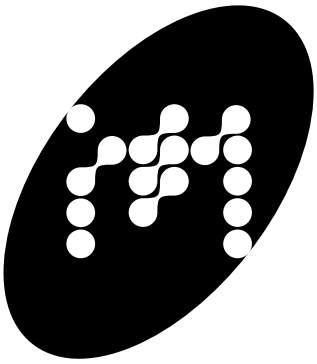

MINEMA

State of the Art Review of Distributed Event Systems

Sasu Tarkoma and Kimmo Raatikainen

Helsinki University Computer Science Department
Helsinki Institute for Information Technology

Abstract

This document presents an overview of event systems and distributed event frameworks with an emphasis on the special requirements presented by mobile computing. An event-based framework can be decomposed into two essential parts: event detection and event notification. The former deals with the detection of the occurrence of a particular event of interest, and the latter is the act of notifying interested parties that an event has occurred.

Document Identifier	C0-04
Document Status	Final
Created	8 November 2005
Revised	19 February 2006
Distribution	Public

Contents

1 Introduction

This document presents an overview of event systems and distributed event frameworks [EFG03] with an emphasis on the special requirements presented by mobile computing. By mobile or ubiquitous computing we mean the new field of research created by wireless communication and the introduction of small, mobile devices. Traditionally, event-based systems are based on a number of event sources and event sinks. Sources produce events and they are delivered to event sinks that have a priori registered to receive them. An event-based framework can be decomposed into two essential parts:

- Event detection, which deals with the detection of the occurrence of a particular event of interest.
- Event notification, which is the act of notifying interested parties that an event has occurred.

Many existing platforms employ the synchronous model of method invocation, in which operations are performed on passive objects. This model is insufficient for reactive environments, where components need to react to changes, or events, within the system and give timely responses. An option would be to poll the states of objects, but too-frequent polling burdens the system and too-infrequent polling delays the communication [BMH+00]. Asynchronous events support different application types as identified by [BMH+00]:

- Group interaction
- Multimedia support (multimedia control through rules)
- Mobility
- Alarms and exceptions
- Management

Reliable and efficient asynchronous event detection and event notification are vital for the development of the next-generation distributed software for mobile Internet-aware devices. Event frameworks provide a plug-and-play architecture for creating distributed applications. Distributed architectures are based on middleware that provides the interoperability layer required for heterogeneous cross operating system and cross-language operation and communication. Components from different systems and different manufacturers can interoperate using middleware such as CORBA, where the interface definitions created using IDL (Interface Definition Language) may be shared.

Currently middleware solutions, such as Java, from the desktop world are being introduced into the wireless world, where the requirements are different. Small and wireless devices have limited capabilities compared to desktop systems: their memory, performance, battery life, and connectivity are limited and constrained. The requirements of mobile computing need to be taken into account when designing an event framework that integrates with mobile devices. From the mobility and wireless viewpoint event systems can be divided into three distinct categories:

1. Traditional event systems designed for fixed network operation.
2. Event systems that support intermittent clients using a client-server protocol and possibly roaming between access nodes.
3. Ad-hoc networks, where clients can also be servers and servers may roam.

The first category is the most researched and most of the architectures presented in this document fall into this category. Several architectures support intermittent clients and roaming between access nodes. Ad-hoc event architectures are currently emerging.

From the small device point of view, message queuing is a frequently used communication method because it supports disconnected operation. When a client is disconnected, messages are inserted into a queue, and when a client reconnects the messages are sent. The distinction between popular message-queue-based middleware and notification systems is that message-queue-based approaches are a form of directed communication, where the producers explicitly define the recipients. The recipients may be defined by the queue name or a channel name, and the messages are inserted into a named queue, from which the recipient extracts messages. Notification-based systems extend this model by adding an entity, the event service or event dispatcher, that brokers notifications between producers of information and subscribers of information. This undirected communication supported by the notification model is based on message passing and retains the benefits of message queuing. In undirected communication the publisher does not necessarily know which parties receive the notification.

This also applies to message-oriented middleware such as JMS [Sun01] that supports publish-subscribe type of communication [SAS01]. Undirected communication decouples producers and consumers from each other. In addition, many systems support filtering and pattern detection that are used to reduce the amount of transmitted information and to improve the accuracy of notifications. Content-based routing is flexible because it does not require configuration information pertaining to channel names. Undirected communication may also be used to deliver the same set of information to a number of client devices. However, this requires associating user subscription information with a set of devices [SAS01, CDN01].

This document is structured as follows: chapter 2 introduces event models, and event routing. Chapter 3 presents event standards and specifications such as the CORBA Notification Service and DSS, JMS, and the COM+ and .NET event models. Chapter 4 presents research prototypes and examines the support for disconnected operation and mobility in each of the presented event systems. Chapter 5 examines recent research areas in pub/sub, namely mobility support, dynamic and peer-to-peer systems, formal modelling of these systems, and security issues. Finally, chapter 6 presents the conclusions.

2 Event Models

Event models consist of event sources, event listeners, notification services, filtering services, and event storage and buffering services. In addition, there may be one or more authentication schemes to enforce security and access control. This section focuses on the general definition of events and event models.

2.1 Events

An event represents any discrete state transition that has occurred and is signaled from one entity to a number of other entities. For example, a successful login to a service, the firing of detection or monitoring hardware and the detection of a missile in a tactical system are all events. The firing of each event is either deterministic or probabilistic. A source can generate a signal every second making it deterministic. A stochastic source follows some probabilistic model that can be described using, for example, a Markov chain. Both event qualities can be modeled by building statistical or stochastic models of the firing behavior of the event source. For example, a correlation analysis can be made between a series of event occurrences in time or between two event sources. Such an analysis would measure how strongly one event implies the other or how two event source firings are related.

Events may be categorized by their attributes, such as which physical property they are related to. For instance spatial events and temporal events note physical activity. Moreover, an event may be a combination of these, for example an event that contains both temporal and spatial information.
Events can be categorized into taxonomies on their type and complexity. More complex events, called compound events, can be built out of more specific simple events. Compound events are important in many applications. For example, a compound event may be fired

- in a hospital, when the reading of a sensor attached to a patient exceeds a given threshold and a new drug has been administered in a given time interval,
- in a location tracking service, where a set of users are in the same room or near the same location at the same time, or
- in an office building, where a motion detector fires and there has been a certain interval of time after the last security round.

Event-based interaction can be:

- discrete or
- continuous, as event streams.

Events can also have different prioritizations. Event aging assigns an expiry time to each event notification. Event expiration prevents the spreading of obsolete information.

2.2 Event Model

The standard client/server communication models in distributed object computing are based on synchronous method invocations. For example, COM+, Java RMI, and CORBA use synchronous calls (CORBA 3.0 supports asynchronous invocations). This approach has several limitations [GCSO01]:

- Tight coupling of client and server lifetimes. The server must be available to process a request. If a request fails the client receives an exception.
- Synchronous communication. A client must wait until the server finishes processing and returns the results. The client must be connected for the duration of the invocation.

- Point-to-point communication. Invocation is typically targeted at a single object on a particular server.

Mobile clients and large distributed systems motivate the use of asynchronous and anonymous one-to-many distributed computing models. Event-based models address the limitations of the standard client/server paradigm by introducing two roles: consumers and producers. Since event models employ differing technical terms, in this chapter we consider event consumers, listeners, sinks, and respectively event producers, sources, and suppliers to be synonymous.

2.3 Event Routing

The event model consists of event listeners and event sources. A listener expresses interest in an event supported by an event source and registers to receive notifications of that event based on a set of parameters. Figure 1 presents a general model of the listener-source paradigm, where the actual filtering and notification are treated as a black box, which can reside either on the source or on the network. Ideally, the event source does not have knowledge of all the parties that are interested in a particular event.

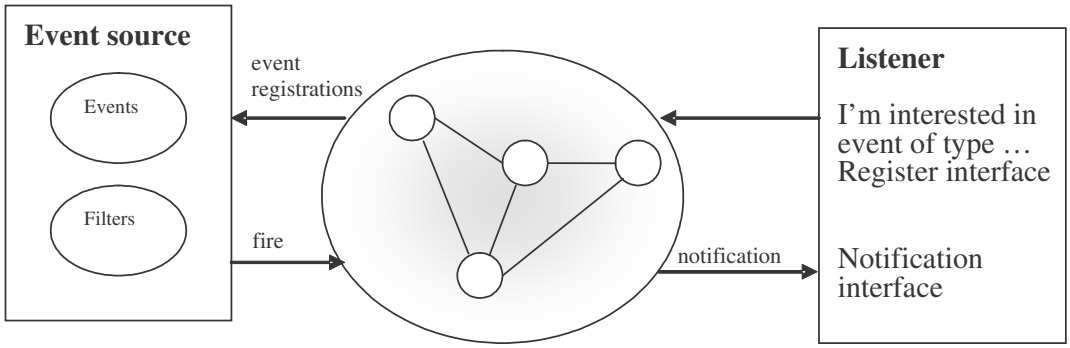

Figure 1. General model of the event source and event listener.

Event source fires events, and the listener is notified using some mechanism on the network or in the client. The event system is a logically centralized component that may be a single server or a number of federated servers. In a distributed system consisting of many servers, there are two approaches for connecting sources and listeners:

- The event service supports subscription of events, and it routes registration messages to appropriate servers (for example, using a minimum spanning tree). One optimization to this approach is to use advertisements, messages that indicate the intention of an event source to offer a certain type of event, to optimize event routing.
- Some other means of binding the components is used, for example, a lookup service.

In this context, by event listener we mean an external entity that is located on a physically different node on the network. However, events are also a powerful method to enable inter-thread and local communication, and there may be a number of local event listeners that wait for local events.

Event routing requires that store-and-forward type of event communication is supported within the network on the access nodes (or servers). This calls for intermediate components called event routers. Each event source is connected to at least one router. Each router needs to know a suitable subset of other routers in the domain. In this approach the request, in the

worst case, is introduced at every router to get a full coverage of all message listeners. This is not scalable, and the routing needs to be constrained by locality or by hop count. Effective strategies to limit event propagation are zones used in the ECO architecture, the tree topology used in JEDI, or the four server configurations addressed in the Siena architecture. Siena broadcasts advertisements throughout the event system; subscriptions are routed using the reverse-path of advertisements, and notifications are routed on the reverse-path of subscriptions. IP Multicast is also a frequently used network-level technology for disseminating information and works well in closed networks, however, in large public networks multicast or broadcast may not be practical. In these environments universally adopted standards such as TCP/IP and HTTP may be better choices for all communication [IBM02a].

2.4 Filtering

Filtering reduces the number of events sent from the sources to the listeners by matching events against a template. Those events that match the template are forwarded to the listeners. Matching is usually done on single events, but may be also performed on compound events. Filtering improves the scalability of the system. Also, the location of the filtering of events affects the scalability of the framework. Here we face two separate issues: the filtering of simple events and the filtering of compound events.

Both kinds of event filtering can be done at several locations:

- At a centralized server (client-server).
- At the listener.
- At the event source.
- In the infrastructure (event routers).

Source-side filtering is more scalable than a centralized server or filtering at the listener. Schemes that use multicasting and listener-side filtering place the burden on listeners and the communication infrastructure. Filtering pertains to routing when it is done by intermediate brokers connected into a routing topology.

2.5 Content-based Routing

Events are published in a named channel, or in an infrastructure of one or more routers that can use the content of the events in making the forwarding decision. Named channels are also called topics, and they represent an abstraction of numeric network addressing mechanisms. With content-based addressing clients can change their interests without changing the addressing scheme. With channel-based messaging, new channels need to be added to the address space.

Content-based	The routing decision is made based on the content, for example strongly typed fields in the event message.
Subject-based	The routing decision is made based on the subject of the event.
Channel-based (or topic-based)	The routing decision is made based on the channel on which the event is published. A channel is a discrete communication line with a name.

The producers and consumers must agree on a channel. Content-based and subject-based are more flexible than channel-based messaging, because this agreement is not necessary. Channel-based messaging, however, allows the use of IP multicast groups. The subjects can

be allocated to multicast addresses. Channel-based routing can be emulated with content-based systems by limiting to a universally defined subject field. Content-based event routing has been proposed as one of the requirements for advanced applications, in particular for mobile users [CW01].

Content-based routing takes place above the network level (level 3) and can be based on e.g. IP multicast networks. In the content information model, the users subscribe to information based on their preferences. The information, when it is available, is then delivered based on these preferences. The subscription paradigm abstracts the publishers of information from the receivers: information is not published to a set of addresses. Work has been done in using multicast networks to deliver the information to the subscribers [CW01] using multicast addresses. The granularity and flexibility of this approach depends on the size and number of the virtual multicast addresses. As an alternative Carzaniga and Wolf present application-level information broker with a rich information selection capability. They define a content-based addressing scheme by considering the predicates that define subscriptions as the destination addresses. Datagrams are implicitly addressed to a node by their content. The predicate model is a set of boolean functions imposed on the datagram model. Content-based routing is done using an algorithm that uses a forwarding table, which is a map of interfaces to their receiver predicates. Content-based systems are contrasted with channel-based and subject-based systems, because the selection is done based on the whole content. The other strategies offer only a set of well-defined attributes for selection purposes. The drawback of content-based systems is scalability.

2.6 Quality of Service

Applications based on event-style communication have varying reliability requirements. The event system may support semantics ranging from "at-most-once" to "exactly-once". In addition, there may be availability, performance, scalability, and throughput requirements. The diverse nature of requirements calls for a number of implementations optimized for different sets of requirements.

2.7 A Taxonomy

Event models can be grouped into a taxonomy by their properties. As contrasted with the client-server paradigm, event models involve one-to-many communication. Other important aspects for event model classification are [Mei00]:

- Does the model support distributed operation, local operation, or both? In a centralized event model the event sources and listeners are located on the same host, whereas in the distributed model they can be located on different hosts.
- Support for detecting composite events (compound events). Compound events require more complicated filtering and history mechanisms.
- Support for Quality of Service requirements, for example, delivery semantics (best-effort, at-most-once, …).
- Support for typed events, generic events, or both. Typed events have a well-defined structure, for example a set of ordered strings, and generic events do not have an expressive structure (datatype any).
- How decoupled the event listeners are from the event sources?
- Is the model subscription-based or advertisement-based?
- Support for channel-based, subject-based, or content-based routing.

Additional aspects are:

- Support for wireless systems and disconnected operation.
- Does the model support event routing, direct notification, etc.?
- How are interests defined and discovered? Not all models include discovery functionality.

Figure 2 and Figure 3 present an example taxonomy based on the event architectures explored in this review.

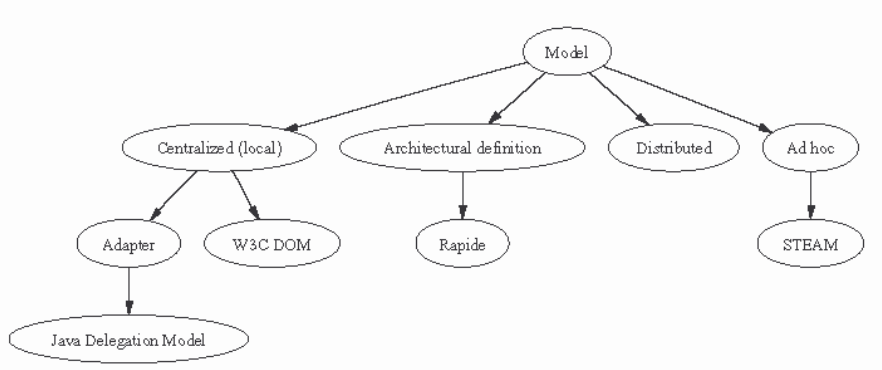

Figure 2. Example event model taxonomy.

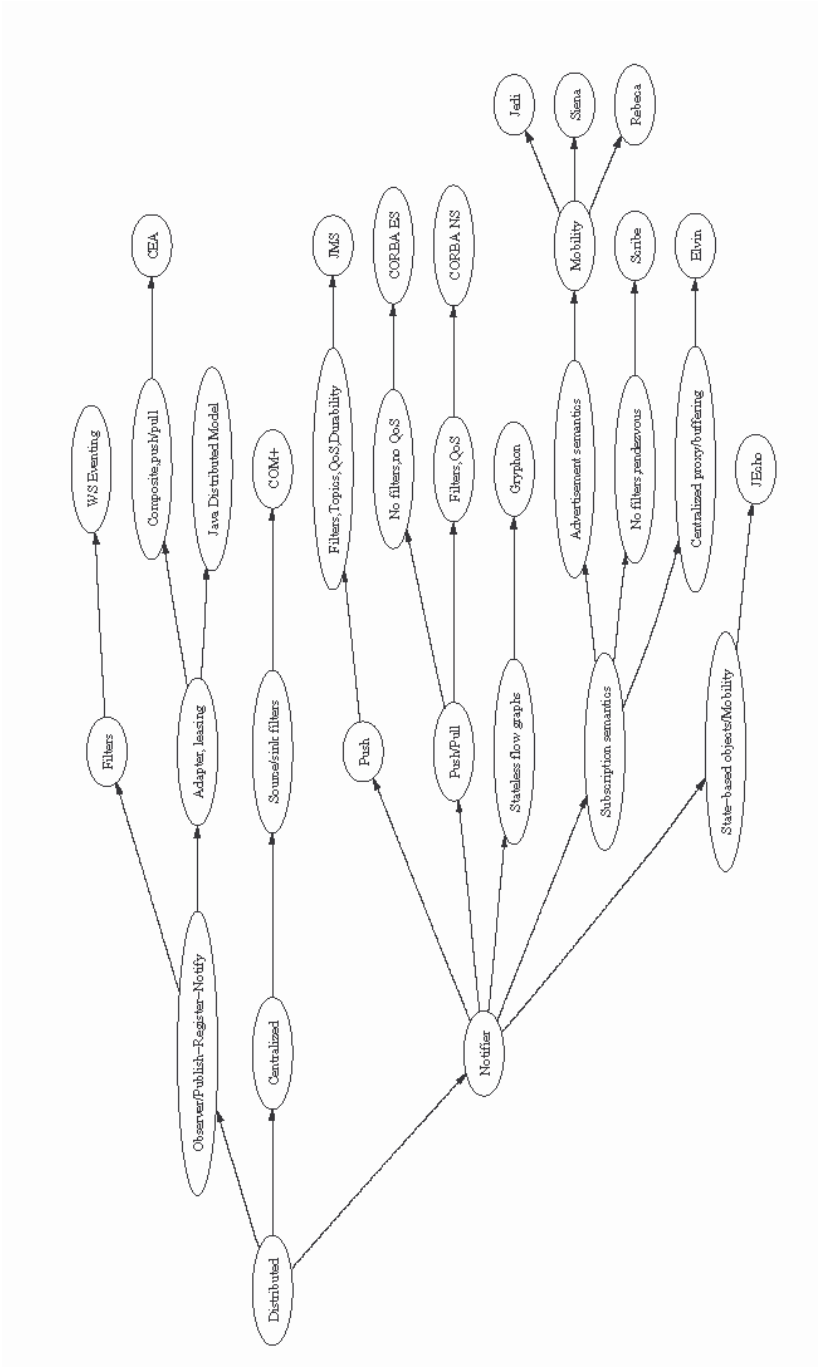

Figure 3. Distributed event systems in the taxonomy.

