A

Related work
Event-based systems

CORBA

The *Common Object Request Broker Architecture* (CORBA) [OMG02b] is a platform and language independent object-oriented middleware architecture designed by the Object Management Group (OMG). CORBA is released as a specification of an open standard that vendors can implement, which facilitates interoperability between different implementations. It is a mature middleware technology that is widely used in financial and telecommunication systems and has inspired many recent middleware initiatives. CORBA tries to address the need for asynchronicity in middleware communication for large-scale systems, its core design is still based on synchronous communication. A more fundamental problem is that many-to-many communication is not part of the basic services provided by the ORB and can only be simulated by less efficient object services.

CORBA Event Service

The OMG acknowledged the need for publish/subscribe communication in a middleware by introducing the *CORBA Event Service* [OMG95] as a CORBA extension. It is centered on the concept of an event channel, which connects producers to consumers of data in a decoupled fashion. The communication among suppliers and consumers can be in pull mode, in which case a consumer requests data from providers via the event channel. The alternative is that producers push data to the consumers in push mode. Data is represented as events that are either typed or untyped. Untyped events are attributes of the CORBA datatype any that can

be cast to any datatype. For typed communication, producers and consumers agree on a particular IDL interface and use its methods to exchange information in pull or push mode.

The Event Service enables CORBA clients to participate in many-to-many communication through an event channel. The asynchronous communication is implemented on top of CORBA's synchronous method invocation and thus has the substantial overhead of performing a remote method invocation for every event communication. Moreover, event consumers cannot limit the events that they receive from an event channel because no event filtering is supported.

CORBA Notification Service

The *CORBA Notification Service* [OMG02a] addresses the shortcomings of the Event Service by providing event filtering, quality of service (QoS), and a lightweight form of typed events. These structured events are divided into a header and a body. Both contain attribute/value pairs that hold the data associated with the event, but only the header attributes are externally visible. This allows event consumers to restrict the events that they receive from the event channel by specifying filters over the attributes using a filter constraints language. In addition, events are categorized according to their type with a type name field. A particular event may be uniquely identified through its event name. With structured events, the Notification Service implements a sophisticated type mechanism for event data. It comes with an expressive content-based filtering language for events and has support for QoS attributes. However, since the Notification Service attempts to maintain backwards compatibility with the Event Service, it suffers from the same problems with regard to communication efficiency over synchronous request/reply.

Java RMI

The *Java Remote Method Invocation* (RMI) specification [Sun99b] describes how to synchronously invoke methods of remote objects using request/reply communication between two Java Virtual Machines (JVMs) running on separate nodes. The Java RMI compiler generates marshaling code for the proxy object and the server skeleton. Because of the homogeneous environment created by JVMs, in which there is only a single programming language, the burden on the middleware is lower. It is even possible to move executable code between JVMs by using Java's object serialization to flatten an object implementation into a byte stream for network transport.

Asynchronous event communication within a single JVM is mainly used in the Abstract Window Toolkit (AWT) libraries for graphical user interfaces. The `EventListener` interface can be implemented by a class to become a callback object for asynchronous events, such as mouse or keyboard events. Some variants of asynchronous communication in Java are provided by the messaging infrastructure of JMS and the tuple spaces in the JavaSpaces specification.

Jini

The *Jini* specification [Sun03b] enables programmers to create network-centric services by defining common functionality for service descriptions to be announced and discovered. For

this, it supports distributed events between JVMs. The Jini framework extends the EventListener interface to provide event communication between JVMs. Event communication in Jini is built on top of synchronous Java RMI. More recently, Jini started to use tuple spaces for asynchronous communication.

JMS

The *Java Message Service* (JMS) [Sun01] defines a messaging API for Java. JMS comes with two communication modes: point-to-point and publish/subscribe communication. Point-to-point communication follows the one-to-one communication abstraction of message queues. Queues are stored and managed at a JMS server that decouples clients from each other. In publish/subscribe communication, the JMS server manages a number of topics. Clients can publish messages to a topic and subscribe to messages from a topic.

