enqueued. Messages can be grouped to form a set that can only be consumed by one consumer at a time. This feature can, for example, be used to transfer a huge payload by a set of messages. Messages can be retained for a given period after consumption. In a message history also the enqueue time and the dequeue time of a message is saved. Retained messages can be related to each other and applications can track sequences of related messages and produce event journals automatically.

Messages can be propagated based their content from a queue to other queues residing either in the same database or in remote databases. This enables applications to communicate that are not connected to the same queue or to the same database. With message propagation, messages can be fanned out to a large number of recipients without requiring them all to dequeue messages from a single queue. This is known as compositing or funneling messages. Messages can also be propagated using `HTTP` or `HTTPS`. AQ allows for message format transformations which are represented by SQL functions. Messages can be transformed during enqueue or during dequeue.

An alternative to *persistent messaging* is *buffered messaging*, which provides a much faster queuing implementation. Buffered messaging is useful for

applications not requiring the reliability and transaction support of persistent messaging. It is faster because it stores messages in main memory and only writes messages to disk if the main memory is too small to hold all current messages. Buffered messaging uses the same API as persistent messaging.

In summary, Oracle Streams Advanced Queuing is a feature-rich messaging system that supports different communication styles including publish/-subscribe. Because of its tight database coupling it exhibits many interesting features that other systems do not expose. However, this comes at the cost of a rather heavyweight implementation.

9.3 Research Prototypes

Many research prototypes have emerged since the second half of the 1990s. The pioneers of this area were the Gryphon (Sect. 9.3.1), the SIENA (Sect. 9.3.2), the JEDI (Sect. 9.3.3), the READY (Sect. 9.3.8), and the Elvin (Sect. 9.3.7) event notification services and the Cambridge Event Architecture (CEA) (Sect. 9.3.6). From the newer approaches we present REBECA in Sect 9.3.4 and HERMES in Sect. 9.3.5. Each of the systems we discuss in the following has its own focus (e.g., routing or matching) and differs from the others in some way. With the above selection of systems we try to cover most of the area. Of course, there are many other research prototypes that are not discussed in this book.

9.3.1 Gryphon

The *Gryphon* project at IBM Research [203] led to the development of an industrial-strength, reliable, content-based event broker that is now part of IBM's WebSphere suite as the IBM WebSphere MQ Event Broker [202]. It is a mature publish/subscribe middleware implementation with a JMS interface that provides a redundant, topic- and content-based multibroker publish/subscribe service. The Gryphon event broker has been successfully deployed for large-scale information dissemination at global sports events, such as the Olympic Games. Opyrchal et al. have also investigated how IP multicast can be used to improve the efficiency of event distribution [291] (cf. Sect. 4.6.7). Gryphon includes an efficient event matching engine [6], a scalable routing algorithm, and security features.

Gryphon is based on an information flow model for messaging [28, 354]. An *information flow graph (IFG)* specifies the exchange of information between information producers and consumers. Information flows can be altered by (1) filtering, (2) stateless transformations, and (3) stateful transformations (aggregation). A logical IFG is mapped onto a physical event broker topology. Figure 9.6 shows an example of a Gryphon deployment. Nodes in the IFG are partitioned into a collection of *virtual brokers* PHB, $IB_{1,2}$, and SHB_{1-4}, which are then mapped onto clusters of physical event brokers called *cells*. Similarly,

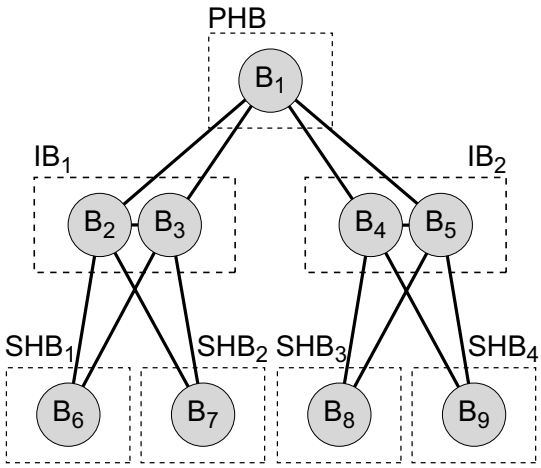

Fig. 9.6. A Gryphon network with virtual event brokers

edges connecting nodes in the IFG are *virtual links* that map onto *link bundles*, containing multiple redundant connections between event brokers for reliability and load balancing.

An event broker that has publishing clients connected to it is called a *publisher-hosting broker (PHB)*. It contains *publisher endpoints* (or *pubends*), which represent a collection of publishers that enter information into the IFG. Correspondingly, a *subscriber-hosting broker (SHB)* consumes information through one or more *subscriber endpoints* (or *subends*) from the IFG according to its subscriptions. An event broker that is neither publisher-hosting nor subscriber-hosting is an *intermediate broker* (IB). The topology mapping is statically defined at deployment time, although more recent work [410] includes dynamic topology changes due to failure and evolution. Several extensions are implemented as part of the Gryphon event broker; these are discussed in the following.

