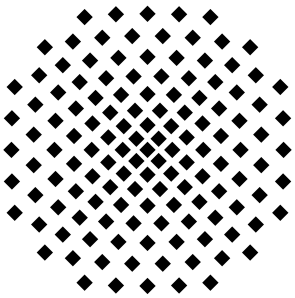


Universität Stuttgart

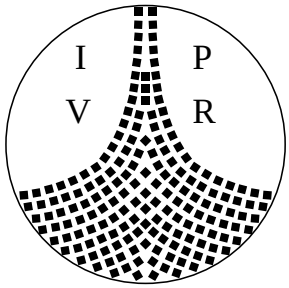
Fakultät Informatik

Prüfer:	Prof. Dr. K. Rothermel
Betreuer:	Dr. Joachim Baumann
begonnen am:	1.1.2000
beendet am:	30.6.2000
CR-Klassifikation:	C.2.4, C.4, H.3.4

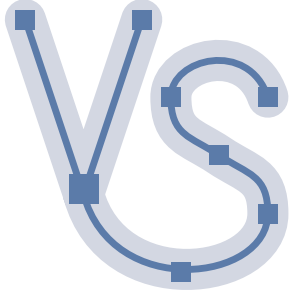
Diplomarbeit Nr. 1836

Event-Management für mobile Benutzer

Martin Bauer



Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Abteilung Verteilte Systeme
Breitwiesenstr. 20-22
70565 Stuttgart



Kurzfassung

Event-basierte Kommunikation ist in vielen Anwendungsgebieten im Computerbereich verbreitet. Dabei reicht das Spektrum von der Programmierung von Benutzerschnittstellen bis hin zu Publish & Subscribe Diensten. Existierende Event-Management Systeme beschränken sich auf die Auslieferung von Event Benachrichtigungen an Gruppen von Empfängern, die physikalisch mit dem Netzwerk verbunden sind. In den meisten Fällen wird das Konzept eines "Event Erzeugers" zugrunde gelegt. Es gibt dann keine eigenständige Komponente zur Beobachtung von Events, besonders was komplexe Events betrifft, deren Beobachtung verteilt erfolgen muss. Außerdem sind die existierenden Event-Management Systeme oft nicht für Umgebungen mit mobilen Computern geeignet, insbesondere wegen der Störanfälligkeit von Funknetzverbindungen.

Das Projekt Nexus an der Universität Stuttgart untersucht Konzepte und Methoden, um ortsbezogene Anwendungen für mobile Benutzer zu unterstützen. Das Ziel ist es, eine globale offene Plattform zu entwickeln, die als Middleware für solche Anwendungen dient. Der Nexus Event Service wird ein zentraler Bestandteil dieser Plattform sein. Seine Aufgabe besteht aus der Beobachtung von Events, insbesondere auch von komplexen, verteilten Events, und der Auslieferung von Benachrichtigungen an interessierte Empfänger.

Diese Diplomarbeit besteht aus zwei Teilen. Der erste Teil diskutiert die relevanten Konzepte aus den Bereichen event-basierte Kommunikation und Mobilität. Die Terminologie und das Event Modell werden konsolidiert und auf räumliche Events erweitert. Wir definieren ein Event als beobachtbare Veränderung im Zustand des Umgebungsmodells. Räumliche Events basieren auf der absoluten Position von Objekten im Raum oder der relativen Position von Objekten zueinander. Ein Beispiel für ein solches Event wäre, wenn ein Benutzer ein bestimmtes Gebiet oder ein bestimmtes Gebäude betritt.

Der zweite Teil präsentiert eine Architektur für den Nexus Event Service auf abstrakter Ebene. Es werden drei logische Komponenten identifiziert: der Observation Service, der Notification Service und eine Komponente zur Verwaltung von Prädikaten, die alle Informationen verwaltet, die sich auf Events und Empfänger beziehen. Darauf folgend werden wichtige Architekturfragen diskutiert, insbesondere die Verteilung des Event Service und die Interaktion zwischen dem Event Service und anderen Nexusdiensten bzw. Empfängern von Nachrichten. Zum Schluss wird gezeigt, wie räumliche Events in der Nexusumgebung beobachtet werden können.