3 Event Standards and Specifications

This chapter presents event standards and specifications. We start from the standard centralized event model in Java, and continue with the Distributed Event Model in Java. We present the Java Messaging Service in section 3.3, section 3.4 presents the CORBA Event Service and section 3.5 the Notification Service. In section 3.6 we examine the CORBA Management of Event Domains. We also examine OMG Data Distribution Service, W3C DOM events, Web Services events, COM+ and .NET, and the WebSphere MQ architecture.

3.1 Java Delegation Event Model

The Java Delegation Event Model was introduced in the Java 1.1 Abstract Windowing Toolkit (AWT) and serves as the standard event processing method in Java. The model is also used in the Java Beans architecture and supported in the PersonalJava and EmbeddedJava environments. In essence, the model is centralized and a listener can register with an event source to receive events. An event source is typically a GUI element and fires events of certain types, which are propagated to the listeners. Event delivery is synchronous, so the event source actually executes code in the listener's event handler. No guarantees are made on the delivery order of the events [Mei00]. The event source and event listener are not anonymous, however, the model provides an abstraction called an adapter, which acts as a mediator between these two actors. The adapter decouples the source from the listener and supports the definition of additional behavior in event processing. The adapter may implement filters, queuing, and QoS controlling.

3.2 Java Distributed Event Model

The Distributed Event Model of Java is based on Java Remote Method Invocation (RMI) that enables the invocation of methods in remote objects. This model is used in Sun's Jini architecture. The architecture of the Distributed Event Model is similar to the architecture of the Delegation Model with some differences. The model is based on the Remote Event Listener, which is an event consumer that registers to receive certain types of events in other objects. The specification provides an example of an interest registration interface, but does not specify such.

The Remote Event is the event object that is returned from an event source (generator) to a remote listener. Remote events contain information about the occurred event, a reference to the event generator, a handback object that was supplied by the listener, and a unique sequence number to distinguish the event globally. The model supports temporal event registrations with the notion of a lease (Distributed Leasing Specification). The event generators inform the listeners by calling the listeners' notify method.

The specification supports Distributed Event Adaptors that may be used to implement various QoS policies and filtering. The handback object is the only attribute of the Remote Event that may grow to unbounded size. It is a serialized object that the caller provides to the event source; the programmer may set the field to null. Since the handback object carries both state and behavior it can be used in many ways, for example to implement an event filter at a more powerful host than the event source.

A mediator component can register to receive events and give a filter object to the source. Upon event notification, the filter is handed back and the mediator can use it to filter the event before handing it to the original event listener. The specification supports recovery from

listener failures by the notion of leasing. A lease imposes a timeout for event registrations. This is used to ease the implementation of distributed garbage collection. Since this model relies on RMI, it is inherently synchronous. Each notification contains a sequence number that is guaranteed to be strictly increasing.

3.3 Java Message Service

Java Message Service (JMS) [Sun01] defines a generic and standard API for the implementation of message-oriented middleware. The JMS API is an integral part of the Java Enterprise Edition (J2EE) version 1.3. The J2EE supports the message-driven bean, a new kind of bean that enables the consumption of messages. However, JMS is an interface and the specification does not provide any concrete implementation of a messaging engine. The fact that JMS does not define the messaging engine or the message transport gives rise to many possible implementations and ways to configure JMS.

JMS supports a point-to-point (queues) model and a publisher/subscriber (topics) model. In the point-to-point model only one receiver is selected to receive a message, and in the publisher/subscriber model many can receive the same message. The JMS API can ensure that a message is delivered only once. At lower levels of reliability an application may miss messages or receive duplicate messages. A standalone JMS provider (implementation) has to support either point-to-point or the publish/subscribe approach, or both. Normally, JMS queues and topics are maintained and created by the administration rather than application programs. Therefore the destinations are seen as long lasting. The JMS API also allows the creation of temporary destinations that last only for the duration of the connection.

The point-to-point communication model consists of receivers, senders, and message queues. Each message queue is addressed to a particular queue, and receivers extract messages from the queues. Each message has only one consumer and the client acknowledges the successful delivery of a message to the component that manages the queue. In this model there are no timing dependencies between a sender and a receiver; it is enough that the queue exists. In addition, the JMS API allows the grouping of outgoing messages and incoming messages and their acknowledgements to transactions. If a transaction fails, it can be rolled back. In the publish/subscribe model the clients address messages to a topic. Publishers and subscribers are anonymous, and messaging is usually one-to-many. This model has a timing dependency between consumers and producers. Consumers receive messages after their subscription has been processed. Moreover, the consumer must be active in order to receive messages.

The JMS API provides an improvement on this timing dependency by allowing clients to create durable subscriptions. Durable subscriptions introduce the buffering capability of the point-to-point model to the publish/subscribe model. Durable subscriptions can accept messages sent to clients that are not active at the time. A durable subscription can have only one active subscriber at a time. Messages are delivered to clients either synchronously or asynchronously. Synchronous messages are delivered using the receive method, which blocks until a message arrives or a timeout occurs. In order to receive asynchronous messages, the client creates a message listener, which is similar to an event listener. When a message arrives the JMS provider calls the listener's onMessage method to deliver the message. JMS clients use JNDI to look up configured JMS objects. JMS administrators configure these components using facilities specific to a provider (implementation). There are two types of administered objects in JMS: ConnectionFactories, which are used by clients to connect with a provider, and Destinations, which are used by clients to specify the destination of messages. JMS messages consist of a header with a set of header fields, properties that are optional header fields (application-specific, standard properties, provider-specific properties), and a body that can be of several types.

Message selection is supported by filtering the message header against the given criteria using an SQL grammar. A JMS message selector allows clients to define the messages they are interested in. Headers and properties need to match the client specification in order to be delivered to that client. Message selectors cannot reference values embedded in the message body. An example is "JMSType='stock' AND company='abc' AND stockvalue > 100".´JMS supports five different messages types: Map, Object, Stream, Text, and Bytes. MapMessage is a set of name/value pairs, where names are strings and values are primitive Java types. ObjectMessage is a message containing a serializable Java object. StreamMessage is a stream of sequential Java primitive values. TextMessage represents an instance using the java.lang.String class and can be used to send and receive XML messages. BytesMessage is a stream of bytes.

Typically a JMS client creates a Connection, one or more Sessions, and a number of MessageConsumers and MessageProducers. Connections are created in the stopped mode. After a connection is started (start() method) messages start arriving to the consumers associated with that connection. A MessageProducer can send messages while a Connection is stopped. A Session is a single-threaded context for consuming and producing messages. Sessions act as factories for creating MessageProducers, MessageConsumers, and temporary destinations. JMS defines that messages sent by a session to a destination must be received in the order in which they were sent.

Messages are acknowledged automatically in the transactional mode (supported by the Java Transaction API), however, if a session is not transacted there are three possible options for acknowledgement: lazy acknowledgment that tolerates duplicate messages, automatic acknowledgement, and client-side acknowledgement. In persistent mode delivery is once-and-only-once, and in non-persistent mode the semantics are at-most-once. JMS messaging proceeds in the following fashion:

1. Client obtains a Connection from a ConnectionFactory
2. Client uses the Connection to create a Session object
3. The Session is used to create MessageProducer and MessageConsumer objects, which are based on Destinations.
4. MessageProducers are used to produce messages that are delivered to destinations.
5. MessageConsumers are used to either poll or asynchronously consume (using MessageListeners) messages from producers.

The JMS API (1.0.2b) does not address load balancing, fault tolerance, error notification, administration, or security. JMS implementations are available from many vendors, such as IBM (it is supported in MQSeries), Sun Microsystems (J2EE), The ExoLab Group (OpenJMS), SoftWired (iBus//Mobile), and Oracle (8i and later). The latest JMS version is 1.1, which incorporates changes approved by a Java Community Process program Maintenance Review that closed on March 18, 2002. In JMS 1.0.2 client code must use the queue and topic interfaces, and it is impossible to reuse queue clients with topics. JMS 1.1 supports client code that works simultaneously with either the point-to-point or publish/subscribe domains. Queues and topics can be accessed through the same session and thus in the same transaction.

3.3.1 JMS and CORBA Interoperability

The communication models of JMS and CORBA are similar, however, integration is necessary in the areas of message conversion, filtering, and the incorporation of point-to-point mode, which uses queues (CORBA uses publish-subscribe). The Notification Service supports structured events defined in IDL, and JMS supports the five different message formats. OMG is working on a Notification Service / JMS Interworking document [OMG02].

The RFP dealt with mappings between message types, reconciliation between different QoS properties, the ability to maintain transactional message contexts across the services, and implementations that facilitate end-to-end messaging between the services. The submission document has been replaced with an OMG Final Adopted Specification, which is currently in the finalization phase.

The specification defines a bridge that manages and interconnects an event channel with a JMS destination. The principles behind the Bridge IDL definitions were to provide backward compatibility with the programming models of NS and JMS. The Bridge is a stateful entity that mediates messages between the two systems. Structured events are used to improve performance. The Bridge is also used to automate the connection setups between channels and destinations. A BridgeFactory object supplies Bridge objects depending on the parameters: channel, destination, type of communication (push/pull), and message type (sequence, single).

Since JMS does not support pull at the source side, this is not supported. In the implementation of PrismTech's OpenFusion [Pri01] (Figures 4 and 5), the JMS event producer is extended by a client-side library that transforms JMS messages to CORBA Notification Service structure events. JMS consumers may use push and pull, but the consumers of the Notification Service may only use one of these two approaches. JMS only allows clients to specify filters on the message properties. To keep the information filterable, this data needs to be included in the filterable body of a structured event. The JMS message interface supports three attributes that are also supported in the Notification Service:

1. DeliveryMode (persistent, non-persistent which maps to best effort in CORBA NS)
2. Expiration (expiration in milliseconds, set to QoS in the variable Timeout)
3. Priority (Mapped to notification Priority QoS in the variable header)

Other user-defined name-value pairs are converted to IDL using the standard primitive mapping. Since Notification Service uses the Extended Trader Constraint Language and JMS uses the where clause of SQL92, the Notification Service needs to be extended to support SQL92.

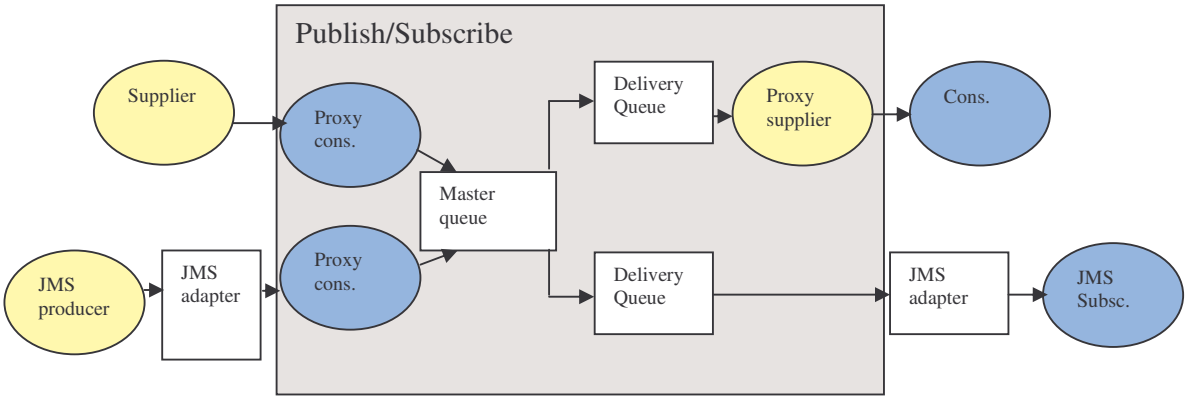

Figure 4. The OpenFusion Notification Service with JMS publish-subscribe interoperability.

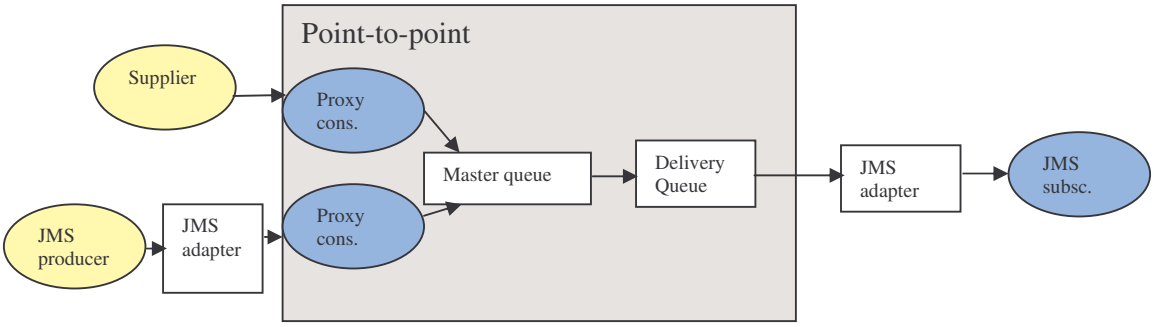

Figure 5. The OpenFusion Notification Service with JMS point-to-point interoperability [Pri01].

3.3.2 Wireless JMS

The iBus//Mobile software from SoftWired consists of a server-side gateway for mobile clients and a JMS compatible messaging server (iBus//MessageServer). The gateway enables communication between a wide variety of devices running different operating systems, such as PalmOS, Symbian, and PocketPC. The gateway supports communication over SMS,WAP, TCP, UDP, and GPRS. The system supports corresponding Java virtual machines, J2ME (CLDC and CDC), PersonalJava, and J2SE [R+01]. All communication between the clients and the gateway is transmitted in binary form. From the JMS provider's viewpoint the gateway is a regular

JMS client and from the client's viewpoint the gateway is a communication hub and a wrapper for different transport and representation formats. In the case of SMS the gateway accepts the incoming messages and a component within the service domain can respond with SMS. The client side library takes a minimum of 70k and at runtime the CLDC version takes a minimum of 50k of Java heap (as a comparison, a 8MB Palm has a 150k Java heap). The iBus system supports security in the form of access control, certificates, and symmetric/asymmetric keys. Cryptographic functions are supported through third-party libraries. If the bearer does not support push-type connections, one connection is used for sending client data to the server and another connection is used for communication from the gateway to the client. Each HTTP request goes over the first connection: send data to the servlet, and return. The second connection is open and blocks until there is traffic; after receiving messages the connection is immediately re-established. The underlying library hides the differences between the protocols.

3.4 CORBA Event Service

The CORBA Event Service specification (current version 1.1) defines a communication model that allows an object to accept registrations and send events to a number of receiver objects [Sie99]. The Event Service supplements the standard CORBA client-server communication model and is part of the CORBAServices that provide system level services for object-based systems. In the client-server model illustrated in Figure 6, the client makes a synchronous IDL operation on a specified object at the server. The event communication is unidirectional (using CORBA one-way operations) [OMG01a]. The Event Service extends the basic call model by providing support for a communication model where client

applications can send messages to arbitrary objects in other applications. The Event Service addresses the limitations of synchronous and asynchronous invocation in CORBA.

The specification defines the concept of events in CORBA: an event is created by the event supplier and is transferred to all relevant event consumers. The set of suppliers is decoupled from the set of consumers, and the supplier has no knowledge of the number or identity of the consumers. The consumers have no knowledge of which supplier generated the event. The Event Service defines a new element, the event channel, which asynchronously transfers events between suppliers and consumers. Suppliers and consumers connect to the event channel using the interfaces supported by the channel. An event is a successful completion of a sequence of operation calls made on consumers, suppliers, and the event channel.

The event channel performs the following functions:

- It allows consumers to register interest in events and stores the registration information.
- It accepts events generated by suppliers.
- It forwards events from suppliers to registered consumers.

The Event Service is defined to operate above the ORB architecture: the suppliers, the consumers, and the event channel may be implemented as ORB applications and events are defined using standard IDL invocations.

