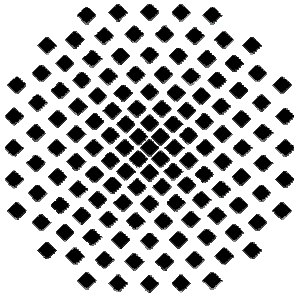


Universität Stuttgart

CR-Klassifikation:

C.2.4 (Distributed Systems)

C.4 (Performance of Systems)

E.1 (Data Structures)

H.3.3 (Information Search and Retrieval)

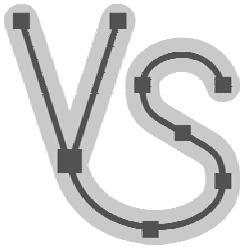
H.3.4 (Information Storage and Retrieval Systems and Software)

Bericht Nr. 2004/02

Event Management for Mobile Users

Martin Bauer

Fakultät Elektrotechnik, Informatik,
Informationstechnik
Universität Stuttgart
Institut für Institut für Parallele und
Verteilte Systeme, Abteilung Verteilte Systeme



Abstract

This technical report presents the results of the project “Event Management for Mobile Users”, a research cooperation between Universität Stuttgart and Microsoft.

In Vienna, Microsoft chief executive Bill Gates recently expressed his vision of a seamless interaction of different computing devices, from PC to mobile phone, that proactively support the users wherever they go. Among other things, he said: “For example, if you want to be notified about something that's changing, that's important to you, software should know which device you have with you and should know what you're doing, know the context to understand if interrupting you with this new information is appropriate or not.” (Bill Gates, January 28, 2004, [Los Angeles Times 2004])

This project may present one step towards realizing his vision. It is about event support for mobile users. Depending on their current context, they want to be informed about events that occur in the world around them – the physical as well as the virtual world of digital information systems. So, to optimally support its users, future generations of web services will need information about the real-world context of the user, especially their spatial context.

This project was carried out in close cooperation with the Nexus project at Universität Stuttgart, whose goal is to support mobile context-aware applications based on a distributed spatial world model. Events of special interest in this context are spatial events, i.e. events that occur when a certain spatial constellation of objects is reached, e.g. when two people meet or when a customer enters a shopping mall. As the underlying world model is distributed, the events have to be observed on a distributed model.

The number of potential spatial events is not restricted, e.g. the event that a user enters an area could be of interest for arbitrary areas. Also, as the spatial world model is distributed over many servers, a local observation is no longer sufficient. Therefore, the well-known publish-subscribe paradigm, in which the observation occurs implicitly within local observers, has to be extended. We propose an observation-notification paradigm, in which the observation of events has to be explicitly initiated by interested clients. The event service that conforms to the new paradigm consists of two components: an observation service, which observes events through a distributed model, and a notification service, which efficiently delivers event notifications. This paradigm applies to all scenarios in which the data needed for the observation of events is distributed over multiple sources.

The user wants the specification of events to be as simple and easy as possible. He only wants to specify *what* event is to be observed, not *how* the observation of the event is realized. Therefore the distribution aspects should be transparent to the user. However, these aspects have a strong influence on the accuracy of the model and thereby directly affect the accuracy of event observation. We propose that the user specifies an event as a predicate, which becomes true when the event has occurred, plus a threshold probability. If the probability that the event has occurred is above the threshold probability, the event is considered to have occurred and an event notification is sent.

The above sketched solution requires calculating the probability with which an event has occurred. We first show which parameters influence the accuracy of the data. We then present update protocols that guarantee a certain accuracy of data in the observer model, i.e. the model where the event is actually observed. Finally we show, how the occurrence probability can be calculated. Based on the identified parameters, the placement of the observation in the system can also be optimized with regard to the accuracy of the data.

We have implemented the event service and integrated it into the Nexus platform. As the Nexus platform is intended as an open platform with possibly world-wide scale, scalability, efficiency and interoperability have been important requirements for the design of the components. To support interoperability, we have built on standard technologies like XML, HTTP and SOAP.

The evaluation of the event service within the Nexus context shows the feasibility of the approach. We were able to show that the event service performs adequately in certain example scenarios. Experiments with a large-scale scenario are the next steps on our agenda.

Table of Contents

Chapter 1 Introduction

The term *event* describes a natural concept of everyday life. Something happens, we observe it and react to it. For example, when the traffic lights turn from green to yellow and eventually to red, the driver of an approaching car will observe this event or rather this sequence of events, and might react to it by hitting the breaks. Another example of an event occurrence in a real-world context is the ringing of a telephone. Usually we react by picking up the receiver to answer the call.

In the world of distributed computing, the event concept also plays an important role. Event services or publish-subscribe services, as they are also often called, are used in areas such as information dissemination services, alerting services, network and distributed systems management, or enterprise application integration. Often the term event is used for both an occurrence and the message informing about the occurrence. This may be due to the fact that the purpose of these event services is to efficiently deliver event notification messages. The observation of events is the task of the event sources or publishers. As our focus is on the observation of events, we want to distinguish between the two. We use the term *event* for the occurrence or change, and the term *event notification* for the message that informs about the event occurrence.

With the proliferation of sensor systems, large amounts of dynamic data become available. At the same time more and more information is processed and generated in information system. Often, the user just wants to be notified when the data indicates that an event of interest has occurred.

If we look at application scenarios in the domains of ubiquitous, pervasive and mobile computing, users walk around with mobile computing devices. An important difference in the way desktop and mobile computers are used is the user's focus. A user at a desktop computer is typically working on a task that is to be solved almost exclusively with the help of the computer, e.g. the user is writing a text with a word processing application. With a mobile computer, however, the main task of the user may be something completely different. He or she may be walking around performing other tasks, and the mobile computer supports the user by helping him or her with subtasks. For example, the mobile computer may help the user to remember certain tasks, to navigate, or to support maintenance staff.

The Nexus project at Universität Stuttgart aims at supporting such mobile users. The goal is to provide a platform for mobile, context-aware applications based on a distributed spatial world model. The spatial world model provides a model of the physical world augmented with additional virtual information, e.g. web pages that can be placed at a location where they are relevant. The spatial world model is distributed over many servers. The distribution is according to the geographic area, but also according to the type of object, e.g. the position data of mobile objects is stored by the Nexus location service, whereas the data of stationary objects is stored by spatial model servers.

Events are well suited for the interaction of the mobile platform and the user. The user can specify the situations he is interested in ahead of time. Then, when he is walking around, the platform can detect if the current situation corresponds to a specified one and the user can be informed about this proactively.

Observing events on the side of the platform also has the advantage that the limited resources of the mobile computing device are not wasted. Mobile computing devices have limited battery power. They also have a wireless network connection that has a lower bandwidth and higher delays than infrastructure-based connections. In the case of events, communication only takes place when necessary, i.e. when an event has occurred. As we will see, observing

events in the infrastructure, “close” to the source also provides a better event semantics than trying to observe events on the mobile device itself by constantly “polling” the information sources.

Events of special interest in the context of the Nexus platform are spatial events, i.e. events that occur when a certain spatial constellation of mobile (or mobile and stationary) objects is reached. Examples of why the user could be interested in certain spatial events are the following:

- If the user is walking past a shoe shop in a shopping mall, he may want to be reminded that he wanted to buy shoes.
- If there are more than 5 people in the meeting room, the user may want to be informed so that he does not miss an important meeting.
- If a user enters the room, the light and temperature could be adjusted to the user’s preferences.

The last example is from a different setting. Here events are used to realize an “intelligent environment”.

The overall goal of the project was to investigate scalable event services with a special focus on the area of mobile and ubiquitous computing. The Nexus project provided the environment, both as a source for requirements and as the context for the evaluation of the event service.

In this report we present the results of our research. We put our focus on the observation of events through a distributed world model, as this seemed to be the most interesting aspect from a research perspective, without neglecting the other aspects.

We present an approach that allows the specification of events by the user, the observation of events through a distributed model and the efficient delivery of event notifications to interested clients. We have implemented a prototype of the event service to show the feasibility of the approach and to evaluate it regarding the requirements that we have set ourselves.

The structure of the report is as follows: In Chapter 2 we discuss the related work. In Chapter 3 we present those aspects of the Nexus project that are relevant for our work. The overall requirements are defined in Chapter 4, before the system architecture is presented in Chapter 5. The theoretical part begins with the event model in Chapter 6 that provides an event classification according to parameters that are relevant for event observation. A uniform approach for specifying events is presented in Chapter 7. In Chapter 8 we show how the specified events can be observed through the distributed model and discuss realization issues. The optimal placement of the observation within the system is the topic of Chapter 9. In Chapter 10, Chapter 11 and Chapter 12 the design and implementation of the integration of events into the location service, the observation service and the notification service are presented. In Chapter 13 we evaluate the integrated system, before concluding the report with a summary and an outlook on future work in Chapter 14.

Chapter 2 Related Work

Events and event-based communication play an important role in many areas of computer sciences: There is an event-based programming paradigm, e.g. graphical user interfaces are typically programmed on that basis. Events are used in active databases, e.g. to trigger further changes when the data in one table has changed. In information dissemination systems, the concept of events can be found, as well as in distributed systems and network management applications. In the first section we look at some of the characteristics of event systems. In the second section, we identify those areas that are more closely related to our work. In the following sections we look at these areas in detail.

2.1 Characteristics of Event Systems

In the following, we first discuss a number of characteristics that are often associated with event systems:

- **Push Communication**

  Event-based communication is typically source-initiated. Clients (or subscribers) can subscribe for events and receive event notifications when an event has occurred. The information is actually *pushed* onto interested clients. Therefore, this style of communication is also called *push* communication.

Push communication is in contrast to the request-response style of communication, e.g. in client-server applications, where the initiative is on the side of the client. There the client *pulls* the information and therefore we also talk about *pull* communication.

The use of push communication can help to reduce the communication overhead, because communication only takes place when something has actually happened. In the alternative case, the client would have to poll the server for the same information regularly. If multiple clients are interested in the same events, the use of a multicast notification service can help to reduce communication overhead even further, because the same event notification has to go over the same link only once. Overall, event-based communication can improve scalability by reducing communication overhead.

- **Asynchronous Communication**

  Event-based communication is often asynchronous. The event source (or publisher) does not need to wait for acknowledgements. It can continue its execution.

- **Anonymous Communication**

  The event sources (or publishers) and interested clients do not have to know about each other. The communication can be completely anonymous. The event notification mechanism just needs to know how to deliver the event notifications.

- **n:m Communication**

  With event-based communication an n:m communication can be realized, i.e. there can be n event sources (or publishers) and m clients.

These characteristics make event-based communication ideally suited for the loose coupling of software components. The software components do not need to know anything about each other, they just have to provide the information that an event has occurred, so that interested components can be informed.

The characteristics also fit very well into our application domain, as push-based communication reduces the load on mobile devices and their wireless communication access. Asynchronous communication is important for the decoupling of the event sources, e.g. servers that are part of the infrastructure, from the mobile clients and anonymity of the event sources fits very well with the idea that the mobile clients should not need to know about the details of the distributed infrastructure.

2.2 Overview of Related Work

In the following we look at those research areas that are closely related to our work. Since our focus is on event observation through a distributed model, we are primarily interested in events in distributed systems. However, we will also look at centralized systems in which events based on complex relations between objects can be specified and at centralized systems that fall into our application domain.

Figure 1 shows four research areas that overlap with our research focus:

- Active Databases – support the specification of triggers that initiate further actions when a certain event occurs.
- Distributed Event Services – focus on the efficient delivery of event notifications to interested clients.
- Global Predicates – allow the specification of predicates on distributed global state.
- Spatial Events – occur when a certain spatial constellation of objects is reached.

These four areas will be discussed in more detail in the following sections.

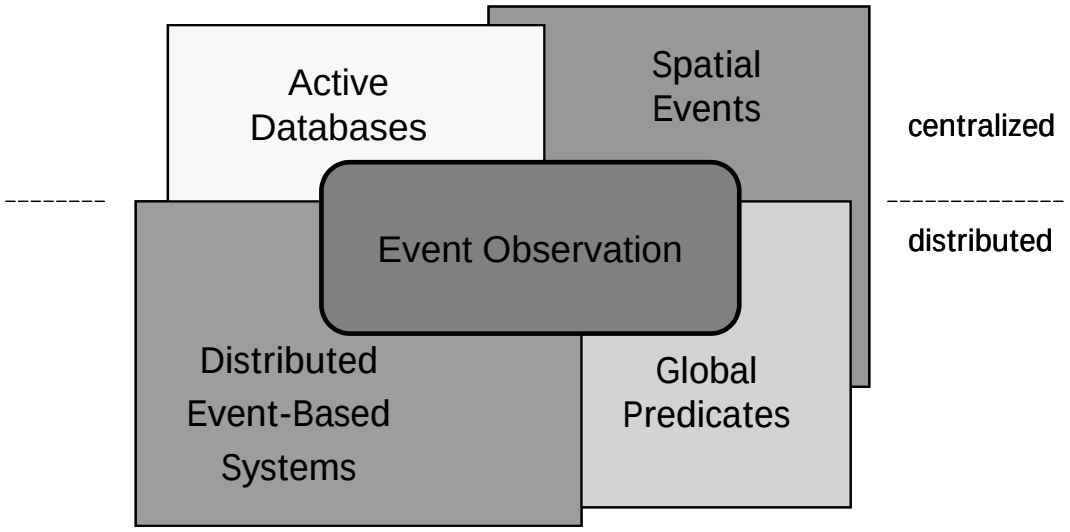


Figure 1: Related Work

2.3 Active Databases

Traditional databases store data persistently and provide efficient access to the data. Active databases extend this functionality through triggers. The triggers are activated by events, e.g. a data update in a certain table. Triggers are typically defined in form of Event-Condition-Action-Rules (ECA). Based on the occurrence of a simple event, a condition is

checked, and if that condition is fulfilled, an action is carried out. The condition can be a method or procedure that returns true or false.

Currently the trigger concept is part of SQL 99 [Matthiessen & Unterstein 2000], which knows data manipulation operations, i.e. insert, update, and delete, as possible events. The standard is supported by DB2 and Sybase [Schmidt & Demmig 2001], Informix additionally supports trigger for select statements and Oracle also provides triggers for changes in database schemas (i.e. create, alter, and drop), user login/logout, database shutdown and server errors.

There has been some research on the composition of simple events in active databases.

[Gehani et al. 1992] [Chakravarty et al. 1993] [Dittrich & Gatziu 2000] and others have proposed event algebras with a number of predicate constructors. The expressiveness of these language is usually restricted to either that of regular expressions or propositional logic extended by operators expressing temporal relationships. These constructors may be sufficient for combining events in active databases, but cannot express the events we want to observe. However, active databases can serve as a basis for implementing event sources on which more complex events can be observed.

2.4 Distributed Event Services and Publish-Subscribe Services

Distributed event services and publish-subscribe systems are often used as synonyms describing the same concepts. Event service may be the more general term, whereas publish-subscribe service describes how the system works.

Figure 2: Event Service Architecture in Publish-Subscribe Systems

Figure 2 shows the underlying event architecture. There are publishers who publish event notifications and there are subscribers who subscribe to certain kinds of event notifications. When an event notification is published the publish-subscribe service is responsible for delivering the event notification to all interested subscribers. Some systems require that before publishers start publishing event notifications of a certain type, they have to advertise this, so that the structure for efficiently delivering event notifications can be set up.

Publish-subscribe systems can be classified according to the communication mechanism, which can be unicast-based or multicast-based, the underlying distribution structure, which can be hierarchical or peer-to-peer, and the filtering mechanism, which can be id-based, subject-based or content-based. Id-based filtering means that clients can specify an event id they are interested in and they receive only event notifications with this specific id. Subject-based filtering allows the specification of a subject, possibly with wildcards. Finally, with content-based addressing, the client can specify the content of event notifications he is interested in, which is typically done in form of attribute-value pairs.

Whereas filtering applies to the content of a singe event notification, event composition concerns the relation between multiple different event notifications. With the available operators event patterns can be specified.

As there are too many different event services to discuss them all, we will only provide a number of examples:

The CORBA Event Service Specification [OMG 2001] describes an event service based on an information channel with multiple *suppliers* and *consumers*. With the event channel a decoupling between suppliers and consumers is realized. Event channels have to be set up explicitly. The specification does not give any details how the event channel has to be implemented. Composite events can only be realized through building a tree with multiple channels and the composition takes place at intermediate supplier/consumer nodes.

TIBCO is a commercial product that is used in financial services likes stock information. It is based on a hierarchical system and provides reliable delivery of event notification. The filtering is subject-based.

The CORBA Notification Service [OMG 2002] goes a step further and allows content-based filtering. However there is only one filter object per channel and the filter constraint language is based on Boolean expressions, so it has a relatively limited expressive power.

The READY [Gruber et al. 1999] Event Service allows the same filtering expressions as the CORBA Notification Service, but additionally supports different composition operators like AND, OR and SEQUENCE. The WHERE operator allows the analysis of relationships between sub-events. For efficiency reasons, the event specification is moved towards the publishers.

JEDI [Cugola et al. 1998] is an object-oriented infrastructure that supports the development and execution of event-based systems. Events in the sense of the JEDI system are special kinds of messages consisting of strings, with the first string being the event name and the following strings the event parameters. JEDI provides filtering with regular expressions over the strings.

Herald [Cabrera et al. 2001] is an event service developed by Microsoft Research in Redmond. The event distribution is based on different rendez-vous points. A special focus has been on the scalability of the service and resilience against failure. The basic service provides no service for finding rendez-vous points, no complex specification and no composition of events. The idea is that such functionality can be layered on top.

The goal of the Siena Project [Carzaniga et al. 1998] is to develop an Internet-scale event service. It supports content-based filtering and event patterns.

Gryphon is a research project at the IBM, Watson Research Center. The goal of the Gryphon event service is to distribute large amounts of data in real-time, e.g. news distribution at large-scale "events" like Olympic games. It supports topic-based and content-based addressing, and addresses security and privacy aspects, but not event composition.

The Overcast service [Janotti et al. 2000] is based on a reliable multicast protocol on an application-layer overlay network. Its goal is the efficient use of bandwidth and it is targeted at content distribution, supporting only 1:m communication.

Scribe [Rowstron et al. 2001] is an event notification infrastructure based on the Pastry [Rowstron & Druschel 2001] framework. Both systems have been developed as part of a cooperation between Microsoft Research in Cambridge, Rice University, Purdue University, the University of Washington and Microsoft Research in Redmond.

Pastry provides a basic structure for peer-to-peer applications. It is based on an overlay network and the routing is done according to node ID. The node with the closest ID can be used as a rendez-vous point. We have used Pastry as a basis to implement a distributed register in the notification service (see Subsection 12.2.3).

The goal of the Hermes project [Pietzuch et al. 2003] at the University of Cambridge is to develop a content-based publish-subscribe system with composite events. Event composition is based on regular expressions extended by operators expressing temporal relationships. The observation is realized by mobile detection objects that are optimally placed in an overlay distribution network.

Overall we can see that there are a huge number of different event services. Their main focus is on the efficient delivery of event notifications in different scenarios. Some services provide the functionality to observe composite events. However the expressiveness of the respective event algebras is usually limited to regular expressions [Pietzuch et al. 2003] or propositional logic with temporal extensions [Hinze & Voisard 2002]. In general, filtering and pattern recognition are strongly intertwined with the delivery of event notification.

The supported composition operators allow the combination of arbitrary event notifications. In contrast to these composite events, we want to observe arbitrarily complex events on the state of a distributed model. This requires a much more complex language for describing events. In our case, the number of possible events is infinite, so they cannot be automatically be provided by publishers; the observation of events has to be explicitly initiated. An additional issue is the limited accuracy of the available data that has to be taken into account. So the existing event services do not provide a solution to our goal of observing events on a distributed model.

2.5 Global Predicates

Global predicates describe global properties in distributed systems and are defined over global state. A typical application area for global predicates is the debugging of distributed applications, where the question of interest is, if the property holds during the distributed execution or in other words, if the predicate is satisfied at runtime. As there cannot be an omniscient observer who can put all local events into a global order, only the causality of events can be taken into account, i.e. the effect must not be considered before its cause.

Causality can be fully characterized using vector timestamps [Schwarz & Mattern 1994]. The vector timestamp includes an element with a logical time for each process. This requires that the number of processes is fixed and known beforehand. Each process updates its logical

time in each step and sends its vector with every message. On receiving a message, the local vector is updated taking the maximum for each vector element of the previous local vector and the vector that came with the message. With this information it can be determined if two events are causally dependent or concurrent.

The approach suggested by Cooper and Marzullo [Cooper & Marzullo 1991] (and others), constructs a lattice of global states. A lattice of global states describes all possible sequences of local states that are consistent with causality. This corresponds to all the sequences of events that could in principle have been observed by an observer. A path through this lattice corresponds to one possible observation. Cooper and Marzullo define three different predicate qualifiers, *possibly* Φ, *definitely* Φ and *currently* Φ and provide algorithms for determining if they hold. *Possibly* Φ holds, if there exists a path through the lattice of states so that Φ holds for some global state on the path. This means that there is one possible observation of the distributed computation for which Φ holds. *Definitely* Φ holds, if all possible paths through the state lattice contain a state for which Φ holds. This means that Φ holds for all possible observations. *Currently* Φ holds, if Φ holds at the current point in the computation. For determining if *currently* Φ holds, it is necessary to temporarily block processes. However, it is guaranteed that there is a logical execution of the unblocked system so that Φ holds. Unfortunately, it can be shown [Schwarz & Mattern 1994] that while blocking a valid Φ can go undetected.

A general problem of *possibly* Φ and *definitely* Φ is that the whole lattice of states has to be considered, a computation which can be prohibitively expensive, because there may be $O(m^n)$ global states, where n is the total number of processes and m the maximum number of relevant execution steps of a single process.

The problem of applying this approach to our problem is that in a lot of cases causality between different (sub-)events may exist in the real world, but this is not necessarily represented in the model. In addition, the model may be distributed over a large and possibly changing number of servers, which would make the use of vector timestamps problematic.

This means that we have to rely on real time for ordering the events. Then, we do not have a lattice of discrete states to determine if the event has occurred. The accuracy of the data may be limited which has to be taken into account for the event observation, i.e. it may only be possible to determine that an event has occurred with a certain probability. *Possibly* ϕ only says that the probability of event occurrence is greater than 0 and *definitely* ϕ that the probability is 100%, which may not correspond to what the user wants to know. Also, we have to detect the occurrence for an event at runtime, not after an algorithm has finished and blocking the system is not possible.

Therefore we cannot rely on the algorithms that have been presented for evaluating global predicates.

2.6 Spatial Events

A number of systems support location events or spatial events, i.e. events that occur when a certain constellation of mobile objects or a constellation of mobile objects with regard to their environment is reached. A number of groups from the University of Cambridge and the Olivetti Research Laboratory (ORL) that later became the AT & T Laboratories in Cambridge have conducted research in this area.

The ORL has developed two different kinds of indoor positioning systems, the Active Badge system [Want et al. 1992] and the Active Bat system [Harter et al. 1999]. The Active Badge system is based on infrared (IR) technology and can locate badges that are in range of a

receiver. The Active Bat system uses ultrasonic signals to locate an active bat. The accuracy is in the range of 10 cm.

Based on these positioning systems a number of location-aware systems have been built.

In [Hayton et al. 1996] composite spatial events are discussed that are all based on the Active Badge event that a certain mobile object was seen at a certain location. The composition operators are:

- WITHOUT (A-B): an event A has occurred without a previous B
- SEQUENCE (A;B): event A has occurred before event B
- OR (A|B): event A or event B has occurred
- WHENEVER ($A): whenever allows multiple independent evaluations. For each occurrence of A a new environment is created and the variables are instantiated accordingly, e.g. $enters(x);leaves(x) would only be true, if the same person had entered and later left the room.

The presented application has been realized specifically for the Active Badge system. It is targeted at a building-sized environment with a centralized location service and the expressiveness of the composition operators is limited.

The CALAIS system was implemented by Giles J. Nelson and presented in his PhD thesis [Nelson 1998]. The system is based on a location service with an in-memory location database and different location sensors, e.g. active badges or active bats, that transmit readings to the location service. The service supports standing queries for monitoring a given region, but does not provide support for more complex spatial events involving multiple mobile objects. The location service is a centralized system that does not scale to larger environments.

The Active Bat system [Harter et al. 1999] allows the efficient monitoring of spatial events. The application can register callbacks with a spatial monitor. The spatial monitor checks for the overlap and containment of areas. Both the location of mobile objects and stationary locations e.g. rooms or the space in front of a computer, are modeled as areas. By modeling locations of mobile objects as areas, the limited accuracy of the sensor information can be taken into account. All possible overlap and containment events are constantly observed. This scales to building scenarios, but not to larger areas. The events that can be registered are limited to areas that already exist within the system. Arbitrary areas are not supported.

QoSDream/FLAME [Naguib 2001] is a middleware for distributed multimedia applications. Position information from different sensor systems are aggregated and provided in a uniform format.

Similar to the original active bat system, locations of mobile objects and stationary locations are modeled as areas. The spatial relations manager observes all overlaps of regions. Filters provide higher-level events for which applications can registers. Again, the scalability beyond the size of buildings is questionable.

Overall it can be said that support for simple spatial events in centralized location services exists. The problem is the limited scalability and the limitations of the event specification languages that make the approaches unsuitable for our purposes.

2.7 Summary

The related work does not provide any approach that would allow us to observe complex real-world events through a distributed model. Active databases are not suitable for the dis-

tributed case, publish-subscribe services do not support distributed observation, approaches for evaluating global predicates based on causality cannot be applied and existing systems that support spatial events do not scale beyond building-sized scenarios.

Chapter 3 Nexus

As the project "Event Management for Mobile Users" was carried out in close cooperation with the Nexus project [Rothermel et al. 2003][Rothermel et al. 2003c], we give an overview of the Nexus project here. We concentrate on those aspects that are relevant for our project.

The Nexus project was both a source of requirements and an environment for the evaluation of the developed event service.

3.1 Nexus Vision

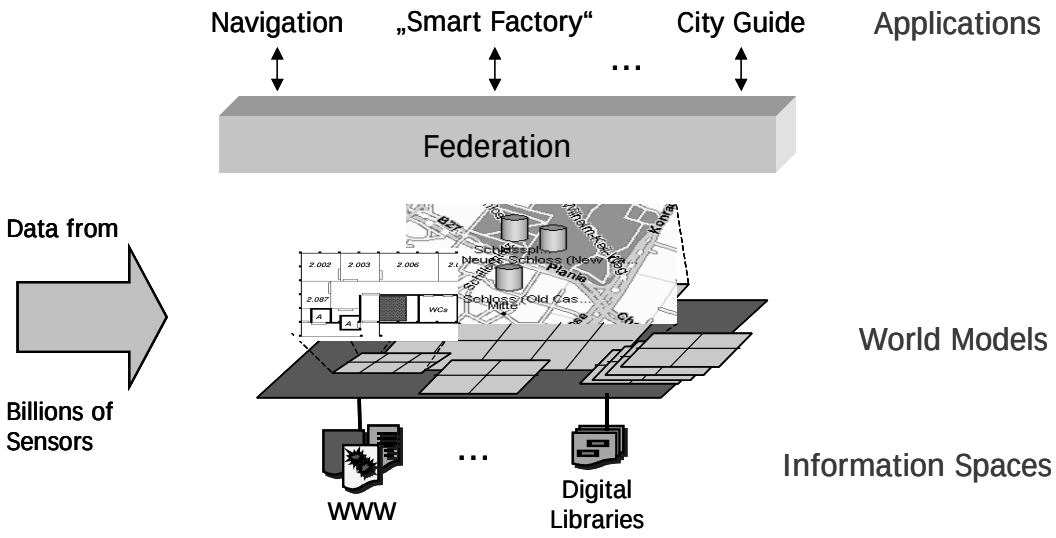


Figure 3: Nexus Vision
Source: [Nexus Project]

There are a large number of different context-aware applications. Typical examples are tourist guides, navigation systems, smart homes or the "smart factory", which is investigated as an application as part of the Nexus project. All these applications need some model of their environment. Some models may be relatively simple, e.g. a car navigation system based on static data only needs a course-grained model of the road network. However, the more sophisticated the applications become, the more detailed the model needs to be. For example, a navigation application for the visually impaired needs a much more detailed and up-to-date model. As the models become more detailed, the costs for building and maintaining them rise. Therefore it would be a good idea, if different applications could share the same model.

Still, it is unrealistic to assume that there will be one single detailed model of the whole world. It is a much more realistic assumption that there will be many different models that each model some part of the real world. For example, a city may decided to provide a two-dimensional model of its streets and houses as we know it from maps. The university may provide a more detailed three-dimensional model of its campus, and the computer science department may model the interior of its building as a set of floor plans.

The Nexus vision is to federate all these models and to provide a uniform view of the federated model to the applications (see Figure 3). So all applications can share the available data and if a new model becomes available all the applications can immediately profit by it.

The model not only contains static data. With the proliferation of sensors, the dynamic data from billions of sensors can be integrated to provide a detailed view of the current state

of the world. A lot of digital information about the real world exists in the World Wide Web and other digital information systems. This data can be integrated into the models using virtual objects like virtual post-its or virtual billboards as metaphors. Thereby the data of the physical world is augmented with virtual information. The resulting federated world model is therefore also called Augmented World Model.

3.2 Nexus Project Organization

The vision that we have just described is ambitious and a lot more research will be needed to realize it. The DFG (German Research Foundation) has recognized the need for more research in this direction and has provided funding for a Center of Excellence (SFB 627 - "Spatial World Models for Mobile, Context-Aware Systems"). The first funding period is four years and it can be extended by two more funding periods to a maximum of twelve years. Currently 30 researchers are working on the project. There is a strong core of researchers from computer science that are joined by colleagues from electrical engineering, photogrammetry, road and transportation science, manufacturing, and philosophy of science and technology.

3.3 Nexus Platform Architecture

In Figure 4 the architecture of the Nexus platform is shown. It is structured in three layers, the application layer, the federation layer, and the service layer. The concept of the federation layer is as follows: The client applications query the federation layer. In the first step, the federation has to identify the servers that manage the data needed for answering the query. This is done by querying the area service register. The area service register has the information about which servers store what kind of data for a given area. In the second step, the federation forwards sub-queries to the identified servers. The results are integrated with the help of model transformation functions that may aggregate or fuse the data making the result consistent. The integrated result is then returned to the requesting client application.

The federation layer also provides a number of value-added services. For example, a client application may request a map of a certain area from the map service. The map service then queries the servers that have the data it needs. Based on the data it draws the map and returns it to the client application. The navigation service can be queried for a navigation route from location A to location B.

