Chapter 4

Review and Analysis of Event Notification Services

It is not once nor twice but times without number
that the same ideas make their appearance in the world.
Aristotle, On the Heavens

In this chapter, we present a comprehensive review and analysis of the state of the art in the area of event notification services. This evaluation of related work is based on our model for ENS as introduced in the previous chapter. The analysis follows the requirements for event notification identified in Chapter 2.

The chapter is divided in two parts: In the first part, related event notification systems are analyzed. For each of the selected systems, we give a brief overview of the main focus and achievements. Our reference model and the event notification sequence are used to structure and to visualize the analysis. Special attention is given to the system's support of the requirements for event notification systems as discussed in Chapter 2. We discuss the limitations of the introduced approaches.

In the second part of the chapter, we provide a brief survey of related research areas and base technologies that are either related to our approach or could be used as base technology for implementation, such as message queuing systems. For each area, we describe the main focus and name relevant approaches. We illustrate the influence of the approaches on our work and discuss, which of the technologies can be used as a basis for an implementation of an integrating ENS.

We conclude the chapter with a summary of the limitations of existing event notification services.

4.1 Analysis of Event Notification Systems

Several implementations of ENS exist, some of these are application-independent, e.g., Elvin [SA97], CORBA Notification Service [OMG99], Siena [CRW01], Keryx [BK97a], and LeSubscribe [PFJ01]. Other services have been designed for certain applications, such as Hermes [FFS+01] for Digital Libraries, DAOS [LCB99] for air traffic control, Gryphon [SBC98] and REBECA [MFB02] for E-Commerce, and Genesis [SF96] for traveller information.

In this section, selected systems are introduced in detail. We identify four types of event notification services. For each type, one example event notification system is analyzed in detail. For each system, we identify the architecture and describe its main parts. The project context and focus of the implementation is given briefly. The systems have been selected for their influence on ENS research – each system example represents a particular approach. Some of these systems will serve as examples for the evaluation of various aspects of event notification systems. In addition to the detailed analysis, we briefly describe a number of ENS and compare the systems to the four types.

Note that the CORBA Notification Service (CORBA NS) and active database systems (aDBS) may be seen as event notification systems or as implementation technology for ENS, e.g., as done in [LYO98]. In this chapter, the CORBA NS and selected aDBS are analyzed as event notification systems. However, ENS can be implemented on top of each other. For example, the OmniNotify [Omn02] and COBEA [MB98] are based on a CORBA NS implementation.

4.1.1 SIFT: Centralized, Document-centered System

One of the earliest notification systems developed was SIFT [YGM95, YGM99], a tool for wide-area information dissemination. It is now commercially operated as news service InReference [InR03]. The Stanford Information Filtering Tool (SIFT) focuses on the distribution of full-text documents. Clients subscribe to selected documents by submitting their profile. The profiles in SIFT include IR-style queries with keywords and additional parameters to control the frequency and the content of notifications. Clients can select between boolean profiles and profiles based on the vector space model.

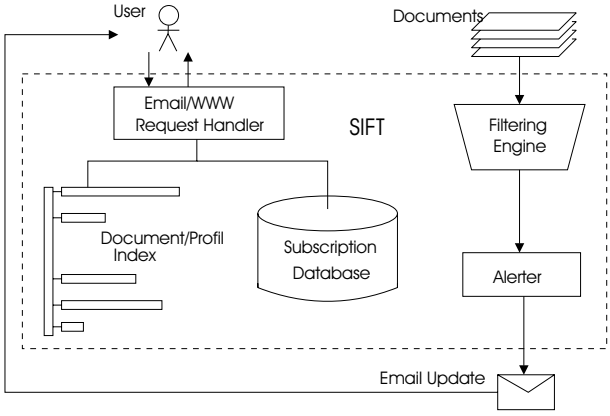


Figure 4.1: SIFT architecture [YGM95]

SIFT has a centralized architecture as shown in Figure 4.1. 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, which is not shown in the architecture. The filter engine matches the new documents against the client profiles, unmatched documents are discarded (see Figure 4.2). Then, the alerter forwards the matched documents to the interested clients, immediately or according to a predefined schedule (indicated by the deferred notification arrow in Figure 4.2).

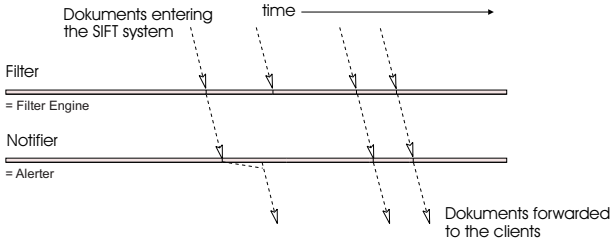


Figure 4.2: Event notification sequence in SIFT: Documents are filtered in SIFT and then directed to the clients

Figure 4.3 shows our reference model of ENS systems as introduced in Chapter 3 with the logical parts that are included in the SIFT system standing out in gray. Because the providers have to send the documents (information objects) 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 profile repository. The filter and notifier parts are implemented via filter engine and alerter. The SIFT project has evaluated efficient filtering techniques to match profiles and documents. The inverted index of profiles [YGM94b] used in SIFT has influenced several other implementations, e.g., Hermes [FFS+01].