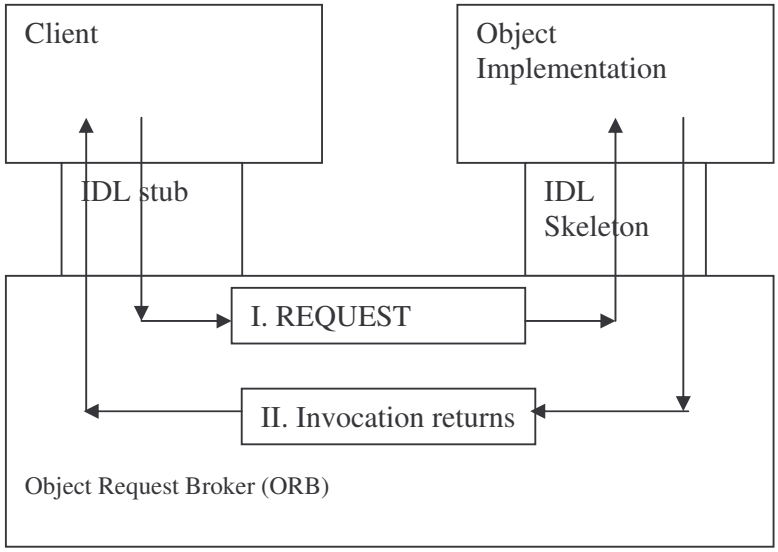

Figure 6. The standard CORBA client-server model of invoking operations from client to the target object.

3.4.1 Push and Pull

The CORBA Event Service provides two models for initiating the transfer of events between suppliers and consumers. The first model is the push model, in which suppliers send events to consumer. In this case the suppliers are active and the consumers passive. Moreover, the event channel actively delivers events to the consumers. In the second model, the pull model, the consumers request events from the suppliers. Now, the consumer actively waits for pull

requests to arrive. Upon the arrival of a pull request, the event is generated and sent to the pulling consumer. CORBA supports both blocking and non-blocking pull.

3.4.2 The Hybrid Model

It is also possible to mix the push and pull models in one application, because the event channel decouples the consumers and suppliers from each other. It is possible to connect suppliers using the push model and consumers using the pull model. In the hybrid model, the event channel does not take an active role in delivering the event to the consumers.

3.4.3 Connecting Suppliers and Consumers

The Event Service specification does not include a mechanism for locating or discovering consumers or suppliers, however, it provides the administrative operations for connecting the suppliers and consumers. Each new event consumer added to the event channel returns a proxy supplier. The proxy supplier follows the supplier interface and adds a new method for connecting a consumer to the proxy supplier. Each new event supplier added to the event channel returns a proxy consumer. The proxy consumer has a new method for connecting to the proxy supplier. A supplier is registered by taking a proxy consumer from the event channel and connecting it with the supplier. Similarly, an event-receiving application takes a proxy supplier from the event channel and connects to it by providing a consumer. Each admin object is a factory that creates the proxy interface that is used in connecting the clients and the event sources. Consumer admins create proxy suppliers and supplier admins create proxy consumers.

3.4.4 Typed and Untyped Event Communication

The data of an event can be passed as invocation parameters or return values. Events are not objects, because the CORBA object model does not support passing objects by value (CORBA 2.3 supports valuetypes). Event data is application-specific and can be either untyped or typed. In untyped communication the event is propagated by invoking a series of generic push and pull operations. The push operation takes a single parameter of type any, which allows any IDL defined datatype to be propagated, and stores the event data. The pull operation has no parameters and transfers event data in its return value, which is of type any. In untyped communication both the supplier and the consumer applications need to agree on the data format of the event. In typed event communication events are propagated through an application-specific interface created by the programmer in IDL. The programmer defines the interface for event propagation that is used by consumers and suppliers. Parameters can be of any suitable datatype supported by the IDL language. To setup typed push-style communication, the consumers and suppliers exchange object references (TypedPushConsumer and PushSupplier).

The supplier invokes a method to get a reference that supports the typed consumer interface. The particular reference is associated with the TypedPushConsumer interface and needs to be agreed on by both the consumer and the supplier. The supplier uses this reference to invoke operations on the consumer. In the typed pull model consumers request event information using some mutually agreed interface. The parties exchange the PullConsumer and TypedPullSupplier interfaces, and an object reference supporting the typed interface is obtained. Once the reference is obtained, the consumer can invoke operations on the supplier.

3.4.5 Discussion

The CORBA Event Service supports different implementations of the Event Channel, and this allows a wide range of approaches for implementing Quality of Service and delivery issues. Moreover, the event consumer and supplier interfaces support disconnection. The CORBA Event Service addresses some of the problems of the standard CORBA synchronous method invocations by decoupling the interfaces and providing a mediator for asynchronous communication between consumers and suppliers. The supplier does not have to wait for the event to be delivered to the consumer. Moreover, the event channel hides the number and identity of the consumers from suppliers using the proxy objects (transparent group communication). The supplier sends events to its proxy consumer, and the consumer receives events from its proxy supplier. However, the specification does not address several important issues, such as Quality of Service support. Applications may have requirements for event notification in terms of reliability, ordering, priority, and timeliness. Furthermore, the specification does not provide a system for event filtering. Event filtering needs to be implemented using a proprietary system within the event channel by adding a mechanism for selective event delivery. Event channels can be composed, because they use the same consumer/supplier interfaces. An event channel can push an event to another event channel. Typed event channels can be used to filter events based on event type [Bar01, OMG01a].

In addition, the specification does not address compound events, but suggests that complex events may be handled by creating a notification tree and checking event predicates at each node of the tree. The drawback of the tree is that the number of hops needed to deliver an event increases.
This motivates the use of a centralized filtering service. The use of proprietary event service implementations restricts the interoperability of applications. Applications that use one proprietary event service implementation may not interoperate with another application that is based on a different event service implementation.

3.5 CORBA Notification Service

The CORBA Notification Service (current version 1.0.1) [OMG01b] extends the functionality and interfaces of the Event Service to support better interoperability [Bar01]. One of the most significant additions to the Notification Service is event filtering. Filters allow consumers to receive particular events that match certain constraint expressions. Filtering reduces the number of events sent to the consumers and improves the scalability of the event handling system. Figure 7 presents the components of the CORBA Notification Service, which derive from the Event Service discussed in the previous section. The event channel has been extended to support a number of admin objects. The Notification Service allows the definition of filters at the proxies. Moreover, each admin object is seen as the manager of the set of proxies it has created. Admin objects may be associated with QoS properties and filter objects. The QoS properties and filter objects of the admin object are transferred to each proxy it creates, however, the QoS properties may be changed on a per-proxy basis.

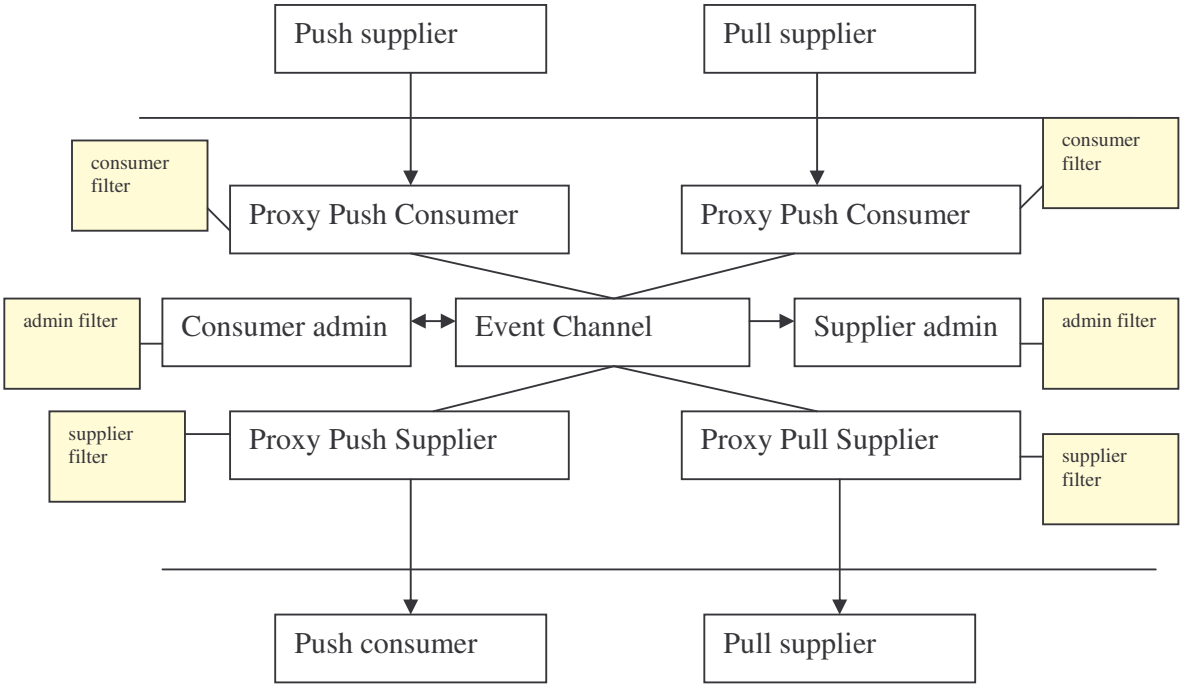

Figure 7. Components in the CORBA Notification Service [GCSO01].

3.5.1 Filters

Filters are CORBA objects that support the addition, modification, and removal of constraints. Constraints are used to match event message values and refer to variables that are part of the event notification message. Constraints are either event types or written in a constraint language. Variable names can refer to all parts of the current notification. The current notification is expressed with the dollar sign '$'.

A sample notification constraint:

$.type_name == StockAlert
$.market_name == 'NASDAQ'
$.ticker == 'Company'
$.price > '100' or $.price < 80

The default constraint grammar is Extended TCL (Trader Constraint Language specified by the Trading Service). The Event Notification specification adds the notion of mapping filter objects. Each proxy supplier may have an association with a mapping filter object, which affects the priority and the lifetime property of the events it receives.

3.5.2 Quality of Service (QoS)

The Notification Service defines standard interfaces that allow the control of characteristics over the delivery of the notification. Service characteristics at different levels in the protocol stack are represented using name/value pairs. QoS properties, tuples of the form <String, Any>, can be used with an event channel, admin objects, proxy suppliers, proxy consumers, and message instances.

Characteristics include:

- Discard policy that determines which notifications are discarded when resource limits apply (queues are full).
- Earliest delivery time.
- Expiration time, which indicates the time range when the event is valid.
- Maximum number of notifications that can be queued for a single consumer. This effectively places an upper bound that lessens the load presented by misbehaving consumers.
- Order policy, which specifies the order in which notifications are buffered for delivery.
- Priority of events.
- Reliability of event delivery
- Both event reliability and connection reliability. If fault tolerance properties are specified, the Notification Service reconnects to the set of clients and delivers all non-expired events to consumers after a crash or disconnection. At the message level: Best effort, persistent.

Furthermore, the event channel supports the following QoS properties:

- MaxQueueLength, which specifies the maximum number of events that can be queued.
- MaxConsumers, which specifies the maximum number of consumers that can be connected to the channel.
- MaxSuppliers, which specifies the maximum number of suppliers that can be connected to the channel.

3.5.3 Structured Events

The Notification Service defines a standard data structure for the events. The structured event illustrated in Figure 8 is a strongly typed event message that consists of a header and a body. The header contains two sections:

- The first stores fixed information, such as domain_name, event_name, and type_name.
- The second section stores the variables and optional information about the event. This is a sequence of properties to hold QoS information related to the notification.

The body of the structured event stores the actual event data and is also divided into two sections:

- The filterable data, which is a sequence of properties. This part contains the fields that the consumers use to base filtering decisions on.
- The payload data.

The header and body are structured into two parts mainly because of performance reasons. When filterable data has its separate compartment, it is not necessary to touch the payload data upon filtering. Moreover, the notification could be contained within the optional header fields leaving the body empty. This would be even more streamlined.

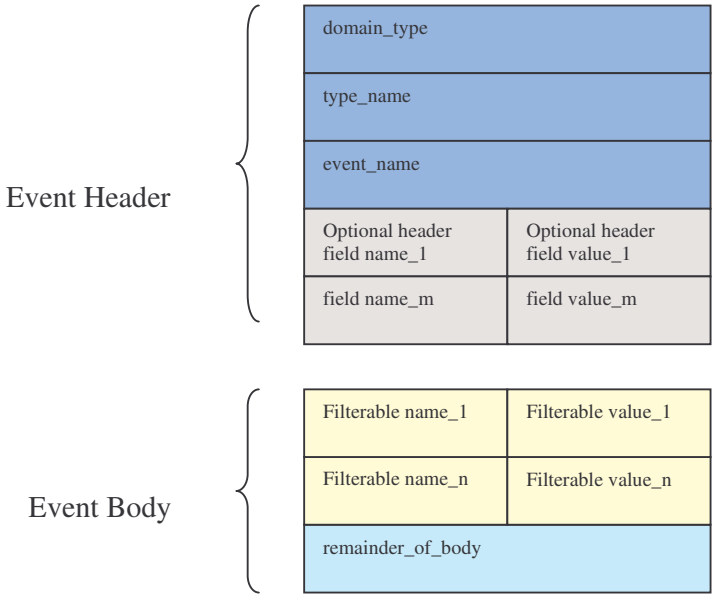

Figure 8. The structured event: Event header and event body.

3.5.4 Discussion

The centralized nature of the Event Channel as a CORBA object limits its scalability. All the registered consumers and suppliers are managed by the channel, which may limit the number of active entities and also the maximum number of notifications that the event channel is capable of processing in a given timeframe. Therefore it becomes important to create, manage, and specify federations of event channels (Figure 9). Each event channel has a master queue and a number of consumer queues. Each queue has some maximum capacity, which may be enforced using QoS policies supported by the specification. One way to relieve the bottleneck of the centralized event channel is to distribute these queues as CORBA objects; however, this kind of solution is still centralized. Since NS supports the federation of channels by connecting the supplier and consumer proxies, the system supports scalability.

Channel federation can be used to:

- Improve performance by distributing consumers on several event channels. Since an event channel is a CORBA object, it may become a bottleneck if the number of consumers (or producers) becomes large. Event channels may also be used to enhance local delivery by assigning to each event channel only local subscribers. In this case there is only one network invocation and a number of local invocations.
- Improve reliability by having multiple event channels for the same information. If one event channel fails, it does not necessarily prevent consumers from receiving the notifications.
- Improve flexibility by grouping consumers and producers into logical units (event channels).

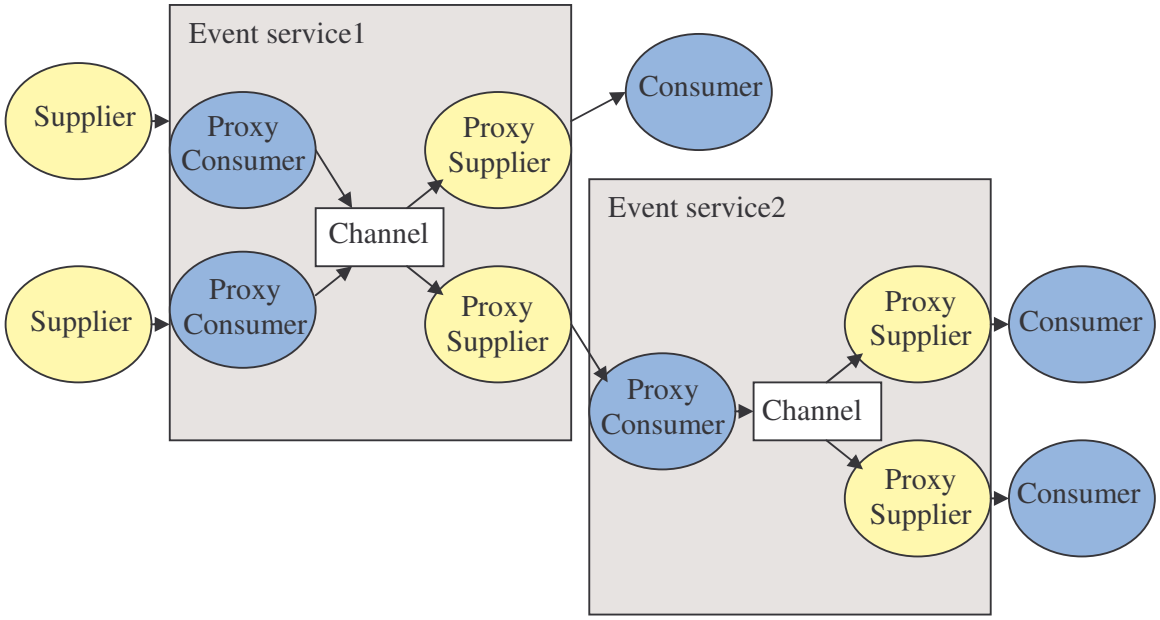

Figure 9. CORBA Notification Service channel federation.

The latest version of the CORBA Notification Service is 1.0.1, which is an editorial update (04/08/2002). The revised version includes changes to Chapter 3: Modules and Interfaces. Specific changes are marked with change bars. The reason for the editorial update was to change all occurrences of the struct name in CosNotification to _EventType. The OMG has started an effort to define a reliable multicast protocol for the CORBA Notification Service. This protocol is used to dispatch publish and subscribe messages to a list of consumers.

3.6 CORBA Management of Event Domains

CORBA Event Service and Notification Service do not specify an event discovery service or a mechanism to federate event channels. Moreover, the procedure for connecting event channels is complex. The OMG Telecommunications Domain Task Force addresses these issues in the CORBA Management of Event Domains Specification [OMG01b], which specifies an architecture and interfaces for managing event domains. An event domain is a set of one or more event channels grouped together for management, and for improved scalability. The specification defines two generic domain interfaces for managing generic typed and untyped channels. Moreover, a specialized domain for both channels and logs is defined by the OMG Telecom Log Service specification.

The specification addresses [OMG01b]:

- connection management of clients to the domain,
- topology management,
- sharing the subscription and advertisement information in an event domain, even when connections between event channels change at runtime,
- event forwarding within a channel topology, and
- connections between event channels.

It supports the creation of channel topologies of arbitrary complexity, allowing cycles and diamond shapes in the graph of interconnected channels. However, if events may reach a point in the graph by more than one route duplicate events need to be detected and removed. Moreover, if no timeouts are specified, events in a cycle will propagate infinitely. Therefore, the specification defines mechanisms that are used to detect cycles or diamonds in the network topology. Graph topology enforcement is done at channel connection time, and the domain management refuses illegal connections. Event suppliers inform the proxy consumers of event type changes using the offer_change callback. The channel is responsible for sharing this information with the consumers by executing offer_change on them. The consumer may be another channel and thus the change may propagate throughout the channel topology. Subscription changes work similarly, and the channel is responsible for invoking the subscription_change operation on all suppliers.