Guaranteed Delivery

A *guaranteed delivery service* [39] provides exactly-once delivery of events, as required for JMS persistent events. The propagation of information (*knowledge*) from pubends to subends is modeled with a *knowledge graph*. Lost knowledge due to message loss causes *curiosity* to propagate up the knowledge graph and trigger the retransmission of events. Curiosity is implemented as *negative acknowledgment (NACK)* messages sent by SHBs. A subscriber that remains connected to the system is guaranteed to receive a gapless ordered filtered subsequence of the *event stream* published at a pubend. A more detailed description of guaranteed delivery and how it can be extended to address congestion in an event-based middleware is given in Sect. 8.3.

Durable Subscriptions

The *durable subscription service* [40] guarantees exactly-once delivery despite periods of disconnection of event subscribers from the system. This means that the event stream is buffered while a subscriber is not available and replayed upon reconnection. As for the guaranteed delivery service, an event log is kept at PHBs and cached at intermediate brokers.

Relational Subscriptions

The final extension is the *relational subscription service* [214]. Its goal is to implement the stateful transformations supported by Gryphon's IFG model, combining messaging with a relational data model. Relational subscriptions can be seen as a continuous query over event streams, providing event subscribers with the expressiveness of a relational language. This relates to the requirement for composite event detection in an event-based middleware, which is discussed in Chap. 7.

The Gryphon event broker includes many of the features that a distributed systems' programmer expects from an event-based middleware. However, the overlay network of event brokers is static, as it is defined in configuration files at deployment time. This makes it difficult for the middleware to adapt to changing network conditions. Failure within a cell of event brokers can be tolerated, but major changes to the IFG cannot be compensated for. Although composite event detection is provided by relational subscriptions, a relational data model for messaging might be too heavy-weight for many applications.

9.3.2 Siena

One of the first implementations of a distributed content-based publish/subscribe system was the *scalable internet event notification architecture* (Siena) [65, 71]. Siena is a multibroker event notification service that targets at Internet-scale deployment. Brokers are called *servers* in Siena. As usual, event publishers and subscribers connect to a server in the logical overlay network. Events published by publishers are then routed through the overlay network of servers depending on the subscriptions submitted by subscribers.

Siena uses covering-based routing in its hierarchical and its peer-to-peer variants. Other routing algorithms are not supported. The algorithms used by Siena are similar to those presented in Sects. 4.5.4 and 4.6.2. In case the peer-to-peer variant is applied, advertisements are supported. The algorithms build upon a *partially ordered set (POSET)*, which allows brokers to keep track of the covering relations among filters. More precisely, the transitive reflexive reduction of the partial order induced by the covering relation is stored. Each server manages a POSET that is accordingly updated when a subscription or unsubscription is processed by the server.

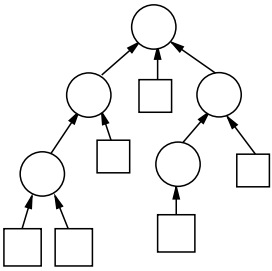

Fig. 9.7. A hierarchical topology in SIENA

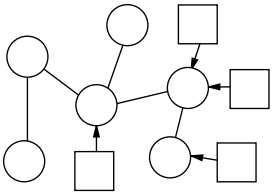

Fig. 9.8. An acyclic peer-to-peer topology in SIENA

The POSET can also be used for matching [71] by traversing it, for example in depth-first order, starting from the root filters, i.e., from those filters which cover all other filters. If a visited filter does not match, then no child filter can match the notification. Carzaniga et al. [67, 70] also presented an alternative matching algorithm that is based on the counting algorithm (cf. Sect. 3.2.2) and that is similar to those presented by Mühl [262].

In SIENA a notification consists of a set of typed attributes. Subscriptions and advertisements are conjunctions of attribute filters, which are simple predicates (e.g., comparisons) over the event attributes. If there is only one attribute filter per attribute, a notification matches a subscription (an advertisement) if it satisfies all attribute filters. However, the interpretation is different for subscriptions and advertisements if there is more than one attribute filter for an attribute. For a subscription, a notification has to match *all* of these attribute filters, while for an advertisement, a notification has to match *at least one* of these attribute filters. Hence, the models of subscriptions and advertisements differ. This fact complicates computing overlapping and covering among filters for more complex data types.