Abstract

Event-based communication is popular in many areas of computing, ranging from programming user interfaces to publish & subscribe services. Existing event management systems concentrate on the delivery of event notifications to groups of clients, which are physically connected to computer networks. In most cases they rely on the concept of an “event producer” and lack a component for observing events themselves, especially regarding complex events whose observation is distributed. Also, existing event management systems are often not well suited for mobile computing environments, particularly with respect to the volatility of wireless connections.

The Nexus project at the University of Stuttgart is examining concepts and methods for supporting location- and spatial-aware applications for mobile users. The goal is to develop a global open platform serving as a middleware for these applications. The Nexus Event Service will be a central component of this platform. Its tasks comprise the observation of events, including complex distributed events, and the delivery of event notifications to interested clients.

This thesis consists of two main parts. The first part discusses the relevant concepts of event-based communication and mobility. The terminology and the event model are consolidated and extended to cover spatial events. We define an event as an observable change in the state of the environment model. Spatial events are events that are based on the absolute position of objects in space or the relative position of objects to each other. An example for such an event is a user entering a given area or building.

The second part presents a high-level architecture for the Nexus Event Service. It identifies three logical components: the Observation Service, the Notification Service and the Predicate Management component, which manages the information pertaining to events and clients. We then discuss important architectural issues, including alternatives for the distribution of the Event Service and the interaction of the Event Service with other Nexus services and clients. Finally, we show how spatial events can be observed in the Nexus context.

Acknowledgements

I would like to thank Professor Rothermel for introducing me to this interesting thesis topic. His ideas have given new impulses multiple times.

I would like to express my sincere thanks to my advisor, Dr. Joachim Baumann, for his continuous feedback that helped to improve the quality of this thesis considerably.

I would also like to thank the whole Nexus group, especially Fritz Hohl, Alexander Leonhardi, Peter Coschurba and Uwe Kubach, for many fruitful discussions that gave me new insights regarding the Nexus system.

Finally, I am grateful for the support of my parents, my girlfriend Vimala and my friends, while I was writing this thesis.

Table of Contents

Part I : Concepts 1

1 Introduction 3
1.1 Overview 4

2 Event-Based Communication 7
2.1 Terminology 7
2.2 Event Observation / Notification Model 8
2.3 Event Paradigm 8
2.4 Events in Related Work 9
2.4.1 Programming Languages 9
2.4.2 Event Notification in Distributed Environments 10
2.4.3 Publish & Subscribe Services 10
2.4.4 Internet-Scale Event Frameworks 11
2.4.5 Active Database Systems 12
2.5 Describing Events 12
2.5.1 Description Language 12
2.5.2 Event Algebra 14
2.6 Event Semantics 14
2.7 Summary 16

3 Events and Mobility 17
3.1 Aspects of Mobility 18
3.2 Event-Based Communication for Mobile Devices 19
3.2.1 Advantages of Event-Based Communication 20
3.2.2 Related Problems and Disadvantages 20

4 Spatial Events 23
4.1 Foundations for the Definition of Spatial Events 23
4.1.1 Dimensions 23
4.1.2 Coordinate System 24
4.1.3 Position of Objects 25
4.1.4 Spatial Attributes and General Relations 26
4.2 Spatial Relations 27
4.2.1 Relations 27
4.2.2 Additional Attributes 29
4.2.3 Specialized Relations 31
4.3 Spatial Events 31

Part II : Design for Nexus 35

5 Nexus 37
5.1 Idea 37
5.2 Example Scenario 37
5.3 Technology 38
5.4 Nexus Model 39
5.5 General Design Objectives 41
5.6 Components 42
5.6.1 Location Service 43
5.6.2 Spatial Model Service 45
5.6.3 Geographic Addressing 45
5.6.4 Hoarding 45
5.7 Summary 46

6 The Event Service 47
6.1 Outside View 47
6.2 Internal Structure 48
6.3 Describing Events 50
6.3.1 Language Elements 50
6.3.2 Registering Events 51
6.3.3 Naming of Predicates and Events 52
6.4 Predicate Management 52
6.4.1 Predicate Template Register 53
6.4.2 Predicate Register 54
6.5 Observation Service 54
6.6 Notification Service 55
6.6.1 Representation of Event Notifications 56
6.7 Summary 57