Event suppliers attached to the channel can obtain the types of subscriptions of event channels anywhere downstream by invoking obtain_subscription_types on the proxy consumers. Similarly an event consumer can obtain the event types offered by suppliers on any event channel downstream by invoking obtain_offered_types on its supplier channels.

3.7 OMG Data Distribution Service (DDS)

The Data Distribution Service for Real-Time Systems (DDS) OMG specification was adopted in June 2003 and finalized in June 2004. The specification defines an API for data-centric publish/subscribe communication for distributed real-time systems. DDS is a middleware service that provides a global data space that is accessible to all interested applications. The specification describes the service using UML [OMG04].

DDS uses the combination of a Topic object and a key to uniquely identify instances of data-objects. In this model, the subscriptions are decoupled from the publications. DDS creates a name space that allows participants to locate and share objects. In case a set of instances are under the same topic, these different instances must be distinguishable.

DDS uses a key to distinguish between these instances. The key consists of the values of some data fields. These fields need to be indicated to the middleware. Different data values with the same key value represent successive values for the same instance. Different data values with different key values represent different instances. If no key is provided, the data set associated with the Topic is restricted to a single instance.

A ContentFilteredTopic may be created for content-based subscriptions. In addition, the MultiTopic can be used to subscribe to multiple topics and combine/filter the received data. The filter language syntax is a subset of the SQL syntax.

The QoS usage follows the subscriber-requested publisher-offered pattern. In this pattern, the subscribers request desired QoS properties and these are matched against those offered by the producers.

DDS is suitable for signal, data, and event propagation. Signals represent continuously changing data, for example from a sensor. In this case, publishers may set the reliability property to best-effort and the history QoS property to retain the last signal (KEEP_LAST). Data delivery, such as exchanging the state of a set of objects, can be realized by using reliable communication and requiring that the last data elements are stored by the system. Events are streams of values and publishers typically use reliable delivery and require that the system keeps a history of all messages (KEEP_ALL).

DDS complements CORBA, because it provides a service more suitable for asynchronous and dynamic operation. CORBA provides support for distributed objects in a client/server environment and supports remote method calls. DDS is more suitable for flexible QoS-aware data dissemination to many nodes in dynamic environments. Therefore, CORBA is object-centric, whereas DDS is data-centric.

The CORBA Event Service decouples producers and consumers, but it is not data-centric and does not offer QoS contracts. CORBA Notification Service offers a more data-centric approach with filters and QoS support. DDS differs from these two services, because it does not have to support the Common Data Representation or use the IIOP protocol. This means that DDS services do not interoperate unless an interoperability specification is adopted. In other words, a DDS implementation does not have to be CORBA based.

Figure 10 presents an overview of information flows in DDS. The data-objects are identified by the Topic. The publishers are objects that use typed accessors, DataWriters, to communicate data-objects. The subscribers are objects that use typed DataReaders to receive information. A subscription is a subscriber with an associated DataReader.

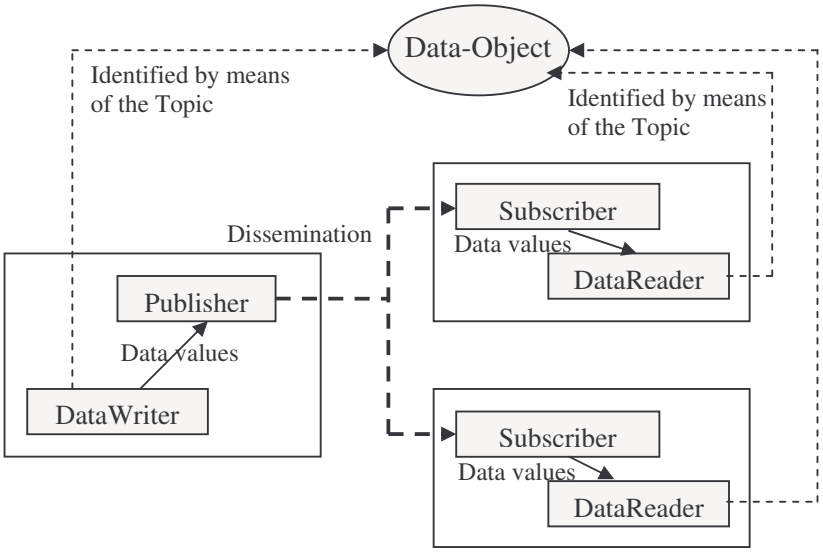

Figure 10. Overview of information flows in DDS.

3.8 W3C DOM Events

W3C's Document Object Model Level 2 Events is a platform- and language neutral interface that defines a generic event system [W3C00]. The event system builds on the DOM Model Level 2 Core and on DOM Level 2 Views. The system supports registration of event handlers, describes event flow through a tree structure, and provides contextual information for each event. The specification provides a common subset of the current event systems in DOM Level 0 browsers. For example, the model is typically used by browsers to propagate and capture different document events, such as component activation, mouse overs, and clicks. The two propagation approaches supported are capturing and bubbling. Capturing means that an event can be handled by one of the event's target's ancestors before being handled by the event's target. Bubbling is the process by which an event can be handled by one of the event's target's ancestors after being handled by the event's target. The specification does not support event filtering or distributed operation.

The specification "An Events Syntax for XML" is a W3C Recommendation (11 October 2003) and defines a module that provides XML languages with the ability to integrate event listeners and handlers with DOM Level 2 event interfaces [W3C03]. The specification provides an XML representation of the DOM event interfaces. The ability to process external event handlers is not required.

3.9 Web Services Eventing (WS-Eventing)

The Web Services Eventing (WS-Eventing) specification describes a protocol that allows Web Services to subscribe or to accept subscriptions for event notifications [BMT04]. An interest registration mechanism is specified using XML Schema and WSDL. The specification supports both SOAP 1.1 and SOAP 1.2 Envelopes. The key aims of the specification are to specify the means to create and delete event subscriptions, to define expiration for subscriptions, and to allow them to be renewed. The specification relies on other specifications for secure, reliable, and/or transacted messaging. The specification supports filters by specifying an abstract filter element that supports different filtering languages and mechanisms through the Dialect attribute. The filter is specified in the Filter element.

3.10 COM+ and .NET

Standard COM and OLE support asynchronous communication and the passing of events using callbacks, however, these approaches have their problems. Standard COM publishers and subscribers are tightly coupled. The subscriber knows the mechanism for connecting to the publisher (interfaces exposed by the container). This approach does not work very well beyond a single desktop. Now, the components need to be active at the same time in order to communicate with events. Moreover, the subscriber needs to know the exact mechanism the publisher requires. This interface may vary from publisher to publisher making this difficult to do dynamically (ActiveX and COM use the IconnectionPoint mechanism for creating the callback circuit; an OLE server uses the method Advise on the IoleObject interface). Furthermore, this classic approach does not allow filtering or interception of events [Pla99, Sri01, Mic02].

3.10.1 COM+ Event Service

The COM+ event service [Pla99, Mic02] is an operating system service that provides the general infrastructure for connecting publishers and subscribers. The service is a Loosely Coupled System (LCS), because it decouples event producers from event subscribers using the event service and a catalog for storing available events and subscription information. In this architecture, an event is a method in a COM+ interface called the event method, and it contains only input parameters.

The following steps are required to produce an event (Figure 11):

1. An event Class is registered.
2. Subscriber registers for an Event.
3. Publisher creates an Event Object at run time.
4. Publisher fires the Event by calling the method in the Event Object.
5. Event Object reads the Subscription List from the Event Store.
6. Delivers the event to the subscriber by calling the appropriate method.

The change in the COM+ Event Service is the addition of the event service in the middle of the communication. The event service keeps track of which subscribers want to receive the calls, and mediates the calls. The event class is a COM+ component that contains interfaces and methods. A subscriber needs to implement the interfaces in order to receive the event, and a publisher calls the methods to fire events. Event classes are stored in a COM+ catalog that is updated either by the publishers or by the administration. Subscribers register their wish to receive events by registering a subscription with the COM+ event service.

A subscription is a data structure that contains the recipient, event class, and which interface or method within that event class the subscriber wants to receive calls from. Subscriptions are also stored in the COM+ catalog either by the subscribers or by the administration. Persistent subscriptions survive restarting the operating system whereas transient subscriptions will be lost on restart or reset. The publishers use the standard object creation functions to create an object of the desired event class. This event object contains the event system's implementation of the requested interface. The publisher then calls the event method that it wants to fire. The event system implementation of that interface searches the COM+ catalog and finds all the subscribers who have expressed interests in that event class and method. The event system then connects to each subscriber, using direct creation, monikers, or queued components, and calls the specified method. Event methods return only success or failure. Any COM+ client can become a publisher and any COM+ component can become a subscriber.

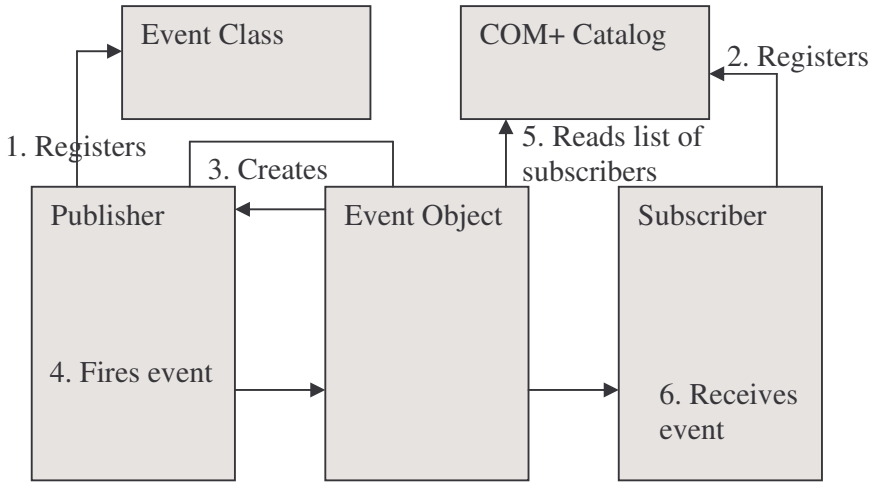

Figure 11. The COM+ Event Service.

The current event system has several limitations. The subscription mechanism is not itself distributed and there is no support for enterprise wide repository. Secondly, event communication in the system is done either by DCOM or Queued Components, which are both one-to-one communication mediums. The delivery time and effort increases linearly with the number of subscribers, which means that the system is not scalable to firing events to many subscribers. However, client-side disconnection is supported with queued components. COM+ supports components that record a series of method invocations (event occurrences) and are able to play them back in the recorded order. These components can be distributed using messages. Since the event object may be defined as queuable, a disconnected client may play back the desired event object upon reconnection. COM+ Events can be extended to support filtering, which needs to be implemented either on the publisher side or on the subscriber side. If an event is filtered by a component on the publisher side, it is never delivered to the event service. If an event is filtered on the subscriber side the event service will make the decision of whether to deliver the event to a particular subscriber [Mic02].

Filtering on the publisher side is done by attaching a filter object to the event object interfaces (which correspond to events). The filter may query the subscription information and, for example, change the firing order for a set of subscribers. The subscriber-side filtering is done using parameter filtering for each subscription and method invocation. Parameter filtering evaluates the subscription FilterCriteria property against the parameters of the event method. The filter criteria string recognizes relational operators, nested parenthesis, and the logical keywords AND, OR, and NOT.

3.10.2 Interoperability with .NET

The COM+ Event System needs to generate some metadata in order to interoperate with the .NET world. However, an abstract definition of the Event Interface, Event Classes, and their attributes is needed [Kis01].

3.10.3 .NET

The .NET framework supports events at many levels. There is support for programming-language-level events and interoperability with COM events. The interoperation of Visual Basic .NET code and legacy COM component events is done using a runtime callable wrapper (RCW). In VB.NET listeners create event handlers, which are added to sources. The connection between events and event handlers is implemented by special objects called delegates. The benefit of the .NET runtime is that the events from components written in different languages, say C# and VB, are interoperable. Microsoft's messaging infrastructure is called Microsoft Message Queuing (MSMQ) [Mic99]. In this kind of architecture, applications receive and send messages using queues. MSMQ supports disconnected operation and is especially useful on intermittently connectedWindows CE/PocketPC devices. MSMQ allows application writers to asynchronously send messages. MSMQ CE version can, for example, be used

- for messages transferred when in range (delivery tracking, quality control),
- for messages transferred once in a while (intelligent set-top boxes, inventory control), or
- when Producer and Consumer are not active at the same time.

3.10.4 MSMQ Product Architecture

The MSMQ architecture is presented in 12. MSMQ queues are either private or public. Public queues are stored in a directory service called Message Queue Information Store. Public queues are more expensive to use because directory access is not free. Moreover, Windows CE clients cannot host public queues. The CE MSMQ independent client can operate independently if the server is unavailable and store messages locally. The servers route and store messages and support clients in the form of a client proxy server and a queue manager. On the other hand, MSMQ supports also dependent clients that cannot store local messages and need the server. The architecture supports three delivery options. Fast memory-based reliable store-and-forward supports network loss, but not reboot, and cannot guarantee exactly-once semantics. Persistent guaranteed store-and-forward supports reboot, and persistent transactional message queuing guarantees exactly-once in-order delivery. Transactional guarantee at commit time is about delivery to the local queue. In essence, the system supports local all-or-nothing guarantee [Mic99].

The MSMQ version for Windows CE (2.12+) supports roaming and dynamic adapter switching. It tracks Network Interface Cards (NIC) and restarts immediately after reconnection. The transparent storage is based on one queue per file. The footprint of the system is around 100-150K. The CE implementation has several limitations: clients must use direct names, only private queues are supported, the routing is limited, transactions are not supported (once and in-order are supported), there is no system support for encryption or ACL, and there is no remote queue access. The system can be deployed in a client-server or client-client environment and also for message-based IPC within a device.

The next version of MSMQ, Message Queuing 3.0, is available in Windows XP and supports messaging over the Internet, a one-to-many messaging model, and message queuing triggers [Mic02]. HTTP is supported as an optional transport protocol and an XML-based SOAP extension is introduced that defines a reliable end-to-end messaging protocol. By default MSMQ uses a proprietary TCP-based protocol. The system also supports real-time messaging multicast using the Pragmatic General Multicast (PGM) protocol [S+01]. This protocol supports only an at-most-once quality of service and does not support transactional sending. The MSMQ 3.0 programming model is extended to allow an application to send a single message to a list of destination queues.

Message Queuing Trigger is a service that allows an application to assign functionality in a COM object to be triggered when a message arrives in a particular queue. Each trigger is associated with a queue and applies a set of rules for every message arriving in that queue. An action is executed when all conditions in a trigger hold [Mic02]. Message routing is done using the lowest-cost route that is available. If a network fails, the next-lowest-cost route is used to deliver the message. Administrators define costs for each network with the management software (MSMQ explorer).

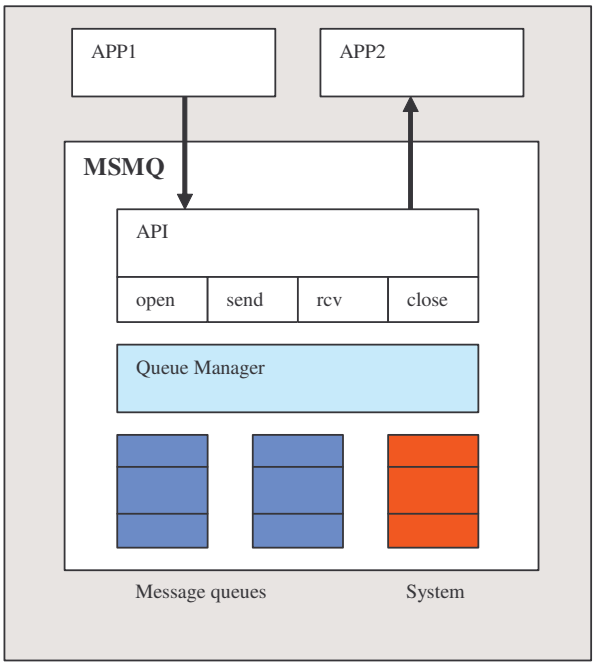

Figure 12. MSMQ Product Architecture. The Queue Manager connects to other Queue Managers in order to communicate between different hosts.

3.11 Websphere MQ

IBM's MQSeries, currently known as Websphere MQ, is one of the most popular MOM products for electronic business. The product supports heterogeneous any-to-any communication between 35 different platforms. MQ is compatible with JMS and integrates with Java Beans 2.0 (EJB), XML, and JSP framework and servlets. MQ also supports SOAP for Web service creation. A JMS 1.0.2 compliant embedded JMS provider supports point-to-point and publish-subscribe messaging [IBM02b]. MQSeries Everyplace enables access to enterprise data and supports mobile workers. Everyplace is available for a number of platforms, for instance Linux, WinCE, EPOC, and PalmOS. The PDA type messaging is similar to messaging for other platforms with queue managers. A queue manager manages queues that store messages, and applications communicate with their local queue manager. Remote queues are owned by remote queue managers, and each message that is inserted into a remote queue gets transmitted over the network. The queue manager may support a local queue, in which case the client is capable of supporting asynchronous communication. If no local queue is present, the client is bound to synchronous communication. Another configuration option is whether the client supports bridges and is capable of exchanging messages with other MQSeries queue managers.

A typical client-server configuration is a scenario where a server hosts the queue manager and clients connect to it with a bi-directional communication link (with a proprietary MQSeries protocol). The client infrastructure is quite lightweight, because it is dependent on the server queue manager. In a multi-server scenario, clients employ message channels, which support unidirectional, safe, and asynchronous message exchange. Channels are a form of end-to-end service provision and consist of the source queue manager, a number of intermediate managers, and the destination queue manager. The footprint of the system is 64K for Palm and 100K for a class file with Java devices [IBM02b].