SIENA considers three different types of topologies: hierarchical (Fig. 9.7), acyclic peer-to-peer (Fig. 9.8), and generic peer-to peer (Fig. 9.9). In contrast to an acyclic topology, a generic peer-to-peer topology is not restricted to be a tree. Here, peer-to-peer only means that there is no master/slave relation among servers as there is for hierarchical topologies. In a hierarchical topology, hierarchical covering-based routing is used. In this case, the protocol that

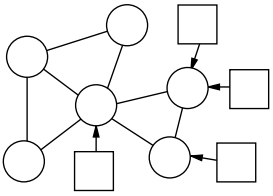

Fig. 9.9. A generic peer-to-peer topology in SIENA

clients use to interact with the respective server they are connected to is the same that a server uses to interact with its master server. Hence, there is an unidirectional flow of subscriptions from servers to their parent servers. In an acyclic or a generic peer-to-peer topology, peer-to-peer covering-based routing is applied. In this case, for the communication among servers a different protocol is used that allows for a bidirectional flow of subscriptions and advertisements. While in an acyclic peer-to-peer topology one common tree is used for filter and notification propagation, in a generic peer-to-peer topology for each producer the minimum spanning tree is used that connects this server with all others servers. A filter is then only forwarded by a server B if it comes from those neighbor servers being on the shortest path from the originating server to B.

There exists no precise specification of the semantics of notification delivery, and the informally described semantics has several peculiarities. A notification should only be delivered to a client if the client had a matching subscription at the time the notification was published, and notifications may be delivered after cancellation of the respective subscriptions. A client that unsubscribes to a filter implicitly unsubscribes to all filters that are covered by the former filter, too. This approach burdens the client with keeping track of covering relations among the issued subscriptions. Hence, it makes clients depend on the applied routing algorithm. The benefit of this approach is that it simplifies routing because (un)subscriptions from neighbors and local clients can then be treated in the same way.

SIENA lacks support for type-checking of events. The complete freedom given to publishers to advertise and publish any event makes it harder to catch type-mismatch errors during system development. SIENA also addressed security issues [390]. Even though the idea of *event patterns* is introduced as a higher-level service, little detail is given on detection and temporal issues. Only the detection of sequences of events is discussed. The topology of the overlay network of event servers is static and must be specified at deployment time. The efficiency of the content-based routing will therefore depend on the quality of the overlay network topology.

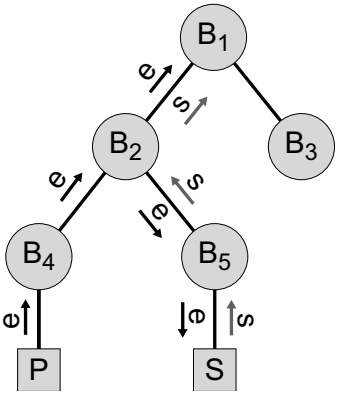

Fig. 9.10. Hierarchical event routing in JEDI

9.3.3 JEDI

The *Java Event-Based Distributed Infrastructure* (JEDI) [92] is a Java-based implementation of a distributed content-based publish/subscribe system from the Politecnico di Milano, Italy. Events in JEDI are *tuples* having a name and a list of values called *event parameters*. Subscriptions are specified as *templates* (cf. Sect 3.1.1). A JEDI system consists of *active objects*, which publish or subscribe to events, and *event dispatchers*, which route events. Event dispatchers are organized in a tree structure, and routing is performed according to hierarchical covering-based routing. Subscriptions propagate upwards in the tree, and state about them is maintained at the event dispatchers. Events also propagate upwards but follow downward branches whenever they encounter a matching subscription, as shown in Fig. 9.10. Since hierarchical routing is applied, advertisements are not used to restrict the propagation of subscriptions.

Support for Mobile Clients

The system has been extended to support mobile computing [89]. Event dispatchers support `moveOut` and `moveIn` operations that enable subscribers to disconnect and reconnect at a different dispatcher in the network. There is no single event dissemination tree for all subscriptions, but instead a tree is built dynamically as a *core-based tree* [24]. The core, called a *group leader*, has to make a global broadcast to announce its presence. A new event dispatcher, wanting to become part of the dissemination tree, directly contacts the group leader. The group leader then delegates the request to an appropriate event dispatcher in the dissemination tree, which becomes the parent of the new node. As a downside, this algorithm requires that every event dispatcher must have knowledge of all group leaders in the system.

Dynamic Reconfigurations

An approach for dynamically reconfiguring the dissemination tree is proposed by Cugola et al. [93, 308]. They focus on the reconfigurations that substitute one link by another one (Fig. 9.11). Instead of intentionally reconfigurations

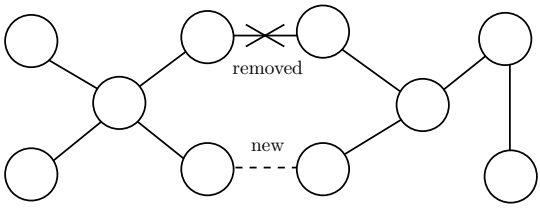

Fig. 9.11. Substituting one link with another link

(e.g., triggered by an administrator), their approach also works for reconfigurations caused by link faults. Regarding routing algorithms, they only consider simple and identity-based routing; however, they state that their algorithms could be generalized to covering-based routing. The use of advertisements is not discussed.