In the service layer, the servers can be found that store the actual model data. Currently there are two different types of servers, spatial model servers and location servers. The spatial model servers store the data of stationary objects, e.g. roads, houses, rooms, … The location servers store the data of mobile objects, e.g. people, cars, trains, … The location servers are part of a hierarchically organized location service. As the position information of mobile objects is the basis for spatial events, we will look at the location service in more detail in the following section.

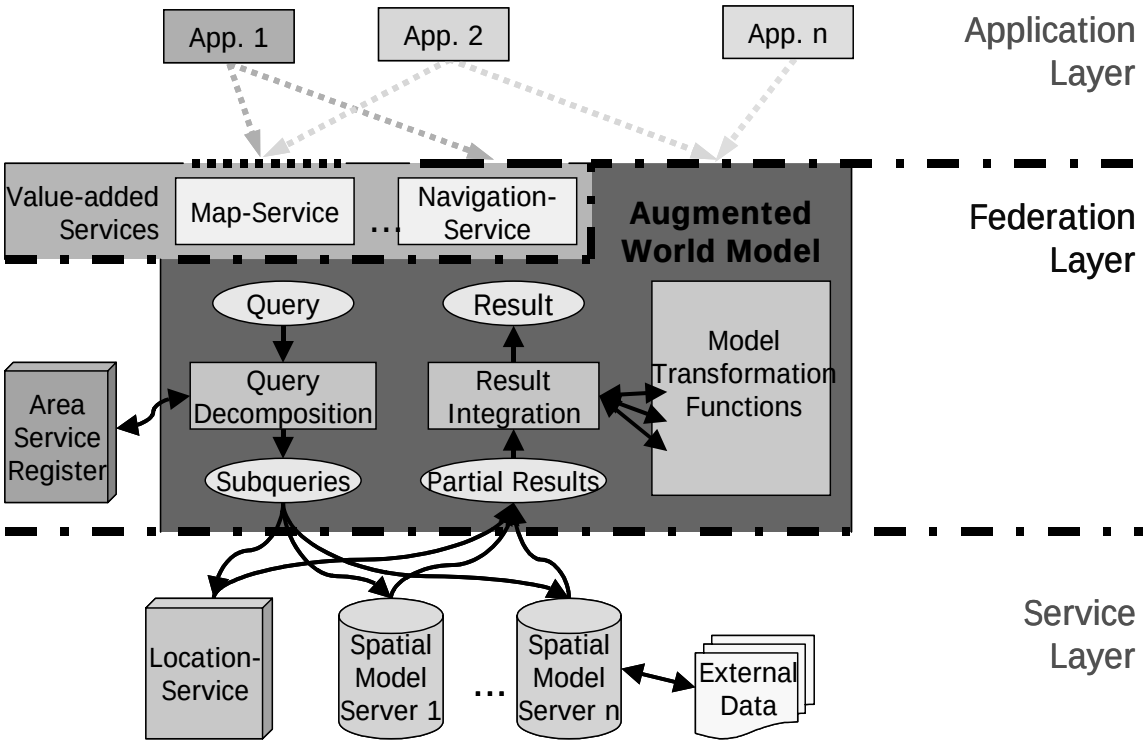


Figure 4: Nexus Platform Architecture
Source: [Nexus Project]

The Nexus platform is designed as an open platform into which different data providers can integrate their models. In order to do that, they have to implement an interface for their server that understands the Nexus query language AWQL (Augmented World Query Language) and returns the data in AWML (Augmented World Modeling Language). When they register their server with the Area Service Register, the federation can utilize the data for answering queries.

In addition to the query-based pull communication that we have just described, the Nexus platform also has to support event-based push communication. This is an important requirement from the perspective of context-aware applications, especially if the users are mobile and want proactive support from their system. This cooperation project has made an important contribution towards reaching this goal.

3.4 Location Service

The Nexus location service was developed by Alexander Leonhardi as part of his dissertation [Leonhardi 2003]. The goal was to design and implement a prototype of a location service that is scalable up to world-wide scale [Leonhardi & Rothermel 2001b][Leonhardi & Bauer 2001]. The supported functionality are position queries, range queries and nearest-neighbor queries. Position queries query the current position of a certain object, range queries query all mobile objects that are currently in a given area and nearest-neighbor queries query for the mobile objects that are currently closest to a given position.

In this cooperation project, we extended the functionality to also support a number of basic events. Examples are onEnterArea, an event that occurs when a mobile objects enters a given area, or onMeeting, an event that occurs when two given mobile objects meet, i.e. the distance between them becomes smaller than a predefined threshold. These and other supported events will be discussed in detail in Section 6.3 and Section 10.1

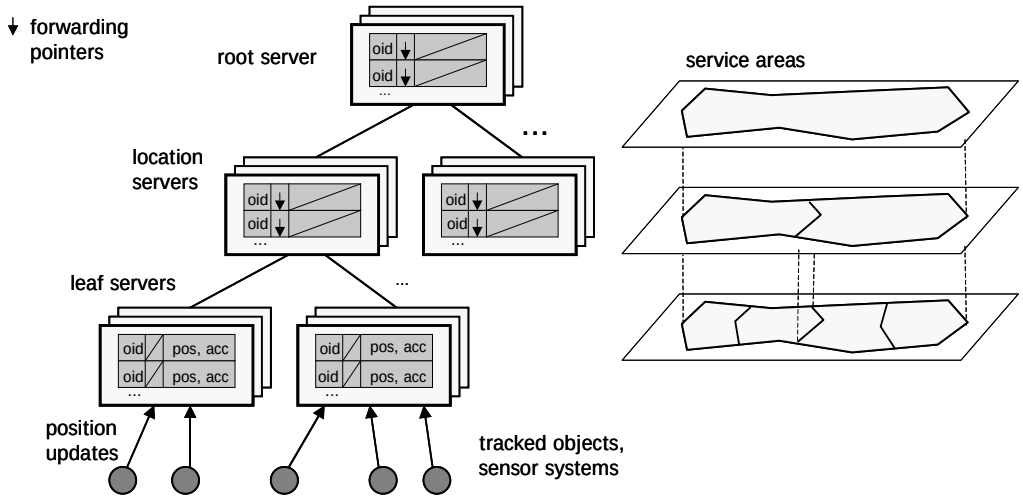


Figure 5: Location Service Architecture
Source: [Leonhardi & Rothermel 2001b]

The Nexus location service is organized in a hierarchy of location servers (see Figure 5). Each location server has a service area, which is the union of the service areas of the child servers. Only the leaf servers keep the actual position data of the mobile objects within their service area. The assumption is that these leaf location servers are “close” to the mobile objects and the position updates need only as much network bandwidth as necessary. The location servers that are higher up in the hierarchy only keep forwarding pointers for each mobile object in their service area that point to the child servers where the position data can be found. This information is necessary to find the position of a particular mobile object.

The location service supports different positioning systems, e.g. GPS for outdoors, and indoor position systems based on infrared, ultrasound or radio technology. Depending on the technology used, the accuracy of the position data can differ widely. The location service can deal with the limited accuracy and the different accuracy levels.

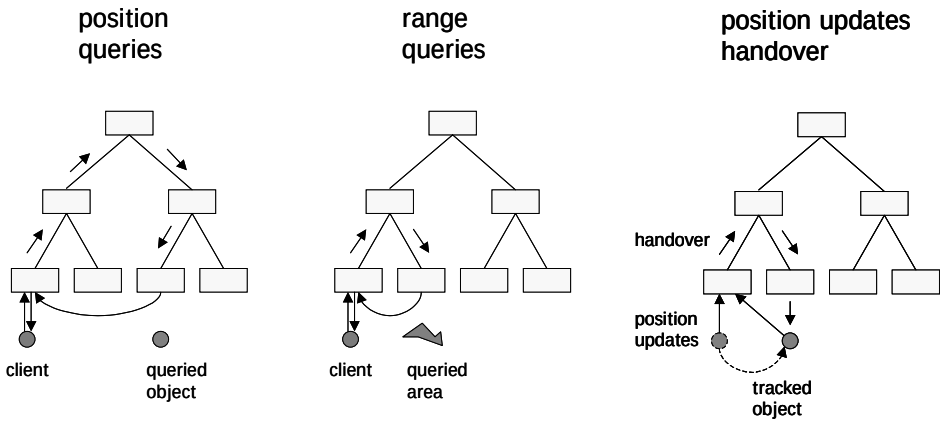


Figure 6: Location Service Functionality
Source: [Leonhardi & Rothermel 2001b]

Since only the leaf location servers store the actual position information, only they are potential event sources. The limited accuracy of the position information also has implications for the observation of events.

Figure 6 shows how the location service functionality is realized. For position queries, i.e. requesting the location of a particular mobile object, clients query their closest location server. If the location server does not have information about this object, the query is forwarded up the hierarchy until a server is found that has a forwarding pointer for the queried object. The forwarding pointers are followed down the hierarchy to the leaf location server that has the position information, which is then returned to the querying leaf location server and eventual to the client.

For range queries, the query is forwarded up the hierarchy until a location server is found whose service area completely covers the area specified in the range query. The location server forwards the query to all child servers whose service area overlaps with the queried area. This is done recursively until the leaf location servers are reached. They return the mobile objects in the area to the querying leaf location server that collects the answers and returns the integrated answer to the client.

Since mobile objects move, they may leave the service area of a leaf location server. In this case a handover has to be performed. When a location server realizes that the position in a position update is outside its service area, it initiates a handover request. The request is passed up the hierarchy until the new position is within the service area of the location server. It is then passed down to the child location server whose area includes the new position, updating the forwarding pointers on the way. The new child location server then informs the tracked mobile object that it is now the responsible server to which position updates have to be sent. The location service internally uses a UDP-based communication mechanism. For the external communication, it provides both UDP and SOAP-based interfaces.

3.5 Communication in Nexus

As the Nexus platform is designed as an open platform, it is important to use standard technology wherever possible. Therefore, it was decided to use XML and SOAP over HTTP, with a WSDL description for the server interfaces. Thereby the Nexus services are made available as web services.

Mobile devices rely on wireless communication and they move between different wireless networks, potentially even between wireless networks with different technologies. This means that mobility support like Mobile IP is required. In the Nexus project we have used the Mobile IPv6 implementation that was developed as a cooperation between Lancaster University and Microsoft Research in Cambridge. The implementation was extended to allow to switch between IEEE 802.11b and GPRS.

Another research goal of Nexus is to investigate what new forms of communication become possible with the existence of a world model and one of them is GeoCast. With GeoCast all the users in a geographic region can be addressed. GeoCast could also be used to deliver event notification to users in a geographic regions, e.g. for warning messages concerning accidents, or natural disasters like fires, floods or storms.

Chapter 4 Definitions and Requirements

In this chapter, we provide an overview of the goals of this project. We define a number of key terms and derive a set of high-level requirements that apply to the overall project. In the chapters describing the respective components the relevant requirements will be taken up, refined and extended.

The chapter is divided into two sections. The first section looks at the requirements for an event service from the user's point of view, whereas the second section concentrates on the requirements of the provider(s) of the event service.

4.1 User View

The user is interested in an event service, because he wants to be notified about events that occur in the world around him. The world around the user is the physical world, but also the virtual world of digital information systems. To observe events occurring in this world, we need a digital model of the world as our basis.

Definition 1: World Model

A world model describes a relevant subset of the world, real or virtual. The state of a model is described through a set of variables and the values assigned to them at a given time.

As the next step we can define an event with respect to this model.

Definition 2: Event

An event is an observable change in the state of the model.

So an event is a change that occurs. This change can be observed by the event service. As the result of observing an event, the user can be notified about the event occurrence.

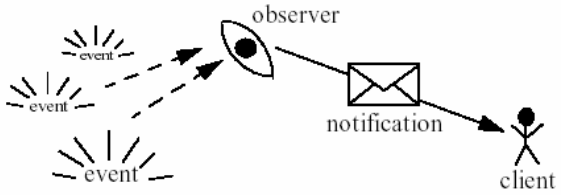


Figure 7: Event Observation

Definition 3: Event Notification

An event notification is a message informing the user about the occurrence of an event.

The underlying concept is shown in Figure 7.

A user interacts with the event service through a client (application). The concrete view the end user gets of the event service is therefore defined by the application programmer. As we want to develop an event service and not a specific client application, we do not distinguish between the view of the end user and the view of the application programmer and subsume both, end user and application programmer, under the term user.

From the point of view of the client, the event service is a single entity with a two-way interface (see Figure 8). It allows the client to register an event with the event service and the event service can notify the client about the occurrence of an event.

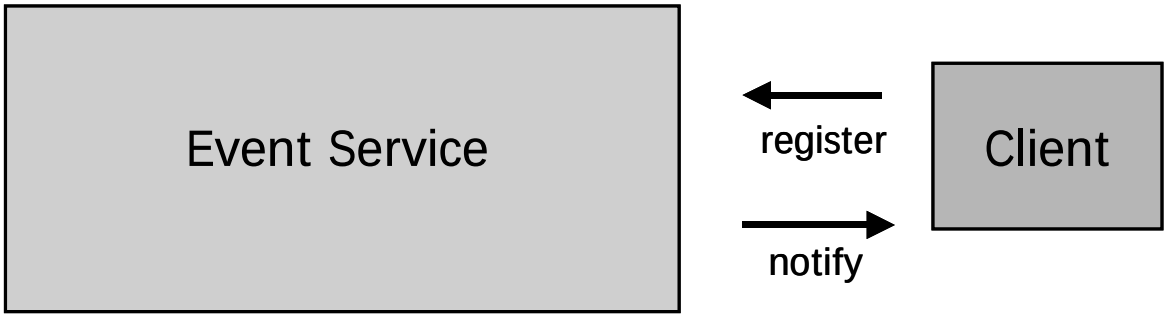


Figure 8: User View of the Event Service

Registering an event includes both the initiation of the observation and the subscription to receive event notifications. Unlike in the classic publish-subscribe paradigm, it is impossible here to have publishers that provide notifications for all possible events. As we will see, a simple change in the model could mean that an infinite number of events have occurred that could be of potential interest to the user. For example, if a user has just moved through a door, he may have entered the property, the house, the ground floor, the left wing of the house, and the corridor, and at the same time he may have met the owner, his wife, the children, and the cat. Therefore, the observation of an event has to be initiated explicitly.

The overall goal for the event service is to provide the user with means to specify and observe all events *of interest* in the given application domain. For us, the Nexus project served as a basis for requirements, so the application domain was context-aware systems with a strong focus on the spatial aspects. However, the results also apply to other application domains with similar characteristics.

In principle, we want to support all those events that can be observed on the available data. The system provider may restrict the events that can actually be observed, because the observation of certain events may be extremely expensive and, as we will see in the next section, the scalability of the system is an important issue.

It is unrealistic to assume that the model of the world will be stored on a single server. As we have seen in the last chapter, the Nexus architecture consists of different kinds of servers, i.e. location and spatial model servers. Each server only has some kind of data for a certain geographic area. To observe an event, data from multiple different servers may be needed, however the user should not have to deal with the resulting problems, which leads to our first requirement:

Requirement 1: Data Distribution Transparency

The distribution of the data should be transparent to the user.

The user has a uniform view of the world model and can specify his events on that model. It is the task of the event service to gather the data and realize the observation. This requirement is taken into account in the general architecture and the specification of events that are presented in Chapter 5 and Chapter 7 respectively.

Requirement 2: Simple Event Specification

The specification of the events should be as simple and easy to understand to the user as possible. It should be clear to the user in its semantics, without oversimplification.

The user should be able to specify an event without needing a very deep knowledge about the underlying system, e.g. it should be straight-forward to understand all parameters that are

needed for the specification of an event. The semantics of the event that is actually observed should be as close as possible to that of the real-world event that the user wants to observe. If differences are not avoidable, they should at least be clear to the user. This requirement is an important topic in Chapter 7.

Requirement 3: Optimal Event Semantics

The event should be observed in such a way that the semantics of the event is optimal with respect to what can be achieved with the available data.

The semantics of events depends on the accuracy of the data that is available, which may be limited due to the limited sensor accuracy and the properties of the distributed system, which are discussed in detail in Chapter 8. Depending on where in the system the event observation takes place, the accuracy of the available data may differ and so does the semantics of the event. In order to optimize the semantics, the accuracy has to be "optimal". This can be achieved through the optimal placement of the observer, which is discussed in Chapter 9.

Requirement 4: Generic Handling of Accuracy Issues

There should be a generic solution for handling the limited accuracy.

Regarding the specification of events, the limited accuracy should be handled in a uniform way. This means that there has to be an explicit and generic solution for all events, it cannot be handled differently for each event. Handling it differently would mean that there are event-specific parameters that influence the observation of the event in a certain way. This would require the user to know the details about how the events is actually observed, which could make the specification unnecessarily difficult and would be in conflict with Requirement 2. How Requirement 4 can be fulfilled is discussed in Chapter 7.

Requirement 5: Minimum Notification Delay

The notification about the occurrence of an event should reach the user with the minimum delay possible

The larger the delay between the occurrence of an event in the real world and the event notification, the less useful it may be. If I want to be informed when I pass a shoe shop and I receive a notification one minute later - in one minute I may have covered 100m – it may not be very useful anymore. The notification delay plays a role when deciding where to place the event observation. This is discussed in Chapter 9.

4.2 System View

After looking at the event service from the user's perspective, we now look at the requirements from the perspective of an event service provider.

From the point of view of the user, the event service is a monolithic entity. As shown in Figure 9, this is not the case for the service provider.

Event sources are typically servers that store some part of the model and in that role we also call them *local model servers*. They become an event source when they can observe events locally. An example for such an event source is a Nexus location server. The event service itself has two functional parts, the observation service and the notification service. The Observation Server observes events that can only be observed with data from multiple event sources. We also call such events *global events*. The notification service delivers event notifications to interested clients. It is also used for delivering event notifications from event

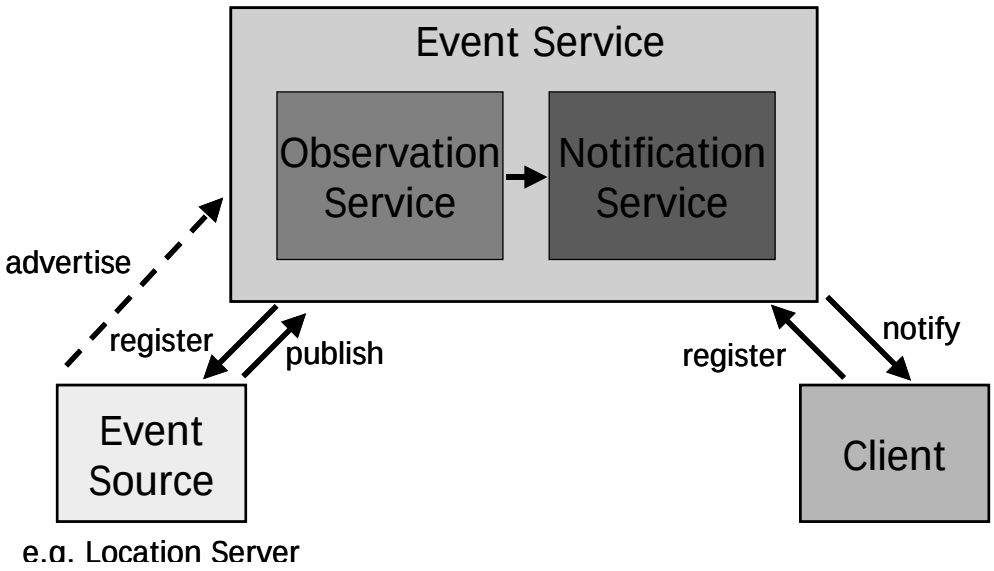


Figure 9: Conceptual Event Service Architecture

sources to the observation service. So, the notification service provides the core functionality of a publish/subscribe service, which does not support the explicit observation of global events.

The components interact as follows: When the client registers an event with the event service, the necessary local sub-events are registered with the event sources and an observer for the event is set up in the observation service. When a local sub-event occurs an event notification is published and sent to the observer using the notification service. When the observer detects the occurrence of an event, an event notification is sent to the client, again using the notification service.

The event service architecture that we have presented here differs from the standard architecture (shown in Figure 2) in that it makes the observation of events an explicit part of the event service. Events are only observed if the event is explicitly registered and the observation of global events through the observation service becomes possible. This can also be seen as a shift from the standard event paradigm, which could be described as an event notification paradigm, to an event observation & notification paradigm.

From the system providers view, there are a number of additional requirements that have to be met, if the system is to be deployed.

Requirement 6: Scalability

The system should be scalable to a large, possibly world-wide scale.

In this case scalability applies to a number of parameters, e.g. the size of the world model, the number of servers, the number of users, the number of events that are observed, the number of event notifications that can be sent per unit of time etc. In the chapters about the respective components, we will discuss in more detail what parameters most strongly influence the scalability of the component and with it the scalability of the event service as a whole. Scalability is an important design criteria. The scalability of the event service as a whole is evaluated in Chapter 13.

Requirement 7: Fault Tolerance

Fault tolerance with respect to failure and unreachability of components

Fault tolerance is an important aspect for any large scale system as it is always possible that components fail or are temporarily not available. This becomes even more important with regard to mobile devices, as they are more likely to fail – due to limited battery power – and as they have to rely on wireless communication, where temporary disconnections are much more likely than in infrastructure-based networks.

Requirement 8: Efficiency

The observation and the notification about the occurrence of events should be as efficient as possible.

Examples for making the observation of events efficient are checking for an event occurrence only when the event could potentially have occurred and reducing the cost of the average check as much as possible. An efficient notification mechanism fits the characteristics of the observed events, especially with regard to the number of clients per event being observed, the distribution of event sources and clients, and the rate of event notifications. The efficiency of the event service components is evaluated in Chapter 13.

Requirement 9: Interoperability

The system should be built on standard technology, to facilitate the interoperability between different providers

Especially if the system is to be deployed on a large scale, this is an important point. Using standard Internet technology like XML, HTTP and SOAP is a step in the right direction.

Requirement 10: Mobility Support

The mobility of the users and the resulting effects on the system have to be taken into account.

The typical usage scenarios we envision are centered around mobile users. Therefore mobility support is a very important issue. Mobility means that we have wireless communication, which has worse communication properties, e.g. less bandwidth, longer delays, and higher loss rates. This may affect the accuracy/availability of the data, e.g. locally determined position information that has to communicated, as well as the notification about the occurrence of events.

4.3 Summary

In this chapter we have defined a number of requirements from both the user's and the system provider's point of view. In summary, the user is mainly interested in an event semantics that corresponds as closely as possible to his expectations of the real-world events that he wants the system to observe. The system provider is mainly interested in a scalable event service that efficiently fulfills the requirements of the user. In the next chapter we describe the system architecture of our event service in detail that we have derived from these requirements.

Chapter 5 System Architecture

In Chapter 4 we have already presented a conceptual system architecture (see Figure 9). Here, we want to look at the event observation in some more detail and derive the internal architecture of our event service.

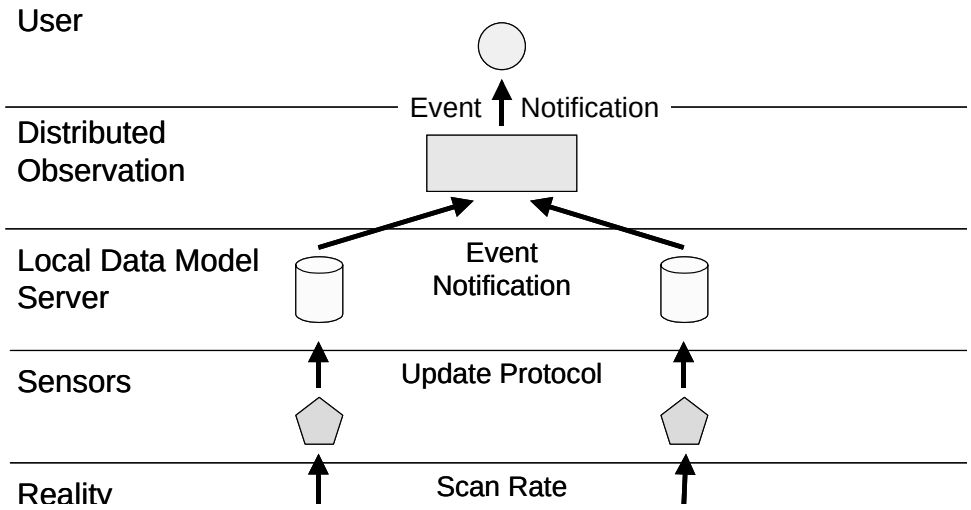


Figure 10: Event Observation

In Figure 10 the process of event observation is shown. An event occurs in the real world. This leads to a change of a physical parameter that is picked up by a sensor. The sensor updates the data of a local model on a local model server. If due to the change of the data, a local event is observed on the local model server, an event notification is sent. If the event has been registered by the observation service that is observing a global event, the event notification is processed there. This may lead to the detection of the global event. Then an event notification is sent to the client.

We assume that the underlying system takes care of the sensors and the update of the local data model servers, so the event service is responsible for the levels above that. In the previous chapter we have already identified two functional parts of the event service (see Figure 9), the observation service and the notification service, where the observation service is responsible for the observation of global events and the notification service is responsible for efficiently delivering event notifications to interested clients.

Requirement 6 states that the system has to be scalable. This excludes a centralized architecture for either the observation service or the notification service. Therefore the observation service consists of a number of observation nodes and the notification service of a number of notification nodes. The nodes communicate with each other in a peer-to-peer fashion.

On the one hand, *Requirement 1* states that the data distribution should be transparent to the client. On the other hand *Requirement 3* demands an optimal event semantics and *Requirement 5* a minimum notification delay. The placement of the observation plays an important role for the notification delay. This is obvious, because if a client in Germany wants to observe a global event based on local events that occur within his vicinity, a significant notification delay is added, if the global event is observed by an observation node in the US.

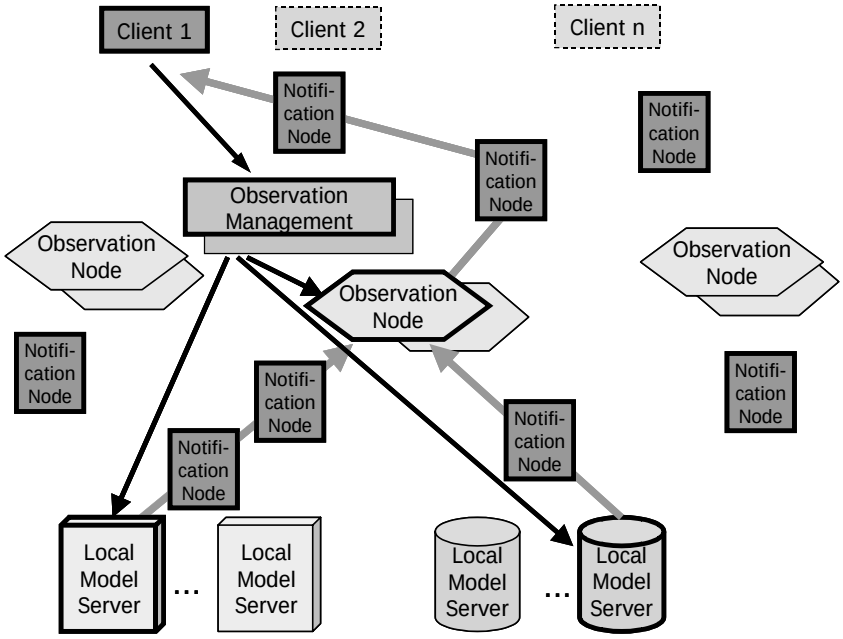


Figure 11: Distributed Architecture

As we will see later, the optimal placement of event observation is also important for the event semantics.

Therefore, an additional component is needed that takes care of the placement of the event observation and the registration of all local events involved. We call this component *observation management*.

Figure 11 shows an example of how the different components interact. In the registration phase indicated by the dark arrows, the client first registers an event with the observation management. The observation management finds the local model servers that have the data needed for the observation, then determines the observation node that is best suited for observing the event and finally registers the global events and the local events needed for the observation.

In the observation phase, whenever a local event occurs, the local model server (in its role as an event source) hands over an event notification to the notification service using the closest notification node. The notification node delivers the event notification to the observation node through its closest notification node. Finally, if a global event has occurred, the observation node passes an event notification to the notification service that delivers it to the client.

Chapter 6 Event Model

In this chapter we develop an event model, i.e. we structure the area, define different abstraction levels and find a suitable event classification. In the first section, we discuss the different conceptual approaches of observing global events on a distributed world model versus observing composite events by combining event notifications describing the occurrence of simpler events. We propose a general event classification in the second section, which we apply to spatial events in the third section. The fourth section discusses the use of spatial events in a number of real world scenarios.

6.1 Observation of Global Events vs. Composite Events

We start by defining the terms global events and composite events and discuss the conceptual difference.

Definition 4: Global Event

A global event is an event that is observed through a distributed model.

Observing an event through a distributed model means that the model data that is necessary for the observation of the event is distributed over a number of local models. Each local model can be seen as a fragment of a distributed global model. To observe the global event a (local) view of that part of the global model has to be realized that is needed for the observation. We also call this view *observer model*. The observer model can be updated using update events. These update events implement certain update protocols that we will discuss in Section 8.4. The observation is based on the state of the model at a certain time.

Definition 5: Composite Event

A composite event is an event that is composed of simpler events.

A composite event combines a number of (simpler) events. To observe the event, the observer has to subscribe for the event notifications that result from the occurrence of the events to be composed. The event is observed directly based on event notifications, i.e. on *messages*,

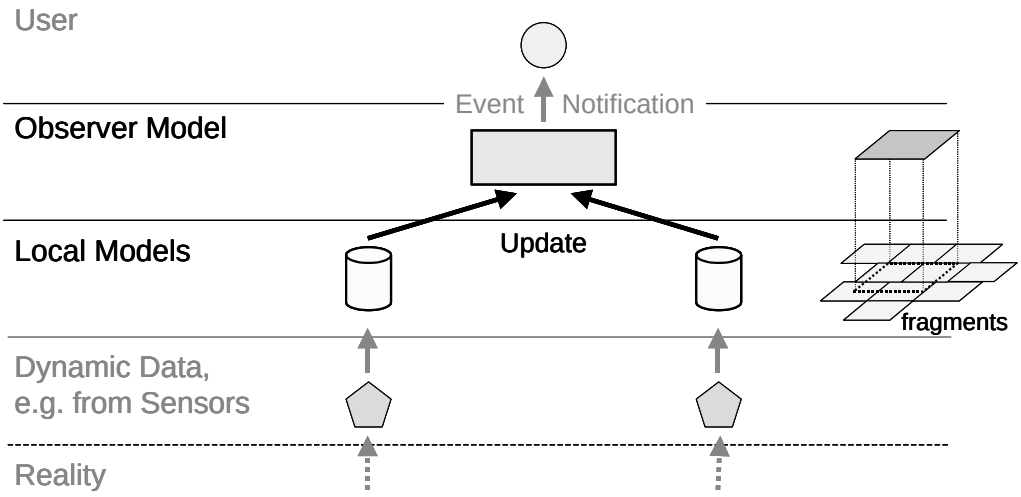


Figure 12: Local and Observer Models

and not through *values* describing the relevant state of the distributed model.