4 Event Systems

This chapter presents event systems and prototypes. We present the Cambridge Event Architecture, Siena, Scribe, Elvin, JEDI, ECho, JECho, Rebeca, Gryphon, STEAM, and Rapide, DADI, Hermes, and the Fuego event system.

4.1 The Cambridge Event Architecture

The Cambridge Event Architecture (CEA) uses the publish-register-notify paradigm [BMH+00], in which the object publishes its interface, for example specified in IDL (Interface Definition Language, which is different from the IDL in CORBA). This interface includes the events of which it is capable of notifying. A client invokes the object synchronously and can register for events by indicating parameters (attributes) or wildcards. Wildcard matching is applied on the parameters of a notification, but it may not be applied on the event type. The template system provides rudimentary filtering by matching parameters one by one. The object accepts registrations and notifies the clients that match the registration template. The notification is performed when the event firing conditions and access restrictions are satisfied (Figure 13). The paradigm supports direct source-to-client event notification.

In CEA an object, if asked, publishes the events it is capable of notifying of in IDL. The object has a register method in its interface that has parameters for the type of event and wildcards. Event occurrences are objects of a specific type, and the set of types defines the level of event detection and notification granularity. CEA enforces access control upon registration, and authentication is based on a parameter value. CEA supports defining intermediate services, which are called event mediators in the architecture. Event mediators act as middlemen between primitive event sources and event clients, and provide the facilities for detecting more complex events. Moreover, if an event source cannot afford the overhead of supporting template matching, it can send all its events to the mediator. The mediator then matches the template on behalf of the source.

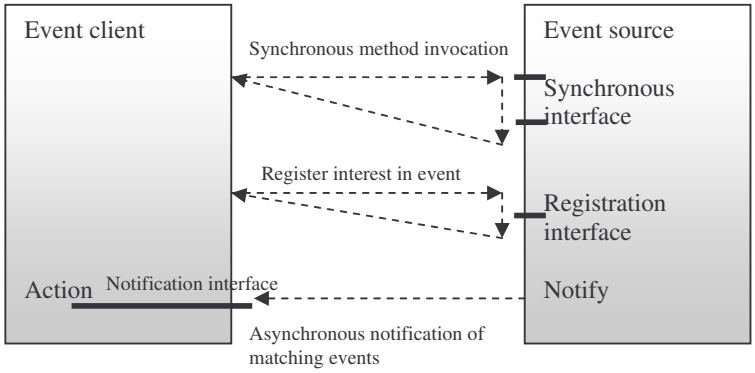

Figure 13. A publish-register-notify event architecture [BMH+00].

The mediator is capable of providing equivalent functionality to the CORBA event service. The CORBA event service registers interests in all notifiable events with event sources and supports both a synchronous pull interface and an asynchronous push interface. Composite events can be detected by giving mediators the capability to filter simple events of different types across different sources. The composite event detection functionality supported in CEA

is a feature that is not present in many event systems. The event composition is supported by the combination of event templates. Composite events are detected by monitors, which are busy until the event is detected and fired. A composite event specification language may be used to design a monitor that detects complex templates. The system has been demonstrated by implementing an active badge system that monitors badges within a building.

Composite events have also been investigated at Cambridge [PSB03] recently. This paper presents a distributed framework for composite event detection and notification in a distributed environment. The system is based on JMS, and leverages the features of the underlying architecture. The key benefits of the proposed approach are the distribution of the detection task, an automata-based detection engine, and the use of an interval time model to detect the causality of events. A Lamport Logical Scalar Clock gives a causal ordering if such exists (but not a strict causal ordering). The paper presents a specification language for composite events. The system transforms the specification language to finite state machines. Formal semantics are given for the interval time model. The problem of translating non-deterministic automata to deterministic is not discussed other than to mention that the current implementation uses non-deterministic automata with a list implementation.

4.2 Scalable Internet Event Notification Architecture

Siena (Scalable Internet Event Notification Service) is an Internet-scale event notification service developed at the University of Colorado. Siena balances expressiveness with scalability and explores content-based routing in a wide-area network. The basic publish-subscribe mechanism is extended with advertisements that are used to optimize the routing of subscriptions [CRW99]. Several network topologies are supported in the architecture, including hierarchical, acyclic peer-to-peer, and general peer-to-peer topologies. Servers only know about their neighbors, which minimizes routing table management overhead. Servers employ a server-server protocol to communicate with their peers and a client-server protocol to communicate with the clients that subscribe to notifications. It is also possible to create hybrid network topologies.

Siena is similar to IP-multicast; however, the two mechanisms differ in the way they support groups of subscribers. IP groups are not very expressive. They partition the IP datagram address space and each datagram can belong to at most one group. Clearly, this creates problems if an event that spans several groups of subscribers is to be delivered. Four different server topologies have been identified in Siena: centralized, hierarchical, acyclic peer-to-peer, and generic peer-to-peer.

4.2.1 Naming and Filtering

Siena is implemented with a flat event namespace, i.e. event names have no structural correlation with each other. An event consists of a set of attribute-value pairs. Each attribute has a name and a value. Siena supports the types null, string, long, integer, double, and boolean. A filter consists of an attribute name, a constraint operator, and a constraint. Siena does not support wildcards in the attribute name so the attribute names must match exactly to the names in the published event. A filter may include several filtering clauses, which are ANDed together. Thus every filtering clause or component must return true in order for the filter to pass the event. Siena supports the operators equal, less than, greater than, greater than or equal to, less than or equal to, string prefix, string suffix, always matches, not equal, and substring.

An example event:
string stock "abc"
int value 2.53

An example filter:
string stock = "cde"
int value > 1.0
int value < 1.5

Siena supports patterns, which are based on event attribute values and event combinations. A pattern is a sequence of filters that is matched to a temporally ordered sequence of notifications. Network latencies may cause some events to arrive in the wrong order, and these are ignored by the Siena solution.

4.2.2 Routing

In Siena, each event consists of a set of attribute-value pairs that are matched with filters. Each server on the event system routes events to other servers based on subscription information, advertisement information, and filters. Each subscriber may specify a filter to constrain the subscription. In the same fashion, each advertisement may also include a filter. Siena evaluates the filters and follows a policy where events are replicated downstream and filtered upstream. This means that events are replicated to the clients at the last possible moment, thus reducing the bandwidth needed to transmit the events. Upstream filtering means that events are filtered as close to the sources as possible in order to reduce the number of uninteresting events transmitted over the network. The simple filter syntax allows the decomposition of a complex filter into several more general filters, which can be evaluated upstream. A filter is only applied if it is less general than the one used in upstream.

The same principle of upstream filtering also applies to event patterns. Patterns are decomposed (factored) into elementary filters that are delegated to other servers. In the delegation process a server tries to assemble sub patterns that are delegable to other servers. Siena uses covering relations to determine when a filter covers a notification, a subscription covers a notification, an advertisement covers a notification, or an advertisement covers a subscription. For example, subscription S1 covers S2 if it evaluates to true in every instance where S2 is true. Servers propagate the most generic subscription that covers a given set of subscriptions. This minimizes the downstream data structures, however, the complex computation cost is paid closer to the subscriber, because the subscriptions need to be matched and evaluated. The results of Siena indicate that the covering relations exhibit a complexity that is quite reasonable for a scalable service.

The Siena system supports two different notification semantics: subscription- based and advertisement-based. In subscription-based semantics subscriptions are introduced at every node of the event service and a notification is routed if it covers a subscription. In advertisement-based routing servers use the information provided by event producers to route incoming subscriptions. A subscription is only forwarded if it covers the advertisement.

4.2.3 Forwarding Algorithm

The forwarding algorithm that was developed in conjunction with the Siena project consists of a forwarding table and a set of processing functions. Conceptually the forwarding table is a mapping between predicates (sets of filters) and interfaces to neighboring nodes. Each predicate is a disjunction of filters, where each filter is a conjunction of elementary conditions. Each elementary conjunction must return true in order for a filter (and predicate) to map to an interface. Each filter may map to several interfaces [CDW01]. The forwarding algorithm iterates over the event attributes. It searches for a partial match from the set of filters, where a constraint belonging to a filter is matched by the given attribute. If the filter (with the partial match) is not yet associated with an interface, the algorithm increases a counter to keep track of matched constraints for the given filter. If the counter size is equal to the number of constraints in the filter, the filter is said to match. After processing one filter the algorithm checks if all filters are matched. The algorithm stops if either all attributes of the notification or all filters are processed. The number of interfaces thus imposes an upper bound on the processing along with the number of attributes and filters. The forwarding algorithm is optimized using binary trees and lookup indices for attributes used in the filters.

The performance and scalability of the forwarding algorithm were demonstrated by running experiments with 1000 messages and various numbers of filters and other parameters. It was found that the algorithm has good absolute performance and good cost amortization over a variety of loads. The constraint index, which acts as a lookup table for attribute names over constraints, is used to quickly detect attribute names that have no matching constraints. If no attributes match the event can be discarded by the router. The filter-matching algorithm has recently been extended with several optimizations [CW03]. The algorithm uses a matching structure based on an index and selections over attribute filters. The paper proposes several enhancements, namely the selectivity table that is used to prune those predicates that cannot be matched.

4.2.4 Implementation

The current Siena implementation is a prototype that consists of Siena servers and client-level interfaces. The C++ version supports the peer-to-peer server and the Java version supports hierarchical servers. Currently, the C++ implementation is not compatible with the Java version. The Siena implementation uses TCP/IP for communication.

4.2.5 Simulation

The algorithms and topologies used in Siena were examined in a simulated environment. The hierarchical client-server architecture should be used when there is a low number of parties that subscribe and unsubscribe frequently. The acyclic peer-to-peer model was found to be more applicable to situations where the total cost is dominated by notifications and there are many ignored notifications [CRW99].

4.2.6 Current and Future Developments

Columbia University has developed the XML-based Universal Event Service (XUES) that consists of three main services that support event handling for the Kinesthetics eXtreme (KX) real-time monitoring architecture. The system inputs events using the Event Packager, analyzes events using the Event Distiller, and dispatches events using the Event Notifier. The system interacts with other event systems using XML, FleXML, and Siena. During the development of the Siena-XML interface [Ere01] several problems with translating an XML-based hierarchical namespace to a flat namespace were identified and addressed. In the

conversion process the nested structure of XML documents is converted into flat names that preserve the hierarchy by separating the hierarchies with dots. This is a typical way of describing hierarchical content; another would be to use the Windows or Unix file system notation. Now, a problem arises when there are duplicate elements in a hierarchy, which translate to an item with multiple values. Siena does not support this, and the Siena-XML interface currently ignores these duplicate values. One solution would be to include support for wildcards or multiple sets of values, for example simple list objects.

In the future Siena is envisaged to integrate at the network service level, coexisting for example with TCP/IP instead of working above the network level. This would eliminate an extra protocol layer, and provide greater efficiency in routing and forwarding. From the Siena viewpoint TCP/IP performs explicit address routing and Siena is based on content-based addressing. The risk in using Siena as a network service is that content-based routing is computationally more expensive than explicit-address or subject-based routing [Ros01]. There is also work to make Siena support satellite-based wireless communication. Satellite-based communication has desirable properties for transmitting events, because routing is not necessary when the events are broadcast rather than sent using point-to-point communication lines. Thus it is possible to notify large numbers of interested parties in one hop. However, wireless networking is more unreliable than wired networking. Moreover, the receiving devices may be different from desktop computers, thus requiring the solution to cope with limited resources.

Siena has also been used as a peer-to-peer network similar to Gnutella. The Java-based Quad uses the Siena prototype and supports query, advertise, and response. One of the differences between Quad and Gnutella is that with Gnutella the messages are propagated to all servers and filtering is performed by the provider at the last step. The main architectural difference between Gnutella and Quad is the separation of clients and servers. Thus the general advantage of peer-to-peer systems in dynamic networking is lost [Hei01].

Siena has been extended to support mobility and wireless clients. The mobility support involves a handover protocol that uses either subscription-based or advertisement-based semantics [CCW03]. One of the findings of the Siena project is that expressiveness and scalability are in conflict. Expressiveness is related to flexibility of notification and routing. Scalability, on the other hand, is about vast dimensions, heterogeneity, decentralization, and the use of resources.

4.3 Scribe

Scribe [CDKR02] is a topic-based publish-subscribe system that explores the scalability of the notification service in peer-to-peer environments. Scribe is built on top of Pastry, which is a scalable, self-organizing peer-to-peer location and routing system. Scribe provides an application-level multicast system. Pastry is based on uniform ID keys that are used as host addresses. The system routes a message to the closest possible key. Scribe provides a best-effort notification delivery on top of Pastry and specifies no particular event delivery order. Moreover, Scribe does not support filtering, buffering, or mobility. The rendezvous point forms the root of a multicast tree. In other words, the responsibility for a given topic (group of subscribers) is hashed over the set of the servers. When a subscribe message is routed towards the rendezvous point, each intermediate node adds the previous node to its table of children. This information is used in the multicast protocol, which is similar to reverse path forwarding. Events may be published directly if the IP address of the rendezvous point is known. However, subscriptions need to be routed within the peer-to-peer topology. Access control can be enforced at the rendezvous point. Pastry can route around faulty nodes by resending the subscription and thus repairing the multicast tree.

4.4 Elvin

Distributed Systems Technology Centre (DSTC) has been developing the Elvin system since 1993 and it has grown from a single person research project to an effort with a team of programmers and researchers. Elvin is a general event notification service, which aims to improve on features identified in a 1995 survey of commercial event filtering software. Elvin started as a publish-subscribe notification service, but currently it is referred to as a content-based routing service. The Elvin team aims to standardize the Elvin protocol through the Internet Engineering Task Force (IETF), and the Elvin protocols are written in the style of IETF drafts. DSTC was a contributor to the OMG Notification Service RFP and one of the submitters of the CORBA Notification Service.

Elvin uses a client-server architecture in notification delivery. Clients establish sessions with Elvin servers and subscribe and publish notifications. An Elvin notification is a list of name-value pairs, similarly to that of Siena. Basic primitives are a 32- and 64-bit integer, a 64-bit double precision floating point, an internationalized string (UTF-8 encoded), and an array of bytes. Subscription expressions are defined using logical expressions with a C-like syntax: “stock == "abc" && value > 80”. The expressions are evaluated with Lukasiewicz’s tri-state logic that uses an additional value of indefinite (i.e. true, false, indefinite). Elvin has language bindings for C, C++, Java, Python, Smalltalk, Emacs Lisp, and Tcl. Elvin is content-based, because it allows routing decisions to be made based on the whole message. Elvin features a decoupled security model, in contrast with the traditional point-to-point model, in which communication between publishers and subscribers is authenticated with keys. Producers and consumers can have overlapping key sets. This supports multi-party authorization.

Service discovery is done using a lightweight protocol that is based on multicast. Once a server has been deployed on the network, clients use the protocol to discover the server and dynamically register. Clients also listen to router advertisements, which are also distributed using multicast. Elvin 4.1 was released on March 19th in 2003. This version includes web-based router management, configurable quality of service, support for automatic failover of standby routers, federation between routers, and new scalability support.

4.4.1 Clustering

Elvin supports local clustering of servers that improves scalability and distributes the local load. Clustering is used to implement a distributed, but single-subscription, address space. Routers within a cluster communicate using a reliable multicast protocol over an IP network. An Elvin router may force a client to reconnect to another server in order to reduce load. The Elvin cluster is similar in functionality to a web farm. An Elvin router is a daemon process that runs on a single server and distributes Elvin messages. Each router in an Elvin cluster shares client subscription information with every other node. Not all subscription information is shared, but only sufficiently in order for a router to decide if a given notification has any subscribers at any server. The initial forwarding decision in server-server communication is done based on a list of terms. Messages are first analyzed at a local router and then multicast to the cluster. The set of destination routers is determined before multicasting by matching the message against the term list. Each packet contains the unique identifiers of the routers that have matching terms. This hasty approach results in a number of unnecessary notifications at the router level. The Elvin team aims to improve this in the next version of the system.

The Elvin cluster topology consists of a single master router and a number of slave routers. The master router maintains management data. All slave routers listen to management traffic within the cluster and keep information about every node. Routers also keep information

about subscription terms of other servers, current states, the list of URLs offered by a router for client connection, and current router load and statistics. Master servers listen for join packets and keep track of the cluster as a whole. Anew master router is elected using an election protocol if the old one fails. Communication between clients and routers employs RPC-style communication with positive and negative acknowledgements. Delivery has best effort, at-most-once semantics. In the client-server protocol the server may drop notifications, but is obliged to warn the client that it has done so.

4.4.2 Federation

There is a different protocol for linking distributed clusters of servers to a federated system. The Elvin federation protocol assumes that the federated topology forms a spanning tree. Moreover, the linking protocol supports the definition of pull filters that constrain the notifications sent to other clusters.

4.4.3 Quench

In Elvin terminology quench means an operation supported by all event producers that gives the producers the possibility to evaluate a subscription expression to cease producing events that are no longer needed. Quench is also used to determine which notifications should be produced. In CORBA this would mean that the first event channel refrains from forwarding unnecessary notifications (CORBA does not support client side filtering). The quench is a semantic extension of the subscribe mechanism In Elvin quench is implemented in the client-server protocol. Any client may request to be notified when the subscription information of the server changes. The client may request information on named attributes in subscriptions. The requested information is sent as an abstract syntax tree. There is also support for an automatic quench, which is implemented in the client library.

4.4.4 Mobile Users

Elvin has been extended to support mobile users. One of the requirements was persistence in order to keep undelivered notifications. Elvin is non-persistent by design so a prototype proxy was designed to store notifications. The proxy model extends the client-server architecture of Elvin by introducing the proxy as a third component. Proxies act as normal clients to servers, but as proxy servers to clients. In this design, clients connect to these proxies, which mediate the Elvin service [SAS01].