First, the authors describe more precisely than previous work the *strawman approach*. With this approach, both endpoints of the removed link behave as if they had received an unsubscription for each of the subscriptions of the other that are currently active. The endpoints of the added link exchange all to establish the delivery of all notifications needed at the other side. The processes of tearing down the old link and establishing the new link are carried out concurrently. As the authors explain, this has the consequence that notifications might get lost, duplicated, or reordered (violating FIFO-producer or causal ordering). The approach is also inefficient with respect to the filter forwarding overhead because subscriptions might be canceled that are shortly later reinserted, and vice versa. The strawman approach also leads to correct routing tables if multiple links are exchanged concurrently.

After discussing the strawman approach, the authors also propose a solution that (is according to their simulation results) more efficient than the strawman approach but which exhibits the same deficiencies with respect to notification loss, duplication, and reordering. With this solution, the new link is established a bounded delay (i.e., a timeout is used) before the old link is removed, i.e., subscription propagation starts earlier than unsubscription propagation. However, choosing a sensible value for the timeout seems difficult. To avoid the propagation of subscriptions that would otherwise be removed a short time later, subscriptions located at an endpoint of a removed link are removed from the routing tables of the respective brokers instantaneously and only their propagation is delayed.

More recently, the authors presented a more advanced approach to deal with reconfigurations [94] based on *reconfiguration paths* which identify the minimal portion of the system affected by a fault. This approach is better suited for controlled administration than for dealing with faults.

9.3.4 REBECA

The REBECA notification service [136] implements the publish/subscribe interface and conforms to the definition of simple event systems (cf. Sect. 2.1). Its basic architecture is a representative example of a distributed notification service, which is comparable to that of other services such as SIENA, JEDI. However, REBECA is different from other services:

Formal Specification. REBECA is based on a formal specification that defines the intended behavior of the notification unambiguously.

Extensible Data and Filter Model. The default data model of REBECA is the name/value pair model. However, the set of datatypes and constraints that can be used is not fixed but extensible.

Extensible Routing Framework. REBECA is designed to support various routing algorithms [263, 267]. Peer-to-peer and hierarchical variants of the algorithms as well as advertisements can be used.

Visibility Control. With REBECA it is possible to control the visibility of notifications [139, 144] by using the scopes.

Architecture

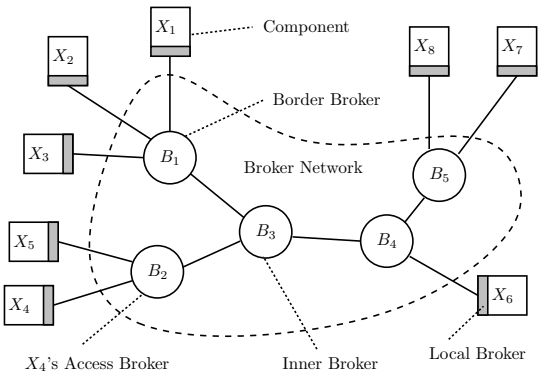

Fig. 9.12. An exemplary router network of REBECA

The constituents of the system are the components (i.e., producers and consumers)and the notification service (Fig. 9.12). The notification service

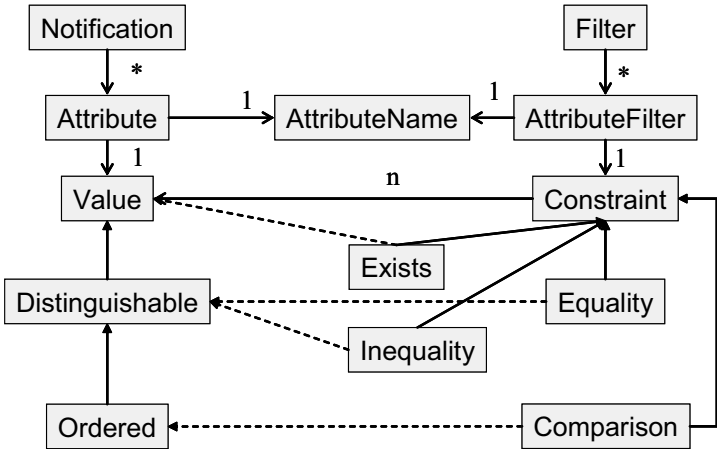

Fig. 9.13. The filtering framework of REBECA

consists of a number of brokers that form an overlay network in the underlying physical network. Brokers are processes that run on physical nodes. The communication topology of the overlay is an acyclic graph. Edges are communication links that are mapped to TCP/IP connections. As an alternative, IP multicast can be used. Obviously, an acyclic topology is can become a bottleneck, but extensions exploiting redundancy are available to tackle problems of scalability and single points of failure [86, 311, 374].