A JMS message is divided into a message header and body. The header contains various fields, including the destination of the message, its delivery mode, a message identifier, the message priority, a type field, and a timestamp. Apart from predefined fields, the header can also contain any number of user-supplied fields. The message body is in one of several formats: a StreamMessage, a TextMessage, and a ByteMessage contain the corresponding Java primitive types. A MapMessage is a dictionary of name/value pairs similar to the fields found in the header. Finally, an ObjectMessage uses Java's object serialization feature to transmit entire objects between clients. JMS provides a topic-based service with limited content-based filtering support in the form of message selectors. A message selector allows a client to specify the messages it is interested in by stating a filter on the fields in the message header.

IBM WebSphere MQ

IBM WebSphere MQ [IBM02a] is a message-oriented middleware platform that is part of IBM WebSphere suite for business integration. Messages are stored in message queues that are handled by queue managers. A queue manager is responsible for the delivery of messages through server-to-server channels to other queue managers. A message has a header and an application body. No type-checking of messages is done by the middleware. WebSphere MQ is a powerful middleware, whose strength lies in the simple integration of legacy applications through loosely-coupled queues. Nevertheless, it cannot satisfy the more complex many-to-many communication needs of large-scale applications, as it lacks natural support for multi-hop routing and expressive subscriptions.

JavaSpaces

The *JavaSpaces* specification [Sun03a] uses Java's type system to express elements in a tuple space and rules for matching them. In addition, it supports transactional access to the tuple space from distributed hosts. It implements a notify operation that makes an asynchronous callback when a matching element is inserted into the space. To do this, it follows Jini's distributed event specification, transmitting a RemoteObject to an EventListener.

CEA

The *Cambridge Event Architecture* (CEA) [BBHM95, BMB+00] was created in the early 90s to address the emerging need for asynchronous communication in multimedia and sensor-rich applications. It introduced the publish-register-notify paradigm for building distributed applications. This design paradigm allows the simple extension of synchronous request/reply middleware with asynchronous publish/subscribe communication. Middleware clients that become event producers or event consumers are standard middleware objects.

The interaction between an event producer and consumer is as following. First, an event producer has to publish the events that it produces; for example, in a name service. In addition to regular methods in its synchronous interface, an event producer has a special register method so that event producers can subscribe to events produced by this producer. Finally, the event producer performs an asynchronous callback to the event consumer's notify method according to a previous subscription. Event filtering happens at the event producers, thus reducing communication overhead. The drawback of this is that the implementation of an event producer becomes more complex since it has to handle event filtering.

The design goal of the CEA is to integrate publish/subscribe with standard middleware technology. Therefore, events are strongly-typed objects of a particular event class and are statically type-checked at compile time. Initially, subscriptions were template-based for equality matching only, but they were then extended with a predicate-based language with key/value pairs. These subscriptions are type-checked dynamically at runtime. Furthermore, the CEA provides a service for complex subscriptions based on composite event patterns.

COBEA

The CEA was implemented on top of CORBA in the *CORBA-Based Event Architecture* (COBEA) [MB98]. COBEA is based on the CORBA Event Service. COBEA is a general event architecture for building distributed active systems; it has been developed by the Opera group at the University of Cambridge. Events are passed between event producers and consumers as parameters in CORBA method calls because fully-fledged CORBA objects would be too heavy-weight as events. Event clients can by typed or untyped, a typed client encodes the structure of an event type in an IDL struct datatype, whereas an untyped client uses the generic any datatype. Type-checking for typed clients is done by the IDL compiler. The subscription language consists of a conjunction of predicates over the attributes defined in the event type. COBEA supports subscriptions regarding composite events. The processing of composite events is based on the CEA. The COBEA system is employed in an alarm correlation system for network management in telecommunications.

SIFT

The *Stanford Information Filtering Tool* (SIFT) [Car98] focuses on the distribution of full-text documents. Clients subscribe to selected documents by submitting their subscription. The subscriptions in SITF include information retrieval style queries with keywords and additional parameters to control the frequency and the content of notifications. SIFT supports two information retrieval models for queries: *boolean* and *vector space model*.