The proxy is able to handle multiple clients with separate sets of subscriptions. Elvin did not support subscription grouping by the client, so support for this was added to the system (the concept of a session). These sessions need not be client-specific, but may rather span multiple clients or applications. This stems from the observation that many people have several devices, but may wish to receive the same set of information regardless of the medium. In order to manage the storage space for undelivered notifications, the proxy supports the definition of a time-to-live (TTL) for each subscription. In addition, clients may specify the maximum number of notifications to keep. In the current prototype clients explicitly connect to the proxy, and they must connect to the same proxy to retrieve notifications. Proxy discovery and roaming between proxies is not supported. The Elvin proxy service is proposed as a solution to proxy roaming and client migration between networks. However, the difficulty lies in that the proxy is a stateful entity, whereas normal Elvin servers are stateless.

4.4.5 Non-destructive Notification Receipt

For users who use many different devices and wish to share notifications, Elvin supports non-destructive notification receipt. This means that the proxy does not destroy a notification upon its successful delivery. Elvin ensures that notifications are never delivered to the same client more than once. Because sessions may contain a number of clients, Elvin supports additional management functionality regarding the set of subscription set by clients. Each client is informed of the current subscription status. There may also be a number of sessions per client, in which case only one notification is sent even if there are multiple matches.

4.5 JEDI

Java Event-based Distributed Infrastructure (JEDI) is a distributed event system developed at Cefriel at Politecnico di Milano. In JEDI the distributed architecture consists of a set of dispatching servers (DS) that are connected in a tree structure [CDN01]. Each DS is located on a node of the tree and all nodes except the root node are connected to one parent DS. Each node has zero or more descendants as shown in Figure 14. Event subscription and unsubscription requests are propagated by each DS upwards towards the root. Event notifications are processed similarly and forwarded by the local DS to its parent. Upon receiving an event, each DS checks its descendants if they have an interest in the event, and, if required, forwards the event down the tree. This strategy requires that a given DS knows the event requests of its descendants in order to make the forwarding decision. Moreover, since all requests and notifications are propagated up the tree, the communication and processing overhead of the nodes near the root may become a bottleneck. If any of the nodes near the root become disabled, parts of the tree become isolated. In this case the system needs to deal with segmentation and to be able to mend the tree or negotiate a new root and a new tree.

A JEDI event is an ordered set of strings, the first string being the name of the event followed by event parameters. An Event Dispatcher can subscribe to a single event or an event pattern. Event patterns are used to filter events based on parameter matching, for example foo(aa*,bb) matches all events named foo that have exactly two parameters and the first parameter starts with aa and the second parameter is exactly bb. JEDI preserves causal ordering of messages, that is, if event e_1 caused the firing of event e_2, e_1 must be delivered first to all interested subscribers. This mechanism allows a pair of components to synchronize through the generation of events [CDNF01].

The JEDI architecture is being extended to support mobile clients and ad-hoc configuration [CDNP00]. Publish/subscribe middleware is a good candidate for context-aware computing. Asynchronous interest-based communication is a good start for building decoupled and adaptive software components. Compositionality and reconfigurability are being emphasized, and JEDI supports mobility with moveOut and moveIn operations. One issue is run-time configuration of the dispatching system, which is also investigated by the JEDI project.

The dispatching servers in the JEDI architecture support mobility by allowing clients to disconnect, move to a new dispatching server, and connect while retaining all the notifications. The dispatching servers manage temporary storage for notifications. They also coordinate that no duplicates are received and that the notifications are causally ordered [CDN01]. The new dispatching server contacts the old one directly in order to receive the accumulated notifications. The old DS notifies its parent-dispatching server to route any further notifications for this client to the new DS. Notifications are routed in the JEDI dispatching tree from producers to consumers and there is no possibility for adapting the routing strategy to reflect changes in the pattern of communication.

The system offers good performance if the tree is organized in a good way that minimizes network traffic. In essence, when clients migrate from one dispatching server to another the load placed on the servers changes. It may be necessary to recreate the dispatching topology to reflect these changes. JEDI approaches the adaptation of publish/subscribe systems to more dynamic environments by extending the event routing mechanism with the addition of a new spanning tree routing algorithm. Now, a delegate leader is responsible for each subscription. The delegate accepts subscriptions of similar type and becomes the leader of the subscribers. It also manages the distribution of the group in the tree. Each dispatcher knows the group leaders for all subscriptions [CDNF01].

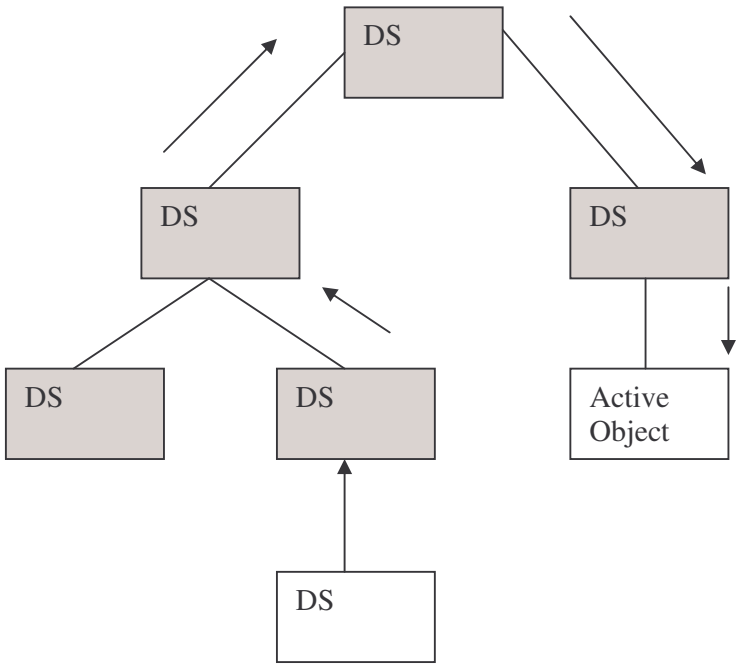

Figure 14. Event propagation in JEDI.

The JEDI approach is based on dynamically defining the dispatching tree by using approaches similar to multicast routing. The first strategy is to create a minimal spanning tree for each pair of publisher and group of subscribers, but this is considered to be inefficient. The second strategy is to have a single routing tree for each group of subscribers and have different publishers for the same class of events use the same tree. JEDI uses a method called the Core Based Tree Strategy, in which the dispatchers are connected in a possibly cyclic graph and each dispatcher knows its neighbors. Dispatchers broadcast all unique subscriptions to all servers, and all subsequent subscriptions of the same type are sent to the party that sent the original subscription. The original source dispatcher has implicitly become the leader of a group of subscribers, and it maintains access to that group. Now, the source may balance load by assigning subscriptions to dispatching servers. All dispatchers know all group leaders, and those dispatchers that belong a group know the dispatching tree of that group. When a component unsubscribes, the associated dispatcher either leaves the group, continues to route notifications, or, if it was a leader, the system needs to elect a new leader for that group.

Mobility support in JEDI is still under consideration, for example the latency of updating the dispatching trees when clients are moving very frequently and in the case of abrupt disconnections are still open issues. The current focus is on what kind of abstractions are needed at lower levels in order to detect disconnections at upper level. The scalability of the

JEDI system to Internet-wide use is an open issue. JEDI was used to implement the Orchestra Process Support System (OPSS) workflow management system (WFMS) [CDNF01].

The JEDI project ended in 2000 and Cefriel has continued to work on event architectures. They have a project on fault tolerance and scalability issues in distributed communication based on the publish/subscribe paradigm. They continue to use the JEDI event dispatchers as a reference implementation. The goal of this research is to implement a fault-tolerant JEDI. Later the JEDI subscription propagation algorithm was improved by introducing advertisements. This new algorithm is similar to the Siena work, and covering relations are used to optimize routing. The impact of advertisements was evaluated using simulation, and the results show that with advertisements the root node spends much less time processing subscriptions. The simulation results on 8–85 dispatchers indicate that the processing time of advertisements is quite low (2.65%–2.9%) [BDNFT00,BDNT00].

4.6 ECho

ECho is a high-performance data transport mechanism that is based on event channels [EBS01]. ECho uses channel-based subscriptions, similarly to the CORBA Event Service. ECho's derived event channel mechanism implements filtering by adding an application-supplied derivation function F to all listeners of a particular event channel, and by transferring all events that are generated by the sources and passed through the filters to a derived event channel. This scheme resolves issues in the delivery of unwanted events. ECho is especially optimized for streaming data and data transmission. ECho has been shown to perform better than Jini (distributed Java events), CORBA Event Channels, and XML-based messaging. ECho was developed at Georgia Tech and the source is available for academic research purposes.

4.7 JECho

JECho is a distributed event system that has been recently extended to support mobility using opportunistic event channels [CSZ03]. The central problem is to support a dynamic event delivery topology, which adapts to mobile clients and different mobility patterns. The requirements are addressed primarily using two mechanisms: proactively locating more suitable brokers and using a mobility protocol between brokers, and using a load-balancing system based on a central load-balancing component that monitors brokers in a domain. The mobility protocol is, in principle, similar to most mobility protocols (Wireless CORBA, Siena, Rebeca, . . .). The filtering model is based on stateful user-defined objects, called modulators, which may transform the event stream. This allows more fine-grained filtering than non-state-based predicate matching. However, possible security problems are not addressed, and it may be difficult to do optimizations between similar modulators. In addition, client-based filtering is not addressed and it may also be difficult to implement efficiently. For example, a mobile producer should download all relevant modulators from the broker. Furthermore, no session management is provided so all user-specific modulators are relocated.

The system supports load balancing and resource monitoring, which are novel features for mobility-aware event systems. The paper presents simulation results for different scenarios, for example, relocation overhead and mobility patterns. Mobility patterns are examined in a 100-node network using BRITE and the evaluation includes scenarios such as random walk, salesman, pop-up, and fixed. Moreover, end-to-end delay and mobility/communication ratio are measured using a real system with two subnets.

4.8 Rebeca

Rebeca is a distributed event system that supports mobile users and context-aware subscriptions [FGKZ03]. The system supports both logical and physical mobility. The basic system is an acyclic routed event network using advertisement semantics. The mobility protocol uses an intermediate node between the source and target of mobility, called Junction, for synchronizing the servers. If the brokers keep track of every subscription, the Junction is the first node with a subscription that matches the relocated subscription propagated from the target broker. If covering relations or merging are used this information is lost, and the Junction needs to use content-based flooding to locate the source broker. A merging system was developed in the Rebeca project for conjunctive filters.

4.9 Gryphon

The Gryphon system was developed at the Distributed Messaging Systems group at the IBM T.J.Watson Research Center. Gryphon is a Java-based publish-subscribe message broker intended to distribute data in real time over a large public network. Gryphon uses content-based routing algorithms developed at the research center. The clients of Gryphon use an implementation of the JMS API to send and receive messages. The Gryphon project was started in 1997 to develop the next generation web applications and the first deployments were made in 1999. Gryphon is designed to be scalable, and it was used to deliver information about the Tennis Australian Open to 50000 concurrently connected clients. Gryphon has also been deployed over the Internet for other real-time sports score distribution, for example the Tennis US Open, Ryder Cup, and monitoring and statistics reporting at the Sydney Olympics.

Gryphon supports both topic-based and content-based publish-subscribe, relies on adopted standards such as TCP/IP and HTTP, and supports recovery from server failures and security. In Gryphon, the flow of streams of events is described using an information flow graph (IFG), which specifies the selective delivery of events, the transformation of events, and the creation of derived events as a function of states computed from event histories.

Information flow graphs contain stateless event transforms that combine events from various sources, and stateful event interpretation functions that can be used to derive trends, alarms, and summaries from published events. Each event is a typed tuple. Stateful events depend on the event history. States are used to express the meaning of an event stream and the equivalence of two event streams.
The Gryphon model consists of information spaces, which are either event histories or states. Event histories grow monotonically over time as new events are published. Event sources and sinks are modeled as event histories. States capture certain relevant information about event streams, and they are typically not monotonic. Information spaces are defined using information schemas. Dataflows are directed arcs that connect nodes in the graph, which needs to be acyclic [BKS+99].

Gryphon supports four types of dataflows. Select is an arc that connects two event histories with the same schema. Each arc is a predicate on the attributes of the event type in the information space. All events that satisfy the constraint are delivered to the destination information space. The transform arc connects any two event histories that may have different schemas. Each arc has a rule for mapping event types between the two spaces. This rule may include functions that transform particular event attributes. The collapse arc connects an event history to a state using a rule. The rule maps a new event and a current state into a new state. The expand arc is the inverse of collapse, and links a state to an information space.

When the state at the source of the arc changes, the destination space is updated in such as way that the sequence of events it contains collapses to the new state. This transformation is non-deterministic. Gryphon has two techniques for the implementation of systems based on IFGs. The first is a flow graph rewriting optimization that allows stateless IFGs to be used with multicast technology. The second is an algorithm for converting a sequence of events to the shortest equivalent sequence of events.

The information flow graph is abstract and separated from the physical topology of the network. The mapping of an IFG to a network of message brokers is nontrivial. Gryphon reduces an arbitrary IFG by rewriting it. All the select operations are moved together and closer to publishers and all the transform operations are also grouped together closer to the subscribers. Transform operations are done at the periphery of the network.

The Gryphon system allows the representation of event histories as states, which is interesting especially for mobile and disconnected users. Wireless users would benefit if a system could inform them with a summary of events that occurred while they were disconnected (the state). The Gryphon system detects failed brokers and reroutes traffic around failed nodes. Moreover, the system incorporates several security mechanisms, such as access control, and four authentication methods. Gryphon supports the JMS publish/subscribe API, and supports topic-based subscription. In addition, clients may specify filters using the WHERE-clause of SQL92 supported by JMS.

Gryphon extends the publish/subscribe one-to-many model with request-reply and solicit-response models. By using unique topics JMS users can use request-reply-style messaging. In the solicit-response model a client may make an advertisement to which one or several clients may respond privately. The basic unit of the Gryphon multi-broker configuration is the cell, which is a group of fully connected servers. Cells may be further linked together for geographical scaling through link bundles. Link bundles provide redundant connections between cells, which includes load balancing and fault tolerance not provided by gateway-based approaches. The internal protocols and systems ensure that cycles are avoided and messages are routed around failed nodes.

4.10 STEAM

The STEAM (Scalable Timed Events and Mobility) event system is specifically designed for wireless ad-hoc networks [MC03]. The system uses three different filters to address the problems related to dynamic reconfiguration of the network topology. Specifically, the STEAM system is intended for WLANs using the ad-hoc network model, and the main application domain is traffic management. The system uses an implicit event model in which entities subscribe to interesting event types locally, and not by using a centralized broker. STEAM exploits a group communication service for notifying interested entities. Groups are geographically bound and nodes are identified using beacons.

The three filter types supported by STEAM are subject, proximity, and content filters. Events consist of a name and a set of typed parameters. The name also determines the structure of the event. A subject filter is matched against the event and mapped onto a proximity group. A proximity filter corresponds to the geographical aspect of the proximity group. A proximity filter specifies the scope in which events are disseminated. A proximity filter applies to an event type and is established when the type is deployed. In essence, upon publication of an event the source matches the subject and proximity, and the subscribers match the content. This requires that the proximity filter at the producer must have location information from the subscriber. The paper does not explain how this information is acquired, how often it is updated, and how the security implications are handled. In essence the protocol is a wireless

application-level broadcast protocol with subject-based filtering at the source and content-based filtering at the client.

Producers announce the event types they intend to raise (publish) with the geographical area, called the proximity, within which events of this type are to be disseminated. The proximities may be defined independently of the physical range of the communication system. The routing layer may support multi-hop communication.

STEAM uses a Proximity-based Group Communication Service (PGCS). In this service, groups are assigned certain geographical areas. A node that wants to join a group needs to be located in the group's area. STEAM provides a Proximity Discovery Service (PDS) that uses beacons to discover proximities. Once a proximity is discovered the associated events are delivered to the client if it has a matching subscription. PDS causes the middleware to join a group if either a subscription or an announcement matches the group. The proximities are static but clients may move. An experimental scenario is presented in [MC03]: traffic lights at an intersection with experimental results with and without filtering. The results suggest that distributed filtering, although simple in this case, is beneficial in ad-hoc environments and may reduce the amount of transmitted traffic significantly.

4.11 Rapide

The Rapide language is designed to meet the requirements for architectural definition [LV95]. The main idea of Rapide is to use asynchronous events and their causal relations to model both static and dynamic architectures. In this context, an architecture consists of interfaces, connections, and constraints — an interface connection architecture. When the architecture specification is executed all causal relations are stored and checked against the constraints. The key requirements for the system were component abstraction, communication abstraction, communication integrity, dynamicity, causality and time, hierarchical refinement, and relativity. The interdependencies of Rapide components are modeled using partially ordered sets. A pattern language is defined for detecting composite events. An event of a particular action is a tuple of information with a unique identifier, a timestamp, and dependency information. The system supports placeholders and universal quantification over types. Patterns are used in interfaces to define behaviors and in architectures to define connections.

4.12 DADI

DADI (Discovery, Analysis and Dissemination of Information) is a research project at Princeton that focuses on discovery, analysis and information dissemination on the Internet. The project investigates the event-based model with emphasis on wide-area pub/sub. The DADI effort includes a number of subprojects that cover the different layers of operation, namely the system, algorithm, and application layers [CLS03, CSi04, CSi05].

The system layer encompasses architecture design for pub/sub and subscription-based content delivery. The algorithm layer covers routing and matching algorithms for content-based pub/sub. The application layer pertains to internet-scale persistent search.