7 Architectural Issues 59
7.1 Interaction with Other Nexus Components 59
7.1.1 Coupling 59
7.1.2 Interaction Level 64
7.1.3 Level of Abstraction Offered to the Clients 67
7.2 Distribution 68
7.2.1 Distribution Structure 69
7.2.2 Relation to the Distribution of Other Nexus Services 70
7.2.3 Handover of Event Observation 70
7.2.4 Distribution of Event Service Components 71
7.3 Summary 71

8 Spatial Events in Nexus 73
8.1 Event Descriptions 74
8.1.1 Examples of Spatial Events 76
8.2 Observation of Spatial Events 77
8.2.1 Classification of Spatial Events 77
8.2.2 Services Involved in the Observation of Spatial Events 78
8.2.3 Observing Events Involving more than one Nexus Service 80

8.2.4 Basic Events Offered by the Location Service 81
8.3 Distributed Observation of Spatial Events 83
8.3.1 onEnterArea 84
8.3.2 onEnterObject 85
8.3.3 onMeeting 85
8.4 Summary 86

9 Conclusion **87**
9.1 Future Work 87

Part III : Appendix 91

A Commercial Notification Services 93
A.1 CORBA: Event Service Specification 93
A.2 Java: Distributed Event Specification 94
A.3 Orbix Talk 95
A.4 TIBCO, TIB/Rendezvous 96

B Mobile IP 99

C More Nexus Events 101

D References 103

List of Tables

Table 3-1: Wireless Network Technologies 18
Table 4-1: Predicates Describing Spatial Events 32
Table 4-2: Predicates Describing Events Involving the Movements of Objects 33
Table 8-1: Predicates Describing Spatial Events in Nexus 74
Table 8-2: Predicates Describing Nexus Events Involving the Movements of Objects 75
Table 8-3: Spatial Events Not Supported in Nexus 76
Table 8-4: Event Example 77
Table 8-5: Event Classification 78
Table 8-6: Observation of Events Involving Different Kinds of Object Pairs 79

List of Figures

Figure 2-1: Event Observation/Notification Chain 7
Figure 2-2: Event Observation /Notification Model 8
Figure 4-1: Position of Objects 25
Figure 4-2: Spatial Attributes 26
Figure 4-3: General Spatial Relations 26
Figure 4-4: in and in_extent Relation 28
Figure 4-5: Direction of Movement 29
Figure 4-6: Relative Deviation 30
Figure 4-7: Absolute Deviation 30
Figure 4-8: Direction Relative to Coordinate Systems 30
Figure 5-1: Interaction between Augmented Area and Augmented Area Model 40
Figure 5-2: Augmented Areas 40
Figure 5-3: Nexus Architecture 42
Figure 5-4: Location Service 44
Figure 6-1: Event Service Interface 47
Figure 6-2: Internal Structure of the Event Service 49
Figure 6-3: Predicate Management Component 52
Figure 6-4: Notification Service 55
Figure 7-1: Service Levels 65
Figure 7-2: Level on which Clients Interact with the Event Service 67
Figure 7-3: Distribution Structure 69
Figure 7-4: Distribution of Event Service Components 71
Figure 8-1: Query for Static Object 80
Figure 8-2: Distributed onEnterArea 84
Figure 8-3: Distributed onEnterObject 85
Figure 8-4: Distributed onMeeting 86
Figure A-1: CORBA Event Channel 93
Figure A-2: Java Distributed Event Model 94
Figure A-3: OrbixTalk Model 95
Figure A-4: TIBCO's Publish & Subscribe Model 96
Figure B-1: Registering with a Foreign Agent 99
Figure B-2: Mobile IP 100

Part I: Concepts

Chapter 1

Introduction

The term *event* describes a natural concept of everyday life. Something happens, we observe it and react to it. For example, if 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 a computer context, something similar happens when we receive an e-mail. An e-mail application may inform us about the occurrence of the event “new e-mail arrived” by beeping and displaying or changing an icon on some visible part of the screen.

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. In many cases, especially if the mobile computer is aware about its current context, e.g. its location, the mobile computer can become active by itself, giving its user information that is relevant in the current situation. For example, the mobile computer may help the user to remember certain tasks.

Let us look at two examples. If the user is walking past a shoe shop in a mall, the mobile computer may remind him that he wanted to buy shoes. In this case, the user had to inform the computer application some time in advance about his intention to buy shoes. In another scenario, the wife of the user may place a virtual note in the supermarket, telling her husband to bring tomatoes. The mobile computer informs the user about the virtual note when he is shopping in the supermarket. So the information is placed at the location where it is most useful.