REBECA distinguishes three types of brokers: local, border, and inner brokers. *Local brokers* provide access to the middleware by offering the publish/subscribe interface to the components. Usually, they are part of the communication library loaded into application components; they are not represented in the graph, but only used for implementation issues. A local broker is connected to one border broker. *Border brokers* form the boundary of the distributed communication middleware and maintain connections to local brokers, i.e., the clients of the service. *Inner brokers* are connected to other inner or border brokers; they do not maintain connections to clients.

Local brokers put the first message containing a newly published notification into the network. Border and inner brokers forward the messages to neighbor brokers according to filter-based routing tables and respective routing strategies. At the end, the messages are sent to the local brokers of the consumers, and from there the notifications are delivered to the application components.

Extensible Data and Filter Model

In the default data model of REBECA, a notification consists of a set of attributes that are name/value pairs. Attribute values can be of different types,

including the usual primitive types such as integers, strings, Booleans, and floats but also composite types such as points or rectangles. It is possible to add new datatypes to the filtering framework (Fig. 9.13) easily. New data types should support the operations which are needed by the applied routing algorithms such that routing optimizations become possible. The set of constraints that can be imposed on attributes contains the usual operator such as equality, inequality, and comparisons. It can be extended by new constraints. For more details regarding the data and filter model of REBECA please refer to Chap. 3.

Extensible Routing Framework

REBECA is based on a flexible routing framework which allows new routing algorithms to be added easily. If a new algorithm is added, it can be used for subscriptions and for advertisement propagation. It can also be combined with other routing algorithms in the sense that, for example, the new algorithm is used for subscription forwarding and a previously existing algorithm is used for advertisement forwarding. In contrast to, for example, SIENA, the publish/subscribe interface used by components is independent of the applied routing algorithm. Thus, applications need not to be changed if a new routing algorithm is applied.

Currently, REBECA supports flooding, simple, identify-based, covering-based, and merging-based routing (cf. Sect. 4.5). The implementation of the routing algorithms closely follows the pseudocode we have presented and so we can place high confidence on the correctness of the implementation. The following combinations of routing algorithms are possible: If only subscriptions are used, any of the four filter-based routing algorithms can be applied. If advertisements are used, for subscription forwarding and for advertisement forwarding one of the filter-based routing algorithms can be used, resulting in ten possible combinations. The use of advertisements can greatly enhance the efficiency of the system if certain kinds of notifications can only be produced in certain parts of the broker network. In this case, the size of the subscription routing tables and the filter forwarding overhead is reduced. In the hierarchical setting, again any of the four filter-based routing algorithms can be used. Together with flooding, this results in altogether 19 different combinations of routing algorithms. Flooding can only be combined with a filter-based routing algorithm in a hybrid routing scheme. In this case, in a subtopology notifications are flooded and filters are only forwarded to the root broker of this subtopology. For more details regarding the routing framework please refer to Chap. 4.

Visibility Control

In large-scale publish/subscribe systems, the ability to control the visibility of notifications is a crucial feature. If a notification should not be visible

in some part of the system, then it is also not necessary to distribute the notification into this part. The visibility of events can be controlled with scopes that facilitate information hiding. Together with input and output interfaces this points the way toward event-based components. Event mapping can be used to transform notifications from one representation to another, which is a necessity in heterogeneous systems. For more details regarding scopes please refer to Chap. 6.

Available Prototypes

Two prototypes have emerged and are available: a Java-based prototype and a prototype based on Microsoft's .NET platform. We are implementing a bridge between the two prototypes to make them interoperable. Other developers in the REBECA project are currently implementing the scoping concept [138, 140, 144] that allows the visibility of notifications to be constrained using a scope graph. Histories supporting caching of past notifications [81] and that support client and broker mobility [141, 142, 408] as well as P2P-based routing [374] are also part of current implementation and research efforts.

9.3.5 Hermes

Another research prototype is HERMES [310], a distributed, event-based middleware platform. HERMES is aimed at a generic class of large-scale data dissemination applications, such as Internet-wide news distribution and a sensor-rich, active building. It follows a type- and attribute-based publish/subscribe model that places particular emphasis on programming language integration by supporting type-checking of event data and event type inheritance.