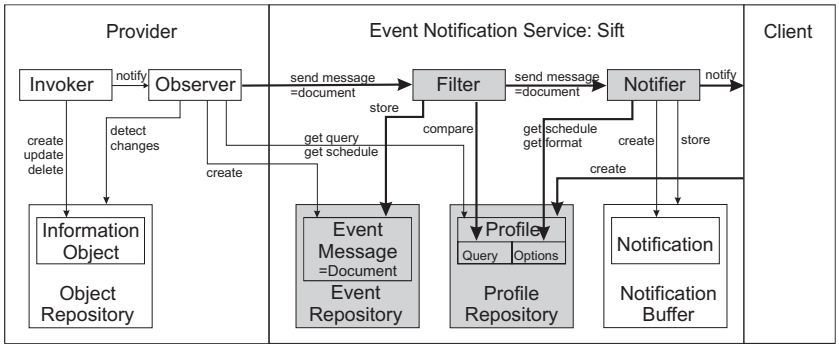


Figure 4.3: Logical parts implemented in SIFT (in gray)

Analysis: The system's functionality is more closely related to Information Retrieval systems than to current event notification 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. The system is not explicitly aware of differing document sources; the filtering follows static rules. No composite events are supported in the SIFT profile language, therefore, only simple cooperation

of different providers is possible. Temporal awareness is not needed because time-dependent profiles are not covered by the profile language. SIFT provides good performance for text-centered filtering as needed in IR systems. Scalability has been addressed by usage of sophisticated document filtering algorithms. A distributed architecture has not been implemented for SIFT.

4.1.2 Elvin: Simple Distributed System

Elvin [SA97, ASB+99] is a notification service for controlling and monitoring applications within a distributed system. Conceptually, it is closely related to the CORBA Notification Service[1] It has been developed at the DSTC, Queensland, Australia, as a testbed for distributed federated algorithms. Applications based on Elvin are, e.g., Tickertape and Orbit. Tickertape [FPSK98] is a news ticker that can be used for group communication (see Scenario 4 on Page 17). Orbit [MT97] is a distributed workspace system that provides, e.g., facilities for distributed authoring on a shared document repository (see Scenario 4 on Page 17). In Orbit, the Elvin service is used as medium for awareness notifications.

The current implementation Elvin4 uses a distributed client-server architecture (see Figure 4.4). Clients subscribe at the service by sending profiles to the read thread of the server. The profiles are stored within the subscription database. Profiles are defined by means of (attribute;value) pairs. Only primitive content profiles are supported.

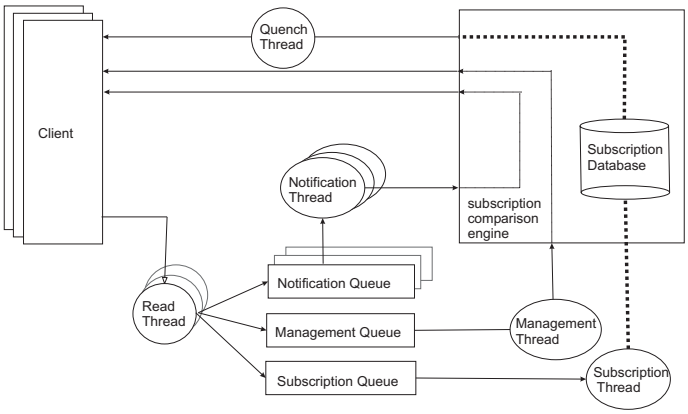


Figure 4.4: Elvin architecture [SA97]

The providers send event messages to the service via the read thread, only active providers are supported. In Elvin, event messages are called notifications. The messages are routed to the clients (see Figure 4.5). The messages are matched to the profiles via the subscription comparison engine. The server filters only those events for which profiles have been defined, this technique is called *quenching*. Due to quenching, all incoming messages have interested clients: Figure 4.5 shows that no messages are rejected and all messages are delivered immediately. Elvin is, therefore, called a 'content-based routing service'. Quench expressions prevent providers from sending event messages to which no client has subscribed. They can additionally be used to establish profile-driven observers. These observers are implemented as independent applications and are not part of the Elvin system.

[1]The DSTC was a contributor to the OMG Notification Service RFP and ultimately, one of the submitters of the CORBA Notification Service standard.

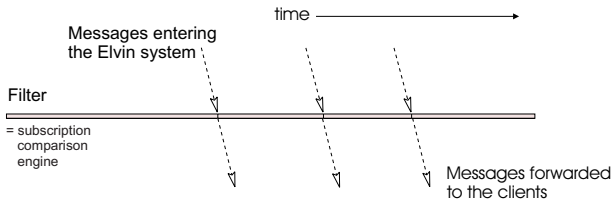


Figure 4.5: Event notification sequence in Elvin: Incoming messages are routed to clients

Figure 4.6 depicts the parts of the reference model that are implemented as components in Elvin. The observation of the events is done at the providers' sites. The subscription database acts as a profile repository. Event message persistency is not supported. The filter is implemented via the subscription comparison engine. Because messages are only routed to the clients, no notifier (as defined in our reference model) is implemented. The current focus of the Elvin project is on security and filter performance.