SIFT has a centralized architecture. The events are sets of new documents. Publishers have to actively forward their full-text documents to the system. The documents are then stored in a central SIFT repository. The filter engine matches the new documents against the client subscribers, unmatched documents are discarded. Then the alerter forwards the matched documents to the interested clients, immediately or according to a predefined schedule.

The providers have to send the documents directly to the service. The SIFT system does not implement an observer. The central document repository acts as event message repository, the subscription database as subscribe repository. The filter and notifier parts are implemented via filter engine and alerter. The SIFT project has evaluated efficient techniques to match subscribers and documents. The inverted index of subscribers used in SIFT has influenced several other implementations, e.g., Hermes.

The system's functionality is more closely related to information retrieval systems than to current event systems. Event messages are passively received by the SIFT system. The system has no event model, only filtering of full-texts of newly published documents is supported. No composite events are supported in the SIFT subscription language, therefore, only simple cooperation of different providers is possible. SIFT provides good performance for text-centered filtering as needed in information retrieval systems. Scalability has been addressed by usage of sophisticated document filtering algorithms. A distributed architecture has not been implemented for SIFT.

Gryphon

The *Gryphon* project at IBM Research [IBM01] led to the development of an industrial-strength, reliable, content-based event broker that is part of IBM WebSphere suite as the IBM Web-Sphere MQ Event Broker [IBM02b]. It is a publish/subscribe middleware implementation with a JMS interface that provides a redundant, topic- and content-based multi-broker publish/subscribe service. The Gryphon event broker has been deployed for large-scale information dissemination at global sports events, such as the Olympic Games. It includes an event matching engine, a scalable routing algorithm and security features. However, the overlay network of event brokers is static, as it is defined in configuration files at deployment time. This makes it difficult for the middleware to adapt to changed network conditions. Composite event detection is provided by relational subscriptions, a relational data model for messaging can prove to be too heavy-weight for many applications.

JEDI

The *Java Event-Based Distributed Infrastructure* (JEDI) [CNF01] is a Java-based implementation of a distributed content-based publish/subscribe system from the Politecnico di Milano, Italy. Events in JEDI have a name and a list of values for event parameters. Subscriptions are specified in a simple pattern matching language. A JEDI system consists of active objects, which publish or subscribe to events, and event dispatchers, which route events. Event dispatchers are organized in a tree structure, and routing is performed according to a hierarchical subscription strategy.

The system has been extended to support mobile computing. Event dispatchers support moveOut and moveIn operations that enable subscribers to disconnect and reconnect at a different point in the network.

Elvin

Elvin [Sal68] is a notification service for controlling and monitoring applications within a distributed system. The current implementation Elvin4 uses a distributed client-service architecture. Clients subscribe at the service by sending subscriptions to the read thread of the server. The subscriptions are stored within the subscription database. Subscriptions are defined by means of (*attribute; values*) pairs.

The providers send event messages to the service via read thread, only active providers are supported. In Elvin, event messages are called notifications. The messages are routed to clients. The messages are matched to the subscriptions via the subscription comparison engine. The server filters only those events for which subscriptions have been defined, this technique is called *quenching*. Due to quenching, all incoming messages have interested clients. Quench expressions prevent providers from sending event messages to which no client has subscribed. They can additionally be used to establish subscription-driven observers. These observers are implemented as independent applications and are not part of the Elvin system. The filter is implemented via the subscription comparison engine. Because messages are only routed to the clients, no notifier is implemented.

Rebeca

The *Rebeca* [FMB01] project investigates the potential of publish/subscribe technology for large-scale e-commerce applications. The focus lies on scalable routing algorithms and software engineering techniques for the design and implementation of event-based business applications.

Narada Brokering

The *Narada Brokering* project [PF03] aims to provide a unified messaging environment for grid computing, which integrates grid services, JMS, and JXTA. It is JMS compliant but also supports a distributed network of brokers as opposed to the centralized client/server solution advocated by JMS. The JXTA specification is used for peer-to-peer interactions between clients and brokers. Events can be XML messages that are matched against XPath subscriptions by an XML matching engine. The network of brokers is hierarchical, built recursively out of clusters of brokers. Every broker has complete knowledge of the topology, so that events can be routed on shortest paths following the broker hierarchy. In general, there is the additional overhead of keeping event brokers organized hierarchically, which can be costly. Dynamic changes of the topology are propagated to all affected brokers.