Since event notifications are used to update the observer model, global events can also be seen as a specialization of composite events. The difference lies mainly in the conceptual view. A composite event is based on other events, a global event on the change of values. In practical use, composite events often provide simple constructors to combine any kinds of event, e.g. AND(A,B), OR(C,D), SEQUENCE(E,F), NOT(G) etc. Global events are usually more specialized and based on values, e.g. the simple spatial event onEnter-Area(PERSON,AREA).

The general architecture of our event system is suitable for both approaches, even the specification of events is rather similar. The main difference can be found in the implementation of the observation modules. In the theoretical part of this report, especially Chapter 8 and Chapter 9, we focus only on the observation of global events.

Figure 12 is a specialization of Figure 10 that shows the models. Changes to the local models can lead to updates of the observer model. The updates are realized in form of event notifications. The local models are seen as fragments of a global distributed model, and the observer model is a view of that part of the global model that is needed for the observation of a particular global event.

6.2 General Event Classification

In this section we develop a general event classification. We look at different parameters that can be used to classify events, especially regarding the observation of events.

6.2.1 Event Observation Hierarchy

We start with the event observation hierarchy. There are multiple abstraction levels on which we have events. Figure 13 shows the event observation chain for our application do-

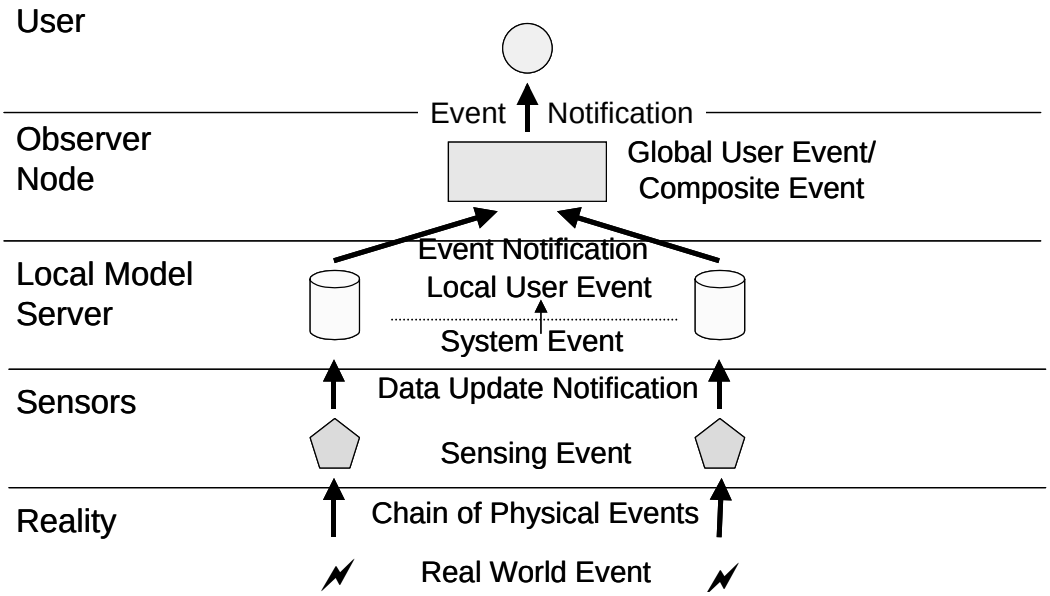


Figure 13: Event Observation Chain

main. We are confident that in other domains the situation will be similar. Table 1 explains the event observation chain and gives examples.

Table 1: Explanation of Event Observation Chain and Examples

Event Observation Chain	Example
On the lowest level we have the real world event.	A user with an IR tag enters a room.
Through a chain of physical events the information is propagated to the sensor that observes a sensing event.	In the case of the IR tag an IR signal would be sent that is picked up by the sensor.
Through some update protocol the data reaches the local model server, where the event is observed as a system event.	The IR signal can be interpreted as a position update that is passed on to a location server.
Within the server, the system event may lead to the observation of a number of user events.	A position update may lead to the observation of the event that a user has entered the house (onEnterArea), that the user has entered the room and that the user has moved by more than 10 m since the last update notification was sent (distPosUpdate).
The event notifications describing the occurrence of these local user events are delivered to observer nodes that may observe more complex user events.	A composite user event, based on onEnterArea and onLeaveArea events, that might be observed on an observer node, is that more than a certain number of people are in the house. A global user event, based on the distPosUpdate event, may be that two people, whose position information is stored on two different local model servers, are within a specified meeting distance (onMeeting).
Finally, on the detection of an event occurrence in the observer node, an event notification is sent to all interested users.	Event notifications for the number of people in the house and the meeting event are sent to the users interested in the respective events.

A more complex multi-level observation is also possible. In this case the observation service would observe global events that depend on other global and local events. As the observation in these cases follows the same principle as in the presented simple case, we concentrate on the simple case here.

An important abstraction is to distinguish between *system events* and *user events*. The problem with providing all system events to the event system as a whole is that this does not scale. The system event rate may be too high, so the distributed system would be overwhelmed with the event notifications. With the introduction of user events that can be observed through the local model servers, it becomes possible to introduce additional observation parameters and policies, like limiting the number of events that can be observed within a

time interval. For example, the maximal accuracy of position data provided by a location server can be limited in this way or the costs for an event could be based on the load that is being generated.

The number of possible user events that can be observed based on a single system event is infinite, e.g. as we have already seen, a position update can cause the detection of any number of onEnterArea or onMeeting events. However, users in practice will only be interested in a very small number of these.

There are an infinite number of different user events, e.g. there is an infinite number of possible constellations of mobile objects, based on which a user event could be defined. However, the number of those that are of actual interest in a specific application domain will be very limited.

6.2.2 Event Triggers

We have defined an event as a change in the state of the model. As we have seen in the previous subsection, we have different models, one on every abstraction level. The change of a model on a certain level can be triggered either by an explicit change in data, i.e. through an update from a lower level, or through a time-induced change.

So we can distinguish between data-based and timer-based events. Data-based events can sometimes be classified into value update events that change the attribute of an object, e.g. its location, and management events that register or deregister objects. Timer-based events are triggered through a timer.

6.2.3 Number of Dynamic Parameters

Events can further be classified according to the number of dynamic event parameters. Each change in a dynamic parameter can potentially lead to the occurrence of an event.

For example, an onEnterArea event has one dynamic parameter, describing the mobile object, whose position can lead to the occurrence of the event. An onMeeting event has at least two dynamic parameters. Both objects can move and potentially the movement of each object could lead to the occurrence of the event.

6.2.4 Specific and Variable Parameters

An event parameter can further be classified according to how specific it is, i.e. if it specifies exactly one variable (or object) or if it specifies a group of objects. In the first case we call the parameter *specific*, in the second case *variable*. For example an onEnterArea event could be specified for John Doe, a specific person, or for an object selector, e.g. a professor.

As we will see in the next subsection, this classification is important for the efficient observation of an event.

6.2.5 Efficient Observation

The efficient observation of events is especially important in cases where one event can cause multiple events on a higher abstraction level. This is especially the case for system events that can cause multiple user events. To increase efficiency, it should be possible to check only for the occurrence of those events that can possibly have occurred. The better this preselection works, the more efficient the service will be.

For this purpose, the event observation can be *attached* to a given parameter. If there is a change, there has to be an efficient index for each dynamic parameter to find the events at-

tached to the given parameter. (The dynamic parameter and the parameter to which the event is attached do not have to be the same!) This works well, if the respective parameter to which the events are attached is specific, but not, if the parameter is variable.

For example, an onMeeting event can be efficiently observed, if both parameters are specific, e.g. if both mobile objects are explicitly specified, e.g. John Doe and Anne Smith. If the onMeeting event is attached to both John Doe and Anne Smith respectively, for every position update for either John Doe or Anne Smith, the event to be checked can be found efficiently. For position updates of other mobile objects, the event does not have to be checked.

For an onMeeting event with one specific parameter, an efficient observation can also be achieved – the implementation details are discussed in Section 10.1. However, with two variable parameters, this may be impossible, e.g. an onMeeting event that any two professors or any two students meet would be very expensive to observe.

The specific parameter does not have to be the same as the dynamic parameter. For an onEnterArea event, the area parameter can be specific and the object parameter variable. With a two-dimensional index structure, e.g. a quad tree, the events that have to be checked for a position update with a given position can be found efficiently.

6.3 Classification of Spatial Events

In this subsection we classify the basic spatial events supported by the location service according to the general event classification presented in the last section.

Definition 6: Spatial Events

Spatial events are events that occur when objects reach a certain spatial constellation.

This definition includes events in which mobile objects reach such a constellation with respect to their environment, e.g. an onEnterArea event, and events in which mobile objects reach a constellation relative to each other, e.g. an onMeeting event.

6.3.1 Basic Spatial Events

In [Bauer 2000] we have identified the following spatial relations:

- Distance (between objects)
- Direction (one object is in a certain direction from another object)
- Orientation (of an object)
- Inclusion (one object is included in another)
- Intersection (two objects intersect)
- Tangent (two objects touch each other)
- Disjoint (two objects are disjoint)

If the changes over time are taken into account, there are additional relationships:

- Change of orientation
- Direction of movement
- Speed
- Change of Speed

…

Based on these spatial relations, spatial events can be defined. Not all spatial events that exist in principle can currently be observed in Nexus. The location service currently does not take the extent of an object into account, the accuracy of the position data is limited and only the current position of an object is available.

Currently the following basic spatial events are supported:

- onEnterArea: an object enters an area
- onLeaveArea: an object leaves an area
- onCrossingLine: an object crosses a line
- onMeeting: two objects meet
- distPosUpdate: an new position of an object is provided, as it has moved further than a specified distance
- contPosUpdate: each (specified) time interval, a new position of an object is provided
- contAreaUpdate: each (specified) time interval, a new position of all objects that are currently within a given area is provided

The following “management” events are also supported

- onRegisterObject: a certain object has registered with the location service
- onDeregisterObject: a certain object was deregistered from the location service
- onRegisterArea: an object has registered in a given area
- onDeregisterArea: an object in a given area was deregistered

6.3.2 Event Triggers

The spatial events can be classified according to the data-based system events that may cause them:

- position update
    - onEnterArea
    - onLeaveArea
    - onCrossingLine
    - onMeeting
    - distPosUpdate
- registration, deregistration (management events)
    - onRegisterObject
    - onDeregisterObject
    - onRegisterArea
    - onDeregisterArea
    - onMeeting

6.3.3 Number of Dynamic Parameters

All basic spatial events, except for the onMeeting event have one dynamic parameter. The onMeeting event has two dynamic parameters.

6.3.4 Specific and Variable Parameters

In the following we show which parameters for the basic events supported by the location service have to be specific and which can be variable. The decision if a parameter can be variable or not is directly related the fact that for an efficient event observation, the observation has to be attached to a specific parameter.

- onEnterArea: The onEnterArea event needs the *area* as a specific parameter, the *mobile object* can be variable.

  The reason that only this combination is supported is that the location service has the information about mobile objects, so an object selector can be supported, whereas there is no basis for an area selector. On the level of the observation node, we can support such an onEnterArea event, e.g. with *[shoe shop]* as an area selector. However, in this case the mobile object has to be specific, as otherwise an efficient observation is not possible.

- onLeaveArea: The onLeaveArea event needs the *area* as a specific parameter, the *mobile object* can be variable.

- onCrossingLine: The onCrossingLine event needs the *line* as a specific parameter, the *mobile object* can be variable.

- onMeeting: The onMeeting event needs one *mobile object* as a specific parameter, the other can be variable. Again, this is due to ensure the efficiency of the observation.

- distPosUpdate: The distPosUpdate event needs a *mobile object* as a specific parameter.

- contPosUpdate: The contPosUpdate event needs a *mobile object* as a specific parameter.

- contAreaUpdate: The contAreaUpdate event needs an *area* as a specific parameter.

- onRegisterObject: The onRegisterObject event needs a *mobile object* as a specific parameter.

- onDeregisterObject: The onDeregisterObject event needs a *mobile object* as a specific parameter.

- onRegisterArea: The onRegisterArea event needs an *area* as a specific parameter.

- onDeregisterArea: The onDeregisterArea event needs an *area* as a specific parameter.

6.3.5 Efficient Observation

For the efficient observation of an event, the observation has to be attached to a specific parameter. In the case of the spatial events, this has to be either a mobile object or an area, i.e. a location. In the case of a timer-based event, it is the (next) occurrence time of the event.

This means the events to be checked after a position update can be found through the object id of the object for which the position is the updated, if the specific parameter is the mobile object, or the position, if the specific parameter is the area. A possible index for the object id is a hash table, for a two-dimensional area the index might be a quad tree and for a three-dimensional space an oct tree might be used as an index. For timer-based events a heap can be used as an index structure. The time when each event is scheduled to occur next are ordered in the heap with the smallest element, i.e. the first event that will occur, being the root. We will discuss our actual implementation in Section 10.1.

The object id, the location and the time is exactly the context that we have identified as the *primary context information* in [Rothermel et al. 2003a]. The primary context can be used as an index to efficiently access all other context information.

So, the onEnterArea, onLeaveArea and the onCrossingLine events that have to be checked after a position update can be found through the position and the location index. The distPosUpdate can be found through the object id and the object index. The onMeeting event has a special implementation and the events to be checked can be found through both the position and the location index.

6.4 Utilization of Spatial Events in Nexus Scenarios

So far, we have classified events according to the characteristics that are important for the observation of events. In this section we will classify the spatial events according to how they will be used, which is important for the design of the notification service.

The problem is that it is difficult to derive requirements for a totally new system that cannot really be compared to any existing system. How will the users actually utilize the features that are being offered? Therefore, we look at a couple of scenarios that we think might be typical for the use of the event service in Nexus: an office scenario, a shopping mall scenario and a city scenario.

6.4.1 Office Scenario

For this scenario we imagine that a company makes Nexus services available to their employees within one office building. Assuming that there are about 2 people per office, 40 offices per floor and 5 floors per building, there may be about 400 people altogether. Taking into account that there are always people who are on holiday, sick or traveling on business, there may be about 250 people in the building at a given time during the day, who may use the Nexus system.

Further, we assume that the company has installed an indoor location system on each floor that can determine the location of people to within one meter from their actual location. For each indoor location system, there is a separate location server that stores the location of all the people within the system, i.e. of 50 people on average in this scenario. The five location servers act as sources for basic events.

A typical application that is viable in such a scenario is having maps of the floors, showing the location of all the people on the respective floor. This can be realized using contAreaUpdates.

In addition to this continuous tracking, employees will also want to be informed about more specialized events that are only of interest to themselves or to a small group of people, e.g. if somebody enters their office. Another important characteristics is, if people are always interested in the event, e.g. as in the last example, or if the event is observed to get a one-time reminder, e.g. inform me when I pass the cafeteria, so that I can buy sweets for my colleagues. As an estimate, we assume that every employee is interested in five such specialized events at any given time. This means that each location service has to monitor about 500 events, taking into account that some of the events are not restricted to a single floor.

Except for the continuous tracking, the other events will occur relatively rarely. Therefore, the load on the notification service is relatively predictable and the continuous update rate can be determined accordingly.

6.4.2 Shopping Mall Scenario

Compared to the office scenario, there is a much higher turnover of people in a shopping mall and the potential users, which, realistically, will only be a subset of the customers, are not known in advance. Also, the variance is much greater, e.g. a Saturday afternoon before Christmas compared to an afternoon in the middle of the week in summer. The customers themselves are not really interested in what the other customers are doing, however the management of the shopping mall might be very much interested in customer behavior. An example in which events might be useful is that additional salespersons could be sent to a certain area to handle the higher demand, which can be determined by the number of people who have entered (and left) it.

Another usage area that would be interesting for the shopping mall is to use events to pass on advertisements, e.g. if a customer enters a shop or area, this event could be observed and advertisements relevant for this particular shop could be sent to the customer. We assume that there are about 20 shops/areas of interest for which the shopping mall provides events.

The events that are interesting to customers are events that remind them that they wanted to buy something when they pass the shop that sells the given article. We assume that each customer using the system registers about 2 such events. Unlike in the office scenario, the number of event occurrences may vary considerably. This has to be taken into account when designing the notification service, but also when planning the infrastructure (e.g. wireless network access), so that the notification service can handle the amount of event notifications.

6.4.3 City Scenario

Here, a city scenario refers to the center of a city the size of Stuttgart. This scenario is much more complex than the two previous scenarios, because it is much harder to predict what type, number and complexity of events people are interested in. In addition, there will be different user groups. Tourists have different interests than locals and people doing business have different interests than people in their leisure time. Ideally, it should also be possible to easily integrate the previous scenarios into the city scenario.

Whereas in the simple scenario the notification service can be mostly operated by a single provider, such an integration requires that different providers cooperate in the delivery of event notifications. As event notifications may generate a significant network load, policies dealing with the cooperation of different providers, which also have to be visible to the clients, will have to be introduced in the future. However, we ignore this aspect for now.

We assume that one of the most popular applications in a city scenario is a reminder system, i.e. a system that reminds the user when he is at a certain location that he wanted to do something there, or even better, when he is at a location that fulfills certain requirements. An example reminder of the first type is “Inform me, when I’m within 1 km of THE OPERA in Stuttgart, because I need to pick up tickets there”. An example reminder of the second type is “Inform me, when I’m near A PHOTO SHOP, because I need to get a new lens for my camera”.

Going one step further, a tourist information system could be based on events. Whenever a tourist gets close to a sight, information about this particular sight is automatically provided. In order to do this, the user just has to register for certain types of events at the beginning of the tour. Using different types of events, the information provided can also be specially customized for the tourist, providing more information about things he is interested in or having special information for children.

Under the given circumstances, it is very difficult to derive half-way reasonable numbers, therefore we will make some rough assumptions: The city is covered by 50 location servers

(not counting the ones in office buildings and department stores), there are about 10000 users of the system and each of them is, on average, interested in about twenty events. Ten of these events are of personal interest only, five events are of interest to about ten different people and the remaining five are of interest to about 200 people.

6.4.4 Parameters of Interest

The following parameters may be of interest regarding the utilization of spatial events:

- Overall number of event sources
- Number of active event sources per time interval
- Overall number of clients
- Overall number of different events
- Number of event sources per event
- Number of clients per event
- Average size of an event notification
- Number of event observations per time interval
- Number of event notifications per client per time interval
- Number of subscription changes per time interval
- Number of subscription changes per client per time interval

6.4.5 Parameter Values for the Scenarios

In Table 2 we give an estimate for the parameter values in each of the scenarios.

Table 2: Parameter Values for the Scenarios

Parameter	**Office Scenario**	**Shopping Mall Scenario**	**City Scenario**
Overall number of event sources	5	5	50
Number of active event sources per time interval	0.5/s	1/s	5/s
Overall number of clients	250	550	10000
Overall number of different events	5: contAreaUpdates 2000: spec. events	100 (shop. mall) 1000: spec. events	100000
Number of event sources per event	5 contAreaUpdate 1-2: spec. event	1-2 (shop. Mall)	1-2 (50%), 5-10 (40%), 50 (10%)
Number of clients per event	250 per contArea-Update, 1-2 per spec. event	1-2 spec. events 50 shop. mall events	1 (50%) 10 (30%) 500 (20%)

Average size of an event notification	2 KBytes	2 KBytes	1-500 KBytes
Number of event observations per time interval	0.5/s	1/s	5/s
Number of event notifications per client per time interval	0.1 event notification/s (contAreaUpdate)	4 per hour	10 per hour
Number of subscription changes per time interval	40 per hour	500 per hour	20000 per hour
Number of subscription changes per client per time interval	4 per day	1 per hour	2 per hour

6.4.6 Comparison with Other Notification/Publish - Subscribe Services

Overall, the numbers presented in the previous section can be summarized providing interpolated results for using the Nexus notification service in a metropolitan area. These results are compared to the requirements for other notification and publish / subscribe services (see Table 3) as presented in [Jacobsen & Llirbat]. There, the following scenarios for publish / subscribe services are presented: Alerting services, e.g. for digital libraries, inform users about new publications in areas that are of interest to the user. Selective information dissemination services are used for disseminating news, including such time-sensitive items as current stock quotes. Network and distributed system management services are used to manage networks on an enterprise scale and may have tough real-time constraint. Enterprise application integration is used to integrate software applications across heterogeneous platforms on an enterprise scale.

Table 3: Comparison of Requirements for Different Notification Services

	Clients	**Event Sources**	**Event obs. Rate**	**Notificat. size**	**Subscr. Updates**	**Notificat. Rate**
Nexus Notificat. Service	**200K +**	**200**	**20/s**	**1-500 KB**	**2/user/hour**	**10/user/hour**
Alerting Services	100K +	10-20	10 / h	.5k - 2 MB	5/user/month	5/user/month
Selective Information Dissemination	millions	1000+	100+/s	variable	3/user/week	100+/sec.
Network Management	10+	10000+	10000+/s	.5 KB	rare	100+/s

Enterprise Application Integration	10	10	Request rate	request size	rare	request rate

Maybe, it should not come as a surprise that the estimates for the Nexus notification service come closest to the selective information dissemination, because the number and kinds of users and at least the type of some of the information are relatively similar.

The main difference is that in Nexus users explicitly register for certain types of events and those events are only observed because of that. This also means that the service can directly find out, who is interested in which event. With selective information dissemination, users usually specify the information they are interested in using filters and information is routed according to those filters. Usually many users are interested in the same information, whereas in Nexus, for the large majority of events, only a few users, if not only a single user is interested. Also, most of the events will occur rarely, potentially only once.

These results have had an influence on the design of the Nexus notification service that we will discuss in Chapter 12.

Chapter 7 Event Specification

In this chapter we look at the expectations of the user towards the observation of events, and especially how he can specify the events he wants the event service to observe for him. The specification here is on a relatively abstract level. Additional parameters that are relevant mostly for the concrete implementation will be discussed in Section 10.3.

The system behavior the user would like to have for the observation of real world events is that

- an event notification is sent if and only if the corresponding event in the real world has occurred and that
- an event notification is available immediately

Unfortunately such a behavior cannot be realized, especially not in a distributed system. In the best case, there is a short delay introduced by the time needed to communicate and process the information. As we will see, the limited accuracy of the available data will make the first point unrealistic. Therefore, we have to consider it in the specification of events.

As the user has to specify the event, we have to take the user requirements that apply to the specification into account. *Requirement 1: Data Distribution Transparency* states that the distribution of the data should be transparent to the user. For the specification this means that for the user there is no difference between specifying a local or a global event.

Requirement 2: Simple Event Specification directly addresses event specification, stating that it should be as simple as possible without distorting the semantics of the event. Finding a reasonable balance between *Requirement 3: Optimal Event Semantics* and *Requirement 5: Minimum Notification Delay* should be the default behavior of the event service. Additional parameters may influence the implementation of the event service with respect to this balance, but as stated before, we want to discuss event specification on a more abstract level here.

Requirement 4: Generic Handling of Accuracy Issues is an important requirement for which we provide a solution in Section 7.2 , but for the next section, we assume that we have exact data.

7.1 Event Specification Based on Exact Data

It should be relatively natural to the user to specify an event in form of a predicate. For the user a predicate is a parameterized statement about the world that is true after an event has occurred. So the occurrence of an event is equivalent to the predicate becoming true, i.e. the predicate evaluated to false in the previous state and to true in the current state.

Predicates are defined over variables, which in our case represent the state of the model.

The following definition formalizes the intuitive understanding of *predicate*:

Definition 7: Predicate

If P is a k-ary predicate symbol defined for variables $v_1,...,v_k$ of types $z_1,...,z_k$ respectively, $P(v_1,...,v_k)$ is a predicate.

For example, if variable x stores the current temperature at location X and y is the temperature at location Y, and there is a predicate $P_1(v_1,v_2):=(v_1>v_2)$, then $P_1(x,y)$ describes the event when the temperature at location X becomes greater than the temperature at location Y.

A typical predicate that could be used for specifying the spatial event that a user enters an area is PersonInArea($Person_1$, $Area_1$). The onEnterArea event has occurred when the PersonI-

nArea predicate becomes true, which is the case when the person specified by $Person_1$ is outside $Area_1$ in the previous state of the model and is inside $Area_1$ in the current state. For easier understanding of the described event, we still use onEnterArea as the name of the predicate.

We do not restrict the complexity of the predicates here, which means that all events observable on the model can be described in form of predicates. If we had exact data, the user would just have to specify the predicate and based on the exact data the predicate would be evaluated.

To specify events, the end user would be able to choose from a set of predicate templates.

Definition 8: Predicate Templates

Predicate templates are predicates that have free variables as parameters.

The user sets the parameters of a predicate template and gets a predicate that can be registered with the event service. For example onEnterArea(<Person>,<Area>) is a predicate template. Setting Person to 'Fritz' and Area to the 'Trafalgar Square' yields the predicate onEnterArea(Fritz, Trafalgar Square).

For each predicate template an implemented observer module has to exist. New observer modules can be added at runtime by putting them on a web server. The observer node can then download them from there. Normal users will usually not implement their own observer modules. They have to utilize what is already provided by the event service. Application programmers writing a new application may add new observer modules. As new observation modules can be computationally expensive and may pose a security threat, event service providers may implement policies for adding new modules and may charge for the service according to computational costs. However, these policies are beyond the scope of this work.

7.2 Event Specification based on Data with Limited Accuracy

In the previous section we have assumed that we have exact data. To decide if an event has occurred we simply had to evaluate a predicate. In reality, exact data will be impossible to achieve, because the accuracy of real sensors is limited and the fact that the observer model has to be updated from different local models reduces the accuracy of the available data in both the value and the time dimension, i.e. a value can only be given as lying within some accuracy interval and the time in which a value has changed can only be given as an interval.

So in the realistic case, we have to deal with the inaccuracy of the data and that means dealing with a potential error. As the error lies in the available model data, the user can only influence how this error influences the observation of the event.

This could be explicitly encoded in the specification of the event. For example for the event that two people meet, it could be defined by how much the location areas (area in which the real location of the user can be found) extended by the maximal meeting distance have to overlap and how the size of the location area influences the required overlap. However, this requires that the user knows about the details of how an event is observed, not just the semantics of the event itself. This information is not required in the exact case and is in contradiction to *Requirement 4: Generic Handling of Accuracy Issues* that requires that there should be a generic solution for handling the accuracy issues.

Since we cannot achieve that an event notification is sent if and only if the corresponding event in the real world has occurred, we could "weaken" the approach by replacing the equivalence relation (if and only if) with the implication:

- If an event in the real world has occurred, an event notification is sent.

- If an event notification is sent, an event in the real world has occurred.

In the first case, there is a notification for every real world event, but there may be also be notifications when no event has occurred. This means there could be false positives.

Definition 9: False Positive

We have a false positive, if an event notification was sent, even though no real-world event has occurred.

In the second case, for every notification an event has occurred in the real world, but there may also be events for which no event notification was sent. This means there could be false negatives.

Definition 10: False Negative

We have a false negative, if a real-world event has occurred, but no event notification was sent as a result.

Both approaches are generic with respect to the handling of accuracy issues, but are somewhat extreme and may not be what the user would like to have. We therefore propose an approach that allows the user to specify a threshold probability that decides if an event is considered to have occurred. If, based on the available data, the probability that an event has occurred is higher than the specified threshold probability, an event notification is sent. Setting the threshold probability to 100% yields the approach where we have no false positives, but possibly a large number of false negatives. Setting the threshold probability close to 0% (an occurrence probability of 0% does not make sense since this would be true for all values) yields the approach where we get no false negatives, but possibly a large number of false positives. So the threshold probability determines the ratio between false negatives and false positives. Given that we can calculate the probability that an event has occurred, we have a generic approach as required by Requirement 4. In the following chapter we show how the probability with which an event has occurred can be calculated.

The user can specify an event as follows:

Definition 11: Specification of an Event

An event is specified as a pair (P, TP) where P is a predicate and TP a threshold probability. For an exact value, the predicate P becomes true if and only if the event has occurred. The threshold probability TP specifies the probability with which the occurrence of the event must at least be detected so that the event is considered to have occurred.

7.2.1 Setting the Threshold Probability

The question that arises from this definition is how the user can determine a suitable threshold probability. The choice of the threshold probability depends on the usage scenario and the model quality, i.e. the accuracy in value and time dimension. The accuracy is limited by the accuracy of the sensor, which can be taken from the fact sheet of the sensor. However, it is the update protocol (see Section 8.4) that ultimately determines the accuracy of the observer model. Given a guaranteed accuracy we can estimate the number of false negatives and false positives for a chosen threshold probability. For this estimation we have to make certain assumptions, e.g. that all possible values have the same probability. If we have more information about how the values typically change, reflecting the changes in the real world, we can improve the estimation by weighting the values accordingly.

The threshold probability directly influences the ratio of false negatives to false positives. Given a concrete scenario, the user has to decide what the relative costs of false negatives and false positives are. If, for example, the application is important for the safety of the user – like a navigation system for visually impaired people, which is investigated as part of the Nexus project – it may be better to have more warnings (e.g. based on a false positive) than a missed warning (e.g. based on a false negative), so the threshold should probably be set to a lower value. For an application that is mostly for the convenience of the user, a higher threshold probability might be selected.

There can be tools that help the user to find the appropriate threshold probability by calculating the number of false negatives and false positives for a predefined accuracy and the expected range of values. We plan to investigate the choice of threshold probability for concrete application scenarios in our future work.

Chapter 8 Concepts and Realization of Event Observation

In the previous chapter, we have proposed that an event can be specified as a predicate together with a threshold probability that decides, if the event is considered to have occurred for a particular state change or not. In this chapter we show how the probability that an event has occurred can be calculated on the available data. In the first section we look at the system parameters that are relevant for the observation. In the second section we propose the concept of an *event domain* to model the identified system parameters. In the third section, the general properties of the observer model on which the events are actually observed are discussed. Update protocols that define the properties are presented in the fourth section, before we finally show in the fifth section how the occurrence probability can be calculated on the observer model. The sixth section about realization issues concludes the chapter.

8.1 System Parameters Relevant for the Observation

Figure 14 shows a distributed system consisting of a number of nodes. The local models and the observers with the observer models are located on different nodes. As the event observation takes place in a distributed system, we have to deal with the typical properties of such a system.

The observation of events is primarily affected by two properties, the delay update messages may experience and the clock skew.