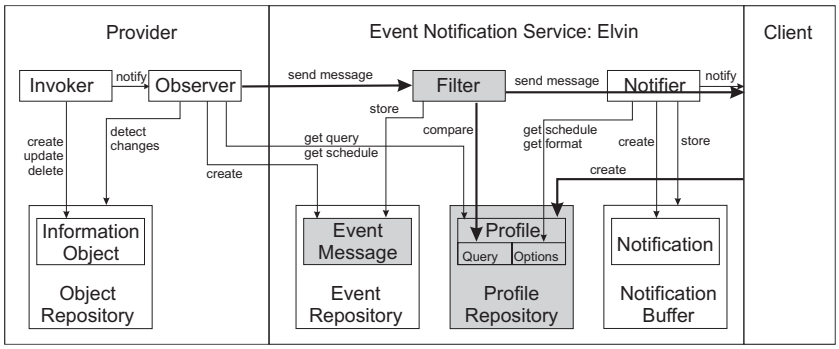


Figure 4.6: Logical parts implemented in Elvin (in gray).

Analysis: The service supports filtering and routing of event messages. It is not based on an event concept but on the handling of structured documents containing (attribute;value) pairs. The observation of events is not part of the service. Only profiles regarding primitive events are supported. It could be argued that the Elvin service is merely an event-based infrastructure on middleware level and not a full event notification service. The service may support coordinating sources, but, due to the lack of composite event support, integration of event information is not possible in Elvin. The service does not adapt to changing sources. The filter engine is based on the Gough-algorithm [GS95], which is one of the fastest filtering algorithms for (attribute;value) pairs [FLPS00]. Scalability of the Elvin system is achieved by distributing the system's components.

4.1.3 Siena: Distributed Network of Servers

The Siena research project [CDRW98, CRW01] 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 (see Figure 4.7). The parts of each event server are implemented as distributed components.

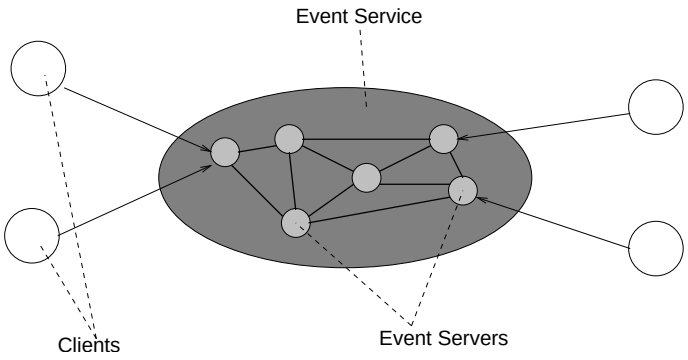


Figure 4.7: Structure of a Siena service consisting of a network of event servers [CRW01]

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 (profiles), and the publication of events. The event messages are (attribute;value) pairs. Advertisements announce the publication of certain event message types. Profiles are simple predicates on attribute values with limited composite operators.

Siena's cooperating event 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 (see Figure 4.8). The profile information are used for the network routers. Figure 4.8 shows primitive and composite events (indicated by the bar resulting in a single arrow).

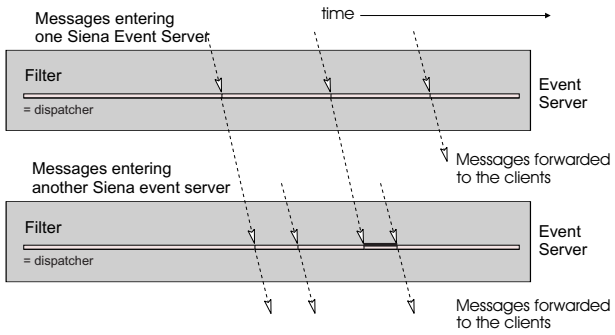


Figure 4.8: Event notification sequence in Siena: Incoming messages are forwarded through the network of servers (routing).

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 recognizers as close to the sources as possible.

Figure 4.9 identifies the logical parts derived from the system's functionality. Only limited information about the event server implementation is available. Because only active providers are supported, observation has to be implemented by the providers. Due to the routing of event messages, no notifier component is necessary. The routing tables in each server act as profile repository.

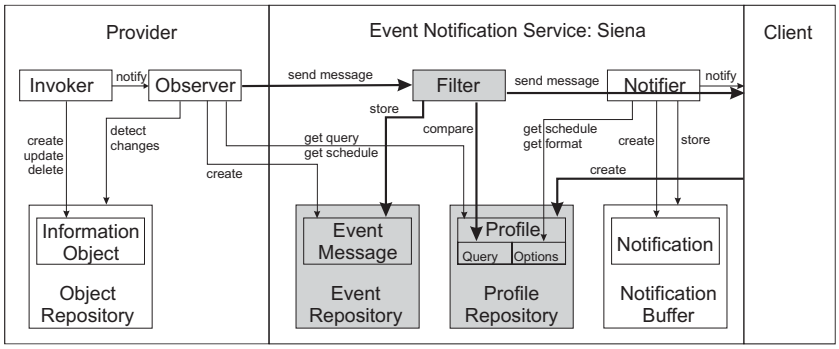


Figure 4.9: Logical parts implemented in Siena (in gray)

Analysis: The Siena service implements filtering and dispatching of event messages that enter the system's network of event servers. Hierarchical, acyclic/generic peer-to-peer, and hybrid architectures of the network of event servers have been simulated. Filters and notifications are dispatched upward and downward the network to achieve scalability within a wide area network.

In the Siena service, event observation is not supported; a rudimentary event model is used in advertisements to announce event messages. Theoretically, Siena supports composite events, but practically only the detection of event sequences has been implemented. Profiles regarding composite events are subdivided, then the matching sub-patterns are monitored and reported to the network. The event filtering does not consider temporal influences and restrictions. The filtering does not flexibly react to changing sources.