In both cases, the mobile computer is proactive, thereby enhancing human capabilities. Also, certain event characteristics are evident. The user is informed automatically, when he comes close to a certain location. The actual events are “getting close to a shoe shop” and “entering the supermarket”.

In order to support such events in a computer system, it must be possible to describe the event, the event has to be observed and, if the event actually occurs, interested clients have to be informed. For this purpose, a suitable communication mechanism is needed.

As part of the Nexus project, a global open platform for supporting location- and spatial-aware applications is being developed. The Nexus project is a research project funded by the German Science Foundation (DFG). Currently, four partners at the University of Stuttgart are involved, the Institute of Photogrammetry, the Institute of Communication Networks and Computer En-

gineering and the Institute of Parallel and Distributed High Performance Systems with the Applications of Parallel and Distributed Systems Group and the Distributed Systems Group.

An event management mechanism is needed as a fundamental component of the Nexus platform. Since the purpose of the platform is to support location- and spatial-aware applications, the event examples given above are typical for a Nexus scenario. This means that the event management mechanism has to deal with mobile users and also has to cooperate with the Location Service planned for Nexus that provides the current location of mobile users.

Existing implementations of event management mechanisms are not well suited for the support of mobile users. They do not take into consideration the special situation regarding the communication with mobile users, they offer insufficient or not precisely specified fault semantics, and, in a lot of cases, they do not scale well enough.

The overall goal of this project is to investigate and evaluate architectures and mechanisms for Internet-scale event management. Important issues that have to be taken into account are the mobility of the users and the scalability, efficiency and reliability of the event-management mechanism.

The Nexus platform will serve as a test environment for analyzing and evaluating alternative architectures for event management mechanisms. A prototype of a Nexus Event Service is to be designed and implemented. Simulations and measurements will be used to validate the analysis.

This thesis can only cover a small part of the whole area of event management described above. The actual goals are the following:

- Structure the existing work in the areas of
  - Event-based communication
  - Mobility
  - The Nexus platform
- Design a high-level architecture for the Nexus Event Service
- Discuss design alternatives
- Define classes of spatial events

The results will serve as a foundation for future work that goes beyond the reach of this thesis.

1.1 Overview

This thesis consists of two main parts. The first part will cover the general concepts of event-based communication, mobility and spatial events. The second part will describe the design of an architecture for the Nexus Event Service, including the discussion of architectural issues and the spatial events to be supported in Nexus.

In Chapter 2 we will define our event terminology and describe the characteristics of event-based communication. This will be followed by an overview of events in related work, the presentation of a description language for events and a short discussion of event semantics. Chapter 3 will cover mobility issues and how they relate to events. Chapter 4 will present the foundations for the definition of spatial events and we will derive a list of interesting spatial events.

Chapter 5 will give a short overview of the planned Nexus platform, the underlying ideas, concepts and components. In Chapter 6 we will define a high-level architecture of the Event Service. Chapter 7 will discuss important architectural issues like the interaction of the Event Service with other Nexus components and the distribution of the Event Service. In Chapter 8, the spatial events presented in Chapter 4 will be discussed in the Nexus context, especially regarding their observation. Chapter 9 will conclude the thesis by giving a short outlook on future work.

Appendix A provides some additional information regarding specifications and commercial implementations of event notification services. Appendix B gives an introduction to Mobile IP, a protocol that allows mobile devices to change their network access point while being in operation. Appendix C lists events that may be relevant for the Nexus system, but are not covered in Chapter 8. Finally, Appendix D contains a list of references.

Chapter 2

Event-Based Communication

As we have seen in the introduction, an important reason for the popularity of the event paradigm in the context of computing is that it is natural for us to think in terms of real-world events. In the following, we will discuss how the term event can be defined in the computer context, what characteristics are associated with the event paradigm, and how events can be described.

2.1 Terminology

In a computer context, all applications are based on some kind of an underlying model of their environment. The state of the model is defined by the data associated with the application. Hence, an occurrence of any kind is reflected by a change in the application data, that is, an observable change in the state of the model. Therefore, we define an *event* in the following way:

Definition 2-1: An event is an observable change in the state of the model.