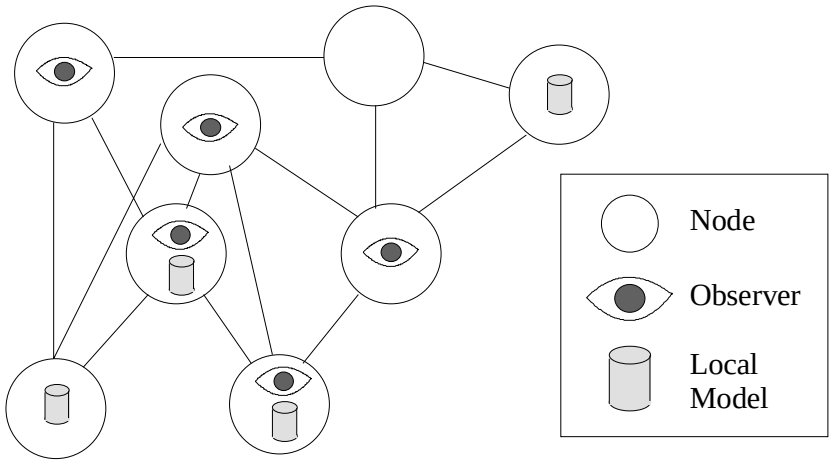


Figure 14: Distributed System

Message Delay

The message delay is important for the evaluation of an event predicate as a predicate can only be evaluated when there cannot be any update message delayed in the network that might change the outcome of the evaluation. The communication delay that is introduced by the network itself is only a lower limit of the overall message delay. The message delay is the sum of all the processing and network delays the message may have incurred on the path from its original source to the observer node. Figure 15 shows the delays that are incurred from the occurrence of the real-world event to the delivery of the event notification to the user.

The communication delay is influenced by the underlying network, e.g. there is a big difference, if we have a backbone network, e.g. a 1 GBit/s LAN, or a wireless connection, in the extreme case a GSM or GPRS network. The communication protocol used also has an influence, e.g. a low-level protocol like UDP vs. SOAP over HTTP.

Clock Skew and Time in Distributed Systems

The clock skew is important as it determines the time interval in which the reported model state has become valid, i.e. the temporal inaccuracy of the information about the state change.

As we are interested in the real-time relation between state changes and causality relationships may not be available, we have to completely rely on real time. As the user may define events with respect to real time, the clocks do not only have to be synchronized with respect to each other, but also with respect to a standard time scale. The most widely used time scale is Coordinated Universal Time (UTC), so we expect that clocks are synchronized with respect to UTC.

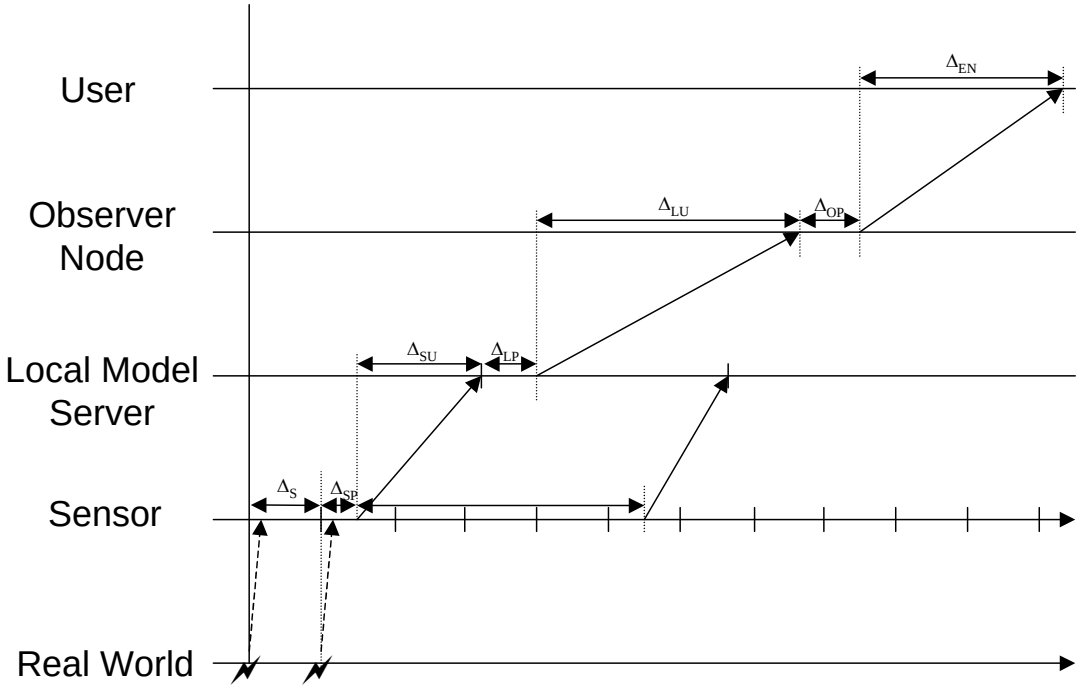


Figure 15: Delay Incurred from the Real-World Event to the User

The clock synchronization mechanism that is used most widely in the Internet is the Network Time Protocol (NTP). The NTP protocol defines a synchronization hierarchy. The root level servers (level 0) are directly synchronized with a primary reference source, e.g. an atomic clock. Level 1 servers synchronize their clocks with level 0 servers, level 2 servers with level 1 servers and so on. The *root dispersion* defines a conservative estimate of a clock's offset to a root source, and the *root distance* difference defines the (estimated) maximum offset between a clock and the root source. The root distance is an estimate, because no real bound can be given NTP provides the method ntp_gettime() that returns the time, the root dispersion and the root distance.

The last available study on clock synchronization in the Internet was conducted by Minar in 1999 [Minar 1999]. It is based on 175,000 systems and looks at the root dispersion. Eliminating the worst 3%of the systems with a root dispersion of up to one year, the median root dispersion is 39 ms, the mean dispersion 88 ms and the standard deviation 175 ms.

8.2 Event Domains

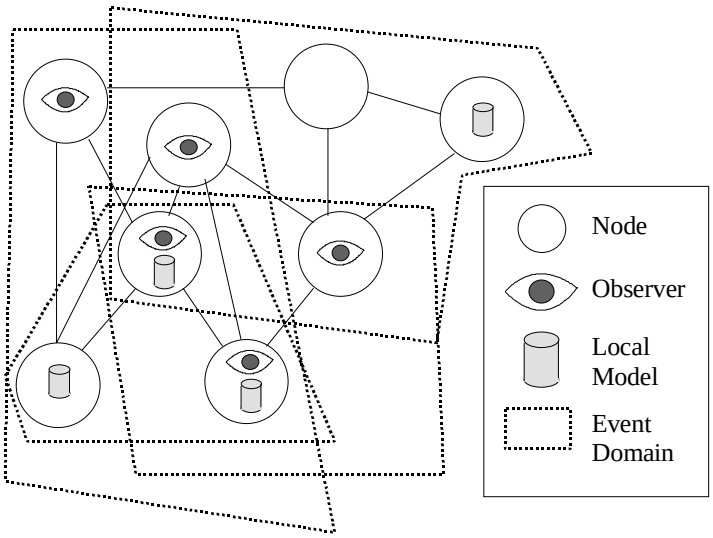


Figure 16: Event Domain

For large networks, e.g. the Internet as one extreme case, there are no real bounds on either delay or clock skew. However, for a given set of nodes, especially within local area networks, reasonable values for delay and clock skew can be given. Unless these networks can guarantee certain real-time bounds, which cannot be assumed in the general case, the given values are no strict upper bounds for delay and clock skew, but "statistical guarantees". This is sufficient for our purposes, because due to the restricted accuracy of the sensor data, we cannot give more than „statistical guarantees" for the event observation anyway.

To specify these values, we introduce the concept of an event domain. An event domain consists of a set of nodes for which statistical guarantees for delay and clock skew are given. Figure 16 shows a number of event domains in an example network.

For the observation of an event, we need to find the event domain with the best values for delay and clock skew that includes all the nodes involved in the event observation, i.e. the local model nodes, the observer nodes and all nodes that are needed for passing on notifications.

In the future, the concept of event domains can be extended to include dynamic information, like the current load of the network, which may be especially useful for optimistic observation strategies, as well as incorporate means for resource reservation in order to guarantee a distinct upper bound for the delay. The necessary dynamic information could be supplied by a system management component.

8.3 Model Properties Relevant for the Observation

After introducing the underlying system model and the concept of event domains, we present the observer model in its general form. The update protocols, which we present in the next section, and the system properties then set the parameters of the general observer model, yielding the concrete observer model on which the event is observed. In principle the same parameters also apply to the local models, but on a smaller scale.

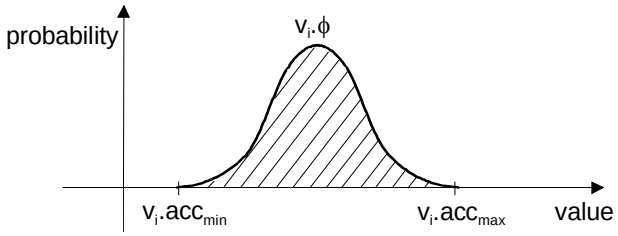


Figure 17: Probability Density Distribution of Value v_i

8.3.1 Accuracy

In Chapter 4 we have defined the state of the model as a set of (variable, value) pairs. In the following we extend the definition of value by providing the accuracy for a given value that needs to be taken into account for the observation. The limited accuracy of the value is introduced through both the limited sensor accuracy and the update protocol as we will see in the next section.

In principle, values can be multi-dimensional and complex. Here we begin with simple, one-dimensional values, but, as we will see, the concepts can easily be extended to the multi-dimensional case.

In the most general case, a value v_i can be specified in form of a probability density distribution $v_i.\phi$ over an accuracy interval $[v_i.acc_{min}, v_i.acc_{max}]$, see Figure 17.

Definition 12: Probability Density Distribution of Value v_i

For a value v_i a probability density distribution $v_i.\phi$ over the accuracy interval $[v_i.acc_{min}, v_i.acc_{max}]$ can be given as $vi.\phi[v_i.acc_{min}, v_i.acc_{max}]$

with

$$\int_{(v_i.acc_{min})}^{(v_i.acc_{max})} v_i.\phi(v)dv = 1$$

If the probability for all the values in the interval is equal, we have an equi-distribution and it is sufficient to give the accuracy interval $[v_i.acc_{min}, v_i.acc_{max}]$. If we do not have any given probability density distribution, we may also have to assume an equi-distribution or a normal distribution.

The case of an exact value v_i is a further specialization where the accuracy interval becomes a single point with a probability of 1.

If we go to n-dimensional values, e.g. a two- or three-dimensional spatial coordinate, the value is given as an n-dimensional value; instead of an accuracy interval, we have an n-dimensional body and the probability density distribution is an n-dimensional probability density distribution over the given body.

For example, if we have a positioning system, the likelihood that the actual position is in the center of the accuracy area may be higher than at its edges, e.g. see [GPS Signal Specification 1995], which can be modeled by a probability density distribution.

8.3.2 Update Interval

The update of model state cannot be attributed to a fixed point in time, but only to a given time interval $[v_i.t\_acc_{min}, v_i.t\_acc_{max}]$, e.g. because of clock synchronization issues. So the updated value is associated with this time interval. The time interval is based on the time stamp, which can already be given as an interval (e.g. from the sensor), and the maximum clock skew. This also means that for the interval in which the new value has become valid, the new and the old value coexist with the probability of the old value decreasing and the probability of the new value increasing over the time interval. This has to be taken into account for the observation.

Again, as the distribution of the time may not be equal over the time interval, a probability density distribution $v_i.\delta$ over the time interval can be given as $v_i.\delta[v_i.t\_acc_{min}, v_i.t\_acc_{max}]$ with

$$\int_{(v_i.t\_acc_{min})}^{(v_i.t\_acc_{max})} v_i.\delta(t)dt = 1$$

8.3.3 Change of Value over Time

For the observation of global events, the observer model has to provide model state over time. If the maximal change of a value over time is known, a "worst-case" estimation for a point in time for which no current value is available (yet) can be given. For example, a pedestrian may move at a maximum speed of 10km/h, so the current location can be estimated as the location of the last update plus the product of the time that has passed since then and the maximum speed. This means that the accuracy interval or body and the probability density distribution can also change over time, which has to be taken into account for the observation. In the extreme case, the probability of the event having occurred can cross the threshold probability simply through the change over time.

For the observation, it not only has to be checked, if the predicate evaluates to true for the current state, but also for the previous state, as we are interested in the predicate becoming true, which signifies the occurrence of the event. In some cases not only the current change, but changes over a longer time interval have to be available for the evaluation of the predicate, so a history of the model state has to be provided. An example for such an event would be that a value has increased for the tenth time within 5 minutes. Over what period of time the history needs to be provided depends on the event.

8.3.4 Resulting Model

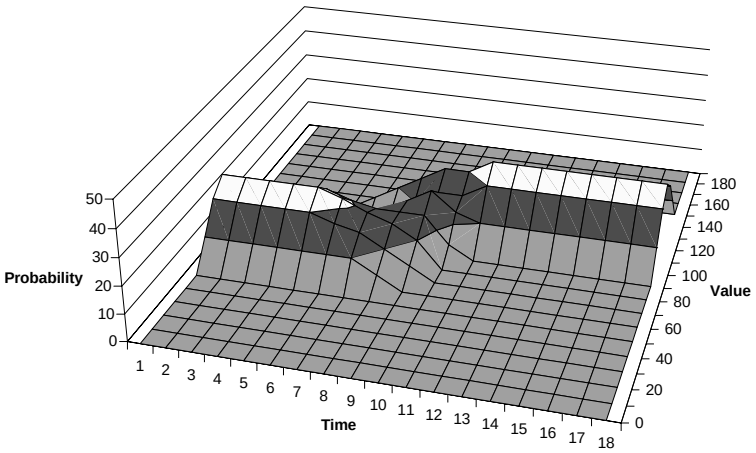


Figure 18: Probability Density Distribution of a Variable over Time

Figure 18 shows the model for a given variable over time. At any point in time the variable has a value that is given in form of a probability density distribution. If there is a new update, the time density distribution defines how the distribution before the update is "faded over" to the new distribution. In Figure 18 an update takes place with an update interval between Time 5 and Time 10, which modifies the accuracy density distribution over time.

As a next step we look at update protocols that can be used to provide such a model with data.

8.4 Update Protocols

Update protocols are used to propagate the data from the data model node to the observer nodes. The update protocols define the data available in the observer model at a given time. So the update protocols determine the accuracy of the model, thereby defining its "quality".

We can classify update protocols according to who initiates the check, if an update is necessary, and how the check is triggered (also see [Leonhardi & Rothermel 2001])

As far as initiating the check is concerned, there are two principle options, either the receiver queries the source, which corresponds to „pull-based" communication, or the source sends an update when necessary, which corresponds to „push-based" communication. As in our case the initiative for observation has to come from within the system, a „push-based" approach is appropriate, therefore we will not consider query-based approaches here.

There are two general classes of push-based update protocols, depending on how the update is triggered: value-based protocols and time-based protocols. Value-based update protocols send update notifications based on a change in value and time-based protocols send update notifications in regular time intervals. Time-based protocols as such cannot guarantee any accuracy of the value. Since this is necessary for the observation, we assume that an initial accuracy interval or density distribution is available; in addition we need a function that models the (maximal) change of the value of a variable over time. With this information, we have a value for any point in time.

Another important aspect is if it has to be guaranteed that the value in the updated model is within the accuracy distribution at any point in time, or if it is sufficient to have accurate information for the evaluation at a later, but known point in time. In the first case, it is possible to do evaluations at real-time with limitations regarding the accuracy, but it also requires that certain information about how values can change over time is available and that the underlying distributed system provides the necessary „quality of service" as described for the event domain.

8.4.1 Value-based Update Protocols

Value-based protocols send an update message whenever the value of a variable has changed in such a way that an update criteria is fulfilled. This update criteria can also be defined as a predicate that becomes true whenever such a change in the value occurs. This means, an "update event" has occurred and an update message is sent, so that the variable in the observer model can be updated by the new value. A typical predicate might specify that a distance between two values is larger than a given threshold, taking into account the accuracy information.

In our case, we assume that the accuracy interval for each value is known. If we want to have a certain accuracy for the observer model – which of course cannot be more accurate than what is available in the local model – we have to specify this accuracy in the predicate, i.e. as a threshold.

As mentioned above, there is a difference between protocols that only update the information after the source has detected that the accuracy requirement has been violated, as we have just seen, and protocols that guarantee that a value on the receiver side has a certain accuracy at any point in time.

In order to realize the latter case, additional information is needed. First, the source has to have a certain guaranteed accuracy that is more accurate than the accuracy to be guaranteed at

the receiver side. Then, the source also has to know the maximum delay between itself and the receiver(s) and information about how the value may change over time in the worst case, which depends on the characteristics of the information and the maximum delay, which is the sum of all processing and network delays from the sensor to the receiver. The idea is that the source has to be able to determine, if the next update it has to send can wait until it receives a new update without violating the guaranteed accuracy.

In this case, the receiver does not have to wait for the maximum delay with the evaluation, since all relevant values are guaranteed to be within their accuracy intervals at any point in time.

Overall, value-based protocols lead to a model with a relatively regular distribution in the value dimension (see Figure 19 and values v_0 (= $v_i[t_0]$) to v_3), as there is a new update value with the granularity of the accuracy interval, but there is no regular distribution in the time dimension (t_1 to t_3). So, as we can see in Figure 19, we have a maximum accuracy interval [$v_i.acc_{min}$, $v_i.acc_{max}$] (shown for v_0), i.e. the maximum inaccuracy, that is constant for the time between updates. Without any other external information, we do not know anything about the probability density distribution $v_i.\phi$ over this accuracy interval, so we have to assume an equi-distribution.

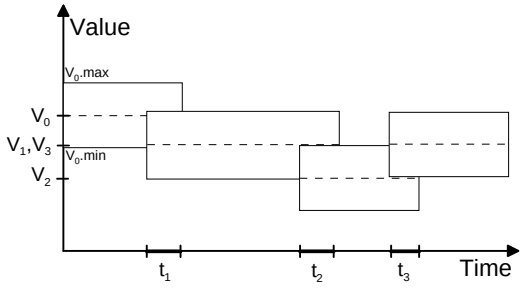


Figure 19: Value-based Update Protocol

In the general case, the predicate can only be evaluated for any given update after the maximum delay has passed that update messages can experience in the event domain. Only after that delay it is clear that no other update messages that have experienced a longer delay could influence the evaluation of a predicate for the time in which the update took place. In case of guaranteed accuracies, however, this is not necessary, as we can always assume to have the guaranteed accuracy level.

One problem of value-based update protocols is that faults of sources or the network may not be detected by the receivers. If no message is received, it will just be assumed that the value is still accurate enough, not that nodes or the network may be down.

8.4.2 Time-based Update Protocols

Time-based update protocol are triggered by the system clock. An event notification is sent in regular intervals as specified. In time-based protocols, there is no guaranteed accuracy. The accuracy interval can only be calculated if an initial value is available and also the maximum change over time is known.

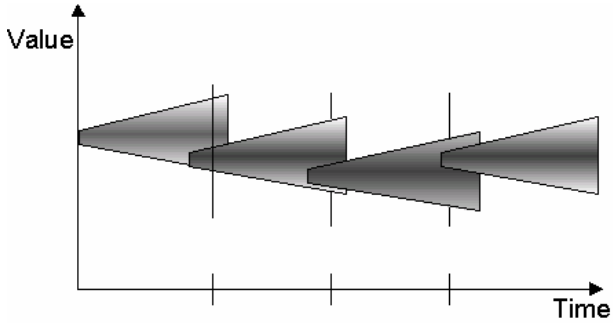


Figure 20: Time-based Update Protocol

With the change function being known to the receiver, this leads to a model, in which the accuracy interval [$v_i.acc_{min}$, $v_i.acc_{max}$] changes over time (see Figure 20), but, again, we do not automatically have any information about the probability density distribution $v_i.\phi$. If we want the distribution in our model, we would need additional information. In Figure 20

a probability distribution is indicated through the shading. If we want the most accurate information for the evaluation of the predicate, we may have to wait for the maximum delay, but there can also be an immediate evaluation on possibly less accurate values.

Unlike, for the value-based protocol, faults of sources or the network can be detected by the receivers. If no message is received after the specified time interval plus the maximum delay, it means that there is a fault in either the source or the network.

8.4.3 Other Protocols

Other variations of the protocols are possible, e.g. there could be a combination of value-based and time-based protocols, which would solve the problem of detecting faults. In addition, if the change of a value over time can be predicted, a dead reckoning protocol can be used, where both sides use the same prediction function and an actual update is only sent when the difference between the real value and the predicted value crosses a given threshold. For a broader discussion of different types of update protocols taking the example of location updates, see [Leonhardi & Rothermel 2001].

8.5 Event Observation

Now that we have defined the observation model and shown how it can be realized using update protocols, it remains to be shown how global events can be observed based on the resulting model. In order to do that, the probability with which an event has occurred has to be calculated and compared to the specified threshold probability.

To decide if an event has occurred, it has to be checked, if there was a state change in the model that leads to the predicate evaluating to true. In other words, evaluating the predicate describing the event returns false before the state change and true afterwards. In case the values of the relevant variables and/or the point in time when the change has occurred can only be determined as intervals with a probability density distribution, as in the model we have defined in Section 8.3, it may not be possible to determine for certain that such a change in the evaluation of a predicate has occurred. For those cases, we want to calculate the probability with which the predicate evaluates to true. This probability can then be compared with a predefined threshold probability to decide, if the event is considered to have occurred or not. In the following we discuss how to calculate this probability. As this gets rather complicated for the general case, we start out with a number of constraining assumptions that we relax step by step to arrive at the general case in the end.

We also make a general assumption about the observer model: Values in the observer model can only change when there is an explicit update, i.e. we assume that there are no automatic model-internal changes of values over time. This means that predicates only need to be evaluated when there is an explicit update. At first sight, this may seem very restrictive, but such changes in the model can be realized through internal updates. So here predicates only need to be evaluated as the result of an update.

8.5.1 Update in the Exact Case

We start the formalization with the case in which we have an update with an exact value at an exact point in time. This means we have to evaluate the predicate for the point before the value was updated as well as for the new value.

> Let P be a predicate that is defined over the variables v_1, ..., v_i given as exact numbers. The variable v_1 is updated at the point in time t_1 and t_0 is defined to be $t_1-\varepsilon$ for $\varepsilon \rightarrow 0$. Then the event specified by P has occurred, if the following holds:

$$P(v_1[t_0], \ldots, v_i[t_0]) = false \ \ and \ P(v_1[t_1], \ldots, v_i[t_1]) = true$$

(Equation 1)

8.5.2 Update with the Value Given as an Accuracy Interval

For the following we no longer deal with exact values, but accuracy intervals. We interpret the predicates as functions that return 0 or 1 for a given set of values and assume that for each point in an interval, it is equally likely that the point is the actual value, which means we have an equi-distribution over the interval. (We could also assume a normal distribution, in which case the next subsection applies.)

As for each value, each point of the interval has the same probability, we have to evaluate the predicate for all possible combinations of points and take the average. Since we do not have discrete points, but continuous intervals, we have to integrate over the intervals, interpreting the predicate as a function over the values and normalize it by dividing it by the length of the intervals. The result is a value between 0 and 1 that can be interpreted as the probability that the predicate holds over the interval.

Let P be a predicate that is defined as above over v_1, ..., v_i given as accuracy intervals, i.e. v_i stands for $[v_i.acc_{min}, v_i.acc_{max}]$. $|v_i|$ is defined as $|v_i.acc_{max} - v_i.acc_{min}|$. Again, the variable v_1 is updated at the point in time t_1. The threshold probability TP is given as a value between 0 and 1.Then the event specified by P is considered to have occurred, if the following holds:

$$\frac{1}{|v_1[t_0]|} \cdot \ldots \cdot \frac{1}{|v_i[t_0]|} \cdot \int_{v_1[t_0]} \left(\ldots \left(\int_{v_i[t_0]} P(v_1, \ldots, v_i) dv_i \right) \ldots dv_1 \right) < TP$$

$$and$$

$$\frac{1}{|v_1[t_1]|} \cdot \ldots \cdot \frac{1}{|v_i[t_1]|} \cdot \int_{v_1[t_1]} \left(\ldots \left(\int_{v_i[t_1]} P(v_1, \ldots, v_i) dv_i \right) \ldots dv_1 \right) \geq TP$$

(Equation 2)

8.5.3 Update with the Value Given as a Probability Density Distribution

Instead of an equi-distribution, we can have any probability density distribution over the given accuracy interval. In this case, we have to integrate over the probability density distributions multiplied by the predicate interpreted as a function. Again, the result is a value between 0 and 1 that can be interpreted as the probability that the predicate holds over the interval.

> Let P be a predicate defined over the variables v_1, ..., v_i for which probability density distributions (ϕ_i) over the accuracy interval are given. Again, the variable v_1 is updated at the point in time t_1 by a new probability density distribution. Then the event that is specified by P is considered to have occurred, if the following holds:

$$\int_{v_1[t_0]} \left(\phi_1[t_0] \ldots \left(\int_{v_i[t_0]} \phi_i[t_0] P(v_1, \ldots, v_i) dv_i \right) \ldots dv_1 \right) < TP$$

and

$$\int_{v_1[t_1]} \left(\phi_1[t_1] \ldots \left(\int_{v_i[t_1]} \phi_i[t_1] P(v_1, \ldots, v_i) dv_i \right) \ldots dv_1 \right) \geq TP \qquad \text{(Equation 3)}$$

8.5.4 Update over a Time Interval without Any Interleaving Updates

So far, we have looked at the case for which it is known that the update has occurred at an exact point in time. As this assumption is not very realistic in a distributed system, we now look at the case, in which it is only known that the update has taken place within a certain time interval. Not all points in the time interval may have the same probability, so again, we have to take a probability density distribution into account. As a result, we know for each point in the time interval with what probability the change has already taken place. In order to focus on the time dimension, we assume exact values, before integrating all aspects in the general case at the end.

If there are no other changes during the time interval of interest, it is sufficient to evaluate the predicate for the begin of the interval, when the update has not yet taken place, and the end of the interval, when we know for sure that the change has taken place, to determine if an event has occurred. So basically we have the same situation as in Equation 1 only that t_0 marks the beginning of the update interval and t_1 the end.

8.5.5 Update over a Time Interval with Interleaving Updates

If the values of other variables that are needed for the evaluation of the predicate can also change during the time interval in which the update has taken place, it is no longer sufficient to check at the end of the update interval. The predicate may have become true during the interval (with a certain probability), but due to other changes, this is no longer the case at the end of the interval. So, when checking for a given point in time, it is possible that an old and a new value for the same variable have to be taken into account with the respective probabilities with which they are valid at that point. Overall, we have to find the point in the time interval where the predicate is true with the maximum probability.

Let P be a predicate that is defined over the variables v_1, ..., v_i given as exact numbers. The variable v_1 is updated in the time interval m between t_0 and t_1. δ_m is the probability density distribution over the interval m.

$pb(v_i,x,t)$ is the probability that variable v_i has the value x at time t. In case there are no overlapping updates for a single variable, it can be calculated for the update of variable v_1 as follows, where x_1 is the value before the update and x_2 the update value, t_k is a point in time within the update interval:

$$pb(v, x_2, t_k) = \int_{t_0}^{t_k} \delta_m(t)dt \qquad pb(v, x_1, t_k) = 1 - pb(v, x_2, t_k)$$

S(v,t) is the set of values x_1, ..., x_i of variable v at time t for which pb(v,x,t) > 0. In the case of no overlapping updates S(v,t) has at most two values.

$max(f(\ldots))|_{t_0}^{t_1}$ is the function that returns the maximum of function f within the interval (t_0 ... t_1].

Then the event that is specified by P has occurred, if the following holds:

$$P(v_1[t_0], \ldots, v_i[t_0]) = false \text{ and}$$

$$max\left(pb(v_1, x_2, t)\sum_{S(v_2,t)}\left(pb(v_2, x_g, t)\cdot\ldots\sum_{S(v_i,t)}(pb(v_i, x_h, t)\cdot P(v_1, \ldots, v_i))\right)\right)\Big|_{t_0}^{t_1} > TP$$

(Equation 4)

We face an additional problem, if the intervals for updates of the same variable can overlap. Now, a given variable can have more than two different values at the same time, each with a certain probability. This is not only the case during the update intervals themselves, as after an update it can only be determined with a given probability which update actually came first. As a result, the probability of a given value for a variable only becomes 0 after a completed update of the same variable whose update interval did not overlap with the update interval for the given value. However, we do not look at this case in detail here, because, assuming that all updates come from the same local model, we can determine the sequence of updates and in that case the described problem does not arise.

8.5.6 General Case

Integrating the cases for values given as probability density distributions and updates over potentially overlapping time intervals yields the general case that allows the evaluation of predicates over the general model as defined in Section 8.3. In order to get the general case, we have to replace the predicate (function) in Equation 4 by Equation 3, which yields Equation 5.

$$\int_{v_1[t_0]} \left(\left(\phi_1[t_0] \ldots \left(\int_{v_i[t_0]} \phi_i[t_0] P(v_1, \ldots, v_i) dv_i \right) \ldots dv_1 \right) < TP \right) \; and$$

$$max\left(pb(v_1, x_2, t) \sum_{S(v_2, t)} \left(pb(v_2, x_g, t) \cdot \ldots \sum_{S(v_i, t)} (pb((v_i, x_h, t) \cdot Q)) \right) \right) \Bigg|_{t = t_0}^{t_1} > TP$$

$$where \qquad Q = \int_{v_1[t]} \left(\phi_1[t] \ldots \left(\int_{v_i[t]} \phi_i[t] P(v_1, \ldots, v_i) dv_i \right) \ldots dv_1 \right)$$

(Equation 5)

8.6 Realization Issues

As can be seen in Equation 5, the calculations in the general case can become rather complicated. So, for efficiency reasons, it may be necessary to only calculate an approximation of the actual result. The best approach may depend on the actual event and the desired semantics. The following aspects should be considered when choosing a heuristic for the approximation:

- The complexity of the predicates: The more detailed the specification of the event has to be, the more complex the predicates are and the more complex the calculation. Sometimes a slightly less accurately described event can be observed much more efficiently. If the number of cases in which the event actually occurs is low compared to the number of cases in which the predicate has to be evaluated, it may be worth to have a test in two steps. The first test just checks a predicate describing a rough approximation of the event, which can be done efficiently, and only if this evaluates to true, the actual predicate is evaluated.
- The probability distribution of the values: It may be sufficient to assume an equidistribution instead of a complex probability distribution. Depending on the update protocol this may be all we have anyway.
- The accuracy interval of the values: It may be possible to simplify the calculation by evaluating the predicate for a few (weighted) representative values instead of the whole accuracy interval.
- The time interval and probability distribution: If there are no overlapping update intervals, the calculation is the same as in the case where we have exact time points. If update intervals do overlap, it may be sufficient to check for a few values to determine, if any event may have occurred, and only do an exact check, if this is the case.