4.1.4 OpenCQ: Distributed System with Sophisticated Language

Within the Continual Queries Project [LPBZ96, LPT99a] at the Oregon Graduate Institute of Science and Technology, a number of tools for update monitoring and event driven information delivery have been developed. The main contribution of the project is the newly developed *continual query language* for profile definition. In Continual Queries (CQ), a client's interest is defined in a SQL-like manner. Profiles consist of a query, a start-trigger, and a stop condition. The language allows for sophisticated definition of profiles. It has been implemented in several prototypes, e.g., OpenCQ [LPT99a], CONQUER [LPTH99], and JCQ [LPT99b], mainly for logistics applications (see Scenario 2 on Page 15).

For the analysis, we concentrate on the OpenCQ system. OpenCQ has a three-tier architecture: client, server, and wrapper (see Figure 4.10). The client tier is primarily responsible for receiving client profiles. The system supports time-based, active content-based events, and composite events. Passive events are not supported. The profiles are stored in the system repository. The CQ server at the second tier evaluates the profiles: For each continual query, an update monitoring program creates distributed programs that keep track of information sources, their availability and changes. After an initial submission of object states, the profile 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

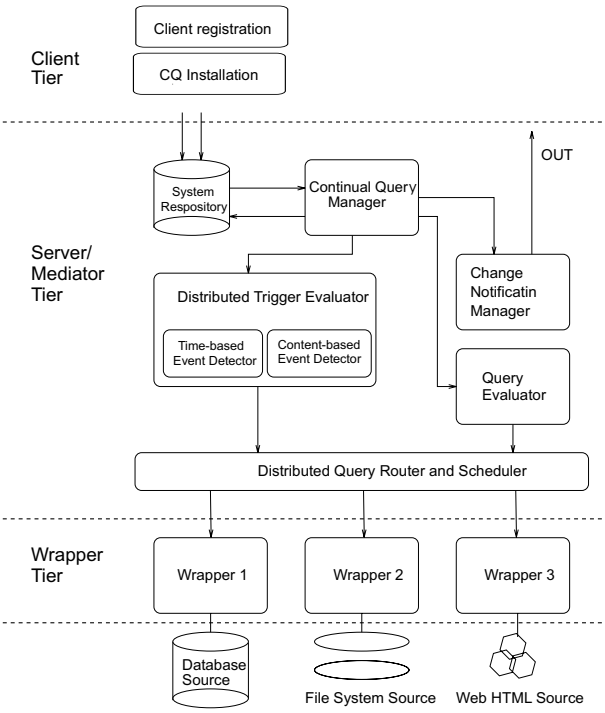


Figure 4.10: OpenCQ architecture [LPT99a]

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 the 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 notifier. In Figure 4.11, we show primitive and composite events (indicated by the bar resulting in a single arrow, as well as scheduled delivery (indicated by the deferred notification arrow). The logical parts of OpenCQ are shown in Figure 4.12. The active and passive wrappers are shown as observers on both the system's and the provider's site.

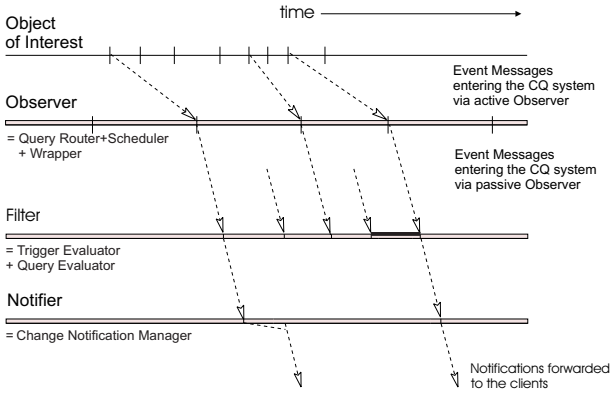


Figure 4.11: Event notification sequence in OpenCQ: Events are actively and passively observed by the service; the event messages are filtered and notifications are sent to clients.

The system repository acts as both the profile and event repository. The filter part is implemented via the trigger and query evaluator, and is controlled by the distributed query router and scheduler. The notifier part is implemented via the change notification manager.

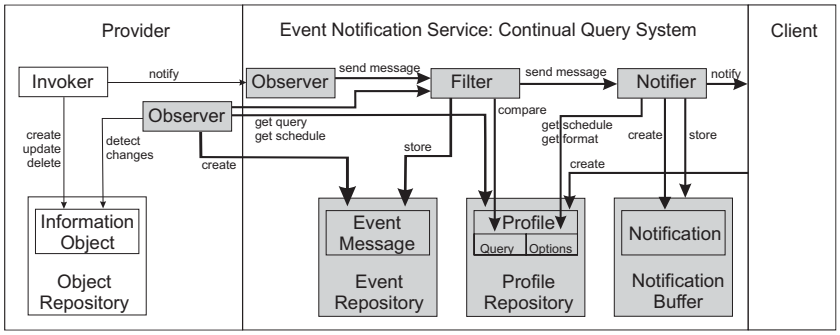


Figure 4.12: Logical parts implemented in OpenCQ (in gray)

Analysis: OpenCQ enables sophisticated profiles in SQL-like style based on an event model. The CQ language supports primitive and composite events. Passive events are not supported. The evaluation style of continual queries is fixed. Passive and active observers are implemented, the influence of different observation styles (temporal awareness) is not considered in the profile evaluation. The system's performance is achieved by the use of trigger indices, database-techniques for multiple-query optimization, and incremental condition evaluation. The system's scalability is based on the distributed trigger facility and distributed event detection. However, OpenCQ does not provide adaptation to changing sources and application as proposed in this thesis.