4.13 Hermes

Hermes [PBa02,Pie04] is a peer-to-peer event system based on an overlay called Pan that supports a variant of the advertisement semantics. Hermes leverages the features of the underlying overlay system for message routing, scalability, and improved fault-tolerance. Hermes supports the basic pub/sub operations introduced previously. Rendezvous points (RP) are used to coordinate advertisement and subscription propagation. The RP manages an event type and Hermes supports chaining RPs into type hierarchies. The RP of an event type is obtained by hashing the event type to the flat addressing space of the overlay.

In type-based routing, any events conforming to the advertisement from the publisher are sent on the forward path of the advertisement to the RP, which then forwards events on the reverse path of any subscriptions. In type/attribute-based routing, the RP sends the subscriptions on the reverse path of advertisements. Any events conforming to the advertisement from the publisher are sent on the reverse path of subscriptions.

The model used by the Hermes system is the familiar advertisement semantics model used in Siena and Rebeca with three key differences:

- All messages (type-based routing) or advertisements and subscriptions (type/attribute-based routing) are sent towards the RP. Thus routing topology is constrained by the RP.
- Advertisements are introduced only on the path from the advertiser to the RP.
- Subscriptions are introduced on the path from the subscriber to the RP. In addition, for type/attribute-based routing subscriptions are sent on the reverse path of any overlapping advertisements.

4.14 Fuego Event Service

The Fuego event service was developed in the Fuego Core 2002/2004 project at the Helsinki Institute for Information Technology HIIT[1]. The event service addresses the challenges in the mobile computing environment by providing an asynchronous content-based publish/subscribe system that supports client mobility [Tar05].

A key component of the architecture is the Fuego event router. The event router is a component that connects the publishers and subscribers and mediates event messages between them. Typically, an event router consists of two parts: a set of neighbouring routers and a set of local clients. Both sets are associated with a routing table that contains information about which event messages should be forwarded to which neighbouring router or local client.

The event service and event router are part of the Fuego middleware service set, which realizes a set of generic service elements pertaining to communication and data synchronization. High-level services use the extendable messaging and RPC facilities provided by the messaging service. The Host Identity Protocol implementation for Linux is an optional component of the architecture, which is used for secure mobility and multi-homing support. The HIP architecture is currently being standardized by IETF and it defines a new cryptographic namespace between the network (L3 in OSI) and the transport (L4) layers. The Fuego middleware has a Java API for HIP, which allows applications to use HIP features [KTK05].

[1] http://www.hiit.fi/fuego/fc

The primary data representation format of the system is XML according to the XML Infoset specification. Since XML parsing is a time consuming activity, and XML documents are not very space-efficient, a more efficient XML encoding is used for transmitting most XML content. Use of XML lead to the selection of SOAP [W3C03b] as the primary communication protocol in the architecture, transported using optimizations for wireless links. SOAP is used for one-way and request-response messaging for relatively small XML documents and fragments, and HTTP is used for bulk transfers of data. The current implementation supports two message transport protocols: HTTP 1.1 and BEEP. BEEP supports reliability, authentication, and the prioritization of messages.

The event service for mobile computing consists of two parts: the client-side API, and the server-side system. The client-side API is similar in functionality to JMS and offers the basic publish/subscribe functionality and session management. The server-side provides an extensible framework for content-based routing with optimizations and mobility support. The generic router implementation allows pluggable routing algorithms and routing table user interfaces.

Events are represented according to the XML Infoset specification. All remote API calls use the SOAP request-response protocol, and the notification of events uses asynchronous SOAP messaging. The event router is implemented as an Apache Axis web service. The client side API implementation uses wireless SOAP [KLT05] by default; however, a lightweight version of the API was created for J2ME end systems that uses HTTP 1.1 and a proprietary binary message format.

Filtering is a central core functionality for realizing event-based systems and accurate content-delivery. Filtering may be optimized by using covering relations or filter merging [TKa05b]. The event service supports both through new generic data structures for content-based routing. Any objects that implement methods for covering and merging are automatically optimized. The system has a default filtering language, which is based on typed tuples and disjunctive normal form based attribute-filters. Typically, events and filters are represented as lists of typed tuples. In this case, an attribute filter is a 3-tuple <name, type, constraint>.

The router toolkit does not specify any particular routing topology, but rather allows the developers to use the generic API methods available, leverage the efficient data structures for various configurations, and develop various mobility protocols. As an example, the Fuego router toolkit has been used to implement an event channel based configuration.

The server-side system consists of a set of event routers. Each router has two components: a local routing table and a remote routing table. The local routing table stores filters set by local clients and provides queue management for mobile and wireless clients. Disconnected clients may retrieve queued events upon reconnection using push or pull semantics. The remote routing table is responsible for communicating with other routers and forwarding events in the distributed system. In order to support extensibility, the local and remote routing tables and algorithms are separate objects, which may be changed if necessary.

This modularity allows the implementation of various distributed event routing semantics and router topologies. Subscription semantics and advertisement semantics are examples of two different interest propagation mechanisms. The former propagates subscription messages throughout the system and events are routed on the reverse-path of subscriptions. In the latter, advertisements are propagated throughout the system and subscriptions are routed on the reverse-path of advertisements. Supported routing topologies include hierarchical, event channel, and peer-to-peer topologies. The system also has separate user interface modules for the routing tables.

The client-side API supports expressive operation with three mechanisms: multiple sessions, expressive pull functionality, and fast subscriptions. The first approach allows clients to create multiple sessions at different access servers for subscriptions with different maximum queue sizes and delivery options. The client may have several sessions at different access servers, for example to support different modes of operation. The second approach is realized by pulling the notifications that match subscription identifiers or arbitrary filters. Thus the client may subscribe different events, which are stored in the session – and when running on a small client only the essential events may be retrieved using the pull operation.

5 Recent Research Areas in Publish/Subscribe

Research in publish/subscribe has recently gained popularity and current active reasearch areas include mobility support, dynamic and peer-to-peer systems, formal modelling of pub/sub systems, and security issues. In this chapter we briefly discuss these research areas.

5.1 Mobility Support

Mobility support [HGM01,PHJ02] is a relatively new research topic in event-based computing. Mobility is an important requirement for many application domains, where entities change their physical or logical location. Mobile IP is a layer-3 mobility protocol for supporting clients that roam between IP networks [JPA04]. Higher-level mobility protocols are also needed in order to provide efficient middleware solutions, for example SIP (Session Initiation Protocol) mobility [SWe00]. Event-based systems require their own mobility protocols in order to update the event-routing topology and optimize event flow.

JEDI was one of the early systems to incorporate support for mobile clients with the move-in and move-out commands. JEDI maintains causal ordering of events and is based on a tree-topology, which has a potential performance bottleneck at the root of the tree with subscription semantics. Elvin is an event system that supports disconnected operation using a centralized proxy, but does not support mobility between proxies.

Siena, Rebeca, and Hermes [PBa02,Pie04] support content-based routing of events using covering relations. To our knowledge, covering relations were first introduced in the Siena project and they support the optimization of event-based communication. Recently, mobility extensions have been presented for several well known distributed event systems, such as Siena and Rebeca.

Siena is a scalable architecture based on event routing that has been extended to support mobility [CCW03]. The extension provides support for terminal mobility on top of a routed event infrastructure. In addition, the Rebeca event system supports mobility in an acyclical event topology with advertisement semantics [FGKZ03]. Context-aware subscriptions have also been investigated in the Rebeca project.

Rebeca supports both logical and physical mobility. The basic system is an acyclic routed event network using advertisement semantics. The mobility protocol uses an intermediate node, between the source and target of mobility, called Junction for synchronizing the servers. If the brokers keep track of every subscription the Junction is the first node with a subscription that matches the relocated subscription propagated from the target broker. If covering relations or merging is used this information is lost, and the Junction needs to use content-based flooding to locate the source broker [MUH04].

JECho is a mobility-aware event system that uses opportunistic event channels in order to support mobile clients [CSZ03]. The central problem is to support a dynamic event delivery topology, which adapts to mobile clients and different mobility patterns. The requirements are addressed primarily using two mechanisms: proactively locating more suitable brokers and using a mobility protocol between brokers, and using a load-balancing system based on a central load-balancing component that monitors brokers in a domain.

Mobility support in a generic routed event infrastructure, such as Siena and Rebeca, is challenging because of the high cost of the flooding and issues with mobile publishers. The standard state transfer protocol consists of four phases:

1. Subscriptions are moved from A to B.
2. B subscribes to the events.
3. A sends buffered notifications to B.
4. A unsubscribes if necessary.

The problem with this protocol is that B may not know when the subscriptions have taken effect – especially if the routing topology is large and arbitrary. This is solved by synchronizing A and B using events, which potentially involves flooding the content-based network.

Recent findings on the cost of mobility in hierarchical routed event infrastructures that use unicast include that network capacity must be doubled to manage with the extra load of 10% of mobile clients [BJL04]. Recent findings also present optimizations for client mobility: prefetching, logging, home-broker, and subscriptions-on-device. Prefetching takes future mobility patterns into account by transferring the state while the user is mobile. With logging, the brokers maintain a log of recent events and only those events not found in the log need to be transferred from the old location. The home-broker approach involves a designated home broker that buffers events on behalf of the client. This approach has extra messaging costs when retrieving buffered events. Subscriptions-on-device stores the subscription status on the client so it is not necessary to contact the old broker. In this study the cost of reconfiguration was dominated by the cost of forwarding stored events (through the event routing network).

The cost of publisher mobility has also been recently addressed [MPJ05]. They start with a basic model for publisher mobility that simply tears down the old advertisement and establishes it at the new location after mobility. Thus a specific handover protocol is not needed. They confirm the high cost of publisher mobility and present three optimization techniques, namely prefetching, proxy, and delayed. The first exploits information about future mobility patterns. The second uses special proxy nodes that advertise on behalf of the publisher and maintain the multicast trees. The third delays the unadvertisement at the source to exploit the overlap of advertisements, but does not synchronize the source and target brokers.

A formal discrete model for both publisher and subscriber mobility was presented in [Tar05, Tka05a, Tka05c]. In this work, two new properties are defined for the pub/sub topology, namely mobility-safety and completeness. A handover protocol is mobility-safe if it prevents false negatives. A topology or a part of a topology is complete if subscriptions and advertisements are fully established (propagated) throughout it.

Mobility-safety of a generic stateful handover is shown in acyclic pub/sub networks. The completeness of the topology is used to characterize pub/sub handover protocols and optimize them. One of the results of this work is that rendezvous-points are good for pub/sub mobility, because they can be used to limit signalling and flooding of updates.

5.2 Dynamic and Peer-to-Peer Systems

A number of overlay-based routing algorithms and router configurations have been proposed. An application layer overlay network is implemented on top of the network layer and typically overlays provide useful features such as fast deployment time, resilience and fault-tolerance. An overlay-routing algorithm leverages underlying packet-routing facilities and

provides additional services on the higher level, such as searching, storage, and synchronization services.

Good overlay routing configuration follows the network level placement of routers. Many overlays are based on Distributed Hash Tables (DHTs), which are typically used to implement distributed lookup structures. Many DHTs work by hashing data to routers/brokers and using a variant of prefix-routing to find the proper data broker for a given data item. Hermes [Pie04] and Scribe [RKC01] are examples of publish/subscribe systems implemented on top of an overlay network and are based on the rendezvous point routing model. The Hermes routing model is based on advertisement semantics and an overlay topology with rendezvous points. This model was found to perform better than the acyclic topology [Pie04].

Typical fixed-network pub/sub routing algorithms are deterministic in nature. Basic routing algorithms do not cope with topology changes. Dynamic connections between routers have been investigated in [PCM03]. Recently, probabilistic algorithms have also been proposed for better routing support in peer-to-peer and ad hoc environments [CPi05].

A topic-based multicast algorithm for peer-to-peer event dissemination is presented in [BEG04]. The algorithm is "data-aware" in the sense that it exploits information about process subscriptions and topic inclusion relationships. This "data-awareness" is used to limit the membership information that each process needs to maintain. The paper discusses tradeoffs between message complexity and reliability.

Gossip-based broadcast algorithms form a family of broadcast algorithms that contrast reliability guarantees with scalability properties. A lightweight probabilistic broadcast, lpbcast, is a decentralized gossip-based broadcast algorithm that offers scalability in terms of throughput and memory-management [EGH03]. Decentralization means that the algorithm is based only on local information.

The construction of an optimal pub/sub dissemination tree for routing information from source to interested recipients was analyzed in [HGM03]. A greedy algorithm was proposed as a solution to the tree building problem that builds the three in a fully distributed fashion.

A protocol for content-based message dissemination for mobile ad hoc networks (MANETs) was presented in [BBC05] with simulation results. The protocol uses broadcast to send a message to neighbour nodes. The forwarding decision made by these neighbours is based on an estimation of their distance from a potential subscriber of the message.

An event dissemination algorithm for a topic-based pub/sub abstraction in MANETs is presented in [BCG05]. The algorithm relies on the mobility of processes and the validity period of events to ensure reliable dissemination. A general deterministic information diffusion scheme was proposed in [ADG02]. The three main features of this scheme are support for network reorganization, anonymity support, and decentralization.

5.3 Formal Modelling

Formal modelling of publish/subscribe systems and the correctness of content-based routing protocols were examined in [Müh02]. A routing protocol is correct if it maintains required safety and liveness properties. Since it may be difficult to maintain these properties in dynamic pub/sub systems they may be relaxed. A self-stabilizing pub/sub system ensures correctness of the routing algorithm against the specification and convergence [Müh02]. The safety property may be modified to take self-stabilization into account by requiring eventual safety.

The safety and liveness properties were extended in [TMü04] with the notion of message-completeness and using propositional temporal logic. A message-complete pub/sub system eventually acknowledges subscriptions and guarantees the delivery of notifications matching acknowledged subscriptions.

A formal framework for modelling pub/sub systems is presented in [BBP05]. The framework is based on two delays, namely the subscription/unsubscription delay and the diffusion delay. The motivation for this abstraction is to model concurrent execution of the system without waiting for the stability of the system state. This work differs from the previous liveness and safety properties, because they focus on analytically to characterize the quality of the system.

A formal discrete model for pub/sub mobility is investigated in [Tar05, Tka05a, Tka05c].

5.4 Security

Security is also a recent research area for pub/sub systems and a fundamental requirement for any real deployments of event systems. Typically, security requirements have not been taken into account in research prototypes of systems. This has created the necessity for a number of security services for pub/sub. These services are built on top of existing solutions and rely on symmetric and asymmetric cryptography.

An overview of pub/sub security topics was given in [WCE02]. They propose several techniques for ensuring the availability of the information dissemination network. Prevention of denial-of-service attacks is essential and customized publication control is proposed to mitigate large-scale attacks. In this technique, subscribers can specify which publishers are allowed to send them information. A challenge-response mechanism is proposed, in which the subscriber issues a challenge function, and the publisher has to respond to the challenge. The use of the mechanism in a distributed environment was not elaborated.

General pub/sub security has been addressed recently, especially requirements, authentication, confidentiality, and payment processing. Security-aware pub/sub systems include Hermes [BEP03], EventGuard [SLi05], and Rebeca [FZB04]. The possibility of unsolicited bogus messages originating from subscribers and producers is addressed in EventGuard. The EventGuard system comprises of a set of security guards to secure pub/sub operations, and a resilient pub/sub network [SLi05]. The basic security building blocks are tokens, keys, and signatures. Tokens are used within the pub/sub network to route messages, which is not directly applicable for content-based routing.

Pub/sub broker networks are vulnerable to message dropping attacks. For example, overlays such as Hermes and Maia [PBa05] may suffer from bogus nodes. The prevention of message dropping attacks has a high cost and only a few systems address them. The EventGuard uses an r-resilient network of brokers [SLi05].

Secure event types and type-checking was proposed in [PBa05]. Secure event type definitions contain issuer's public key, version information, attributes, delegation certificates, and a digital signature. We believe that secure event types and schemas are important for spam prevention. Scope-based security was discussed in [FZB04], in which trust networks are created in the broker network using PKI techniques. A proxy-based security and accounting solution was proposed in [Khu05] for untrusted broker networks.

6 Conclusions

Message-oriented middleware and event notification are becoming more popular in the industry with the advent of the CORBA Notification Service and DSS, the Java Messaging Service, and other related specifications and products from many vendors. Many research projects have addressed and are addressing issues of scalability, compound event detection, mobility, and fault tolerance, to name a few topics. There are many ways to classify event systems, and many possibilities for their use depending on the requirements. Traditional MOM systems are getting influences from event-based systems. For instance, JMS supports both queues and publish-subscribe style communication with filtering. However, these systems usually lack support for distributed coordination in notification delivery, and they employ topic-based routing. Current event systems are evolving towards content-based routing, which uses the whole notification as an address. In content-based systems clients can change their interests without changing the addressing scheme (adding a new topic).

Scalability has been emphasized in Siena, and it has been designed for Internet-wide scalability and tested in a simulation environment with various network topologies. However, scalability introduces latency, which creates problems for notification semantics and mobility. Other systems address scalability and fault tolerance by creating clusters (Elvin) or cells (Gryphon) that contain connected servers. These clusters are connected using point-to-point links and possibly different protocols. Multicast and fault tolerance can be provided within the clusters. Event systems are logically centralized, however, the CORBA Event Channel is also physically centralized, creating a possible bottleneck.

Recently, creating event systems on top of DHTs and overlays has become an active area of research. Systems such as Scribe and Hermes use the properties of the underlying overlay to provide separation of concerns over basic message delivery in peer-to-peer environments, and event dissemination. For example, DHTs are useful for building multicast trees rooted at rendezvous-points, as demonstrated by Hermes and Scribe.

Ad-hoc networks are emerging with the introduction of short-range radio communications. Ad-hoc event systems support the dynamic addition and removal of event servers (or event dispatchers). However, ad-hoc event topologies are currently an emerging research topic and some research issues have been raised in JEDI. STEAM supports ad-hoc event dissemination and proximity groups. Many research papers also address multicast information delivery in MANETs and the challenges posed by node mobility.