As we can see, the calculations that have to be done in the general case can be simplified for given specific cases. However, what is reasonable in a given case depends on the actual event and the desired semantics. As an example, [Dudkowski 2002] looks into the observation of spatial events. For the observation of the event that two users meet (onMeeting), whose locations are given in form of circular accuracy areas, an approximation based on a number of

representative values is used. We are confident that this approach will work in other cases and we plan a more detailed investigation for a number of specific cases.

Chapter 9 Observer Placement

In the previous chapter, we have shown how events can be observed through a distributed world model and what the relevant parameters are that determine the quality of the model. The event semantics depend directly on the quality of the model. Therefore, it is important to observe an event at the location with the best possible observer model, so the placement of the observer in the network is crucial.

Figure 21 shows two observers. One is connected to the event sources through a fast local area network, whereas the other is connected through a slow modem. The observer models on top show the accuracy of a certain value over time as it is available in the respective model. Whereas on the left side, the accuracy interval is very small for each point in time and the time when the value changes is relatively accurately defined, this is not the case on the right side, where the interval in which the value may have changed is relatively long (see Figure 18 for a larger picture of the model).

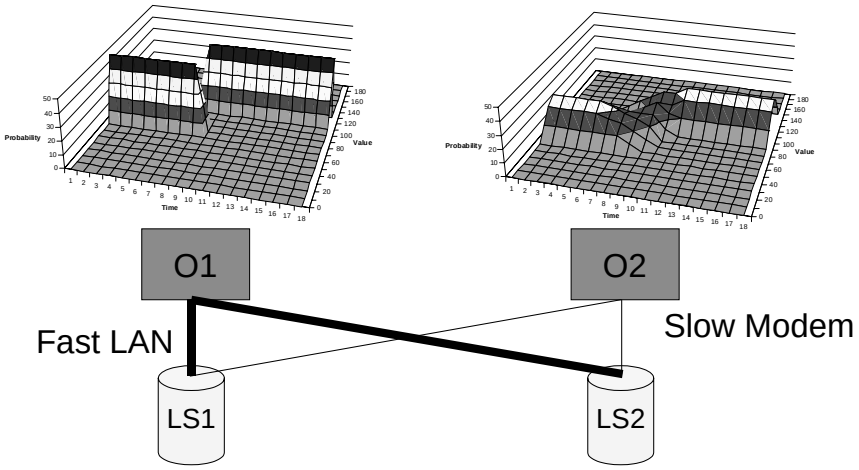


Figure 21: Different Observers and their Respective Observer Models

To derive possible optimization criteria, we first look at the goals of the different parties involved.

The user of the service wants the best possible event semantics:

- maximal observation accuracy
- minimum delay

The operator of the service is interested in performance, stability and scalability, which corresponds to the following optimization goals:

- balance server load
- minimum network load

As some of the goals are potentially in conflict, e.g. balanced server load vs. maximal observation accuracy, there have to be trade-offs.

Table 4: Mapping Optimization Goals to Parameters

Optimization Goal	**Parameters**
Observation Accuracy	Local accuracy permissions, maximal notification rate allowed, clock skew
Minimum Delay	Delay, (network load)
Restrict/Balance Server Load	Load/node: number of observers/node, [sum(load by observers)/node]
Restrict/Minimize Network Load	Link load, Sum(link loads)

Table 4 shows a mapping between the optimization goals and the parameters whose values can be optimized. We can further distinguish between system-dependent parameters, e.g. delay and clock skew, and policy-dependent parameters, e.g. maximal notification rate allowed.

Different optimization strategies can be investigated:

- System load: to optimize for system load, the load of the possible observation nodes has to be compared. The advantage is that this is a single value, however the load is a dynamic parameter that can change over time.
- Delay: to optimize for delay, the delay of the communication paths between the event sources and the observation nodes has to be optimized. As this involves at least two values, it is not a-priori clear, if the sum or some other relation, e.g. the difference of the values, should be optimized. This may also be event-dependent.
- Observation accuracy: as the observation accuracy depends on two parameters, the maximal notification rate allowed (affects the accuracy in the value domain) and the clock skew (affects the accuracy in the time domain), a suitable method to combine the two values has to be found.

If we abstract from the concrete optimization strategies, we have to solve a general optimization problem in which a tree (of logical observation nodes) has to be mapped to a graph (consisting of event sources and observation nodes) and minimize “costs”, which are defined by the parameters, e.g. delay, clock skew, 1/(notifications/s), or a combination of the parameters.

In general, multi-parameter optimizations are difficult. Our plan is to focus on the observation accuracy, and evaluate how that affects the other parameters. Due to time limitations, we have not been able to do that yet. As the next steps we will look at concrete application scenarios to have a basis for evaluating the optimization strategies.

Another issue that has to be addressed with respect to the observer placement is the dynamic reconfiguration of the observation, Mobile objects move between service areas of location servers and handovers are performed on that level. To keep the observation optimal, the placement of the observation has to be adapted.

So far, we have assumed that there are a number of observation nodes that exist and we optimize based on those. However, the question for a system operator may be where to place observation nodes in the first place. Thus, a lot of research questions remain to be solved here.

Chapter 10 Integration of Events into Event Sources

Event sources are an important foundation for any event service. We have identified spatial events as the events of primary interest in Nexus. For the observation of spatial events, the position data of mobile objects is required. In Nexus, the location service stores the position data of mobile objects. Therefore, the location service has to be instrumented to serve as a an event source.

In the first section, we describe a specialized event component for the location service, which was developed as part of a student thesis [Dudkowski 2002] early in the project, as it was important to have a reference event source. It is specialized with respect to the fixed set of events that are supported and the interface for registering events. In the second section the general format for event notifications based on XML is described, and in the third section a generic XML-based format for the registration of events is presented. In the fourth section, we describe a generic event component for the location service that was developed as part of the a diploma thesis [Minder 2003]. This interface supports the registration of events using the generic XML-based format and allows the integration of new event types through the download of plug-in triggers.

10.1 Specialized Event Component for a Leaf Location Server

The goals for developing the first specialized event component were to have an event source that can observe a fixed set of event types in a single leaf location server.

The supported event types correspond to those that we have classified in Section 6.3. We repeat them here with their specific parameters. The classification of the respective parameter is added in brackets. A specific parameter has to uniquely identify a single object or area, whereas a variable parameter can be given as an object selector, i.e. specify attribute values that an object has to have. For example an object selector can specify that a mobile object has to be a student or a car.

Data-based Events

- onEnterArea(mobile object (variable), area (specific)): specifies the event that a mobile object enters a given area
- onLeaveArea(mobile object (variable), area (specific)): specifies the event that a mobile object leaves an area
- onMeeting(mobile object (specific), mobile object (variable), distance): specifies the event that the distance between the two mobile object becomes closer than the given distance
- onCrossingLine(mobile object (variable), line (specific)): specifies the event that a mobile object crosses a line
- distPosUpdate(mobile object (specific), report distance (specific)): specifies the event that a mobile object has moved further than the report distance from the previously reported position; the new object position is sent with the event notification.

Timer-based Events

- contPosUpdate(mobile object (specific), report interval (specific)): specifies the event that after each report interval, the new object position is sent with the notification.

- contAreaUpdate(area (specific), report interval (specific)): specifies the event that after each report interval, the positions of all objects in the given area are sent with the event notification.

Management Events

- onRegisterObject(mobile object (specific)): specifies the event that the given object has registered with the location service.
- onUnregisterObject(mobile object (specific)): specifies the event that the given object has deregistered from the location service.
- OnRegisterArea(mobile object (variable), area (specific)): specifies the event that a mobile object has registered with the location service with a position in the given area.
- OnUnregisterArea(mobile object (variable), area (specific)): specifies the event that a mobile object has deregistered from the location service with the last position having been within the given area.

Implementation of the onMeeting Event

As indicated in Subsection 6.3.5, the onMeeting event with one mobile object given in form of an object selector can be efficiently observed: The onMeeting event has two parameters with dynamic position attributes, i.e. a change in either of these attributes can result in the occurrence of the event. We now consider the specific mobile object to be the primary mobile object. If the primary object moves, it can be checked if any mobile object within the given distance fits object selector, but has not already been within the given distance for the previous position of the primary object. Whenever the primary object moves, a temporary onEnterArea event is registered. The event area is given as a circle around the current position of the primary object with the given distance as the radius and the mobile object has to specify the object selector for the onMeeting event. If such a mobile object moves so it is within the given distance of the primary object, the temporary onEnterArea occurs and as a result, the onMeeting event can be detected.

Assumptions

For the first implementation we have made a number of assumption. The most important assumption is that we have exact position information. As we have already seen, this is not very realistic, but got us started. With the implementation of the generic event component and the support for the distributed observation of location service events, we now also support position information with a limited accuracy.

For the movement of the mobile objects, we have assumed that between position updates they move in a straight line and with constant speed between two position updates. This is especially important for onCrossingLine events.

Event Triggers

As we discussed in Subsection 6.3.2, user events can be triggered by a number of system events. The system events in a location server are:

- position update
- register
- deregister
- handover

In the following we will list the user events that can possibly be triggered for each of the system events. For each of these system events the predicates describing those user events that can possibly have occurred have to be checked:

- position update: onEnterArea, onLeaveArea, onCrossingLine, onMeeting, distPosUpdate
- register: onRegisterObject, onRegisterArea, onMeeting,
- deregister: onDeregisterObject, onDeregisterArea
- handover: onEnterArea, onLeaveArea, onCrossingLine, onMeeting, distPosUpdate

On a timer system event, the contPosUpdate and contAreaUpdate can occur.

To support event observation, observation hooks have to be integrated into the location server that initiate the checking for event occurrences for each system event.

Efficient Event Observation

Requirement 8: Efficiency states that the observation of events should be as efficient as possible. This also applies to the observation of events in the event component.

To make the event observation more efficient, the number of predicates that have to be checked for each system event have to be kept to a minimum. As discussed in Subsection 6.3.5, efficient index structures can help to easily access only the predicates describing those user events that have to be checked.

As the data-based events are directly related to system events in the location server, whereas the timer-based events are not, there are two different managers: a spatial events manager and a timer-based event manager. The timer-based event manager is run as a separate thread so it does not interfere with the operations of the location server.

Data-based Events

For the events that are attached to specific mobile objects, a simple hash table can be used as an index structure to access the predicates to be checked. For the events attached to a specific area, the predicates to be checked have to be found through a spatial index structure. We have used a quad tree, a two-dimensional index structure. In the three-dimensional case an oct tree could be used.

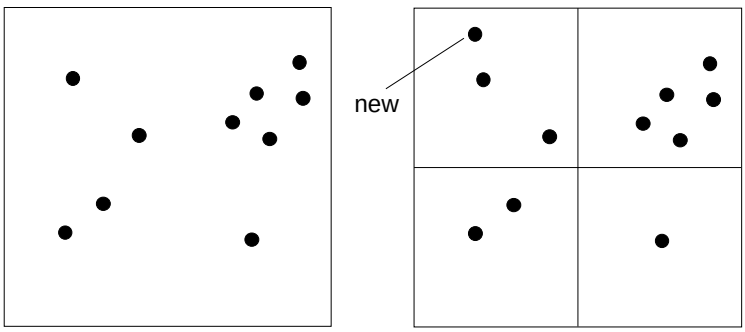


Figure 22: Quad Tree Example
Source: [Dudkowski 2002]

Figure 22 shows a visualization of a quad tree, to which a point has just been added. Every quad tree node has the capacity to store a number of points. If the number of points are larger than the capacity, the node is split into four parts that are added to the quad tree as new leaf nodes. A quad tree can be defined to cover a certain geographical area, i.e. the root node covers the whole area and the areas of the child nodes taken together give the area of the parent node.

The predicates for area events are stored in the quad tree node with the smallest area that completely covers the event area. The quad tree can be queried for a point location, e.g. as returned by a position update, to find all the quad tree nodes whose areas overlap with the given point. Exactly the events stored in these quad tree nodes have to be checked.

Timer-based Events

The predicates for timer-based events can be efficiently accessed by a heap index structure. For each registered event the next occurrence is calculated and this time is inserted into the heap. The smallest element, i.e. the timer-based event that occurs next is always on top of the heap. At the respective time the top element is removed from the heap, an event notification is sent, the next occurrence time is calculated and reinserted into the heap. The overall complexity for all operations on the heap (read next element, remove next element and add new element) is O(log n).

Congestion Avoidance

If too many timer-based events per unit of time occur, the event component can no longer keep up with sending event notification. Therefore, a mechanism was implemented that detects a possible overload and prevents the registration of more timer-based events.

Soft State Approach

Requirement 10: Mobility Support states that the effects resulting from the mobility of the user and the wireless connections used should be taken into account. Mobile devices can easily run out of battery or loose their network connection. However, the events they have registered will continue to be observed in the system. To avoid an overload of the system with orphaned event observations, a soft state approach is followed. Clients can register events only for a certain time interval. If they continue to be interested in the observation, they can refresh the observation before the end of the time interval. If there has been no refresh by the end of the time interval, the event is (implicitly) deregistered.

Registration Interface

The specialized event component has a SOAP-based event registration interface. The interface has a separate method for each supported event.

Event Notification

On the occurrence of an event, an event notification is handed over to the notification service. We discuss the general content of this event notification in the next section.

The evaluation of the specialized event component will be presented in Section 13.1, together with the evaluation of the event service as a whole. More information about the specialized event component can be found in [Dudkowski 2002] .

10.2 Event Notification Format

An event notification has the following elements:

- ID: The (notification) ID uniquely identifies the event notifications, which corresponds to a concrete event occurrence. For example the onEnterArea(Tom, Office12) event that occurred at 09:30:25. In Nexus, IDs are given as NOLs (Nexus Object Locators).
- Predicate ID: The predicate ID uniquely specifies the event that is being observed, e.g. onEnterArea(Tom, Office12)

- Template ID: the template ID uniquely specifies the type of event being observed, e.g. onEnterArea(<mobile object>, <area>)

```
<?xml version="1.0"?>
<notification xmlns="http://www.nexus.uni-stuttgart.de/Notification">
<id>nexus://nexus.uni-
stut-
tgart.de/0x9a06f8e11c3f9dd326e31a6a2437c964/0x9a06f8e11c3f9dd326e31a6a243
7c964</id>
  <predicateId>nexus://nexus.uni-
stutt-
gart.de/onEnterAreaEvent/0xcf2d68a644dd219d09f66deee6874a5f</predicateId>
  <templateId>nexus://nexus.uni-
stutt-
gart.de/onEnterAreaEvent/0x00000000000000000000000000000000</templateId>
  <name>onEnterAreaEvent</name>
  <service>Location Service</service>
  <server>127.0.0.1</server>
  <counter>1</counter>
  <scope>2002-01-17T20:51:14.418+00:00</scope>
  <timestamp>2002-01-17T18:51:14.418+00:00</timestamp>
  <variableList>
    <variable>
      <name>Entering Object</name>
      <type>Mobile Object</type>
      <restriction>Mobile Object</restriction>
      <value>00000000ea600000000000000000</value>
    </variable>
    <variable>
      <name>Entering Space</name>
      <type>Nexus Object</type>
      <value><SEGMENT WGS84: 48.7746077, 9.18255183, 0.0; WGS84:
48.77415773, 9.183014, 0.0></value>
    </variable>
  </variableList>
</notification>
```

Figure 23: Event Notification for an onEnterArea Event

- Name: the human-readable name of the event type, which does not necessarily have to be unique, e.g. onEnterArea
- Service: specifies the service that has observed the event, e.g. the location service
- Server: specifies the server that has observed the event, e.g. the location server with the IP address 129.99.99.99.

- Counter: specifies the number of times this event has been observed so far by the given server, e.g. the 10^{th} occurrence.
- Scope: specifies for how long the event notification is considered valid.
- Timestamp: specifies the time when the event was observed.
- Comment: optional element
- Variable List: the list of event variables, specified as 4-tuples:
    - Name: the name of the variable, e.g. Entering Object
    - Type: the type of the variable, e.g. mobile object
    - Restriction: a possible further restriction of the object, e.g. origin: Germany
    - Value: the value of the variable, e.g. Tom's NOL

An example event notification is shown in Figure 23. The XML schema that defines the event notification together with more examples can be found in the appendix.

10.3 Generic Event Registration

For the specialized event component we have implemented an interface that has a separate registration method for each event type. This approach is not very flexible and does not allow to flexibly add new event types at run-time. For this purpose a generic approach is needed. We therefore propose a registration message with a generic format for the registration of events.

In Chapter 7 we have proposed to specify events as predicates. We now propose to make predicate templates available, one for each type that the user can register. A predicate template defines the parameters for the event type. In addition to the predicate template an observation module has to exist that implements the observation of the event.

This observation module is then loaded by either an observer node or an event source and instantiated with the parameter values from the event registration message.

Every event has a number of event specific parameters, e.g.

- onEnterArea: the *entering object*, which is of type *mobile object*, and the *entered area*, which is of type *area*
- onMeeting: the *primary object*, which is of type *mobile object*, and the *secondary object*, which is also of type *mobile object*

Then, due to the limited accuracy of the data, a probability threshold has to be defined that determines the minimum occurrence probability up to which an event is considered to have occurred, so that an event notification is sent (see Chapter 7):

- Probability Threshold: in the range (0, 100] percent

In addition to these required parameters, there are a number of optional parameters, that can be specified. If a parameter is not given, a suitable default will be used. These parameters can be grouped into different categories that we discuss now. The parameters in each category may have to be extended depending on the application domain and the exact requirements.

Quality of Service Parameters

The quality of service parameters are optional, but they could be used to influence the observer placement (see Chapter 9) in a certain way:

- Maximal clock skew: The maximal clock skew refers to the maximal time difference of any two clocks within the observation hierarchy. (of course this is a “statistical maximum” and not an absolute bound)
- Maximal delay: The maximal delay refers to the maximal time it takes from the observation of the occurrence of an event to the delivery of the event notification, possibly taking multiple processing steps into account.
- Average delay: The average delay refers to the average time it takes from the observation of the occurrence of an event to the delivery of the event notification, possibly taking multiple processing steps into account.
- Package loss: The package loss refers to the average rate of lost packages over a certain amount of time.
- Duplicates: Specifies, if duplicate event notifications are allowed. Duplicate event notifications could either be introduced by the underlying system or in the event observation, e.g. when a handover of an event observation is performed.

Evaluation Parameters

The evaluation parameters are those that directly influence the evaluation of a predicate, i.e. they determine if an event has occurred.

- Blocking interval: The blocking interval determines the time after an event occurrence, in which the same event is not observed again. For example, after a user has entered an area, due to the limited accuracy of the position data, there may be an oscillation of the position data showing the user inside and outside the area, which could lead to a number of events being observed. With a blocking interval, this behavior can be avoided.
- Event consumption: If an event is a composition of multiple other events, i.e. an event pattern, there can be multiple occurrences of one of these events, before the occurrence of another event, e.g. a sequence A A A B. The question is what event(s) is/are observed and when the event notification is finally “consumed”, i.e. that it is no longer available for future event observations. A number of different event consumption policy have been proposed: chronicle (oldest), recent (newest) [Liebig et al. 1999], continuous (all subevents initiate a new observation), accumulative (subevents are accumulated).

Notification Parameters

The notification parameters affect the sending of an event notification, i.e. if the notification is sent and what the content of the event notification can be.

- Sub-event parameter values: This parameter decides how the values given in the event notifications of sub-events is included in the event notification of the complex event. The general options are: do not include, include as a flat list, i.e. as a simple list of variables of the complex event, or include as a hierarchy to keep the original structure from the notifications of the sub events. Of course, other, event-specific solutions are possible.

Management Parameters

The management parameters directly concern the management of the event observation and the characteristics of the event occurrences themselves.

- Handovers: decides if handovers of the observation are allowed
- Maximal notification rate: limits the maximal number of notifications per second
- Lifetime of registration: specifies for how long an event has to be observed (soft state)

Error Handling Parameters

The error handling parameters specify how to react if an error occurs.

- Error semantics: different semantics are possible: *ignore*, i.e. ignore the error and continue, *stop*, i.e. stop if an error occurs, or *warn*, i.e. warn the application, e.g. using a special kind of event notification

Based on these parameters we have defined an XML-based registration language. The schema and an example can be found in the Appendix.

10.4 Generic Event Component with Plug-In Triggers

The motivation for developing the generic event component for leaf location servers was that the specialized event component was very efficient concerning the observation of a fixed set of event types (see Subsection 13.1.1), but it lacked extensibility. To add a new event type, changes deep in the location server and in different parts of the event component are necessary. The goal was to allow the observation of new event types at run-time through the download of special observer modules, which we sometimes also call *plug-in triggers*. In addition, we wanted to develop an event component that can easily be adapted and integrated into other (system) event sources.

The general motivation for observing events in the event source as opposed to an observation node is that the event semantics of a locally observed event is much better than the semantics of an event observed further away (see Chapter 8 and Chapter 9), so events should be observed locally wherever that is possible.

The generic event component was developed by Daniel Minder as part of his diploma thesis [Minder 2003].

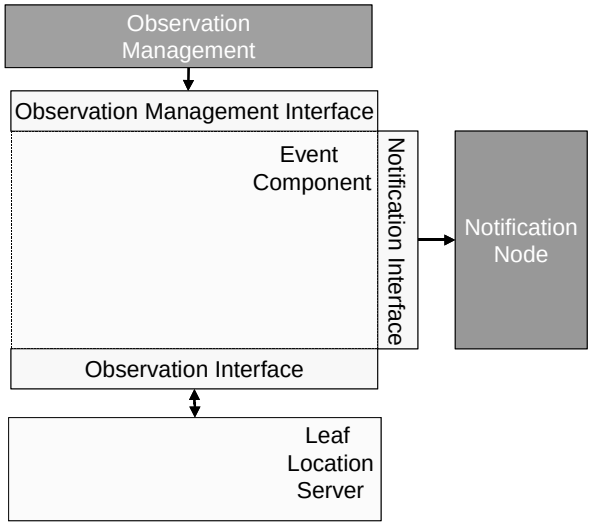


Figure 24: Interfaces of Event Component

Interfaces

Figure 24 shows the three interfaces of the generic event component. The interface towards the observation management is for management purposes, e.g. event registration. The observation interface is towards the (system) event source, in our case the leaf location server. It is responsible for the efficient handover of system events. Finally, the notification interface is responsible for handing over event notifications to the notification service.

An important design decision was how generic the interfaces could and should be to allow an efficient management and observation of events.

Observation Interface

The advantage of having a generic observation interface is that new events can easily be integrated. However, in this case parameters have to be checked first to find out which observer module can handle the event. The advantage of a specialized interface is that this is automatically known, but, on the other hand, the interface needs to be changed for every new observer module. In this case, the decision was made to support a combination of both: a general interface to provide extensibility and a specialized interfaces for certain events for which efficiency is crucial.

Observation Management Interface

The observation management interface is responsible for the registration of events, the deregistration of events, for the refresh of event registration, as a soft state approach has been implemented, and for the handover of event observations. A handover of the event observation has to take place, if the mobile object to which the observation is attached moves out of the service area of a leaf location server. Then it is handed over to another leaf location server and the same handover has to happen on the level of the event components.

As management operations are relatively rare, the general drawbacks of a generic interface do not play an important role. Therefore a SOAP interface was implemented that takes the XML-based event registration message presented in Section 10.3.

Notification Interface

The notification interface is defined by the notification service. The event notification has the format described in message as discussed in Section 10.2.

Recovery

Requirement 7: Fault Tolerance states that the event service should be tolerant to failures. In case a location server crashes, the event component will also crash and with it its current state is lost. For those events, for which that makes sense, the initialization data can be kept on persistent storage. When the location server recovers, the respective events can be reregistered and the observation resumed.

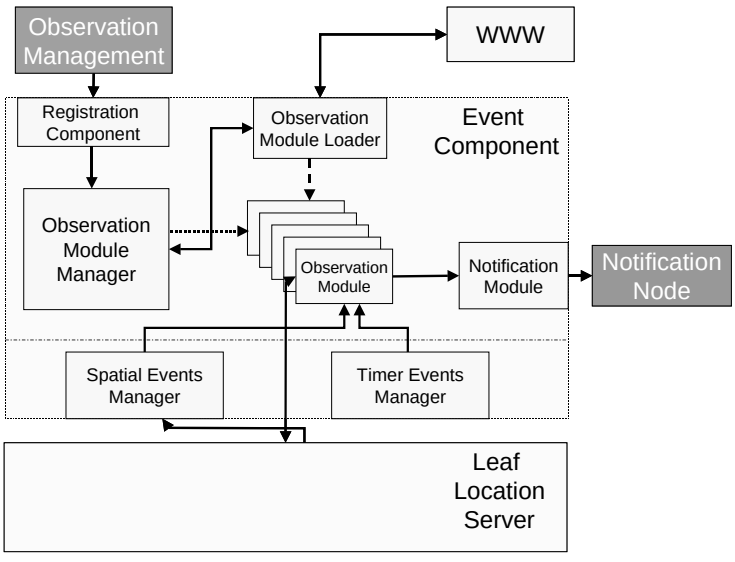


Figure 25: Architecture of Event Component

Reuse of Observation Modules

Certain events provide the basis for the observation of other events. There can be a whole hierarchy of observation modules. The same basis modules may be needed for different observations, so it would be inefficient to observe the same event multiple times with different observation modules. Therefore, the reuse of observation modules is supported in the event module. Whenever a new event is registered, it is checked, if the event is al-

ready being observed.

Architecture of the Event Component

Figure 25 shows the architecture of the generic event component.

When the observation of an event is registered, the main module checks, if an existing observation module can be reused. If this is not the case, it requests an observer module from the observation module loader. It registers all necessary sub-events. There are checks that prevent possible cycles in the observation module hierarchy.

The observation module loader loads the locally available modules on startup. When a new observation module is requested, it checks if such a module is locally available. If this is not the case, it tries to download it using a URLClassLoader. Finally, it returns the instantiated observation module.

The observation modules have to implement different interfaces depending on what kind of system events they require. Observation modules that need data for a specific object have to implement the ObjectModule interface, observation modules that require data for a specific area have to implement the AreaModule interface. Both interfaces are derived from the SpatialModule interface. Observation modules that are timer-based have to implement the TimerModule interface. To implement an event component for a different event source, new interfaces have to be provided.

The spatial events manager manages all observation modules that directly depend on the location server system events. Just as in the specialized event component a hash table (or alternatively a linked list index structure) is used for object-based events and a quad tree (or alternatively a linked list) is used for area-based events. The events that depend on both object and area are registered in both indexes. A duplicate elimination mechanism is used to prevent double evaluation. The timer events manager manages all time-dependent events.

The purpose of a manager is to efficiently determine for each system event, which observation modules need this information. If events are to be observed that are neither spatial nor timer-based, a new manager has to be implemented.

With its general approach, the generic event component can easily be integrated in other event sources. We currently plan to integrate it into the new version of the Nexus spatial model server that is just being developed.

The performance of the generic event component is evaluated and compared to the specialized event component in Section 13.1. For more information about the generic event component see [Minder 2003].

Chapter 11 Observation Service

The observation service is responsible for observing the events that have to be observed on distributed data, i.e. those events that cannot be observed locally within an event source. A general observation service framework has been developed as part of a diploma thesis [Boronas 2003]. Within this framework, the observation of events as described in Chapter 8 can be realized. An important goal when developing the framework was its modularity, so that components can easily be extended or replaced.

11.1 Requirements

The general requirements that are important with respect to the observation service are Requirement 1: Data Distribution Transparency, Requirement 3: Optimal Event Semantics, Requirement 5: Minimum Notification Delay, Requirement 6: Scalability, Requirement 7: Fault Tolerance, Requirement 8: Efficiency, Requirement 9: Interoperability, and Requirement 10: Mobility Support. In the following we will discuss the detailed requirements that follow from these general requirements with respect to the observation service:

- To fulfill *Requirement 1: Data Distribution Transparency*, a client has to have a single point for registering events. So the event service is responsible for registering and observing the event. This includes:
  - the registration of sub-events
  - the optimal placement of the observation of global and composite events (*Requirement 3: Optimal Event Semantics* and *Requirement 5: Minimum Notification Delay*)
  - the efficient observation (*Requirement 8: Efficiency*) of the global event based on data from multiple sources or the composite event based on multiple sub-events
- It should be possible to add observers for new event types at runtime. Following from that, there have to be mechanisms and policies to control the addition of new event types, as they can pose a possible security threat.
- The handover of observations should be supported, in order to keep the event semantics optimal (*Requirement 3: Optimal Event Semantics*) during the course of the observation, e.g. a mobile object may move into the service area of another leaf location server. The handover on the level of leaf location servers may make the observation suboptimal, which may be corrected by a handover between observation nodes.
- For efficiency reasons (*Requirement 8: Efficiency*) the same event should only be observed once, i.e. the observation should be reused, if multiple clients are interested in the same event.
- The replication of the event observation should be possible, e.g. if a server crashes or it gets (temporarily) disconnected from the network (*Requirement 7: Fault Tolerance*).
- Again, a soft state approach should be used to deal with observations that have not been properly deregistered (*Requirement 7: Fault Tolerance*, Requirement 10: Mobility Support).
- Scalability (*Requirement 6: Scalability*) and interoperability (*Requirement 9: Interoperability)* are of major importance for the whole event service, which of course includes the observation service.

- As already mentioned in the introduction to this chapter, the modularity of the observation service is important, so that components can easily be extended or replaced.

11.2 Overview

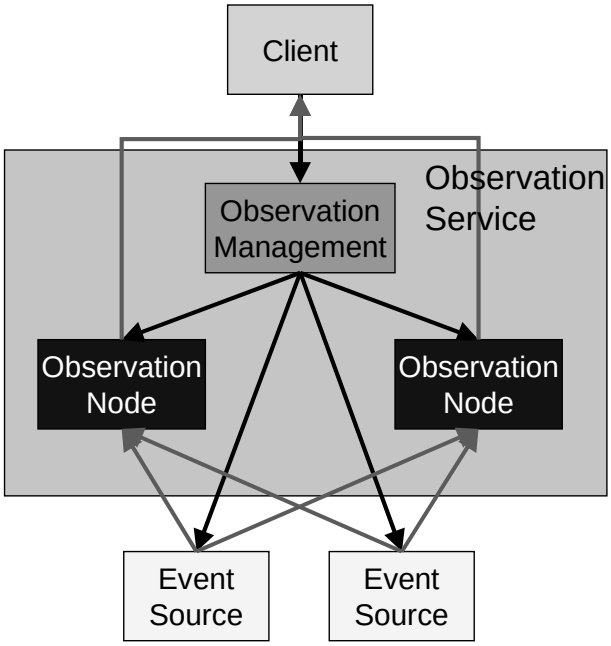


Figure 26: Observation Service Overview

Figure 26 shows a simple configuration of the observation service. There are two types of components: the observation management and the observation nodes. The observation management is responsible for the placement of the event observation and the event registration as indicated by the dark arrows, whereas the observation nodes are responsible for the actual observation. The event notifications that are sent are indicated by the lighter arrows.