4.1.5 Overview of Further Event Notification Services

In this subsection, we briefly analyze additional event notification systems. The systems' capabilities are compared to the four extensively evaluated event notification systems.

CORBA Notifications Service. The CORBA Notification Service [OMG99] is part of the widely used CORBA Object Request Broker (ORB). It is a standardized model for a decoupled, event-based communication via channels. The service enhances the CORBA Event Service [OMG97b] by structured events, event filtering, transactions, and reliability considerations. Providers supply events to notification channels while clients register simple filters with channels. A notification channel forwards all supplied events that pass its filters to each registered consumer. Filter constraints can only be defined for structured event messages.

The notification service filters only actively submitted event messages, it provides neither an event model nor active observation of events. Similar to Elvin (Section 4.1.2), the notification service supports profiles regarding primitive events in a distributed environment and implements only the filter part of our ENS model. The notification service is covered only by some CORBA implementations [ION01, Dco02, Ope03]. The CORBA notification service is the basis for several ENS; examples are the READY/OmniNotify system [GKP01] and the COBEA system [MB98].

YEAST/READY/OmniNotify. The YEAST event action system [KR95] and its successor, the READY notification service [GKP99], have been developed at AT&T. READY is a distributed system similar to Siena: READY 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: Profiles can be grouped such that clients may share their profiles, session grouping allows for uniform control over the profiles' QoS and delivery properties. Here, 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 [Omn02]. Similar to Siena, the system supports efficient filtering of composite events in a distributed environment.

COBEA. COBEA [MB98] is a general event architecture for building distributed active systems; it has been developed by the Opera group at the University of Cambridge. It is implemented as a distributed service and supports profiles regarding composite events. COBEA's implementation is based on the CORBA notification service; the project focuses on scalable event delivery in networks. The system architecture does not support active event observation. COBEA uses an event model for the handling of event messages. The processing of composite events is based on the Cambridge Event Architecture (CEA) [BBHM96]. The COBEA system is employed in an alarm correlation system for network management in telecommunications [Ma97]. COBEA is conceptually similar to OmniNotify and Siena.

Keryx. Keryx [BK97b, LRW97, Low97], the former Nexus Event Service, was a project at Hewlett Packard Laboratories, Bristol, for distributing event messages on the Internet. Similar to Siena, the Keryx system implements a hierarchy of event distributors in order to achieve scalability. Event messages are routed through the system. Clients subscribe to notifications, the service does not implement every filter completely or at all: When an event distributor is unable to implement a requested filter, this is indicated to the client and the distributor delivers the events unfiltered. The event dispatcher implements only the filter and the notification part, observer and invoker reside on the event publishers' sites. Keryx is not based on an event concept, but provides a profile definition language that covers sophisticated primitive events supporting extensive combinations of attribute predicates. The Keryx system is the basis of a distributed virtual environment [WH98], which can be used, e.g, to display an active office plan indicating the location of all employees in the building.

LeSubscribe. LeSubscribe [PFLS00, PFJ01] is a content-based publish-subscribe system developed at INRIA, Paris, in the context of the Caraval project. LeSubscribe is a centralized system that supports passive and active observation of events, namely Internet auctions. Similar to the Elvin system, only primitive events are supported. The system is based on an event model that is rather document centered but considers semi-structured event messages and various event message types. The system's filter algorithm aims at balancing the tradeoff between performance and maintenance. A prototype called WebFilter [FJL+01] extends the LeSubscribe system to enable XML processing.

Gryphon. Gryphon [SBC98, BCM+99] is a distributed content-based message-brokering system developed at IBM. While Gryphon's structure is similar to Siena's, its filtering techniques are complementary to those of Siena. Gryphon uses distributed filtering to match a notification to a large set of profiles [ASS+99]. The main difference to Siena is that Gryphon propagates every profile everywhere in the network, whereas Siena propagates only the most generic profiles. Gryphon is not based on an event concept; only event message routing is supported. The system neither implements active observation nor filtering of composite events.

REBECA. The REBECA Event-Based Electronic Commerce Architecture [MFB02, FMG02] is a project at the Technical University Darmstadt. The project focuses on event filtering in a distributed environment. Profiles are distributed within the service network using several optimization strategies, e.g., the prevention of repeated profile distribution. Event messages are distributed to event filters according to their content [MFB02]. Based on the proposed techniques, the REBECA architecture can be seen as an extension of the Siena approach. Awareness of temporal delays in event composition has been addressed in [LCB99]. The current prototypes support stock exchanges applications and meta-auctions. The services' design follow a modular approach (scopes in [FMG02]) to hide internal configurations and to support for heterogeneous environments. Scopes in REBECA an extension of the concept of groups and zones in READY.

NiagaraCQ. NiagaraCQ [CDTW00] is part of the Niagara project at the University of Wisconsin, which aims at developing a distributed database system for querying distributed XML data sets. The system filters streaming XML data using a simple form of continuous queries as proposed for OpenCQ (regarding primitive and time events). A large number of clients may register continuous queries using the query language XML-QL [DFF+99]. The filtering in NiagaraCQ uses filter grouping techniques as proposed for query optimization in active databases [HCH99]: NiagaraCQ builds static plans for the different profiles in the systems, and allows two profiles to share a filter module if they have the same input. The grouped profiles share computation and save memory and I/O costs. Niagara employs an event model and uses active and passive observation. No profiles regarding composite events are supported. Similarly to SIFT, NiagaraCQ focuses on (XML) documents. It is implemented as a distributed service without taking advantage of the distribution for efficient filtering.