Siena

The *Siena* research project at the University of Colorado focuses on the design of a generic scalable event service that routes messages through a wide-area network. The Siena

architecture is implemented as a distributed system that consists of a network of event servers. The parts of each event server are implemented as distributed components.

The system acts as a dispatcher of event messages. Clients of Siena can be providers or clients or both. The dispatching of event messages is regulated by advertisements announcing events, subscriptions, and the publication of events. The event messages are (*attribute; value*) pairs. Advertisements announce the publication of certain event message types. Subscriptions are simple predicates on attribute values with limited composite operators.

Siena's cooperating servers can be organized in hierarchical or in peer-to-peer manner, the components can be dynamically reconfigured. In Siena, delivering a notification to all interested clients means forwarding the event message through the network of servers. The subscription information is used for the network routers.

The Siena filter implementation is based on principles from IP multicast routing protocols: downstream duplication and upstream monitoring. In Downstream duplication, each message is routed as far as possible and then duplicated downstream. Upstream monitoring is the placement of filters and composite event pattern recognizes as close to the sources as possible.

Observation has to be implemented by the providers because only active providers are supported. Due to the routing of event messages, no notifier component is necessary. The routing tables in each server act as subscription repository.

Theoretically, Siena supports composite events, but practically only detection of event sequences has been implemented. Subscriptions regarding composite events are subdivided, and then the matching sub-patterns are monitored and reported to the network. The event filtering does not consider temporal influences and restrictions.

OpenCQ

OpenCQ has a three-tier architecture: client, server, and wrapper. The client tier is primarily responsible for receiving client subscriptions. The system supports time-based, active content-based events and composite events. Subscriptions consist of a query, a start-trigger and a stop condition. The subscriptions are stored in the system repository. The CQ server at the second tier evaluates the subscriptions: for each continual query, an update monitoring program creates distributed programs to keep track of information sources, their availability and changes. After an initial submission of object states, the subscription query is evaluated after the start trigger fires. Until the termination condition is met, OpenCQ sends notifications about matched events to the interested clients. At the third tier, wrappers keep track of time events or update-specific data. Only events whose attribute values cross a given threshold are presented to the server.

The OpenCQ system supports passive and active observation: passive observers are triggered by the providers. For each source, a push-client agent has to be implemented that listens to the broadcast sources and presents the collected information to the service. The other components of the server interact on a pull basis with wrappers. Active observers are triggered by the trigger evaluator. The event detector and the trigger evaluator implement the passive observer functionality and execute the task of a filter, the query evaluator

provides the content of the notification, and the change notification manager acts as a notifier.

Sentinel

Sentinel [Cha97] is an integrated active database system developed at the University of Florida. It supports the evaluation and management of ECA rules. Sentinel uses the Open OODB Toolkit as underlying platform. The rule specification has been integrated in the C++ language. The rule specification implements *Snoop* [CM94], an expressive event specification language for a DBS. Snoop supports composite events with temporal restrictions and additional consumption policies (recent, chronicle, continuous and cumulative). Sentinel supports database-internal events and external events. Internal events are observed by the system, external events are externally submitted to the system. Sentinel does not support the active observation of events outside the database. Sentinel is implemented as a centralized system. However, Snoop supports event detection in a distributed environment. The *Global Event Manager* [CL01] is an extension to Sentinel that detects events in a distributed environment. The extended Sentinel is comparable to OpenCQ.

SAMOS

The active database *SAMOS* [GD92] has been developed at the University of Zurich. The system supports Event-Condition-Action (ECA) rules that can be seen as client subscriptions. SAMOS supports primitive and composite events. SAMOS use Petri nets for the detection of composite events. A Petri net consist of states (input and output) modeling classes, and of transitions. The system supports parameters for event specification that allow for the selection of event instances based on database-related characteristics, such as the same transaction or the same client. SAMOS does not support distribution. The focus of the SAMOS prototype is on the client-friendly ECA rule specification language and on the Petri net implementation of rules. The rule specification in SAMOS is similar to the subscription language in OpecCQ restricted to database-internal events.