The observation of events is realized through observation modules, which can be loaded at runtime

In the following section we look in detail at the observation nodes and observation modules. In Section 11.4 we discuss the observation management.

11.3 Observation Nodes & Observation Modules

Figure 27 shows the internal structure of an observation node. In the following we present the different components:

- **Registration Component**: The registration component provides the interface for the observation management component, or the client, if the client should directly contact an observation node. It accepts registration messages of the format defined in Section 10.3. In principle, the registration component may support different communication technologies. Currently only SOAP is supported.
- **Observation Module Manager**: When a new event is registered, it delegates the loading of the respective observation module to the observation module loader and then initializes the returned observation module. During the lifetime of the observation, it is responsible for the management of the observation modules and their deregistration; again, we employ a soft state approach, so the registration has to be renewed regularly.
- **Observation Module Loader**: The observation module loader loads the observation modules. The observation module loader first checks, if an observation module is available locally. If not, it has to be downloaded from a web server, i.e. by a Java class loader over HTTP. Of course there are security issues related to downloaded code. Therefore, the download can be restricted to trusted servers and only signed observation modules can be accepted.

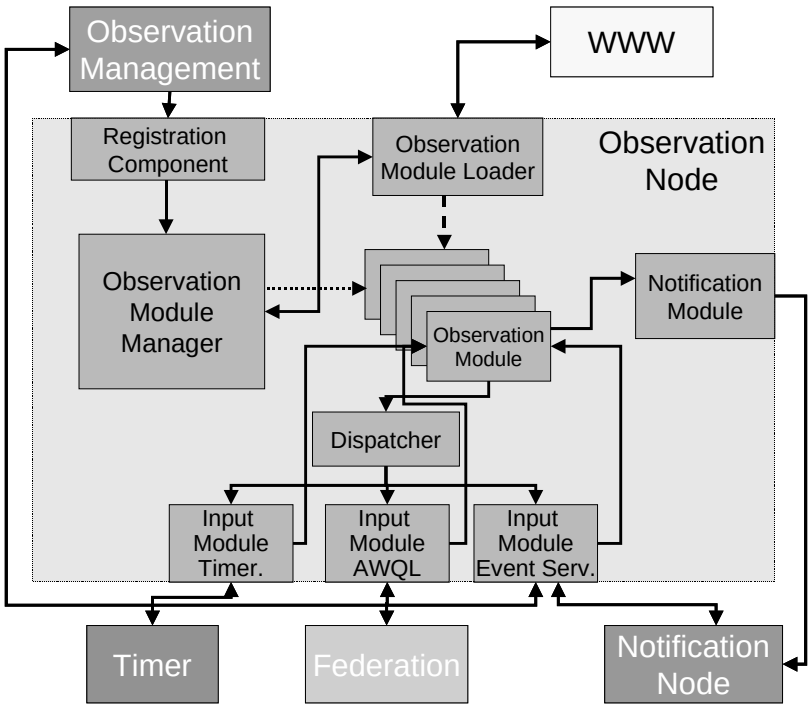


Figure 27: Internal Structure of Observation Node

- **Observation Module**: An observation module implements the evaluation of a predicate describing an event. At startup, it is initiated with the event parameters provided by the registration message (see Section 10.3). During the event observation, it receives event notifications that are stored in a queue. This information is then integrated into the internal state of the observation module. It is checked, if the resulting state change makes the predicate become true. If additional information is needed, an observation module can also query services, e.g. a Nexus spatial model server.
- **Input Modules**: Input modules provide an infrastructure for the observation modules. This means that observation modules do not have to implement their communication themselves. Instead they only communicate with a single component, the dispatcher.
- **Dispatcher**: Observation modules hand over registrations for sub-events, queries etc. to the dispatcher, that passes it on to the correct input module, so the observation module does not need to know about the input modules themselves.
- **Input Module "Event Service"**: The input module for the event service is used by the observation modules to register events and receive event notifications. The module passes on event registrations from the observer modules to the observer management and passes the event notifications from the notification service on to the observation modules.
- **Input Module "AWQL"**: The input module sends AWQL queries to a spatial model server/federation and returns the AWML answers to the requesting observation modules.
- **Input Module "Timer"**: The timer input module registers timer events and passes on event notifications when a timer event has occurred.

- **Notification Module**: When the occurrence of an event is detected, the observation module passes on a notification to the notification module, which hands it over to the notification service to deliver it to all the clients interested in the event.

Replication

The idea behind replication is that multiple observation modules on different observation nodes observe the same events. There is one active, primary observation module sending notifications and multiple passive observation modules that observe the event, but do not send any event notifications. Through regular synchronization with the primary observation module, it can be detected when the primary module has failed and another observation module can take over. The full replication strategy has not yet been implemented, but the mechanisms to do so are available.

Handovers

For the handover of an observation, the state of the observer module has to be transferred from one observation node to the other. A two phase handover protocol is needed to realize a safe handover. In the first phase, the observation on the new observation node is prepared, i.e. the module is initialized and the necessary sub-events are registered. In the second phase, the state of the old observation module is transferred. This requires that the observation module has to implement a method to extract its state and also a method to set the state in the new observation module. For the handover itself, there is a trade-off: either the observation is "frozen" during the handover, i.e. no event notifications are sent during the second handover phase, or both observation modules may observe the event in parallel for some time, in which case there is no freeze, but there may be duplicate event notifications, which are difficult to detect as they come from different observation nodes and may have slightly different timestamps.

A performance evaluation of the observation node can be found in Section 13.3. For more information about the observation service see [Boronas 2003].

11.4 Observation Management

The task of the observation management component is to accept event registration requests from client applications, find the event sources, optimally place the observation of global and composite events on an observation nodes and register the event and all sub-events.

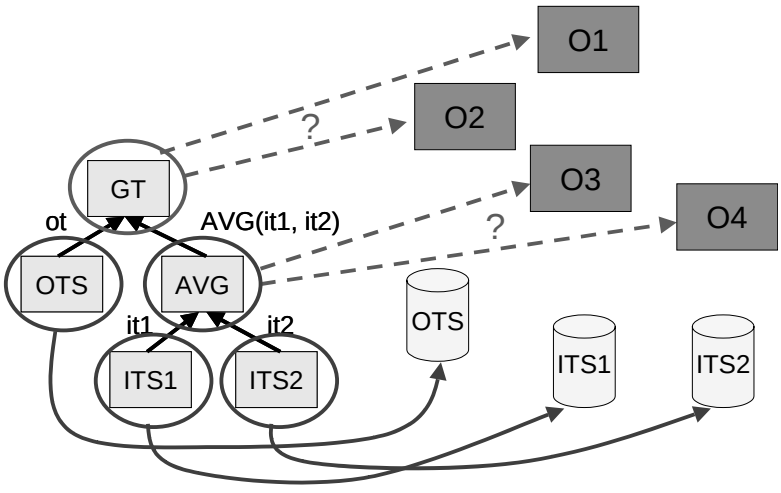


Figure 28: Observation Placement in the Observation Management

Figure 28 shows an example: "Inform me when the outdoor temperature is greater than the average indoor temperature." There are three sensor, one outdoors (OTS), two indoors (ITS 1) and (ITS2).

On the left side, the logical observer structure is shown. First the average of the two indoor sensors

has to be calculated (AVG), then it has to be compared to the value of the outdoor sensor (GT). The mapping to the event sources (OTS for the outdoor sensor, ITS1 and ITS2 for the indoor sensors) is clear, as only they have the necessary data. For the observation nodes, there is a choice, i.e. O3 or O4 for AVG, and O1 or O2 for GT. For this purpose a suitable observer placement strategy is needed (see Chapter 9). Finally, the registration of sub-events should be bottom up, because if a registration with an event source fails for some reason, the event cannot be observed, whereas if it fails for an observation node, alternatives can be tried.

To optimize the event observation, after receiving an event registration, it should first be checked, if the event or any sub-events are already being observed, so the observation module can be re-used for the new observation.

The finding of the respective event sources is system dependent. In Nexus we can distinguish between object-dependent data and location-dependent data. Location-dependent data can be found with the help of the Area Service Register, which given an area and the object type of interest returns the servers that have data for the given area and object type. Object-dependent data can be found using the ID, in case of stationary object, as the server that stores an object is encoded into the ID. For mobile objects, the location service has to be queried.

Based on the event sources, the event domains with the best characteristics that contain all the event sources have to be found. Within these event domains the optimal observation node has to be found.

The current implementation of the observation management is very rudimentary and does not support this functionality yet. So far, the existing observation nodes are hard-coded into the observation management. We also have not been able yet to evaluate the different observer placement strategies (see Chapter 9) due to time restrictions. A good observer placement strategy is a prerequisite for a good observation management. We plan to investigate these issues in the near future.

Chapter 12 Notification Service

The purpose of the notification service is to efficiently deliver event notifications to interested clients. As the notification service is an essential part of the event service, we had decided to develop two prototypes. The purpose of the first prototype, described in Section 12.1, was to get started with a simple version and gain experience. Based on that, a second prototype was developed as part of a diploma thesis [Till 2002]. It is described in Section 12.2. A .NET-based version of the second prototype was implemented in C# as part of a student thesis [Dieterle 2003]. The purpose was to evaluate the interoperability of the SOAP-based communication and to compare the performance of the different platforms. This implementation is described in Section 12.3. The evaluation of the performance and scalability of the event service can be found as part of the general evaluation in Chapter 13.

12.1 First Prototype

12.1.1 Requirements

The notification service is a fundamental part of the event service, because it is used by all other event service components. Therefore we began the project with a first implementation. Not all requirements were known at the beginning of the project. They only became clear in the course of the project. Some requirements like *Requirement 6: Scalability, Requirement 7: Fault Tolerance,* and *Requirement 8: Efficiency,* only played a secondary role here.

As we wanted to gain some experience with the web service technology that was establishing itself as a standard at that time, we decided to use the SOAP protocol for the communication. SOAP (which used to stand for Simple Object Access Protocol) is a light-weight XML-based protocol that defines the serialization of data and is mostly used over HTTP. Using standards that are available for all relevant platforms fits well with *Requirement 9: Interoperability*.

12.1.2 Design

As we have seen in Section 6.4, in some typical Nexus scenarios only few clients are interested in the observation of a certain event. Therefore, the components communicate directly with each other. Using multicast on an overlay network does not make sense in such a scenario, as the overhead for keeping the structure is not justified with regard to the efficiency that could be gained.

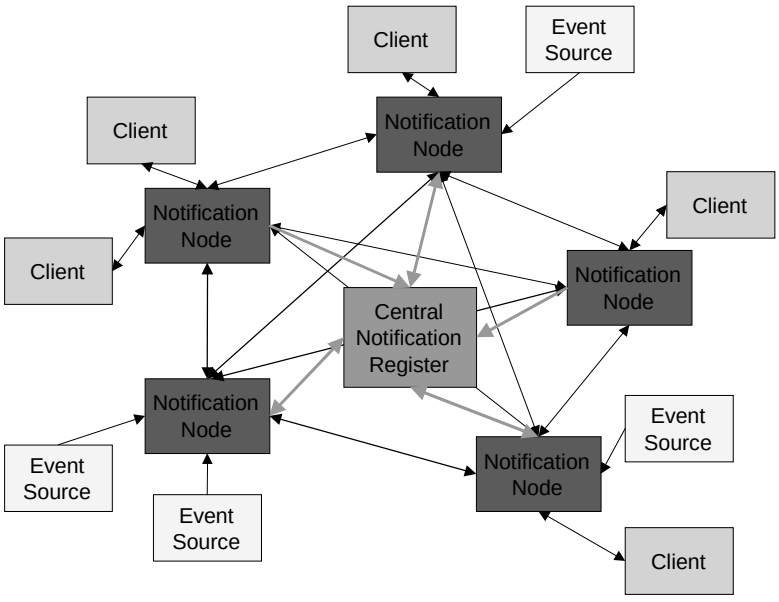


Figure 29: First Event Service Prototype

As the clients explicitly initiate the event observation, they know the ID of the events they are interested in. Therefore, an id-based notification mechanisms is sufficient. Content-based addressing would not help, because no similarity between different events can be exploited. For example, if somebody is interested in an onEn-

terArea event for his office, he will not automatically be interested in an onEnterArea event for the office of his colleague.

The communication interface was kept general, so other notification mechanisms could be added later.

Figure 29 shows the architecture of the first notification service prototype. The arrows indicate the directions in which the components communicate with each other. The notification service consists of a number of notification nodes and a central register. A client registers its interest in an event (based on the event ID) with its pre-configured notification node. The idea is that a notification node runs on each device, but a configuration with a distant notification node is also possible. The nodes in turn register their interest in the event with the central register (light arrows).

An event source passes an event notifications to its pre-configured notification node (dark arrows). The notification node queries the central register for the notification nodes that are registered for this event (light arrows) and directly send the event notification to each of them (dark arrows). The notification nodes then pass the event notification on to all registered clients (dark arrows).

12.1.3 Evaluation

The notification service prototype was implemented in Java based on Apache SOAP, the only SOAP implementation for Java that was available at that time. Apache SOAP is implemented as a servlet that was run inside the Apache Tomcat Servlet container. The performance of the first prototype was sufficient for small-scale examples, but the scalability of the underlying architecture is limited due to the centralized register.

A general problem with SOAP was that for the delivery of SOAP notifications the event clients have to act as servers with respect to the SOAP/HTTP connection. Therefore they have to run a web server, which is not always possible on mobile devices like PDAs due to their limited resources.

12.2 Second Prototype

After gaining some experience with the first notification service prototype and after the requirements for a notification service in the given environment became clearer, a new notification service was designed and prototypically implemented as part of a diploma thesis [Till 2002].

12.2.1 Requirements

The general requirements, especially the Requirement 6: Scalability, Requirement 7: Fault Tolerance, Requirement 8: Efficiency, Requirement 9: Interoperability, and Requirement 10: Mobility Support have been the focus of attention for the design of the second notification service prototype.

On a more detailed level, we have identified the following requirements:

- Scalability: In the context of the notification service, scalability means scalability to large networks, in the extreme case the whole Internet. It also means scalability with respect to a number of parameters that we have identified in Section 6.4, i.e. the number of clients, the number of event sources, the number of event types, the notification rate and the number of subscriptions.
- Number of clients registered for an event: The investigation of more scenarios led to the observation that in a large number of cases only a small number of clients is inter-

ested in a certain event. There are only a few scenarios with a higher number of interested clients, e.g. in the city scenario, see Subsection 6.4.3.

- Number of event sources per event: There are possibly multiple event sources for a single event, which has to be considered in the design of the notification service.
- Locality principle: Another important observation is that most events are only of interest locally.
- Configuration: Dynamic and automatic configuration and possibly reconfiguration is important for a fault tolerant system. It also simplifies the operation of the event service.
- Interoperabiltiy: As the notification service may be operated by different providers in a large-scale environment, the interoperability between different platforms is important.
- Mobile clients: The mobile clients have limited computing resources, limited battery power and wireless communication that is not as reliable as the cable-bound communication in the infrastructure. This has to be taken into account in the design of the notification service.

12.2.2 Design

As scalability and fault tolerance is required, the architecture may not include a single point of failure. So a central register as in our first prototype is not acceptable, which also means that we need a new design.

Figure 30 shows the new notification service architecture. Again the arrows indicate which components communicate with each other.

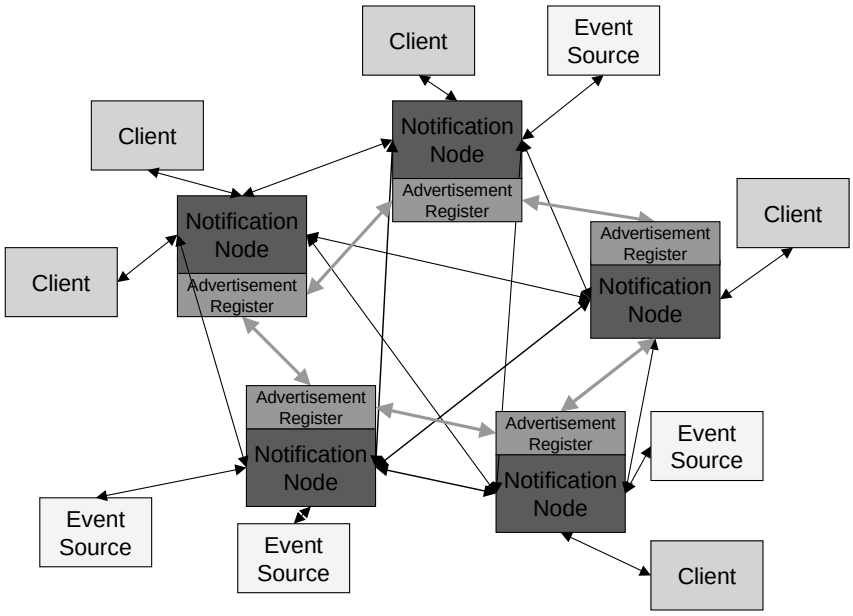


Figure 30: Second Notification Service Prototype

Notification Source

The Notification Source, i.e. event sources or observation nodes, use a notification node as connection point to the notification service. They create event notifications (see Section 10.2) and pass them on to the notification node. If they start observing an event they advertise this to the notification service and unadvertise it again, when they stop the observation.

Notification Client

Notification clients also use notification nodes as connection points. They register (or subscribe for) events with the notification node and receive event notifications.

Notification Node

Taken together, the notification nodes make up the notification service. A notification node provides the access point to the notification service for both the notification sources and the clients. It handles advertisements, subscriptions, and event notifications. The notification service has two registers, a subscription register that is local to each node (not shown in Figure 30), and an advertisement register that is distributed over all the notification nodes.

Subscription Register

The local subscription register manages subscriptions for local clients and subscriptions from other notification nodes for events that are observed by local notification sources. For the subscription register, there is no difference between the two kinds of subscriptions.

Advertisement Register

The advertisement register is distributed over the notification nodes. The advertisement register of each notification node manages its share of mappings from event IDs to notification sources that observe events with the given event ID. So, the advertisement register can answer which notification nodes have event sources that observe an event with a given event ID. The mapping of event ID to notification node and the implementation of the advertisement register is discussed in the next subsection.

Functionality

- Advertise: The notification source advertises an event ID to the notification node. The advertisement is passed on to the advertisement register on the responsible notification node that stores the mapping of event ID to event sources.
- Subscribe: The client subscribes for an event ID with the notification node. The notification node checks, if it has already subscribed for this event. If this is not the case, it queries the advertisement register with the given event ID. The advertisement registers returns a list of all the notification nodes that have notification sources observing the event with this event ID. Finally, the notification node registers with all notification nodes on the list. It also registers with the advertisement register to receive an event notification, if a new notification source for this particular ID becomes available.
- Notify: The notification source passes the event notification to its notification node. The notification node notifies all registered subscribers, both notification nodes and local clients. The notification nodes receiving the notification pass it on to their local clients.
- Startup: At startup, the notification node tries to find another notification node. If no notification node is preconfigured, it uses an expanding ring search to find the closest notification and integrates itself in the notification service.

Discussion of Design Decisions

- Structure: The problem with using a fixed hierarchical structure for delivering event notifications is the problem of failures, especially of the root node, which can become a single point of failure. Therefore, we have decided to use a peer-to-peer structure.

If a node fails, the other nodes are not affected and the share of the node's advertisement register can be taken over by other nodes. To avoid temporary problems with the advertisement information, it can be replicated on other nodes, see Subsection 12.2.3.

- The following subscription strategies exist:
  - Local: If subscriptions are only stored locally, the event notifications have to be broadcast, which results in poor scalability.
  - Distributed: In the distributed case, the subscriptions have to be broadcast and stored on each node. This results in poor scalability regarding subscriptions.
  - Hierarchical: The hierarchical subscription strategies requires a fixed hierarchical structure that we want to avoid.
  - Explicit advertisement of event observations: In this case, the subscription is only stored locally, whereas the advertisement information is available for the whole notification service. Event notifications are delivered directly to local clients and the notification nodes with local clients. Advertisements are stored in a distributed register. They are only needed to find the notification nodes that have event sources, so that notification nodes with clients can subscribe there.

  We have decided to take the last alternative. This requires another step, i.e. that the event sources advertise an event observation. Compared to the alternatives, this is still a much better solution, if we can efficiently realize a distributed register. This is shown in Subsection 12.2.3.

- Communication: We have decided to support different communication protocols. Therefore, the communication is encapsulated using interfaces. Thereby the use of communication mechanism is transparent to the notification service components and can be easily changed. Currently we support communication mechanisms using SOAP and plain TCP/IP. Other communication mechanisms can easily be added, e.g. for events with a large number of subscribers.
- Fault tolerance: Again, we have decided to implement a soft state approach, so state related to nodes that are permanently unreachable is eventually discarded. As already mentioned, replication of advertisement register entries can make the overall service more reliable.

The evaluation of the notification service performance can be found in Chapter 13. More information about the second notification service prototype can be found in [Till 2002].

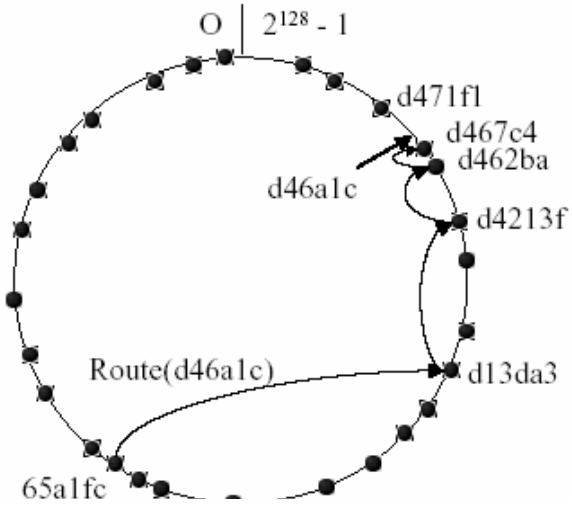


Figure 31: Routing in Pastry
Source: [Castro et al. 2002]

12.2.3 Distributed Advertisement Register based on Pastry

We have decided to implement the distributed advertisement register based on Pastry, a peer-to-peer system that was developed by Microsoft Research in Cambridge, in cooperation with Rice University, Purdue University, the University of Washington and Microsoft Research in Redmond. In contrast to Scribe [Rowstron et al. 2001], we have used Pastry only for implementing the advertisement register and not for the notification service itself, because we need to know the communication characteristics as specified for the event domains and this is not possible on an overlay network with a changing topology.

In Pastry, every node has a unique node ID. Messages with an ID can efficiently be routed to the node with the node ID that is closest to the message ID (see Figure 31 for an example). In addition to the closest node, the n nodes with the closest IDs can be addressed, which can be used to improve the fault tolerance of the system. Nodes stay in contact with their numerically closest neighbors, so they can react to node failures. A detailed description of Pastry can be found in [Rowstron & Druschel 2001].

The distributed advertisement register is realized as follows: The advertisement for an event with a given event ID is stored on the notification node with the node ID that is closest to the event ID. To increase the reliability, the advertisement can be stored at the n nodes with IDs closest to the event ID. If a node fails this is detected and the information can be replicated on the new n-closest node. If a new node is added to the system, the respective information can be transferred to the new node and deleted from the n+1-closest node.

12.3 .NET-based Implementation

We have reimplemented the notification service (second prototype) in C# based on the .NET platform as part of a student thesis [Dieterle 2003]. The goal was to test the interoperability between the different SOAP implementations, to compare the performance between the implementations and to get some experience with the .NET platform.

.NET platform

Visual Studio .NET provided a very nice environment for software development. Especially, the .NET support for web services was extremely helpful to get started.

Pastry

Unfortunately, we had to realize that no full implementation of Pastry for .NET was available, so the distributed register could not be realized on the .NET platform. Instead, we implemented a centralized version.

SOAP and XML Web Services in the Internet Information Server

With SOAP, we had the problem that, in order to receive SOAP calls, a component has to run within the Internet Information Server (IIS). As event notifications have to be delivered to clients using SOAP, the client has to run in IIS. The reason is that, with respect to the SOAP communication, it acts as a server. So, the notification client has to be an XML Web Service. Unlike their Java equivalents, XML Web Services under .NET cannot have user interfaces, which is of course inconvenient for client applications. So, in practice, another solution has to be found, e.g. a TCP/IP version could be implemented to avoid the problem. The use of IIS on mobile devices with limited resources is also not possible.

Limitations of IIS on Non-Server Editions of Windows

The notification service was tested on Windows XP Professional. IIS has a limitation of 10 open connections in parallel on non-server editions. This was just sufficient for the tests, but might represent a problem for notification nodes in any larger scenarios.

Interoperability

The .NET-based version was implemented in such a way that the interoperability between .NET-based and Java-based components was possible. Due to the incompatibilities of the advertisement register (which would require interoperating pastry implementations), it was only possible to test how .NET clients and .NET notification sources interoperate with Java-notification nodes. This test was successful, so the SOAP communication based on the two different SOAP implementations, Apache SOAP on the Java side and .NET, actually provides the interoperability that we expected.

The performance comparison between the two notification service implementations can be found in Chapter 13 together with the overall evaluation of the event service. More information about the .NET-based implementation of the notification service can be found in [Dieterle 2003].

Chapter 13 Evaluation

In this chapter we describe the integration of the event service into the Nexus platform and the evaluation of the different event service components and their interaction. The focus of the evaluation lies on the performance of the event service, especially regarding *Requirement 6: Scalability* and *Requirement 8: Efficiency*. The evaluation is based on the prototype implementation of the Nexus platform, primarily on the location service prototype.

In the first section the performance of the specialized event component of a location server is evaluated and then compared to the generic event component that allows the integration of plug-in triggers. In the second section we investigate the throughput and delay characteristics of the notification service, before the performance of the observation node and the observation of global events is evaluated in the third section. In the fourth section, we describe the conceptual integration of the event service components into the Nexus platform and the use of spatial events in Nexus applications. The fifth section summarizes the evaluation results.

13.1 Leaf Location Server As Event Source

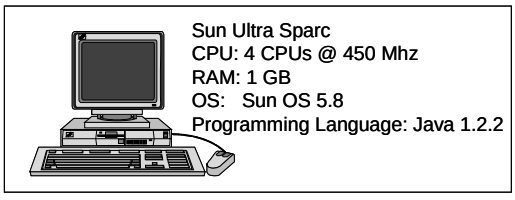
The primary event sources within the Nexus system are the location servers, or more precisely the leaf location servers, because only they store the position information of mobile objects. On the one hand the position information of mobile objects is highly dynamic, on the other hand location is the primary context information for a large class of context-aware applications. So the spatial events that can be observed in the location service provide the basis for evaluating our event system.

Even though the location service itself is not part of the project, the efficiency of its event component, especially the improvements achieved through the use of special index structures, is of general interest. Other event sources can profit from the experience we gained with a location server as an event source.

As discussed in Chapter 10, there are two implementations of the event component of the location server. The first was developed by Dominique Dudkowski as part of his student thesis [Dudkowski 2002]. It tightly integrates the event component with the location server and it has a specialized interface for registering certain basic events. The second is an event component developed by Daniel Minder as part of his diploma thesis [Minder 2003]. It has a generic interface that allows the integration of plug-in triggers. With this concept new types of events can be integrated at a later time. We first present a summary of Dominique Dudkowski's evaluation of his specialized event component and then compare it with the evaluation results from Daniel Minder's generic component.

13.1.1 Evaluation of Specialized Event Component

For the performance evaluation Dudkowski used the following computer [Dudkowski 2002]:



The performance experiments were run locally within a single Java Virtual Machine. Instead of the real notification service, a UDP-based one was used that provided only the functionality that was needed for evaluation purposes.

The parameter that measures the performance of an event source is the number of system events that can be processed per second. In the case of the location server the most important system event is the position update, because it is by far the most frequent system event.

The parameters that influence the performance are:

- Mobile Objects
  - Number
  - Step size, i.e. how far do they move between updates
- Registered User Events
  - Type
  - Number

For area-/location-based events:

- Size of the observation area

For timer-events:

- Report interval

For a number of special events, other parameters are also of importance. We discuss them together with the evaluation of that particular event.

An important issue for the observation of events is to reduce the number of user events that have to be checked for each system event. As discussed in Subsection 6.3.5 and in Section 10.1, index structures can be used to efficiently access those events that have to be checked.

For the events that were attached to an object, we used a hash table over the object id. For the events that were attached to a geographical area, we used a quad tree, a suitable two dimensional index structure (see Section 10.1). Each quad tree node could hold ten elements before it was split and four children were added to the tree.

We compared the results achieved with the index structure with the results achieved with a linked list containing all registered events. As all the elements of the list have to be checked every time, this corresponds to the worst case.

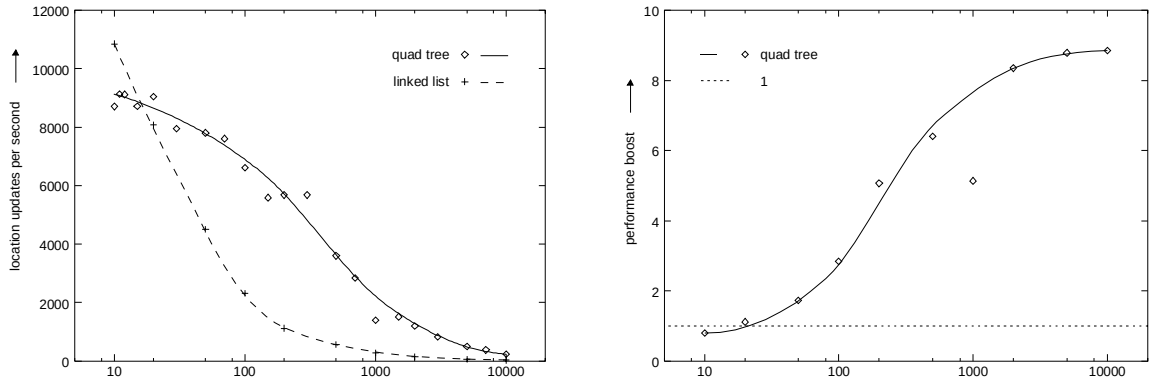


Figure 32: onEnterArea Performance with Standard Parameters
a. Performance Comparison, b. Performance Boost with Quad Tree Implementation
Source:[Dudkowski 2002]

onEnterArea

The onEnterArea event is attached to a specific area, so the quad tree index structure is used.