SAMOS. The active database system SAMOS [GD92, GD94] has been developed at the University of Zurich. The system supports Event-Condition-Action (ECA) rules that can be seen as client profiles. Only database events are supported, external events cannot be specified. SAMOS supports composite events; the event detection is implemented based on colored Petri-Nets. 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 profile language in OpenCQ restricted to database-internal events.

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 [CM93], an expressive event specification language for aDBS. 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. Active database systems with support for external events can be used as base technology to implement event notification systems, as done, e.g., in the Ariel system [HICD+98].

In this section, we analyzed in detail four event notification systems that represent basic system types for ENS. Additional systems have been described in relation to these basic types. However, a large number of event notification systems are referenced in research literature.[2] We could not name all systems related to the topic of this thesis – our analysis is illustrative rather than comprehensive. We focussed on approaches that are closely related to ours. Further systems could be named in this context, e.g., CEA [BBHM96], Deeds [DMDP99], EVE [TGD97], GEM [Man95], Herald [CJT01], Scribe [RKCD01], and A-TOPSS [LJ02].

4.2 Related Event-based Technology

Having analyzed the directly related approaches in the previous section, we now present an overview of areas closely related to the topic of this thesis: event-based infrastructures, Internet notification protocols, low-level event-based communication, and design patterns for event-based communication. We describe the main focus of these areas and name relevant approaches. For each topic, the connections and differences to the area of event notification systems are discussed briefly. These approaches are introduced to further illustrate the focus of this thesis and to consider adjoining areas that influenced our work. We discuss, how these technologies may be used as a basis for an implementation of an ENS.

4.2.1 Event-based Infrastructures

Event-based infrastructures (EBI) support asynchronous message exchange between objects in a distributed environment. These infrastructures are also called message-oriented middleware (MOM) or Message Queuing System (MQS). EBI define a middleware for interoperability of independent, heterogeneous systems; they build an abstraction for program-to-program communication. The communication is performed in a peer-to-peer manner, where the communication objects (messages) remain anonymous. Programs send messages to a passive communication abstract, often called queue or channel.

[2] A selection can be found in a link collection on event-based computing [Buc03] that is maintained by the University Darmstadt.

Consumer programs have to actively request the messages.

Primary applications for EBI are independent, heterogeneous application systems, e.g., business-to-business transactions. Examples for EBI are JEDI [CDF98], Talarian SmartSockets [Tal98], and Java Distributed Event and Message Service [HBS99]. Several systems are closely based on a database system, e.g., Advanced Queuing [Ora99] by Oracle. Others are self-contained systems, such as MessageQ [BEA00] and MQSeries [IBM95]. We briefly introduce MQSeries in the following paragraph. The services provide very different functionalities, ranging from simple put/get-messages for single clients to publish/subscribe and multicast functions to serve multiple clients.

WebSphere MQ (formerly MQSeries). The WebSphere MQ system [IBM95] by IBM implements a sophisticated communication system with a wide range of options. It supports a variety of communication styles, such as multicast ('distribution list'), point-to-point, and group communication. The system consists of providers of messages, several message queues, and clients receiving messages. The providers post their messages to selected queues, clients receive all messages of the queues to which they are subscribed. This method is similar to channel-based and subject-based filtering.

By using trigger messages, an event-based processing of the messages can be realized. For example, the triggers can specify whether the client is notified on the first message posted to an empty queue or on all messages in the queue. The message queue is a passive object, the queue manager actively processes messages and evaluates triggers. It is possible for the clients to read, retrieve, or browse in the message queue. Blocking, or non-blocking get-methods are available to the clients. The messages can be sent within a transactional context; logging of messages is also supported. The system is well suited for high-reliability applications, such as ECommerce systems.

Event notification services can be built on top of event-based infrastructures, e.g., as done in an early version of the Hermes system [FFS+01]. The profile definition languages provided in event-based infrastructures are too limited for an integrating ENS. The trigger messages in MQSeries have inspired our work on adaptive event handling.

4.2.2 Internet Notification Protocols

Several protocols have been proposed for detecting changes in Internet sources. Most of these approaches have not been implemented but remained as W3C working drafts or standard proposals. The proposals for new protocols are usually characterized in their relation to HTTP [RK98b]: (a) HTTP extensions (e.g., GENA Base [CA98], DRP [Hof97], ENP [Red98], SWAP [Swe98]), (b) loosely based on HTTP (e.g., RVP [CD97]), and (c) not based on HTTP (e.g., BLIP [Jen98], SGAP [Day98]).

Other approaches are protocols for widely used services such as NNTP for Newsgroups, SMTP for List addresses, and HTTP for Callbacks. A third group of Internet protocols supports the information exchange in personalized Internet applications, for example, P3P [Rea98], PICS [W3C03a], and OPS [NET99b]. These so-called social protocols support simple personalization and filtering of services. Emphasis is on the definition of profiles that describe general personalization parameters. Here, we briefly introduce an example for each of the three groups: the HTTP-extension GENA Base, the newsgroup protocol NTTP, and the P3P standard approach.