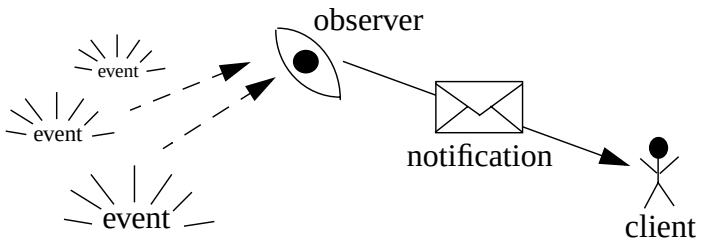


Figure 2-1: Event Observation/Notification Chain

The change of state can be observed, but the occurrence of an event does not depend on the observation. It can take place whether it is being observed or not. In Figure 2-1 the event observation/notification chain is shown. An observing entity determines that an event of a certain kind has taken place. It can then initiate the sending of an event notification to interested clients. A notification is a message that describes the event, e.g. its type, the entity in which it occurred, and the time of occurrence.

2.2 Event Observation / Notification Model

Based on the terminology given above, we now define a simple model that describes the components involved in an event observation and notification architecture. This will help us in the following discussion about event-based communication.

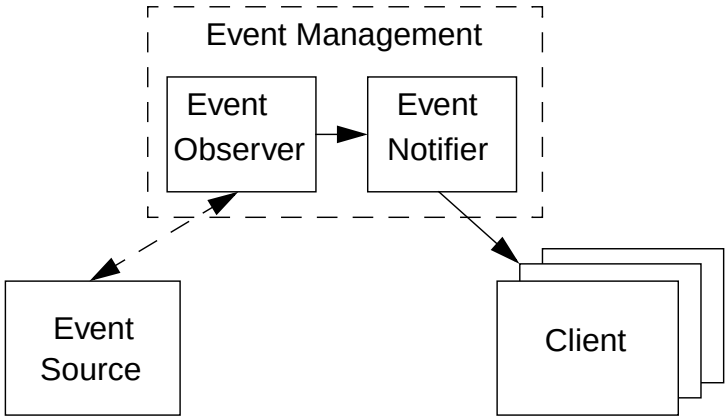


Figure 2-2: Event Observation /Notification Model

The model follows the event/observation chain described in the previous section. There is an event source of some kind, which can consist of one or more active or passive components. The event source comprises the environment model mentioned above. We have defined an event to be a change in the state of that environment model (Definition 2-1).

The event observer component observes the events taking place “in” the event source. The model says nothing about how this is achieved, because this depends very much on the nature of the event source. Therefore a dashed double-arrow is used to symbolize the observation. Whenever the event observer has detected the occurrence of an event, the information about the event is handed over to the event notifier. The event notifier sends an event notification to all interested clients.

The event management combines the event observer and event notifier components. Clients have to register for event notifications with the event management, which enables the observation of the event and the subsequent notification of the client.

2.3 Event Paradigm

In the following, we will discuss a number of characteristics that are often associated with event-based communication.

Event-based communication is typically source-initiated. Clients can register for events they are interested in. If an event has taken place, the registered clients receive an event notification, so the information is actually *pushed* onto interested clients. Therefore, this style of communication is also called *push* communication.

The source in “source-initiated” is the source of the event notification, which in our model may coincide with the event observer component.

Push communication is in contrast to the request-response style of communication that is common in distributed systems, for example in client-server applications. Here the initiative is on the side of the client - it pulls the information. Therefore, the term *pull* communication is used.

Even though event-based communication is conceptually a type of push communication, event notification services like the CORBA Event Service [OMG 1997] also support pull communication. In this case the event notifications have to be buffered. They are delivered only when the client asks for them.

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 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 message has to go over the same link only once. Overall, event-based communication can improve scalability by reducing communication overhead.

Event-based communication allows the decoupling of event source and client. Event-based communication can be asynchronous. The source (or event observer) does not need to wait for an acknowledgement from the client. It can continue its execution.

The event source and interested client(s) do not even 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.

These characteristics make event-based communication ideally suited for loosely coupling software components. There is a strict separation of concerns. 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. In that way, it is even possible to integrate new components into the system without affecting the components that are already there. The old components implicitly invoke the new components through event notifications.