For the evaluation of the onEnterArea event the following default parameters were used:

- Number of mobile objects registered: 10,000
- Step size of mobile objects: 10m
- Size of observation area: 200m * 200m (= 40,000 m²)

As explained above, the performance is measured in number of position updates per second. Figure 32 shows the performance of onEnterArea with default parameter settings. With very few events registered, the linked list slightly outperforms the quad tree index. The performance boost achieved reaches a maximum factor of about 8.8.

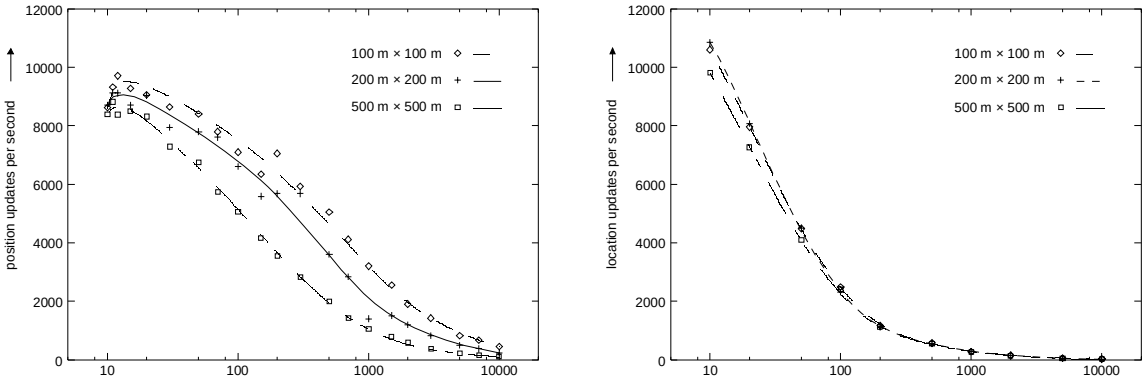


Figure 33: onEnterArea Performance for Different Area Sizes
a. Quad Tree, b. Linked List
Source:[Dudkowski 2002]

In Figure 33 the results for varying area sizes are shown for both the quad tree index structure and the linked list. The area sizes are:

- 100 m * 100 m (10,000 m²)
- 200 m * 200 m (40,000 m²)
- 500 m * 500 m (250,000 m²)

The remaining parameters are set to the default values.

For larger area sizes the performance of the quad tree index gets worse. This is due to the fact that larger areas have to be stored higher up in the quad tree and therefore have to be checked more often. For the linked list, the difference between the different area sizes is very small, because the traversal of the list has a high overhead compared to the evaluation of the event.

distPosUpdate

The distPosUpdate event is attached to a specific object, so the hash table index is used. The relevant parameters are the report distance and the mobile object step size. The following parameter settings were used for the experiments:

- Number of mobile objects registered: 10000
- Mobile object step size: 10m
- Report distance: 30m

As shown in Figure 34, the maximal performance boost achieved for the distPosUpdate event by the use of the hash table is about 20.5.

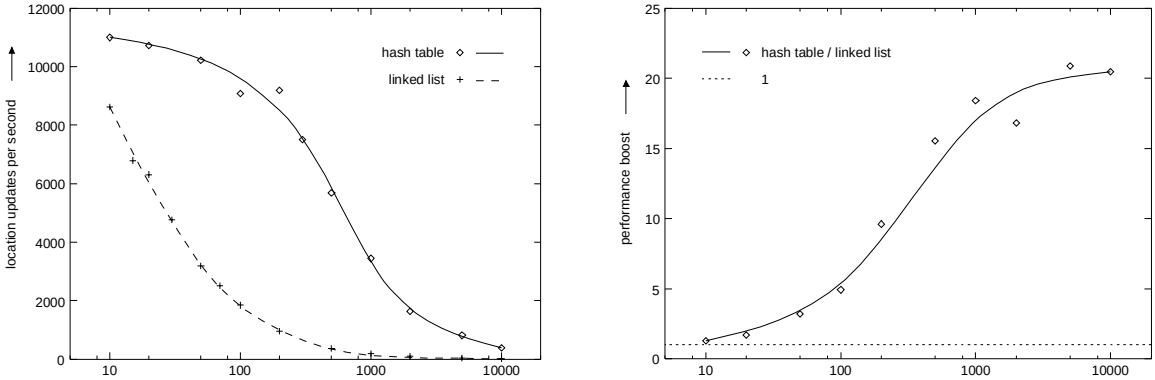


Figure 34: distPosUpdate Performance
a. Performance Comparison, b. Performance Boost with Hash Table Implementation
Source:[Dudkowski 2002]

contPosUpdate

The contPosUpdate is a timer-based event. Timer-based events are managed by a separate timer thread that runs in parallel to the main thread and competes with it for CPU time. The performance of timer-based events is given by the number of event notifications that can be sent within a certain time interval. The parameters of interest are the number of events registered and their report interval. With a longer report interval more events can be registered and vice versa.

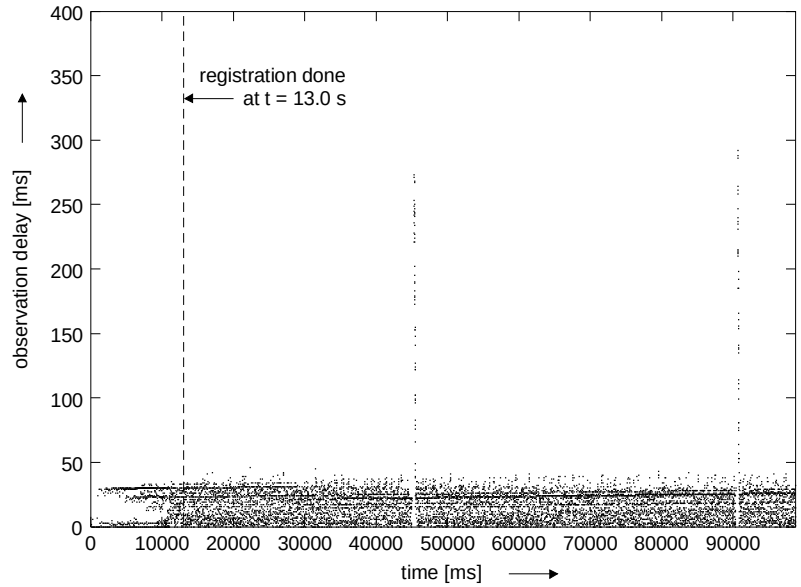


Figure 35: contPosUpdate with 100 Notifications/s

The observation delay, i.e. the time an event notification is “late” compared

to the expected notification time, tells us, if the timer thread can keep up with the processing of event notifications. If a large number of event notifications experience a high delay, we have an overload of the event component.

In Figure 35 the performance for the following parameter setting is shown:

- Number of position updates/s: 1000 (background load)
- Number of events : 100
- Report interval: 1s
- *Notifications/s: 100*

As we can see, the event component can easily handle the number of 100 event notifications per second, since the observation delay stays well below the 50 ms margin for the vast majority of cases.

In Figure 36 the performance for a second experiment with the following parameter setting is shown:

- Number of position updates/s: 1000 (background load)
- Number of events: 500
- Report interval: 2 s
- *Notifications/s: 250*

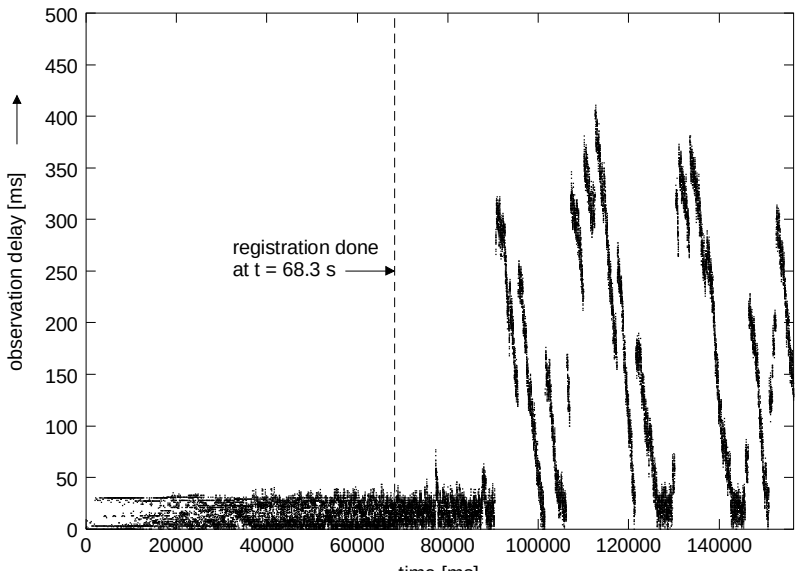


Figure 36: contPosUpdate with 250 Notifications/s

As we can see, the event component can no longer keep up with the sending of event notifications. After about 90 s delays of more than 300 ms appear. The reason may be that some other process has temporarily taken more CPU time, so the event component can no longer handle the 250 notifications per second. As discussed in Section 10.1, a mechanism for congestion avoidance was implemented that rejects new registrations, if the event component is at the limit of its capacity.

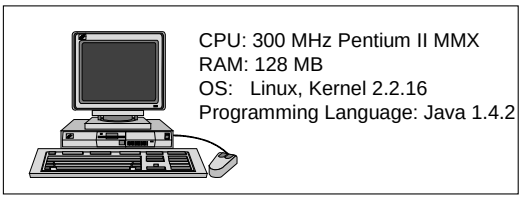
For more information concerning the evaluation of the specialized event component see [Dudkowski 2002].

Overall, we can see that the specialized event component of the location server can easily fulfill the requirements of simple Nexus scenarios like the office scenario that was described in Subsection 6.4.1. For larger scenarios multiple location servers are needed. In that case some of the spatial events, like onMeeting can no longer be observed locally and observation nodes are needed for the observation.

13.1.2 Evaluation of Generic Event Component with Plug-in Triggers

The evaluation of the generic event component with plug-in triggers, which Daniel Minder developed as part of his diploma thesis [Minder 2003], focuses on the comparison to Dudkowski's specialized event component.

The experiments were run on the following computer:



Again, the measurements were made with a pseudo notification service, but this time the regular notification service interface was used and again the performance was measured with respect to the number of position updates per second and the observation delay for the timer-based events.

The distPosUpdate events showed a very similar performance compared to the specialized event component. With only 10 registered events there is a noticeable performance loss due to duplicate elimination, which was not necessary in the specialized event component.

The onEnterArea events showed a performance loss of about 10% compared to the specialized event component when using a quad tree. This may again be due to the duplicate elimination.

Just as in the case of the specialized event component, a separate thread for timer-based events was used. When registering 100 contPosUpdates with a report interval of 1 s, 65% were of the event notifications were less than 10 ms late and 95% less than 25 ms. However, occasional peaks could be seen that were due to the notification service interface. Without the notification service interface, 99.2% of the event notifications were less than 20 ms late. This points to a strong influence of other components on the event observation, which has to be investigated further.

The delay encountered between the time an event was observed and the time the event notification was received by the notification service was also measured for distPosUpdates: In 97.4% of the cases, the delay was less than 3 ms. However there are regular peaks of a delay of about 30 ms, possibly due to a background thread, and in 0.08% of the cases, the delay was over 31 ms.

As distPosUpdate events are very simple, the experiment was repeated for an observation module with a more complex observation ("more than a certain number of objects in a given area"). Here, only in 82.7% of the cases, the delay was less than 3 ms, but in 3% of the cases, the delay was longer than 30 ms, so there is some dependence on the event type.

For more information concerning the evaluation of the generic event component see [Minder 2003].

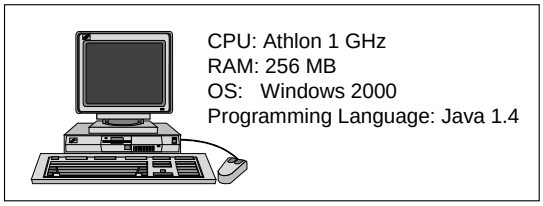
Overall, the generic event component provides similar performance compared to the specialized event component, but much more flexibility regarding the observation of new types of events, even at runtime.

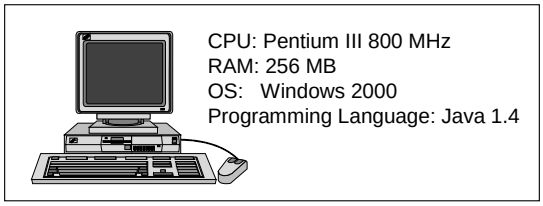
13.2 Notification Service

The most important requirement for the notification service is *Requirement 6: Scalability*. The peer-to-peer architecture of the notification service aims at scalability. As we have seen in Section 6.4, the events in most Nexus scenarios are only of interest for a relatively small number of clients, so if the event sources and clients are regularly distributed, the notification service should scale with the number of nodes; the nodes directly contact each other for subscribing for event notification and delivering event notifications and the overhead for the advertisement register is balanced over the nodes.

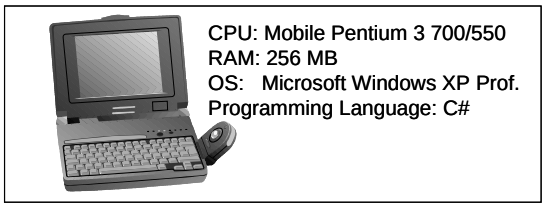
However, one limitation on the scalability of the notification service is the throughput of a single notification node. Therefore we have measured the maximum throughput of event notifications.

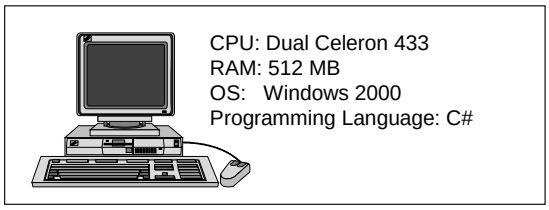
The tests were run on the following computers:





The tests for .NET were run on the following computers:





For the communication mechanism based on Apache SOAP a throughput of about 35 notifications per second was achieved. The values for SOAP on the .NET platform are similar, as a throughput of 24 notifications was achieved, albeit on a slower computer.

For the TCP/IP- based communication mechanism, a throughput of about 600 notifications per second was achieved.

Whereas the performance of the SOAP-based communication mechanism was somewhat disappointing, the TCP/IP-based mechanism provides a performance that is adequate for the Nexus scenarios in Section 6.4. We will further investigate the poor SOAP performance and switch to a faster implementation.

Another important aspect is the delay event notifications encounter in the notification service. Tests have shown that the TCP/IP-based communication mechanism can deal with one source sending permanently, which results in maximum delays of about 80 ms. With more event sources sending continuously in parallel, delays of more than 10,000 ms can be observed.

The SOAP-based communication mechanism can also deal with one event source sending permanently, which results in maximum delays of about 40 ms. With multiple event sources, delay values of more than 1000 ms are possible.

In case of the .NET-based SOAP implementation the maximum delay was also around 40 ms for a single source, but in general stayed under 100 ms even for multiple event sources. A notable exception is the first event notification delivery that can take as much as 3 seconds. This may be due to just-in-time compilation or instantiation overhead.

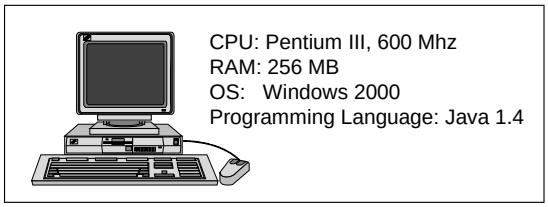
13.3 Observation Service

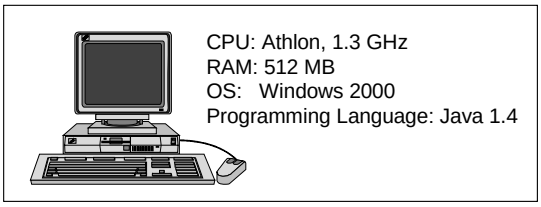
In this section we look at the performance of the observation nodes. The measurements are taken from [Boronas 2003], who developed the observation service, and [Csallner 2003], who developed observation modules for the observation of location service events in the distributed case.

13.3.1 Performance Measurements for a Single Observation Node

The parameters of interest are the throughput of the observation node for different observation modules and the delay. Regarding the delay we first look at the processing delay in the observation node itself and then at the overall delay from the position update to the notification about the occurrence of the event.

The following tests were conducted by Andreas Boronas with the following computers:





Maximum Throughput

The first test shows the number of event notifications a notification service can handle in the best case, i.e. when the respective method is called locally. The registered observation modules were *dummy modules* that did not do any processing. The number of modules registered was varied between 1 and 125 (see Figure 37).

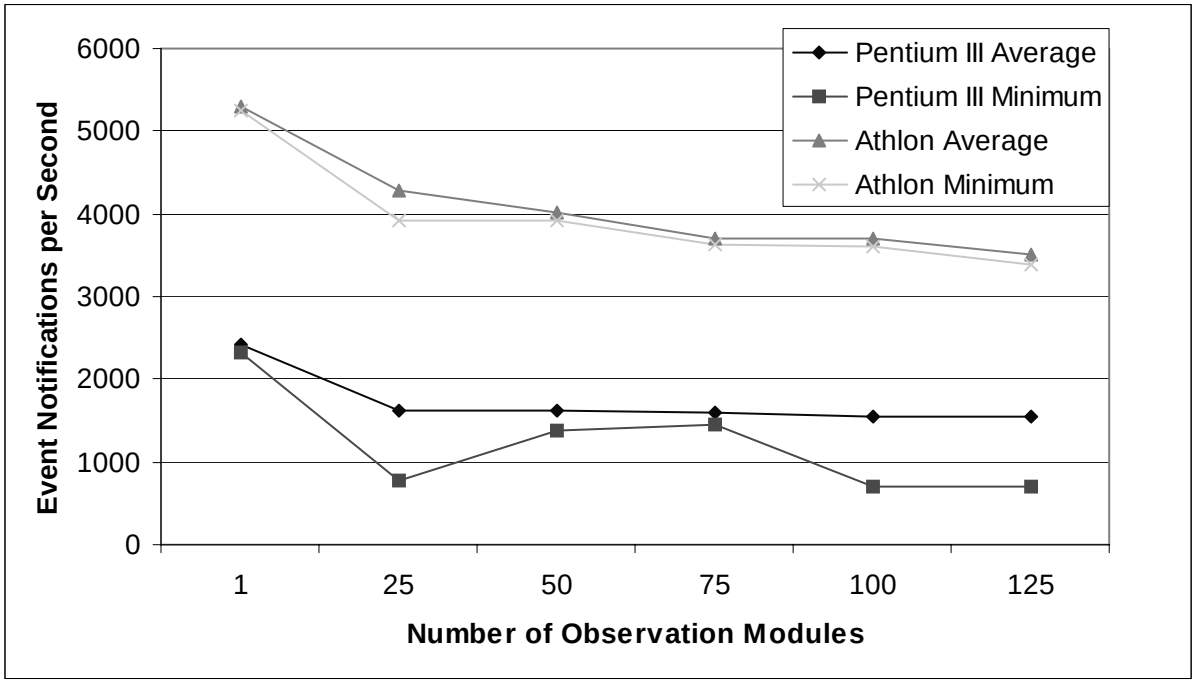


Figure 37: Event Notifications per Second
adapted from [Boronas 2003]

Table 5: Event Notifcations per Second [Boronas 2003]

	1	25	50	75	100	125
Pentium III Average	2414,53	1610,92	1630,97	1592,62	1544,62	1545,03
Pentium III Minimum	2325	765	1378	1458	710	711
Athlon Average	5292,92	4270,88	4007,48	3711,13	3710,38	3503,95
Athlon Minimum	5250	3924	3928	3639	3597	3391

As we can see in Table 5, for 125 registered modules on average 1500 event notifications can be processed on the Pentium 3 and 3500 on the Athlon (based on 10 measurements per value). Compared to the 600 event notifications the notification service can deliver per second, we do not have a bottleneck yet, even though the dummy modules do not represent typical observation modules. The comparison between the Pentium 3 and the Athlon shows that the performance scales with the clock rate of the processor.

Processing Time

The second test investigated the processing time for event notifications given a notification rate of 500 notifications per second. Again, dummy modules were used. All the following tests were carried out on the Pentium 3.

On average, the processing time was 1 ms, in 0.3% of the cases longer than 9 ms and in 0.03% longer than 100 ms with the maximum being about 250 ms (based on 2000 measurements).

The third test investigated the effects of how the registration of new events in parallel influenced the processing time. The results show that, if events were registered one at a time, there was a slightly higher rate of processing times of 100 ms than in the normal case, but no qualitative difference. With 20 registrations in parallel, times of more than 300 ms are possible. This suggests that the number of registrations that can occur in parallel should be restricted.

Event Registration

The fourth test looked at the time it takes to process an event registration and initialize the observation module. If the module was available locally, the measured registration time on the Pentium 3 was always under 500 ms. If the module has to be downloaded, the registration time depends on the network connection between the observation node and the web server.

Handover of Event Observations

In the fifth test, the time for a handover of an event observation was measured. The handover took place between two observation nodes on the Pentium 3 and the Athlon that were connected by a 100 MBit LAN. The handover took between 1200 and 1400 ms (based on ten measurements).

13.3.2 Event Observation for a Distributed Location Service

In his diploma thesis [Csallner 2003] Christoph Csallner investigated the observation of spatial events for a distributed location service configuration. He considered all the events that Dominique Dudkowski had implemented for a single leaf location server and implemented observation modules for the distributed case.

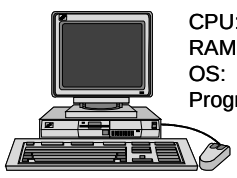
As a basis for the evaluation, two scenarios within a shopping mall were chosen (also see shopping mall scenario in Subsection 6.4.2): a bargain reminder and a friend finder service.

The bargain reminder allows customers to register events for certain special offers. When the customer passes the area where the product is offered, he receives a reminder. The service is realized based on the onEnterArea event. The event is registered for the area around the location where the product is offered. When the customer enters this area he receives an event notification reminding him about the bargain.

With the friend finder service, the user can register a number of friends with the service. If he comes within a specified meeting distance he receives a message. The service is realized based on the onMeeting event. The event is registered for two users and a distance. An event notification is sent, when the two users come closer than the specified distance.

The following experiments were run on the following computers:

CPU: 1,2 GHz Mobile Pentium 3
RAM: 512 MB
OS: Microsoft Windows XP
Programming Language: Java 1.4



CPU: 1 GHz Pentium 3
RAM: 256 MB
OS: Microsoft Windows 2000
Programming Language: Java 1.4

An object mover component simulated the movements of the mobile users. It sent position updates to the location service every 10 seconds using the UDP protocol of the location service.

Bargain Reminder - Evaluation of the Local User Event

As the onEnterArea events needed for implementing the bargain reminder are fixed to a given area, it is sufficient to observe an onEnterArea event on those leaf location servers whose service area overlaps with the given area. So the experiment shows the performance of the observation of a local user event.

The object mover updated the position information in such a way that for every second position update an onEnterArea event was observed and an event notification was sent.

This lead to a mean observation delay (from the movement of the object mover over the position update at the leaf location server to the detection of the event in the event component of the leaf location server) between 13 ms and 18 ms. The measurements were based on three series of 100 position updates each.

Assuming that there is no parallelism within the system, i.e. from the updating of the position by the object mover to the detection of the event occurrence, the leaf location service can, in the worst case, handle a maximum of 55 system events (position updates) resulting in 27 onEnterArea events per second. Taking an update rate of one update every 10 seconds, the leaf location server can manage a maximum of 270 mobile objects. The capacity of the location service can be enhanced by dividing up the service area and adding more leaf location servers.

Friend Finder - Distributed User Event

The observation of the onMeeting event needed for implementing the friend finder cannot be restricted to a single leaf location service. As an example take two mobile objects that are within meeting distance, but on both sides of a border between adjacent service areas of two leaf location servers. Therefore, we need an observation module that implements the distributed observation on an observation node. The onMeeting event is based on distPosUpdates for each mobile object that can cause the onMeeting event.

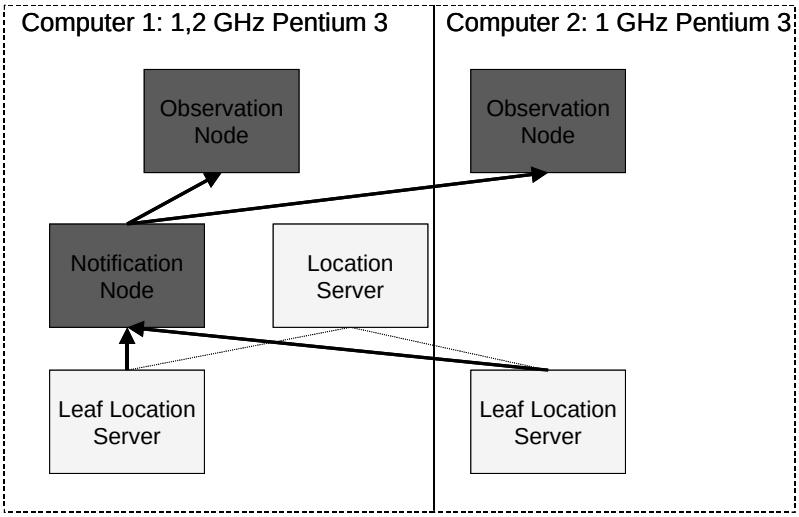


Figure 38: Configuration of the Friend Finder Experiment with Two Observation Nodes

We compare two different scenarios: In the first we have a single observation node on one computer, in the second we have two observation nodes on two computers (see Figure 38).

Again, the object mover updates the position information every 10 seconds. Every position update system event causes a distPosUpdate user event. Every second distPosUpdate user event causes an onMeeting event.

This time the mean observation delay is between 39 and 43 milliseconds for the single observation node and between 55 and 71 milliseconds for two notification nodes. The result is somewhat surprising. A possible explanation is that the two computers were different with regard to their computational power. Since the position updates were sent as fast as possible, the faster node flooded the slower node with event notifications and then had to wait for input from the slower node.

In the first case we have about 23 system events per seconds, leading to 11 onMeeting events that can be observed per second. Under the same assumptions as above, the service can handle 110 mobile objects. In the second case, we get about 14 system events, or 7 onMeeting events per observation node, so we have a total of 14 onMeeting events.

Further experiments have to be carried out to investigate the reason that we did not get the increase in capacity that we expected. However, we are confident that this is mostly due to the setup of the experiment.

13.4 Integration in Nexus

13.4.1 Conceptual Integration

Figure 39 shows the conceptual integration of the event service in Nexus. In the case of queries, the federation layer hides the distribution of the data from the applications and provides them with a homogeneous view (also see Figure 4). The observation management components and the observation nodes support the same goal. When registering an event observation the applications do not need to know about the distribution of the data and the existence of multiple event sources. They register their request with the observation management that, like a federation node, queries the area service register for the servers on which events have to be registered or which have to be queried (e.g. in the case of spatial model servers) to observe an event. Then the observation management registers local events with the event sources and global events with the observation node. For the observation, the observation node plays the role of the federation component as it gathers the data that is needed for the observation of an event, and when an event is detected, it sends a notification to the client application.

The notification nodes cannot easily be assigned to any of the Nexus layers, as they are responsible for the communication and the components responsible for the communication are not explicitly considered in the architecture.

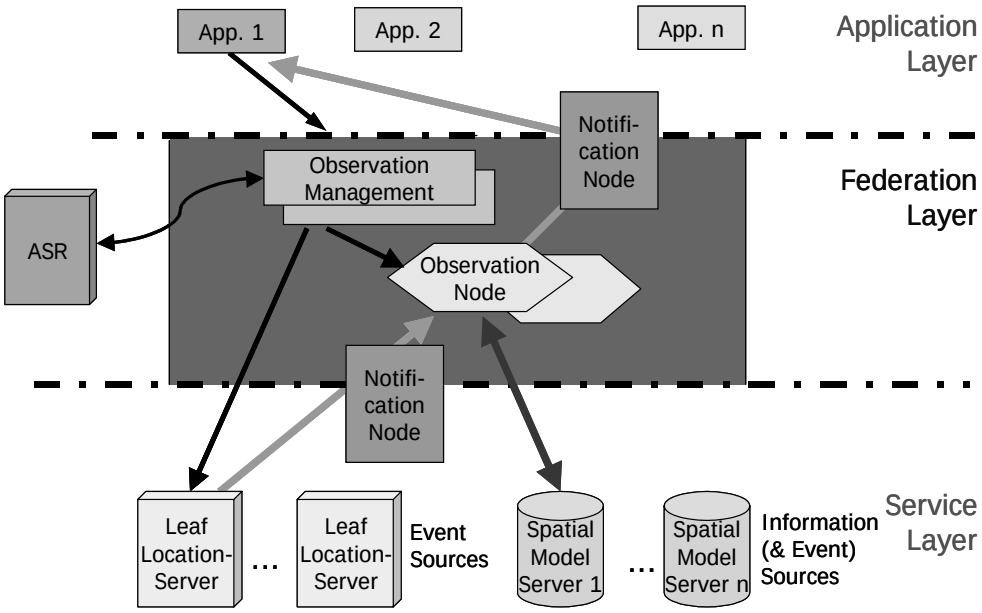


Figure 39: Conceptual Integration of the Event Service in Nexus

13.4.2 Use of Spatial Events in Nexus Applications

Spatial events have been used in a number of example applications that have been developed based on the Nexus platform:

- The Nexus Scout is a city information system that shows the functionality of the Nexus platform. With the Nexus Scout, the user can register spatial reminders based on the onEnterArea event, i.e. he is reminded that he wanted to do something at a certain location, when he gets there. He also can register reminders for the meeting with other users [Nicklas et al. 2003].
- The NexusRallye is a kind of game application where users walk around in a certain area and have to solve tasks. The tasks are associated with locations, so based on an onEnterArea event, a new task is presented to the users.
- The Intelligent Door Plate application tells people where the person they are looking for currently is, if they are not in their office, by showing the current location on the door plate of the office. The location of the user is tracked based on distPosUpdate and onEnterArea events.

13.5 Summary

Overall, we can see that our implementation of the event service is efficient enough to handle scenarios of the scale of the office scenario (see Subsection 6.4.1) and the shopping mall scenario (see Subsection 6.4.2), where the number of events observed on distributed data is limited. We still need to evaluate the event service for use in larger scenarios, especially the scalability of the distributed observation has to be further investigated.

Chapter 14 Conclusion & Outlook

The goal of this project was to develop a scalable event service for the area of mobile and ubiquitous computing.