General Event Notification Architecture Base (GENA Base). The General Event Notification Architecture Base [CA98] defines a HTTP notification architecture that supports the transmission of notifications between HTTP resources using call-back. Different to ENS, the resources are not mutually anonymous. The protocol defines the components subscription and notification. Resources transmit notifications about events (resource change) as HTTP request using the notify method, client pull is also supported. Event messages reference state attributes and their current value. Subscriptions establish a relationship where automatic notifications are triggered by certain events, i.e., attribute changes; the communication partners are not anonymous. The subscribers of a resource receive all event messages for attributes that have changed. The GENA Base format is used for event notification in the UPnP^{TM} architecture that offers peer-to-peer network connectivity, e.g., in Windows XP^{TM}.

Network News Transfer Protocol (NNTP). The Network News Transfer Protocol [KL86] is the main protocol used for the Usenet news system. It implements a many-to-many communication. The messages provide additional header-information that can be evaluated by NNTP-commands to direct their distribution. NNTP servers store messages, forward them to clients, and exchange messages with other servers. Usenet messages are posted in newsgroups that are organized in a hierarchical manner based on subjects. NNTP provides simple filter capabilities for selecting messages according to group names and message date. The filter mechanism is not very sophisticated, thus limiting the application of the NNTP for event notification services.

Platform for Privacy Preferences (P3P). The Platform for Privacy Preferences Project [Rea98] defines a standard format to describe privacy information for web-sites, e.g., the data category describes the site's content, the data collection purposes, and the advised practice of usage. The profiles can be used for content regulation, web-markets, and privacy. These protocols are also called social protocols. The focus is on the privacy information definition, which is similar to a profile definition.

The P3P format can be automatically read by P3P client agents in order to inform clients about the site's practices and to allow for automatic decision-making based on these practices. The P3P information will be exchanged in RDF/XML format. P3P is a recommendation from the W3C, currently only a small number of client agents and policy generators has been implemented.

The first two groups of Internet notification protocols define communication means to establish event-based service on the system level. The protocol implementations can be seen as simplified and system-related implementations of event notification systems. The third group of social protocols is related to profile definition languages.

4.2.3 Low-level Event-based Communication

A number of techniques for network communication implement transport mechanisms for notifications. These technologies implement wide-area notification, even though none of these technologies is designed to realize an Internet-scale event notification service as addressed in this thesis.

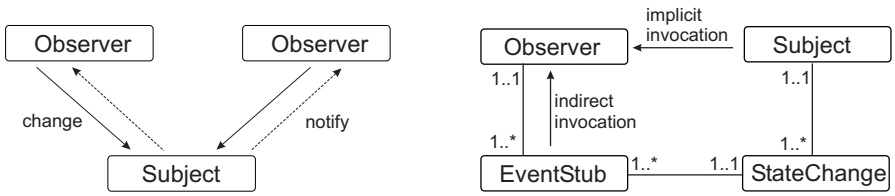
IP Multicast. IP Multicast is a network-level infrastructure that extends the Internet Protocol (IP) to realize one-to-many communication. The network implementing the extension is called MBone. Here, a multicast address is a virtual IP address that corresponds to a group of hosts. Data addressed to a host group are routed to every host that belongs to the group. The cooperation of web-based network management and IP Multicast has been analyzed by Martin-Flatin [MF98b, MF98a].

Domain Name Service. A Domain Name Service (DNS) maps symbolic domain or host names onto IP addresses. DNS has a distributed architecture, where DNS servers form a hierarchical structure. This hierarchical structure is also present in the data that are managed (host names are partitioned in domains and subdomains). In contrast, event notifications are not hierarchically structured.

Although the communication techniques introduced here cannot be employed directly to implement an ENS, several architectural ideas can be adopted. Examples are the routing and filtering techniques applied in the Siena system.

4.2.4 Design Patterns for Event Notification

Several design patterns for event observation and notification have been proposed, e.g., the observer pattern [GHJV95] and the event notification pattern [Rie96]. Both patterns are approaches to instantiate the observation process. The structure of the observer pattern [GHJV95] is consistent with our reference model (see Figure 4.13(a)). It consists of observers and subjects. The observer (*Invoker*)[3] changes the subject's (*Information object*) state and, subsequently, various observers (*Observer*) are notified about the state change. The pattern defines a one-to-many dependency between a subject object and any number of observer objects so that, when the subject object changes its state, all its observer objects are notified and updated automatically.



(a) Observation Pattern (b) Event Notification Pattern

Figure 4.13: Object and observer interactions in design patterns

The event notification pattern [Rie96], in contrast, additionally models a StateChange as an object, as well as an EventStub (see Figure 4.13(b)). The Subject defines its abstract state in terms of StateChange objects. The StateChange object offers operations to register and unregister observers via EventStub objects. Observers provide the EventStubs with an operation reference to be called in case of invocation. The StateChange can be compared to event messages in our model, the EventStub models simple routing filters and observer interfaces.

[3] In parentheses, we show the respective terms used in our model.