NAOS

The *NAOS* (Native Active Object System) incorporates an active behavior within the object-oriented database management system O2. Active behavior has been incorporated in O2 as Event-Condition-Action (ECA) rules which belong to database schemas. The rules are defined at the same level as classes and applications. This approach provides a flexible approach for controlling different kinds of operations (low level operations, methods, programs) on persistent or transient entities (objects or values).

The event part of a NAOS rule specifies the type of events that may trigger the rule. NAOS detects primitive and composite events. Primitive events may be generated by manipulations of entities or parts of entities. The moment of generation (i.e., when an event occurs) can be either before or after the actual operation depending on the event type. Events may also be generated by executions of transactions, programs or applications. Composite events are made of events (both primitive and composite) connected by operators: disjunction, conjunction, sequence, strict sequence, iteration, strict iteration, negation and

strict disjunction. The condition part specifies predicates over persistent or transient entities. These predicates are defined as OQL queries. The target of a query is the actual database or the data associated with the triggering event/operation (an entity or the parameters of a method call). The action part is made of O2C statements that may operate on persistent and transient entities.

Chimera

Chimera is a novel database model and language which has been designed as a joint conceptual interface of the IDEA project, a major European cooperation initiative aiming at the integration of object-oriented, active and deductive database technology.

Ode

Ode [AG89] is an object oriented database and environment developed at AT&T Bell Labs. The database is defined, queried and manipulated using the database programming language O++. O++ is based on C++; it borrows and extends the object definition model of C++ in order to provide persistence of objects. Events happen at specific points in time and are of basic, logical and composite type.

YEAST/READY/OmniNotify

The *YEAST* event action system and its successor, the *READY* notification service, have been developed at AT&T. READY is a distributed system which implements a number of event servers that can form differently structured hierarchies. READY focuses on high-level constructs, such as primitive and composite events, and ordering properties for event delivery. The profile handling in READY is focused on grouping of client interactions and zones for administrative interactions. The main feature of READY is the support for composite events and its grouping functionality. The service uses the principle of structured events as introduced in the CORBA Notification Service. Recently, the system has been re-implemented to support CORBA and the CORBA notification standard; it is now called *OmniNotify*.

ADEES

The *ADEES* (Adaptable and Extensible Event Service) [Var00] is a software component for integrating and executing database applications built out of distributed and heterogeneous components. It is an infrastructure for specifying and generating event services. Generated event services implement event types and event management meta-models that can be specialized and instantiated to specify and generate event managers. Manager functionalities are described through schemata that associate event type models with event management models. Using schema instances, managers interact with event producers and consumers to detect and notify events. Event services and managers can adapt themselves to applications needs and to changes stemming from their execution environment due to the dynamic and static adaptability and extensibility properties. Active systems that integrate event and rule services have been specified and implemented using ADEES. Such systems generate event and rule managers that cooperate to execute active rules. ADEES has also

been used for specifying and building a workflow management system. This system uses an event service and active systems to execute workflows.

A-MEDIAS

A-MEDIAS [Hin03] is an integrating event notification service that is adaptable to different application requirements. The service can (i) adapt to changing applications, (ii) integrate data from differently structured and changing sources, and (iii) efficiently filter primitive and composite events under changing workload.

Hermes

Hermes [Pie04] is a distributed, event-based middleware platform. Hermes follows a type- and attribute-based publish/subscribe model that places particular emphasis on programming language integration by supporting type-checking of event data and event type inheritance. To handle dynamic, large-scale environments, Hermes uses peer-to-peer techniques for autonomic management of its overlay network of event brokers and for scalable event dissemination. Its routing algorithms, implemented on top of a distributed hash table, use rendezvous nodes to reduce routing state in the system, and include fault-tolerance features for repairing event dissemination trees.