From the mobile Internet and ubiquitous computing viewpoint JEDI and Elvin were one of the first systems to examine support for disconnected operation. JEDI supports both mobility and disconnected operation as a service and Elvin only disconnected operation (with a few additional features) as an extension to the original architecture. Almost all message queue products support disconnection with various semantics. One important decision is whether to include this support as an extension or as an integral part of the event service. If fault tolerance or mobility are to be supported, it may be necessary to integrate this functionality at the service level. Another open issue is whether the event service should reside at the network level or at the application level. For Internet-scale routing, as proposed in Siena, it might be beneficial to have some support at the network level. Siena has recently been extended to support mobile clients, and JECho and Rebeca also have a handover procedure for relocating subscriptions. Content-based routing and mobility is a challenging combination and most routed event systems that use covering relations or filter merging may need to use flooding in the mobility protocol.

Only a few architectures support complex compound event filtering. Usually event filtering is done using simple parameter wildcard matching (JEDI), simple clauses (COM+, Elvin), SQL (JMS, Gryphon), or Extended TCL (CORBA). Compound event detection is supported in CEA with event templates and in Siena by detecting a sequence of simple filters. Compound event detection is also a feature that may be integrated as an external component or within the infrastructure. Many systems do not consider the process of locating and connecting producers, or locating event channels (Notification Service). Some architectures, such as Siena, JEDI, and Elvin, support this within the infrastructure. There are two completely different problems: one is locating an access point using multicast or unicast to a known address, and another is to use either the infrastructure or some other service to locate channels or to subscribe. In systems such as Siena and JEDI, subscribing is accomplished by using simple string-format requests. With CORBA it is necessary to obtain an event channel and go through a more complicated procedure in order to obtain references to proxy objects. This process of obtaining the event channel reference according to interests is not specified. OMG DSS addresses this issue by making the discovery of topics easier.

Many message queue products are now supporting XML-based solutions, such as SOAP, as one of the transport options. MQSeries, MSMQ, and .NET support SOAP, and Siena has XML bindings as well. XML has many applications in messaging and event-based communication. XML can be used to define the content of messages. For example, JMS facilitates XML-based messages and the routing of XML documents.

The building blocks of the semantic web, such as ontologies, are not yet supported in pub/sub systems. Ontologies and XML-derived languages could well be used to define events and event systems, and to improve interoperability. XML and a suitable ontology would enable the specification of complex event monitoring tasks that are uploaded to routers or, for example, web services. In the future, it is envisaged that event applications have policies (specified in XML, for example) for event semantics, access control and authorization, buffering, and other information that affects the delivery of notifications.

References

[ADG02] Anceaume, E., Datta, A.K., Gradinariu, M., Simon, G. Publish/subscribe scheme for mobile networks. In: Proceedings of the 2002 Workshop on Principles of Mobile Computing (POMC 2002).

[Bar01] Dave Bartlett. CORBA junction: CORBA 3.0 notification service, May 2001. http://www-106.ibm.com/developerworks/webservices/library/co-cjct8/.

[BBC05] Baldoni, R., Beraldi, R., Cugola, G., Migliavacca, M., Querzoni, L. Structureless Content-Based Routing in Mobile Ad Hoc Networks. In: In proceedings of the International Conference on Pervasive Services (ICPS05), Santorini, Greece, July 2005.

[BBP05] Baldoni, R., Beraldi, R., Piergiovanni, S.T., Virgillito, A. On the modelling of publish/subscribe communication systems. Concurrency and Computation: Practice and Experience 17 (2005) 1471-1495.

[BCG05] Baehni, S., Chhabra, C., Guerraoui, R. Frugal Event Dissemination in a Mobile Environment. In the ACM/IFIP/USENIX 6th International Middleware Conference, November 2005.

[BEP03] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based access control for publish/subscribe middleware architectures. In Proceeding of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03), ACM SIGMOD, San Diego, CA, U.S.A., 2003.

[BDNFT00] Giovanni Bricconi, Elisabetta Di Nitto, Alfonso Fuggetta, and Emma Tracanella. Analyzing the behavior of event dispatching systems through simulation. In Proceedings of the 7th International Conference on High Performance Computing, pages 131–140, December 2000.

[BDNT00] Giovanni Bricconi, Elisabetta Di Nitto, and Emma Tracanella. Issues in analyzing the behavior of event dispatching systems. In Proceedings of the 10th International Workshop on Software Specification and Design, page 95, November 2000.

[BEG04] Sébastien Baehni, Patrick Th. Eugster, and Rachid Guerraoui. Data-Aware Multicast (In Proceedings of the 5th IEEE International Conference on Dependable Systems and Networks (DSN' 04), pages 233-242, June 2004.

[BJL04] I. Burcea, H.-A. Jacobsen, E. de Lara, V. Muthusamy, and M. Petrovic. Disconnected operation in publish/subscribe middleware. In Mobile Data Management. IEEE Computer Society, 2004.

[BKS+99] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, andWei Tao. Information flow based event distribution middleware. In Wei Sun, Sam Chanson, Doug Tygar, and Partha Dasgupta, editors, ICDCS Workshop on Electronic Commerce andWeb-based Applications, pages 114–121, June 1999.

[BMH+00] Jean Bacon, KenMoody, Richard Hayton, et al. Generic support for distributed applications. IEEE Computer, 33(3):68–76, March 2000.

[BMT04] BEA, Microsoft, TIBCO. Web Services Eventing (WS-Eventing), August 2004. Available at: http://www-128.ibm.com/developerworks/webservices/library/specification/ws-eventing/

[CCW03] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. IEEE Transactions on Software Engineering, 29(12):1059– 1071, Dec. 2003.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communication, 20(8), October 2002.

[CDN01] Gianpaolo Cugola and Elisabetta Di Nitto. Using a publish/subscribe middleware to support mobile computing. In Middleware for Mobile Computing Workshop, November 2001.

[CDNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Transactions on Software Engineering, 27(9):827–850, September 2001.

[CDNP00] Gianpaolo Cugola, Elisabetta Di Nitto, and Gian Pietro Picco. Content-based dispatching in a mobile environment. In Workshop su Sistemi Distribuiti: Algorithmi, Architectture e Linguaggi, September 2000.

[CDW01] Antonio Carzaniga, Jing Deng, and Alexander L. Wolf. Fast forwarding for content-based networking. Technical Report CU-CS-922-01, Department of Computer Science, University of Colorado, November 2001.

[CRW99] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, October 1999. Revised May 2000.

[CRW01] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluationof a wide-area event notification service. ACM Transactions on Computer Systems, 19(3):332–383, Aug. 2001.

[CLS03] Mao Chen, Andrea LaPaugh, and J.P. Singh. Content Distribution for Publish/Subscribe Services. In Proceedings of ACM/IFIP/USENIX International Middleware Conference 2003, pp. 83 - 102.

[CSi04] Fengyun Cao and Jaswinder Pal Singh. Efficient Event Routing in Content-based Publish-Subscribe Service Networks. In Proc. of IEEE INFOCOM 2004.

[CSi05] Fengyun Cao and Jaswinder Pal Singh. MEDYM: Match-Early and Dynamic Multicast for Content-based Publish-Subscribe Service Networks. In the 6th ACM/IFIP/USENIX International Middleware Conference, 2005.

[CSZ03] Yuan Chen, Karsten Schwan, and Dong Zhou. Opportunistic channels: Mobility-aware event delivery. In ACM/IFIP/USENIX International Middleware Conference 2003,pages 182–201, June 2003.
http://link.springer.de/link/service/series/0558/bibs/2672/26720182.htm.

[CSZ03] Y. Chen, K. Schwan, and D. Zhou. Opportunistic channels: Mobility-aware event delivery. In Middleware 2003, pages 182–201.

[CW01] Antonio Carzaniga and Alexander L.Wolf. Content-based networking: A new communication infrastructure. In NSF Workshop on an Infrastructure for Mobile and Wireless Systems, October 2001.

[CW03] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In Proceedings of 2003 conference on Applications, technologies, architectures, and protocol for computer communications, pages 163–174, August 2003.

[EBS01] Greg Eisenhauer, Fabián Bustamante, and Karsten Schwan. A middleware toolkit for client-initiated service specialization. ACM SIGOPS Operating Systems Review, 35(2):7–20, April 2001.

[EFG03] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. ACM Comput. Surv., 35(2):114–131, 2003.

[EGH03] P. Th. Eugster, R. Guerraoui, S.B. Handurukande, P. Kouznetsov, and A.-M. Kemarrec. Lightweight Probabilistic Broadcast, in ACM Transactions on Computer Systems (TOCS), 21(4), pages 341-374, November 2003.

[Ere01] Justin R. Erenkrantz. Handling hierarchical events in an internet-scale event service, March 2001. http://www.ucf.ics.uci.edu/~jerenk/siena-xml/SienaPaper.html.

[FGKZ03] Ludger Fiege, Felix C. Gartner, Oliver Kasten, and Andreas Zeidler. Supporting mobility in content-based publish/subscribe middleware, June 2003. http://citeseer.nj.nec.com/kasten03supporting.html.

[FZB04] L. Fiege, A. Zeidler, A. P. Buchmann, R. Kilian-Kehr, and G. Mühl. Security aspects in publish/subscribe systems. In Third Intl. Workshop on Distributed Event-based Systems (DEBS'04), Edinburgh, Scotland, UK, May 2004. IEE The Institution of Electrical Engineers.

[GCSO01] Pradeep Gore, Ron Cytron, Douglas Schmidt, and Carlos O'Ryan. Designing and optimizing a scalable CORBA notification service. ACM SIGPLAN Notices, 36(8):196–204, August 2001.

[Hei01] Dennis Heimbigner. Adapting publish/subscribe middleware to achieve Gnutella-like functionality. In Proceedings of the 2001 ACM Symposium on Applied Computing, pages 176–181, March 2001.

[HGM01] Huang, Y., Garcia-Molina, H. Publish/Subscribe in a Mobile Environment. In: Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), 2001.

[HGM03] Huang, Y., Garcia-Molina, H. Publish/subscribe tree construction in wireless ad-hoc networks. In: 4th International Conference on Mobile Data Management (MDM 2003).

[IBM02a] IBM. Gryphon: Publish/subscribe over public networks., December 2002. (White paper). Available at: http://www.research.ibm.com/distributedmessaging/gryphon.html

[IBM02b] IBM. MQSeries Everyplace for Multiplatforms Version 1, Release 2, 2002. (White paper).

[JPA04] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. IETF, June 2004. [Standards Track RFC 3775].

[Khu05] H. Khurana. Scalable security and accounting services for content-based publish/subscribe systems. In SAC '05: Proceedings of the 2005 ACM symposium on Applied computing, pages 801–807, New York, NY, USA, 2005. ACM Press.

[Kis01] Roman Kiss. Using the COM+ event system in .Net applications, September 2001. http://www.codeproject.com/csharp/solutionlcenotification.asp.

[KLT05] J. Kangasharju, T. Lindholm, and S. Tarkoma. Requirements and design for XML messaging in the mobile environment. In N. Anerousis and G. Kormentzas, editors, Second International Workshop on Next Generation Networking Middleware, pages 29–36, May 2005.

[KTK05] Miika Komu, Sasu Tarkoma, Jaakko Kangasharju, Andrei Gurtov. Applying a Cryptographic Namespace to Applications. Dynamic Interconnection of Networks (DIN 2005) ACM workshop in conjunction with Mobicom 2005.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. IEEE Transactions on Software Engineering, 21(9):717–734, September 1995. http://citeseer.nj.nec.com/luckham95eventbased.html

[MC03] René Meier and Vinny Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In 4th International Conference on Distributed Applications and Interoperable Systems, pages 285–296, November 2003.

[Mei00] René Meier. State of the art review of distributed event models. Technical Report TCD-CS-2000-15, Department of Computer Science, Trinity College, Dublin, Ireland, March 2000.

[Mic99] Microsoft. Message Queuing on Windows CE, June 1999. Windows CE Developers Conference, http://www.microsoft.com/msmq/downloads/devcon99.ppt.

[Mic02] Microsoft. Message Queuing in Windows XP: New Features, 2002. (White paper) http://www.microsoft.com/msmq/MSMQ3.0_whitepaper_draft.doc

[MPJ05] V. Muthusamy, M. Petrovic, and H.-A. Jacobsen. Effects of routing computations in content-based routing networks with mobile data sources. In MobiCom '05: Proceedings of the 11th annual international conference on Mobile computing and networking, pages 103–116, New York, NY, USA, 2005. ACM Press.

[Müh02] G.Mühl. Large-Scale Content-Based Publish/Subscribe Systems. PhD thesis, Darmstadt University of Technology, September 2002.

[MUH04] G. Mühl, A. Ulbrich, K. Herrmann, and T. Weis. Disseminating information to mobile clients using publish/subscribe. IEEE Internet Computing, pages 46–53, May 2004.

[OMG01a] Object Management Group. CORBA Event Service Specification v.1.1, March 2001.

[OMG01b] Object Management Group. Management of Event Domains Specification, June 2001. Available at: http://www.omg.org/cgi-bin/doc?formal/2001-06-03

[OMG02] Object Management Group. Joint Initial Submission regarding the JMS Notification Service RFP, January 2002. Available at: http://www.omg.org/cgi-bin/doc?telecom/02-01-02

[OMG04] OMG Data-Distribution Service for Real-Time Systems. Available at: http://www.omg.org/cgi-bin/doc?ptc/2004-03-07.

[Pie04] P. R. Pietzuch. Hermes: A Scalable Event-Based Middleware. PhD thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.

[Pla99] David Platt. TheCOM+ event service eases the pain of publishing and subscribing to data. Microsoft Systems Journal, September 1999. Available at: http://www.microsoft.com/msj/0999/comevent/comevent.aspx

[PBa02] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02), 2002.

[PBa05] L. Pesonen and J. Bacon. Secure Event Types in Content-based, Multi-Domain Publish/Subscribe Systems. In Proceedings of SEM 2005. ACM, sep 2005.

[PCM03] Picco, G.P., Cugola, G., Murphy, A.L. Efficient content-based event dispatching in the presence of topological reconfiguration. In: 23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA. (2003) 234-243.

[Pri01] Prism Technologies. Open Fusion Notification Service, May 2001. (White paper)

[PHJ02] I. Podnar, M. Hauswirth, and M. Jazayeri. Mobile push: Delivering content to mobile users. In Proceedings of the 22nd International Conference on Distributed Computing Systems, pages 563–570. IEEE Computer Society, 2002.

[PSB03] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In Proceedings of the 4th International Conference on Middleware, pages 62–82, June 2003.

[R+01] Bill Ray et al. Professional Java Mobile Programming. Wrox Press, Birmingham, United Kingdom, July 2001.

[Ros01] David Rosenblum. A tour of Siena, an interoperability infrastructure for internet-scale distributed architectures. In Ground System Architectures Workshop, February 2001.

[RKC01] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In J. Crowcroft and M. Hofmann, editors, Networked Group Communication, volume 2233 of Lecture Notes in Computer Science, pages 30–43. Springer, 2001.

[S+01] Tony Speakman et al. RFC 3208: PGM Reliable Transport Protocol Specification. Internet Engineering Task Force, December 2001. Available at: http://www.ietf.org/rfc/rfc3208.txt

[SAS01] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness—transparent information delivery for mobile and invisible computing. In Proceedings of the 1st International Symposium on Cluster Computing and the Grid, page 277, May 2001.

[Sie99] Jon Siegel. Anoverview ofCORBA3. In Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems, July 1999.

[Sri01] Paddy Srinivas. Introduction to COM+ events, March 2001. Available at: http://www.idevresource.com/com/library/articles/com+eventsintro.asp.

[SLi05] M. Srivatsa and L. Liu. Securing publish-subscribe overlay services with eventguard. In CCS '05: Proceedings of the 12th ACM conference on Computer and communications security, pages 289–298, New York, NY, USA, 2005. ACM Press.

[SWe00] H. Schulzrinne and E. Wedlund. Application-layer mobility using SIP. SIGMOBILE Mob. Comput. Commun. Rev., 4(3):47–57, 2000.

[Tar05] Sasu Tarkoma. Efficient and Mobility-aware Content-based Routing Systems. Ph. Lic Thesis. University of Helsinki, Department of Computer Science, 2005.

[TKa05a] Sasu Tarkoma and Jaakko Kangasharju. Handover Cost and Mobility-Safety of Content Streams. In Eighth ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems, October 2005.

[TKa05b] Sasu Tarkoma and Jaakko Kangasharju. Filter Merging for Efficient Information Dissemination. In 13th International Conference on Cooperative Information Systems, Lecture Notes in Computer Science 3760, Springer-Verlag, October 2005.

[TKa05c] Sasu Tarkoma and Jaakko Kangasharju. Mobility and Completeness in Publish/Subscribe Topologies. In IASTED International Conference on Networks and Communication Systems, ACTA Press, April 2005.

[TKa05d] Sasu Tarkoma and Jaakko Kangasharju. A Data Structure for Content-based Routing. In Ninth IASTED International Conference on Internet and Multimedia Systems and Applications, ACTA Press, February 2005.

[TMü04] Andreas Tanner and Gero Mühl. A Formalisation of Message-Complete Publish/Subscribe Systems. Technical Report 2004/11, Berlin University of Technology.

[Sun01] Sun Microsystems. Java Message Service Specification, June 2001.

[WCE02] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In the proceedings of the Hawaii International Conference on System Sciences, jan 2002.

[W3C00] World Wide Web Consortium. Document Object Model (DOM) Level 2 Events Specification,Version 1.0, November 2000. [Recommendation]. Available at: http://www.w3.org/TR/DOM-Level-2-Events/

[W3C03] World Wide Web Consortium. XML Events—An Events Syntax for XML, February 2003. [Candidate Recommendation]. Available at: http://www.w3.org/TR/2003/CR-xml-events-20030207

[W3C03b] World Wide Web Consortium. SOAP Version 1.2, June 2003. [W3C Recommendation].