2.4 Events in Related Work

Event-based communication is very popular in all kinds of scenarios. However, these scenarios have very different characteristics, different environments and, as a result, they use different terminology. This is especially true for the definition of the term *event* itself. In the following, we will look at some important areas, in which event-based communication is used.

2.4.1 Programming Languages

In the area of programming languages an event is mostly considered to be equivalent to a message. The sending of the message itself is interpreted as an event, so event and event notification fall together, and there is no need to distinguish between them. In this environment, people often talk about “firing an event” for sending an event notification.

Compared to our model, event source and event management are combined into a single component there, the *initiator of the event*.

If an event is the result of a real-world occurrence, like a user action, as is commonly the case when programming user interfaces, one can still distinguish between the event (message) and the real-world event (user action).

In a lot of other cases, the term “real-world event” does not really apply. However, it is often important to make a distinction between the actual occurrence and the notification that is sent as a result of it. Therefore, in our terminology, we use the term event to describe the occurrence and the term event notification to describe the message.

2.4.2 Event Notification in Distributed Environments

A number of protocols for the delivery of event notifications in distributed environments have been defined, e.g. in the CORBA Event Service Specification [OMG 1997] and the Java Distributed Event Specification [SUN 1998]. Both specifications describe services that deliver event notifications between distributed objects. They also specify how client objects (called consumers and listeners depending on their role) can register for events in order to receive event notifications.

The CORBA Event Service Specification gives no explicit definition for the term *event*. Instead, it merely says: “Suppliers produce event data and consumers process event data. Event data are communicated between suppliers and consumers...” [OMG 1997], p. 4-2.

In the Java Distributed Event Specification [SUN 1998], p.13, an event is defined as “something that happens in an object, corresponding to some change in the abstract state of the object.” In this case, the term notification is used for the message just as in our definition.

There are also a number of event notification services that are available as commercial products, e.g. Orbix Talk [IONA 1996] and TIB Rendezvous [TIBCO 1999]. In both cases, the term event is avoided and the term *message* is used as the basic concept.

All specifications and products mentioned so far concentrate solely on the distribution of event notifications. The observation or detection of events is left to *event suppliers*. In our model, an event supplier can be described as the combination of event source and event observer.

This thesis, however, will propose an architecture that allows the observation of distributed events, which implies the existence of a separate observer component. The event notification services just mentioned could be used in the notifier component that finally delivers the event notifications.

A more detailed description of the mentioned service specifications and the products can be found in Appendix A.

2.4.3 Publish & Subscribe Services

Publish & subscribe services are used for the efficient distribution of content ranging from simple text information to multimedia information with audio and video, or software updates. As underlying technology, the products mentioned in the previous subsection can be used. There are also products like Marimba’s Castanet [MARIMBA 1997] that focus especially on the publish & subscribe paradigm.

In the publish & subscribe paradigm, *publishers* offer information to *subscribers*. The information is often organized in channels to which the subscribers can subscribe to receive the information automatically. Conceptually, this looks like push technology, but in a lot of cases, regular polling is used to receive regular updates.

When compared to our model, the publisher is a combination of the event source and the event observer, the subscriber is the client, and the service itself is the event notifier. The occurrence of real events is restricted to the availability of new information that is distributed as part of the event notification.

2.4.4 Internet-Scale Event Frameworks

Internet-scale event frameworks target the delivery of event notifications over the Internet, e.g. [ROSENBLUM & WOLF 1997], [RIFKIN & KHARE 1998]. The focus is primarily on the coupling of heterogeneous systems over the Internet. The term *event-based software integration* is also often used in this context [BARRET ET AL. 1996].

Rosenblum and Wolf [ROSENBLUM & WOLF 1997] describe a design framework for Internet-scale event observation and notification. Their framework consists of seven models - an object model, an event model, a naming model, an observation model, a time model, a notification model and a resource model. Their event definition is based on their object model: “An event is the instantaneous effect of the (normal or abnormal) termination of an invocation of an operation on an object, and it occurs at the object’s location.”

This means that they only consider the local observation of events, not the distributed observation of global predicates. However, their general design framework fits into our scenario, if the models are modified accordingly.