To handle dynamic, large-scale environments, HERMES uses peer-to-peer techniques for autonomic management of its overlay network of event brokers and for scalable event dissemination. It is based on an implementation of a peer-to-peer routing layer to create a self-managed overlay network of event brokers for routing events. Its content-based routing algorithm is scalable because it does not require global state to be established at all event brokers. Its routing algorithms use rendezvous nodes, as explained in Sec. 4.6.3, to reduce routing state in the system, and include fault tolerance features for repairing event dissemination trees. HERMES is also resilient against failure through the automatic adaptation of the overlay broker network and the routing state at event brokers. An emphasis is put on the middleware aspects of HERMES so that its typed events support a tight integration with an application programming language.

A primary feature of the HERMES event-based middleware is scalability. HERMES includes two content-based routing algorithms to disseminate events from event publishers to subscribers. The *type-based routing algorithm* only supports subscriptions depending on the event type of event publications. It is comparable to a topic-based publish/subscribe service but differs by

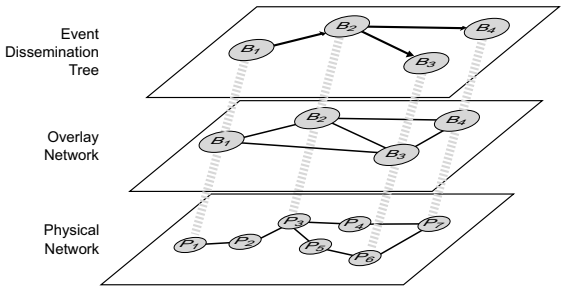

Fig. 9.14. Layered networks in HERMES

observing inheritance relationships between event types. The second algorithm is *type- and attribute-based routing*, which extends type-based routing with content-based filtering on event attributes in publications. In both algorithms, event-type specific advertisements are sent by publisher-hosting brokers to set up routing state. Advertisements are not broadcast to all event brokers, but instead event brokers can act as special rendezvous nodes that guarantee that event subscriptions and advertisements join in the network in order to form valid event dissemination trees.

System Model

Both routing algorithms use a distributed hash table to set up state for event dissemination trees. The distributed hash table functionality is implemented by a peer-to-peer routing substrate, called PAN, formed by the event brokers in HERMES. PAN is an extended implementation of the Pastry routing algorithm. The advantage of such peer-to-peer overlay networks are threefold: first, the overlay network can react to failure by changing its topology and thus adding fault tolerance to HERMES. Second, the peer-to-peer routing substrate that manages the overlay network is responsible for handling membership of event brokers in a HERMES deployment. Third, the discovery of rendezvous nodes, which must be well-known in the network, is simplified by the standard properties of the distributed hash table.

The three layers of networks in HERMES are illustrated in Fig. 9.14. The bottom layer is the physical network with routers and links that HERMES is deployed in. The middle layer constitutes the peer-to-peer overlay network that offers a distributed hash table abstraction. The top layer consists of multiple event dissemination trees that are constructed by HERMES to realize the event-based middleware service. When a message is routed using the peer-to-peer overlay network, a callback to the upper layer is performed at every hop, which allows the event broker to process the message by altering it or its own state.

In addition to scalable event dissemination, HERMES supports event typing, the creation of event type hierarchies through inheritance, and generic, su-

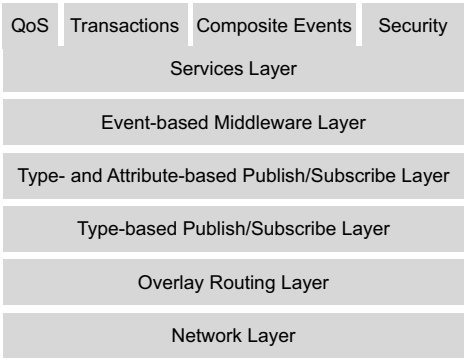

Fig. 9.15. Overview of the HERMES architecture

pertype event subscriptions. This enhances its integration with current object-oriented programming languages such as Java or C++.

Architecture

As shown in Fig. 9.15, the architecture of HERMES has six layers. Each layer builds on top of the functionality provided by the layer underneath and exports a clearly defined interface to the layer above. Apart from that, the layers are independent of each other. A layered architecture for a communications system has the advantage that each layer can have its implementation easily replaced by a different implementation if necessary. For example, if a more efficient implementation of a distributed hash table becomes available, HERMES can benefit from this without major modification. Since HERMES is implemented by the event brokers, its layered structure is also reflected in the implementation of an event broker. Next, we describe the role of each layer, starting with the lowest one.

Network Layer. The lowest layer is the network layer that represents the unicast communication service of the underlying physical network. This assumes that HERMES is deployed in a network with full unicast connectivity between nodes, such as the Internet. No other network-level services, such as group communication primitives, are necessary.

Overlay Routing Layer. This layer implements an application-level routing algorithm that provides the abstraction of a distributed hash table. A peer-to-peer implementation of this layer is chosen for reasons of scalability and robustness. It takes application-level nodes, which are HERMES event brokers, and creates routing state in order to hash keys to nodes. It also handles the addition, removal, and failure of nodes in the overlay network. The topology of the overlay routing layer is optimized with respect to a proximity metric of the underlying physical network.