System	R1 Elaborated Event Model	R2 Active/Passive Observation	R3 Integration of Composite Events: Profile Support	R3 Integration of Composite Events: Integration	R4 Flexible Filtering	R5 Temporal Awareness	R6 Performance	R6 Scalability
SIFT	- -	- -	- -	- -	- -	- -	-	- -
Elvin	+ - 2)	- 1), 2)	- -	- -	- -	- -	+	+ -
Siena	+ - 3)	- -	+ -	+ -	- -	- -	+	++
OpenCQ	++	++	++	++	-	-	+ -	+
CORBA NS	- -	- -	- -	- -	- -	- -	+ -	++
OmniNotify	+ -	- -	+	+	- -	+ -	+ -	+
COBEA	+ -	- -	+	+	- -	+ -	+ -	+
Keryx	-	- -	- 8)	- -	- -	- -	+	++
LeSubscribe	-	- -	- -	- -	- -	- -	+	+
Gryphon	-	- -	- -	-	- 7)	+ -	+	++
REBECA	+ -	- -	+	+	+ -	+ 9)	++	++
NiagaraCQ	+ -	++	- -	- -	- -	- -	++	++
SAMOS	+ - 4)	+ -	+	+ -	+ - 5)	-	+ -	-
Sentinel	+	+ -	++	+	+ - 6)	-	+ -	+

1) External extension possible
2) Quenching comprises event basis
3) Advertisments
4) Event model with restricted notion of (external) events
5) Fixed parameters
6) Parameters not sufficient and operator-based
7) Few simple parameters (first)
8) no composites but sophisticated profiles on promitive events
9) with restrictions

Figure 4.14: Comparison of related work with regard to the requirements for event notification services as identified in Chapter 2 at Page 19.

4.3 Summary of the Analysis

In this chapter, we compared selected event notification services to our model of ENS. Additionally, we analyzed related technologies from which basic concepts can be adopted for implementation in event notification services.

Figure 4.14 summarizes the results of our analysis according to the requirements R1 to R6 defined in Chapter 2. Along the requirements, different systems and technologies were evaluated whether they provide strong support (++), weak support (+), partial support (+ -), weak exclusion (-) or complete exclusion (- -) of the requirements. Our analysis of selected services led to the following observations:

- R1: Elaborated Event Model. Most of the evaluated systems are not based on an event model as described in Chapter 3, but support only the filtering of documents (event messages). Systems based on event models are OpenCQ and the active database system SAMOS and Sentinel. Samos only supports database-internal events; Sentinel supports database interval and external events that are actively provided to the service. The systems with partial support (+ -) of an event concept support only selected event types.

- R2: Active and Passive Observation. Most of the evaluated event notification systems use passive observation, i.e., event messages are sent by the providers. Hence, the systems do not implement the full event notification sequence from event observation to notification as described in Chapter 3, but only the filtering of notification messages that enter the system. So far, only OpenCQ and NiagaraCQ implement the complete structure as described in our reference model.

- R3: Integration of Composite Events. In general, composite events are rarely supported in ENS; we specifically selected those systems for our evaluation, which consider composite events. If composite events are covered by a system, passive events are rarely implemented (only in OpenCQ, Sentinel, and SAMOS). Therefore, most of the systems only provide *cooperation* of information from different sources but no *integration* of information. Additionally, most active database systems support database-internal primitive events (as in SAMOS), only few systems consider events that occurred outside the database.

- R4: Flexibility in Event Filtering. The REBECA system and the active database systems SAMOS and Sentinel enable control of the filter semantics via parameters for event identification and consumption. In SAMOS, these parameters are fixed and cannot be changed during the matching phase. REBECA supports only a few parameters that cannot be changed at runtime; in Sentinel the parameters are operator-based and cannot be flexibly changed. In general, flexible (adaptable) filtering that adapts to changing sources and applications is neither supported by current implementations of event notification services nor by event-based technologies.

- R5: Temporal Awareness. The only system with awareness for temporal delays affecting the event composition is REBECA. The approach of REBECA is not sufficient, because no active observation is supported, which would influence the event ordering. Additionally, the assumed FIFO characteristic of message queues (First In First Out – no message overtaking occurs) does not hold in open distributed systems as the Internet. The influence of temporal delays due to observation and message exchanged in combination with the used time systems has not been addressed so far.

- R6: Performance and Scalability. Performance and scalability are achieved in the analyzed ENS by efficient (local) filtering techniques and/or event routing in the network. Efficient filtering based on profile trees has been used for Elvin, LeSubscribe and Gryphon. NiagaraCQ and OpenCQ use database-inspired trigger grouping and query optimization.

  Most of the systems employ distributed software components, e.g., CQ, LeSubscribe, and Elvin. However, a distributed system architecture is not necessarily implied. The architectural styles vary between centralized systems and a network of event servers that are organized hierarchically or in peer-to-peer manner. A number of services is based on a centralized architecture, e.g., SIFT, Elvin, LeSubscribe. A network of notification systems is used in Siena, Keryx, Gryphon, REBECA. The events and profiles are routed through the network, different routing strategies have been proposed. REBECA is the first system that takes advantage of profile locality, i.e., the characteristic that profiles are not uniformly distributed.

  We believe that this profile property of non-uniform distribution also applies for events and that this characteristic can be used to further enhance a system performance.

Summarizing, we can state that none of the examined systems or technologies provide support for all of the requirements arising from scenarios for the different applications. Consequently, we propose an adaptive integrating event notification service that gives sufficient support for the requirements identified. While designing this service, we considered the lessons learned from the analysis, adopting

promising strategies from the analyzed approaches. In the next four chapters 5 to 8, the specific aspects of our design are discussed. In chapters 9 and 10, we describe our prototypical implementation of the proposed service and present selected test results. In Chapter 11, we resume the present discussion of the state of the art in event notification services and compare the design proposed in this thesis with the related work.