The events of interest in this domain are events that occur in the real world. As there are a huge number of events that are of potential interest, the observation of these events has to be initiated explicitly. The data based on which the events can be observed can be distributed over different local models stored by different servers. Therefore, global events have to be observed through a distributed model and the observation of these events has to be realized as part of the event service. As this together with the explicit observation of events is not part of the standard event paradigm, we proposed a new event paradigm called event observation & notification paradigm. An event service that fits this paradigm has two different functional components – for the observation of events and the delivery of event notifications.

We developed an event model that provides a general classification of events according to parameters that are relevant for the observation of events. We classified spatial events that are relevant in the context of the Nexus project according to this general classification. We also investigated how spatial events are utilized in a number of typical Nexus scenarios, which was important for the design of a suitable event notification mechanism.

A special focus of this work was on the observation of global events through a distributed model that takes the limited accuracy of the available data into account. We proposed a uniform approach for specifying events based on a predicate that defines the occurrence of events in the exact case, together with a threshold probability that defines the minimum probability with which the event is considered to have occurred, so that an event notification is sent. We showed how different parameters influence the accuracy of the model and how the probability with which an event has occurred can be calculated.

We designed and implemented an event service prototype based on which we evaluated our approach in the context of the Nexus project. We could show that the approach is feasible and that our requirements could be fulfilled. Therefore, this project has been completed successfully.

To show the scalability of the approach, we still need to conduct a final evaluation of the event service for a large scale scenario. So far this has not been possible, partly due to technical problems outside the scope of our project, e.g. with the Nexus location service. However this evaluation is next on our agenda.

For the observer placement we have identified the relevant parameters, but, due to time limitations, different strategies for the observer placement still have to be evaluated. The dynamic reconfiguration of observation, e.g. after handovers that are due to the movement of mobile objects, also remains as future work.

Storing position information of people immediately raises privacy concerns. The support of spatial events does not improve the situation. Therefore a real system needs strict access control mechanisms to ensure that only authorized people can access position data with a certain accuracy and only those people will be able to register spatial events based on this data. It may also be possible to provide anonymized position data with a more limited accuracy. Obviously, there will be a trade-off between the level of privacy protection and the available functionality. In a diploma thesis that will start soon, we will investigate suitable access control mechanisms for the event service.

Appendix: Event Notification

XML-Schema

```
    <?xml version="1.0" encoding="UTF-8"?>

    <xsd:schema targetNamespace="http://nexus.informatik.uni-stuttgart.de/Notification"
xmlns:nol="http://nexus.informatik.uni-stuttgart.de/NOL"
xmlns:notify="http://nexus.informatik.uni-stuttgart.de/Notification"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">


        <xsd:import namespace="http://nexus.informatik.uni-stuttgart.de/NOL"
schemaLocation="http://nexus.informatik.uni-stuttgart.de/NOL.xsd"/>

        <xsd:annotation>

                <xsd:documentation xml:lang="en">

                Schema defining event notifications for the Nexus

                Event Service.

        </xsd:documentation>

        </xsd:annotation>

        <xsd:simpleType name="NOL">

                <xsd:restriction base="xsd:string">

                        <xsd:pattern value="nexus://(.*(:.*)?@)?.*(:[1-9][0-9]{0,3})?/(.*|0x([0-
9]|[A-F]|[a-f]){32}|\.home)/(.*|0x([0-9]|[A-F]|[a-f]){32})"/>

                </xsd:restriction>

        </xsd:simpleType>

        <xsd:element name="notification" type="notify:NotificationType"/>

        <xsd:complexType name="NotificationType">

                <xsd:sequence>

                        <xsd:element name="id" type="notify:NOL"/>

                        <xsd:element name="predicateId" type="notify:NOL"/>

                        <xsd:element name="templateId" type="notify:NOL"/>

                        <xsd:element name="name" type="xsd:string"/>

                        <xsd:element name="service" type="xsd:string"/>

                        <xsd:element name="server" type="xsd:string"/>

                        <xsd:element name="counter" type="xsd:unsignedLong"/>

                        <xsd:element name="scope" type="xsd:dateTime"/>

                        <xsd:element name="timestamp" type="xsd:dateTime"/>

                        <xsd:element name="comment" type="xsd:string" minOccurs="0"/>

                        <xsd:element name="variableList" type="notify:VariableListType"/>

                </xsd:sequence>

        </xsd:complexType>

        <xsd:complexType name="VariableListType">

                <xsd:sequence>

                        <xsd:element name="variable" minOccurs="0" maxOccurs="unbounded">

                                <xsd:complexType>

                                        <xsd:sequence>

                                                <xsd:element name="name" type="xsd:string"/>

                                                <xsd:element name="type" type="xsd:string"/>

                                                <xsd:element name="restriction" type="xsd:string"
minOccurs="0"/>

                                                <xsd:element name="value" type="xsd:string"/>
```

```
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:schema>
```

Example for onMeeting Event

```
<?xml version="1.0"?>
<notification xmlns="http://www.nexus.uni-stuttgart.de/Notification">
<id>nexus://nexus.uni-stuttgart.de/0x8cc8775ba13a3440ae9ce1c2b6cec756/0x8cc8775ba13a3440ae9ce1c2b6cec756</id>
  <predicateId>nexus://nexus.uni-stuttgart.de/onMeeting/0x30b9a4e2d6256d30d663aa828ff5d5b5</predicateId>
  <templateId>nexus://nexus.uni-stuttgart.de/onMeeting/0x00000000000000000000000000000000</templateId>
  <name>onMeetingEvent</name>
  <service>Location Service</service>
  <server>127.0.0.1</server>
  <counter>29</counter>
  <scope>2002-01-18T01:31:14.586+00:00</scope>
  <timestamp>2002-01-17T23:31:14.586+00:00</timestamp>
  <variableList>
    <variable>
      <name>Meeting Object</name>
      <type>Mobile Object</type>
      <restriction>Mobile Object</restriction>
      <value>00000000ea6000000130000000000</value>
    </variable>
    <variable>
      <name>Meeting Object</name>
      <type>Mobile Object</type>
      <restriction>Mobile Object</restriction>
      <value>00000000ea6000000b610000000000</value>
    </variable>
  </variableList>
</notification>
```

Example for contAreaUpdate Event

```
<?xml version="1.0"?>
<notification xmlns="http://www.nexus.uni-stuttgart.de/Notification">
<id>nexus://nexus.uni-stuttgart.de/0x0cbb2aa5b8fd6e3c718a11d2b56986b0/0x0cbb2aa5b8fd6e3c718a11d2b56986b0</id>
  <predicateId>nexus://nexus.uni-stuttgart.de/contAreaUpdatesEvent/0x613e20bccbc8844cdb9ce28983b4f9f0</predicateId>
  <templateId>nexus://nexus.uni-stuttgart.de/contAreaUpdatesEvent/0x00000000000000000000000000000000</templateId>
  <name>contAreaUpdatesEvent</name>
```

```
<service>Location Service</service>
<server>127.0.0.1</server>
<counter>3</counter>
<scope>2002-01-17T15:14:54.070+00:00</scope>
<timestamp>2002-01-17T13:14:54.051+00:00</timestamp>
<variableList>
  <variable>
    <name>Tracked Space</name>
    <type>Nexus Object</type>
    <value><SEGMENT WGS84: 48.722, 9.125, 0.0; WGS84: 48.72173, 9.126192, 0.0></value>
  </variable>
  <variable>
    <name>Object</name>
    <type>Mobile Object</type>
    <restriction>Mobile Object</restriction>
    <value>00000000ea60000000130000000000</value>
  </variable>
  <variable>
    <name>Location</name>
    <type>Geodetic Coordinate</type>
    <value>WGS84: 48.72199594, 9.12517916, 0.0</value>
  </variable>
  <variable>
    <name>Location Timestamp</name>
    <type>Datetime</type>
    <value>2002-01-17T13:14:53.169+00:00</value>
  </variable>
  <variable>
    <name>Object</name>
    <type>Mobile Object</type>
    <restriction>Mobile Object</restriction>
    <value>00000000ea600000001b0000000000</value>
  </variable>
  <variable>
    <name>Location</name>
    <type>Geodetic Coordinate</type>
    <value>WGS84: 48.72194515, 9.12509107, 0.0</value>
  </variable>
  <variable>
    <name>Location Timestamp</name>
    <type>Datetime</type>
    <value>2002-01-17T13:14:53.179+00:00</value>
  </variable>

  ... 57 more mobile objects ...

  <variable>
```

```
      <name>Object</name>
      <type>Mobile Object</type>
      <restriction>Mobile Object</restriction>
      <value>00000000ea60000000050000000000</value>
    </variable>
    <variable>
      <name>Location</name>
      <type>Geodetic Coordinate</type>
      <value>WGS84: 48.72176165, 9.12602461, 0.0</value>
    </variable>
    <variable>
      <name>Location Timestamp</name>
      <type>Datetime</type>
      <value>2002-01-17T13:14:52.168+00:00</value>
    </variable>
  </variableList>
</notification>
```

Appendix: Event Registration Language

XML-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.4 U (http://www.xmlspy.com) by Martin Bauer (IPVR-Iniversität Stuttgart) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xs:element name="predicate" type="PredicateType">
		<xs:annotation>
			<xs:documentation>This is the root element of the Predicate Definition Language.</xs:documentation>
		</xs:annotation>
	</xs:element>
	<xs:complexType name="PredicateType">
		<xs:sequence>
			<xs:element name="templateID" type="NOLType"/>
			<xs:element name="name" type="xs:string" minOccurs="0"/>
			<xs:element name="observerURI" type="xs:anyURI" minOccurs="0"/>
			<xs:element name="predicateId" type="NOLType" minOccurs="0"/>
			<xs:element name="thresholdProbability" type="xs:float"/>
			<xs:element name="parameterList" type="ParameterListType"/>
			<xs:element name="filterList" type="FilterListType" minOccurs="0"/>
			<xs:element name="qosParameterList" type="QosParameterListType"/>
			<xs:element name="predicateManagementParameterList" type="PredicateManagementParameterListType"/>
			<xs:element name="predicateEvaluationParameterList" type="PredicateEvaluationParameterListType" minOccurs="0"/>
			<xs:element name="eventNotificationParameterList" type="EventNotificationParameterListType" minOccurs="0"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="ParameterListType">
		<xs:sequence>
			<xs:element name="Parameter" type="ParameterType" maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="ParameterType">
		<xs:sequence>
			<xs:element name="name" type="xs:string"/>
			<xs:element name="type" type="xs:string"/>
			<xs:element name="restriction" type="xs:string" minOccurs="0"/>
			<xs:choice>
				<xs:element name="value">
					<xs:complexType>
						<xs:complexContent>
							<xs:extension base="xs:anyType">
```

```
                                                    <xs:attribute name="unit"
                                                    type="xs:string" use="optional"/>
                                            </xs:extension>
                                    </xs:complexContent>
                            </xs:complexType>
                    </xs:element>
                    <xs:element name="predicate" type="PredicateType"/>
            </xs:choice>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="ValueType">
    <xs:choice/>
</xs:complexType>
<xs:complexType name="FilterListType">
    <xs:sequence>
        <xs:element name="filter" type="FilterType" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="FilterType">
    <xs:sequence>
        <xs:element name="attributeReference1">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs:string">
                        <xs:attribute name="unit" type="xs:string"
                        use="optional"/>
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
        <xs:element name="compareFunction" type="CompareType"/>
        <xs:element name="attributeReference2">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs:string">
                        <xs:attribute name="unit"
                        type="xs:string" use="optional"/>
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:simpleType name="CompareType">
    <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:complexType name="QosParameterListType">
    <xs:sequence>
```

```
                <xs:element name="qosParameter" type="QoSParameterType"
                maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="QoSParameterType">
        <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="type" type="xs:string"/>
                <xs:element name="value">
                        <xs:complexType>
                                <xs:complexContent>
                                        <xs:extension base="xs:anyType">
                                                <xs:attribute name="unit"
                                        type="xs:string" use="optional"/>
                                        </xs:extension>
                                </xs:complexContent>
                        </xs:complexType>
                </xs:element>
        </xs:sequence>
</xs:complexType>
<xs:simpleType name="ClockDriftValueType">
        <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="DelayValueType">
        <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:complexType name="PredicateManagementParameterListType">
        <xs:sequence>
                <xs:element name="predicateManagementParameter"
                type="PredicateManagementParameterType" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
<xs:simpleType name="HandoverType">
        <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="DeregistrationIntervalType">
        <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:complexType name="PredicateManagementParameterType">
        <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="type" type="xs:string"/>
                <xs:element name="value">
                        <xs:complexType>
                                <xs:complexContent>
                                        <xs:extension base="xs:anyType">
                                                <xs:attribute name="unit"
                                        type="xs:string" use="optional"/>
                                        </xs:extension>
```

```
                                </xs:complexContent>
                        </xs:complexType>
                </xs:element>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="PredicateEvaluationParameterListType">
        <xs:sequence>
                    <xs:element name="predicateEvaluationParameter"
                type="PredicateEvaluationParameterType" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
<xs:simpleType name="notificationConsumptionPolicyType">
        <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:complexType name="PredicateEvaluationParameterType">
        <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="type" type="xs:string"/>
                <xs:element name="value">
                        <xs:complexType>
                                <xs:complexContent>
                                        <xs:extension base="xs:anyType">
                                                    <xs:attribute name="unit"
                                                type="xs:string" use="optional"/>
                                        </xs:extension>
                                </xs:complexContent>
                        </xs:complexType>
                </xs:element>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="EventNotificationParameterListType">
        <xs:sequence>
                <xs:element name="eventNotificationParameter"
type="EventNotificationParameterType" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="EventNotificationParameterType">
        <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="type" type="xs:string"/>
                <xs:element name="value">
                        <xs:complexType>
                                <xs:complexContent>
                                        <xs:extension base="xs:anyType">
                                                    <xs:attribute name="unit"
                                                type="xs:string" use="optional"/>
                                        </xs:extension>
                                </xs:complexContent>
                        </xs:complexType>
```

```
                </xs:element>
            </xs:sequence>
        </xs:complexType>
        <xs:simpleType name="NOLType">
            <xs:restriction base="xs:string">
                <xs:pattern value="nexus://(.*(:.*)?@)?.*(:[1-9][0-9]{0,3})?/(.*|0x([0-
9]|[A-F]|[a-f]){32}|\.home)/(.*|0x([0-9]|[A-F]|[a-f]){32})"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:schema>
```

Example for onEnterArea Event

```
    <?xml version="1.0" encoding="UTF-8"?>
    <predicate xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="G:\Nexus\XML Schemas\PredicateDefinitionLanguage.xsd">
        <templateID>nexus://nexus.stuttgart.de/0x84a6fe8cc8b4e101d62ff005ecb97a88/
0x84a6fe1238000000000000000000000000</templateID>
        <name>onEnterArea</name>
        <observerURI>http://trompete.informatik.uni-stuttgart.de:8081/soap/servlet/rpcrouter
urn:registerOnEnterArea</observerURI>
        <thresholdProbability>0.7</thresholdProbability>
        <parameterList>
            <Parameter>
                <name>Entering Object</name>
                <type>Nexus Object Reference</type>
                <restriction>Mobile Object</restriction>
                <value>nexus://nexus.stuttgart.de/.home/Timo%20Heiber</value>
            </Parameter>
            <Parameter>
                <name>Entered Space</name>
                <type>Nexus Object Reference</type>
                <value>nexus://nexus.uni-
stuttgart.de/InformatikFakultaet/Room2.069</value>
            </Parameter>
        </parameterList>
        <qosParameterList>
            <qosParameter>
                <name>clock drift</name>
                <type>real</type>
                <value unit="milliseconds">100.0</value>
            </qosParameter>
            <qosParameter>
                <name>maximum delay</name>
                <type>real</type>
                <value unit="milliseconds">100.0</value>
            </qosParameter>
            <qosParameter>
                <name>message loss</name>
```

```
                <type>real</type>
                <value unit="messages/second">0.00001</value>
            </qosParameter>
        </qosParameterList>
        <predicateManagementParameterList>
            <predicateManagementParameter>
                <name>handover</name>
                <type>String</type>
                <value>allowed</value>
            </predicateManagementParameter>
            <predicateManagementParameter>
                <name>deregistration interval</name>
                <type>String</type>
                <value unit="seconds">10000</value>
            </predicateManagementParameter>
            <predicateManagementParameter>
                <name>maximum notification rate</name>
                <type>String</type>
                <value unit="messages/second">0.1</value>
            </predicateManagementParameter>
        </predicateManagementParameterList>
        <predicateEvaluationParameterList>
            <predicateEvaluationParameter>
                <name>blocking interval</name>
                <type>real</type>
                <value unit="seconds">10</value>
            </predicateEvaluationParameter>
            <predicateEvaluationParameter>
                <name>use estimated occurrence time</name>
                <type>boolean</type>
                <value>true</value>
            </predicateEvaluationParameter>
        </predicateEvaluationParameterList>
    </predicate>
```

References

Project-Related Publications

[Bauer et al. 2001]
Bauer, Martin; Becker, Christian; Rothermel, Kurt: Location Models from the Perspective of Context-Aware Applications and Mobile Ad Hoc Networks. In: Workshop on Location Modeling for Ubiquitous Computing, UbiComp 2001

[Bauer et al. 2002]
Bauer, Martin; Becker, Christian; Rothermel, Kurt: Location Models from the Perspective of Context-Aware Applications Mobile Ad Hoc Networks. In: Personal and Ubiquitous Computing. Vol. 6(5-6)

[Bauer et al. 2003]
Bauer, Martin; Becker, Christian; Hähner, Jörg; Schiele, Gregor: ContextCube - Providing Context Information Ubiquitously. In: Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003)

[Bauer et al. 2003a]
Bauer, Martin; Jendoubi, Lamine; Rothermel, Kurt; Westkämper, Engelbert: Grundlagen ubiquitärer Systeme und deren Anwendung in der "Smart Factory". In: Gronau, Norbert (ed.); Krallmann, Hermann; Scholz-Reiter, Bernd (ed.): Industrie Management - Zeitschrift für industrielle Geschäftsprozesse. Bd. 19(6)

[Bauer & Rothermel 2002]
Bauer, Martin; Rothermel, Kurt: Towards the Observation of Spatial Events in Distributed Location-Aware Systems. In: Wagner, Roland (ed.): Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops

[Becker et al. 2002]
Becker, Christian; Bauer, Martin; Hähner, Jörg: Usenet-on-the-fly - Supporting Locality of Information in Spontaneous Networking Entironments. In: Liscano, Ramiro (ed.); Kortuem, Gerd (ed.): Workshop on Ad Hoc Communications and Collaboration in Ubiquitous Computing Envirnments

[Lehmann et al. 2004]
Lehmann, Othmar; Bauer, Martin; Becker, Christian; Nicklas, Daniela: From Home to World - Supporting Context-aware Applications through World Models. In: to appear: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications

[Leonhardi & Bauer 2001]
Leonhardi, Alexander; Bauer, Martin: “Managing Highly Dynamic Location Data”; “Infobox: Spatial Events”, In: GIS Geo-Informations-Systeme. Vol. 6

[Rothermel et al. 2003]
Rothermel, Kurt; Bauer, Martin; Becker, Christian: SFB 627 - "Nexus" Umgebungsmodelle für mobile kontextbezogene Systeme. In: Molitor, Paul (ed.); Küspert, Klaus (ed.); Rothermel, Kurt (ed.): it - Information Technology. Bd. 45(5)

[Rothermel et al. 2003a]
Rothermel, Kurt; Bauer, Martin; Becker, Christian: Digitale Weltmodelle - Grundlage kontextbezogener Systeme. In: Friedemann Mattern (ed.): Total vernetzt - Szenarien einer informatisierten Welt

[Rothermel et al. 2003b]
Rothermel, Kurt; Dudkowski, Dominique; Dürr, Frank; Bauer, Martin; Becker, Christian: Ubiquitous Computing - More than Computing Anytime Anyplace?. In: Proceedings of Photogrammetrische Woche

[Rothermel et al. 2003c]
Rothermel, Kurt; Fritsch, Dieter; Mitschang, Bernhard; Kühn, Paul J.; Bauer, Martin; Becker, Christian; Hauser, Christian; Nicklas, Daniela; Volz, Steffen: SFB 627: Umgebungsmodelle für mobile kontextbezogene Systeme. In: Proceedings Informatik 2003

Diploma and Student Thesis

[Boronas 2003]
Andreas Boronas, “Ein Framework für verteilte Ereignisbeobachtung”, Diplomarbeit 2045 (German), Studiengang Informatik, Universität Stuttgart, 2003.

[Csallner 2003]
Christoph Csallner, “Verteilte Beobachtung von Ereignissen im Nexus-Lokationsdienst”, Diplomarbeit 2064 (English), Studiengang Softwaretechnik, Universität Stuttgart, 2003.

[Dieterle 2003]
Ralf Dieterle, “Notifikationsdienst auf Basis von Microsoft .NET”, Studienarbeit Nr. 1885 (German), Studiengang Informatik, Universität Stuttgart, 2003.

[Dudkowski 2002]
Dominique Dudkowski, "Events in the Nexus Location Service", Studienarbeit 1825 (English), Studiengang Informatik, Universität Stuttgart, 2002.
[Li 2001]
Chiangping Li, "Notification Service Based on the SOAP Technology", Diplomarbeit Nr. 1942 (German), Studiengang Master of Information Technology, Universität Stuttgart, 2001
[Minder 2003]
Daniel Minder, "Generische Integration der Beobachtung von Ereignissen in Ereignisquellen", Diplomarbeit 2080 (German), Studiengang Softwaretechnik, Universität Stuttgart, 2003.
[Till 2002]
Alexander Till, "Erweiterter Notifikationsdienst für Nexus", Diplomarbeit 2020 (German), Studiengang Informatik, Universität Stuttgart, 2002.

Related Work

[Bauer 2000]
Martin Bauer, "Event-Management für mobile Benutzer", Diplomarbeit 1836 (English), Studiengang Informatik, Universität Stuttgart, 2000.
[Cabrera et al. 2001]
Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer, "Herald : Achieving a Global Event Notification Service", Proceedings of HotOS VIII, May 2001, pp. 87-94.
[Carzaniga et al. 1998]
Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf, "Design of a Scalable Event Notification Service: Interface and Architecture", …
[Castro et al. 2002]
M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks", SIGOPS European Workshop, France, September, 2002
[Chakravarty et al. 1993]
S. Chakravarty, V. Krishnaprasad, E. Anwar, and S.-K Kim, "Anatomy of a Composite Event Detector. Technical Report", UF-CIS-TR-93-039, University of Florida, Department of Computer and Information Science, December 1993.
[Chase & Garg 1998]
C.M. Chase, and V.K. Garg, "Detection of Global Predicates: Techniques and Their Limitations", Distributed Computing, Vol. 11, No. 4, Springer, 1998, pp. 191-201.
[Cooper & Marzullo 1991]
Cooper, Robert and Marzullo, Keith, "Consistent Detection of Global Predicates", Conference proceedings on ACM/ONR workshop on parallel and distributed debugging, May 20 - 21, 1991, Santa Cruz, CA USA http://www.acm.org/pubs/citations/proceedings/onr/122759/p167-cooper
[Cugola et al. 1998]
Gianpaolo Cugola, Elisabetta di Nitto, and Alfonso Fuggetta, "Exploiting an event-based infrastructure to develp complex distributed systems", Proceedings of the 20th International Conference on Software Engineering, IEEE Computer Society, Kyoto, Japan, 1998, pp. 261-270
[Dittrich & Gatziu 2000]
Klaus R. Dittrich and Stella Gatziu, "Aktive Datenbanksysteme", dpunkt Verlag, Heidelberg, 2., völlig neubearb. und erw. Auflage, 2000
[Eugster et al. 2003]
P. Th. Eugster, P.A. Felber, R. Guerraoui, and A,-M. Kermarrec, "The Many Faces of Publish/Subscribe", ACM Surveys, Vol.35, No. 2, June 2003, pp. 114-131.
[Gehani et al. 1992]
N.H. Gehani, H.V. Jagadish, and O. Shmueli, "Composite Event Specification in Active Databases: Model and Implementation", Proceedings of the 18th International Conference on Very Large Databases, 1992, pp. 327-338.
[GPS Signal Specification 1995]
GPS SPS Signal Specification, 2nd Edition (June 2,1995), Annex B, Section 5.0, "Accuracy Characteristics", pp. B-15-B21.
(http://www.navcen.uscg.gov/pubs/gps/sigspec/default.htm)
[Gruber et al. 1999]
Robert Gruber, Balachander Krishnamurthy, and Euthimios Panagos, "The Architecture of the READY Event Notification Service", Proceedings of the 19th IEEE International Conference on Distributed Computing, System Middleware Workshop, 1999.

[Harter et al. 1999]
A. Harter, A. Hopper, P. Steggles, A.Ward, and P.Webster, "The Anatomy of a Context-Aware Application", In Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom'99), Seattle, Washington, USA, August 1999, pp. 59 - 68

[Hayton et al. 1996]
Richard Hayton, Jean Bacon, John Bates, and Keb Moody, "Using Events to Build Large Scale Distributed Applications", ACM SIGOPS European Workshop, September 1996, pp. 9-16

[Hinze & Voisard 2002]
Annika Hinze and Agnès Voisard, "Composite events in notification services with application to logistics support", in Proceedings of the 9th International Symposium on Temporal Representation and Reasoning (TIME-2002), 7-9 July 2002.

[Hohenstein & Schmatz 2003]
Uwe Hohenstein and Klaus-Dieter Schmatz, „Webanwendungen entwickeln mit Oracle9i", dpunkt Verlag, Heidelberg, 1. Auflage, 2003.

[Hohl et al. 1999]
F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, and M. Schwehm, "Next Century Challenges: Nexus – An Open Global Infrastructure for Spatial-Aware Applications", Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99), Seattle, USA, 1999, pp. 249-255.

[Jacobsen & Llirbat]
Hans-Arno Jacobsen and F. Llirbat, "Publish / Subscribe Systems Tutorial", http://www.eecg.toronto.edu/~jacobsen

[Janotti et al. 2000]
John Canotti, David K. Gifford, Kirk L. Johanson, M. Frans Kaashoek, James W. O'Toole, Jr., "Overcast: Reliable Multicasting with an Overlay Network", Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI), October 2000, pp. 197-212.

[Los Angeles Times 2004]
Vanessa Gera, Associated Press Writer, "Gates Forsees Seamless Software Links", Los Angles Times, January 28, 2004
http://www.latimes.com/technology/ats-ap_technology12jan28,1,3889909.story?coll=sns-ap-toptechnology

[Leonhardi & Rothermel 2001]
A. Leonhardi and K. Rothermel, "A Comparison of Protocols for Updating Location Information", Baltzer Cluster Computing Journal, 4(4), pp. 355-367, Baltzer Science Publishers, 2001.

[Leonhardi & Rothermel 2001b]
Alexander Leonhardi and Kurt Rothermel, "Architecture of a Large-scale Location Service", Technical Report No. 2001/01, Universität Stuttgart, Fakultät Informatik, 2001.

[Leonhardi 2003]
Alexander Leonhardi, "Architektur eines verteilten skalierbaren Lokationsdienstes", Dissertation (German), Fakultät Informatik, Elektrotechnik und Informationstechnik, Universität Stuttgart, 2002.

[Liebig et al. 1999]
C. Liebig, M. Cilia and A. Buchmann, "Event Composition in Time-dependent Distributed Systems", Proceedings of the 4th International Conference on Cooperative Information Systems (CoopIS 99), IEEE Computer Society, 1999, pp. 70-78.

[Matthiessen & Unterstein 2000]
Günter Matthiessen and Michael Unterstein, Relationale Datenbanken und SQL", Addison-Wesley, München, 2000.

[Minar 1999]
Nelson Minar, "A Survey of the NTP Network", December, 1999
http://www.media.mit.edu/~nelson/research/ntp-survey99/

[Naguib 2001]
Hani Naguib and George Coulouris, "Location Information Management", Proceedings of Ubicomp 2001, pp. 35-41

[Nelson 1998]
Giles John Nelson, "Context-Aware and Location Systems", PhD thesis, Clare College, University of Cambridge, UK, January 1998

[Nicklas et al. 2001]
Daniela Nicklas, Matthias Grossmann, Thomas Schwarz, Steffen Volz, and Bernhard Mitschang, "A Model-Based, Open Architecture for Mobile, Spatially Aware Applications", Proceedings of the Symposium on Spatial and Temporal Databases, Los Angeles, USA, 2001.

[Nicklas et al. 2003]
Nicklas, Daniela; Grossmann, Matthias; Schwarz, Thomas: NexusScout: An Advanced Location-Based Application On A Distributed, Open Mediation Platform. In: Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

[Nicklas & Mitschang 2001]
Daniela Nicklas and Bernhard Mitschang, “The Nexus Augmented World Model: An Extensible Approach for Mobile, Spatially Aware Applications”, Proceedings of the 7th International Conference on Object-Oriented Information Systems (OOIS) 2001, Calgary, Canada, 2001.

[OMG 2001]
Object Management Group, “Event Service Specification”, 2001
htpp.//www.omg-prg/cgi-bin/doc?formal/01-03-01.pdf

[OMG 2002]
Object Management Group, “Notification Service Specification”, 2001.
htpp.//www.omg-prg/cgi-bin/doc?formal/02-08-04.pdf

[Pietzuch & Shand 2002]
Peter R. Pietzuch and Brian Shand, “A Framework for Object-Based Event Composition in Distributed Systems”, 12th Workshop for PhD Students in Object-Oriented Systems (PhDOOS'02), Malaga, Spain, June 2002.

[Pietzuch et al. 2003]
Peter R. Pietzuch, Brian Shand, and Jean Bacon, “A Framework for Event Composition in Distributed Systems”, ACM/IFIP/USENIX International Middleware Conference, Springer-Verlag, 2003.

[Rosenblum & Wolf 1997]
David S. Rosenblum and Alexander L. Wolf, “A Design Framework for Internet-Scale Event Observation and Notification”, Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), 1997, pp. 344-360

[Rowstron & Druschel 2001]
Antony Rowstron and Peter Druschel, “Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-To-Peer Systems”, IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.

[Rowstron et al. 2001]
Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel, “SCRIBE: The Design of a Large-Scale Event Notification Infrastructure”, NGC2001, UCL, London, November 2001.

[Schmidt & Demmig 2001]
Meinhardt Schmidt and Thomas Demmig, “SQL GE-PACKT”, mitp Verlag, Bonn, 1. Auflage, 2001.

[Schwarz & Mattern 1994]
R. Schwarz, and F. Mattern, “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail”, Distributed Computing, Vol 7, No. 3, Springer, 1994, pp. 149-174.

[Want et al. 1992]
Roy Want and Andy Hopper and Veronica Falcão and Jonathan Gibbons, “The Active Badge Location System," ACM Transactions on Information Systems, vol. 10, pp. 91--102, Jan. 1992.