Type-Based Publish/Subscribe Layer. This layer exports a primitive type-based publish/subscribe service on top of the distributed hash table established by the previous layer. Type-based routing supports subscriptions according to an event type and observes the inheritance relationships between event types. Event dissemination trees are then created with the help of rendezvous nodes in the system. Trees are also repaired by retransmitting messages after state at event brokers has been lost.

Type- and Attribute-Based Publish/Subscribe Layer. This layer extends the type-based service with content-based filtering on event attributes. The same rendezvous node mechanism is used for the construction of event dissemination trees. However, the trees are annotated with filtering expressions derived from the type- and attribute-based subscriptions. These filtering expressions are placed at strategic locations in the network, usually as close to event producers as possible in order to discard unnecessary events as early as possible.

Event-Based Middleware Layer. At this layer, event-based middleware functionality is added to the content-based publish/subscribe system of the previous layers. Typing information is maintained by the rendezvous nodes so that event publications and subscriptions can be type-checked automatically by HERMES. The event-based middleware layer also extends the API used by event clients to invoke HERMES.

Services Layer. The services layer is a set of pluggable extensions to the event-based middleware layer. It allows the HERMES middleware to provide a wide range of higher-level middleware services. For example, different guarantees of publication and subscription semantics can be supported by a QoS module at the services layer. Another service may deal with composite event detection or transaction support. Services may violate the strict layering of the architecture and obtain direct access to lower layers if this is necessary for their functionality.

9.3.6 Cambridge Event Architecture (CEA)

The *Cambridge Event Architecture* (CEA) [18, 20] was created in the early 1990s to address the emerging need for asynchronous communication in multimedia and sensor-rich applications. It introduced the *publish–register–notify* paradigm for building distributed applications. This design paradigm allows the simple extension of synchronous request/reply middleware, such as CORBA, with asynchronous publish/subscribe communication. Middleware clients that become *event sources* (publishers) or *event sinks* (subscribers) are standard middleware objects.

The interaction between an event source and sink is illustrated in Fig. 9.16. First, an event source has to advertise the events that it produces, for example, in a name service. In addition to regular methods in its synchronous interface, an event source has a special `register` method so that event sinks can subscribe (*register*) to events produced by this source. Finally, the event source

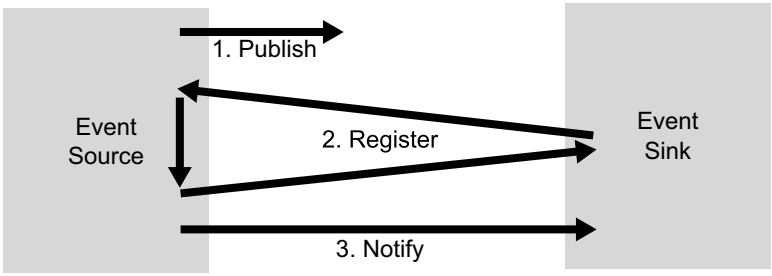

Fig. 9.16. The publish–register–notify paradigm in the CEA

performs an asynchronous callback to the event sink's `notify` method (*notify*) according to a previous subscription. Note that event filtering happens at the event sources, thus reducing communication overhead. The drawback of this is that the implementation of an event source becomes more complex since it has to handle event filtering. Another drawback is that the transmission of notifications to multiple consumers are independent unicast communications.

Direct communication between event sources and sinks causes a tight coupling between clients. To address this, the CEA includes *event mediators*, which can decouple event sources from sinks by implementing both the source and sink interfaces, acting as a buffer between them. Chaining of event mediators is supported, but general content-based routing, as done by other distributed publish/subscribe systems, is not part of the architecture. More recent work [192] investigates the federation of separate CEA event domains using contracts that are enforced by special mediators acting as gateways between domains. A Java implementation of the CEA, *Herald* [346], supports storage of events.

The design goal of the CEA is to seamlessly integrate publish/subscribe with standard middleware technology. Therefore, events are strongly typed objects of a particular event class and are statically type-checked at compile time. Initially, subscriptions were template-based for equality matching only, but they were then extended with a predicate-based language withname/value pairs. These subscriptions are type-checked dynamically at runtime. Furthermore, the CEA provides a service for complex subscriptions based on composite event patterns [189]. This is an important requirement for an event-based middleware. We presented our approach for detecting composite events in Chap. 7.

COBEA

The CEA was implemented on top of CORBA in the CORBA-*based event architecture* (COBEA) [244]. Events are passed between event sources and sinks as parameters in CORBA method calls. Event clients can by typed or untyped: a

typed client encodes the structure of an event type in an IDL `struct` datatype, whereas an untyped client uses the generic `any` datatype. Type-checking for typed clients is done by the IDL compiler. The subscription language consists of a conjunction of predicates over the attributes defined in the event type.

ODL-COBEA

The use of CORBA IDL to express event types is cumbersome since its original purpose is the specification of interfaces for remote method calls. In [309], COBEA is extended with an event type compiler that transforms event type definitions in the *Object Definition Language* (ODL) [72] into appropriate CORBA IDL interfaces. ODL is a schema language defined by the Object Data Management Group (ODMG). With ODL, objects can be described language-independently for storage in an object-oriented database. The advantage of using ODL for event definitions is that it provides support for persistent events because it unifies the mechanisms for transmission and storage of events [19].

An example of an ODL-defined event type, as it would be used in the Active Office application scenario, is given in Fig. 9.17. Event types consist of a set of typed attributes and form an ODL inheritance hierarchy, in which all types are derived from the `BaseEvent` ancestor class. The `BaseEvent` type has attributes that all event types inherit, namely a unique `id` field, a `priority` field, a `source` field with the name of the event source that generated this event, and a `timestamp`. ODL-COBEA is aware of inheritance relationships between event types and supports *supertype subscriptions*. When an event subscriber subscribes to an event type, it will also receive any published events that are of a subtype of the type specified in the subscription. This means that an event subscriber that subscribes to the `BaseEvent` type will consequently receive all events published at a given event source.

The CEA and in particular the ODL-COBEA implementation recognize the importance of type-checking for events in a publish/subscribe system. The object-oriented approach for defining event types cleanly integrates with current object-oriented programming languages and middleware architectures. Static type-checking, as done by an event type compiler, does not introduce a runtime cost, but it tightly couples event sinks to sources.

The main disadvantage of the CEA is the lack of content-based event routing between event mediators. This limits the scalability of the architecture as it forces a subscriber to know the publisher (or mediator) that offers

```
1  class LocationEvent extends BaseEvent {
2    attribute short id;
3    attribute string location;
4    attribute long lastSighting;
5  };
```

Fig. 9.17. An ODL definition of event types in ODL-COBEA

a particular event type. In addition, it makes the implementation of event sources challenging because they are required to perform event filtering depending on subscriptions. Several distributed content-based publish/subscribe systems were proposed after the CEA to address these problems.

9.3.7 Elvin

Elvin [341] is a notification service for application integration and distributed systems monitoring developed by the Distributed Systems Technology Centre in Australia. It features a security framework, internationalization, and pluggable transport protocols, and has been extended to provide content-based routing of events [340]. Events are name/value pairs with a predicate-based subscription language. An interesting feature of Elvin is a *source quenching mechanism*, where event publishers can request information from event brokers about the subscribers currently interested in their events. This enables publishers to stop publishing events when there are no subscriptions, reducing computation and communication overheads.

Clients for a wide range of programming languages are available, which led to the implementation of many notification applications. Applications, such as a ticker-tape, were evaluated as means for collaboration in a pervasive office environment [148]. Other work investigates event correlation and support for disconnected operation in mobile applications [368].

9.3.8 READY

The READY event notification service [184] introduced *event zones* to partition components based on logical, administrative, or geographical boundaries and to delimit the visibility of events. Boundary brokers connect zones and control the communication between them, and may enforce security policies on connected clients. Although similar to scoping, zones resemble more the domain idea of CORBA as it mainly addresses control on the physical routing network; the engineering aspect is lacking. For instance, in READY a component belongs to exactly one zone so that there is only a two-level hierarchy. The system is structured only based on one specific point of view, prohibiting composition and mixing of aspects [188]. Heterogeneity issues are only mentioned in READY: boundary brokers could apply transformations on crossing notifications. Following the idea of CORBA domains, brokers operate here on a rather coarse and static granularity, whereas event mappings (Sect. 6.4) allow for syntactic and semantic mappings in the formal model and at every layer of abstraction in a scoped system.

9.3.9 Narada Brokering

The *Narada Brokering* project [293] aims to provide a unified messaging environment for grid computing, which integrates grid services, JMS, and JXTA.

It is JMS compliant (Sect. 9.1.3), but also supports a distributed network of brokers as opposed to the centralized client/server solution advocated by JMS. The JXTA specification [180] is used for peer-to-peer interactions between clients and brokers.

Events can be XML messages that are matched against XPath [398] subscriptions by an XML matching engine. The network of brokers is hierarchical, built recursively out of clusters of brokers. Every broker has complete knowledge of the topology, so that events can be routed on shortest paths following the broker hierarchy. In general, there is the additional overhead of keeping event brokers organized hierarchically, which can be costly. Dynamic changes of the topology are propagated to all affected brokers.