In the context of their framework, they also define a timeline of activities regarding the handling of events.[ROSENBLUM & WOLF 1997], p. 347:

1. determination of which events will be made observable;
2. expression of interest in an event or pattern of events;
3. occurrence of each event;
4. observation of each event that occurred;
5. relation of the observation to other observations to recognize the event pattern of interest;
6. notification of an application that its pattern of interest has occurred;
7. receipt of the notification by the application; and
8. response of the application to the notification;

The last of the activities is considered to be already out of the domain of concern of the event observation and notification facility.

As far as the timeline of activities is concerned, their model fits very well into our simple event observation - notification model. The same activities have to be performed there, and all components we have identified are involved in doing so.

2.4.5 Active Database Systems

Traditionally, database systems have been seen as passive components that are used for storing data efficiently and safely. In a lot of cases, however, knowledge does not exclusively consist of passive data, but also includes active rules that pertain to the data. For example, there might be a rule pertaining to a bank account that says that whenever the balance of the account is greater than a certain threshold, money should be transferred to another account. In an active database system [DITTRICH & GATZIU 2000][PATON & DÍAZ 1999] this rule could be part of the database system, and the transfer action could be executed automatically.

In [PATON & DÍAZ 1999], p. 63, active database systems are described in the following way: "Active database systems support mechanisms that enable them to respond automatically to events that are taking place either inside or outside the database system itself."

The rules in active database systems are usually defined as *event-condition-action* (ECA) triples. Events describe an occurrence to which the rule may respond. Conditions examine the context in which an event has taken place. Depending on the result of the evaluation of the condition the action is executed. The action itself consists of a task that has to be carried out.

In the active database literature, the term *event* is used for both the actual occurrence and the description of that occurrence. In most cases, active database systems are not distributed, so the observation of events can be carried out by a central component. Events are the result of operations on the database or of user-initiated invocations, or they are based on the system clock. Actions are either database operations that may again trigger events, or invocations of user-defined methods.

So, in active database systems there is no need for the delivery of event notifications within the system, therefore it does not fit well in our event observation-notification model. On the other hand, active database systems can produce input for other event management systems, e.g by invoking a method that informs the observer component of such a system that an event has taken place. Therefore, the whole active database system could be an interesting event source.

2.5 Describing Events

In order to utilize event-based communication, events have to be represented in some way. The clients have to uniquely specify an event when subscribing for event notifications. They also have to associate the event notifications they receive with the registered event, so that the event notification can be handled appropriately.

If the set of events that can be observed is not fixed and may change dynamically, there has to be a way to specify and integrate new events. In other words, the event has to be registered with the event management, and the event management has to know how to observe the event. For all these reasons, a description language for events is needed.

2.5.1 Description Language

We propose a language with three main components - *predicates*, *filters* and *actions*. Predicates are used to describe the actual events. If an event occurs, the predicate becomes true and this can

result in an action. A filter can be used to specify further conditions that decide whether the action is actually executed.

The only action we will consider at this point is the sending of event notifications. The reason is that we want to look at event observation and event notification in general, independent of any underlying system. Actions are usually system-dependent, but even if we are looking at a specific system, it is advantageous to keep the event management separate to allow a separation of concerns. Actions can be integrated by having active components register for event notifications. The components then carry out the action after receiving the event notification.

The *predicate-filter-action* triple corresponds to the *event-condition-action* triple that is used in active database systems (Subsection 2.4.5).

In the following we look at the language elements in some more detail.

Predicates. As mentioned before, predicates are used for describing events. The occurrence of an event is equivalent to a predicate becoming true. A predicate describes properties of entities and relationships between entities in time and space. So an event is a change in a property or a change in a relationship between entities. In a computer context, the entities, properties and relationships are modelled as data, which in turn defines the state of the model. Therefore, an event is a change in the state of the model as we have defined it in Definition 2-1.

We distinguish two types of predicates, *basic predicates* and *complex predicates*:

- *Basic predicates* describe events that are atomic changes or appear to be atomic changes outside the entity in which they occur or become visible. Therefore basic predicates are the basic elements of the predicate description language.
- *Complex predicates* are combinations of other predicates. In order to combine predicates, the predicate language can have certain constructor elements, e.g. logical constructor elements (`and`, `or`, `not`,...) and possibly time-related elements (X `after` Y,...). In the next subsection, we will discuss a simple event algebra that can be used for that purpose. Complex predicates might also maintain state information and include complicated calculations, e.g. the event *"more than N people are in room X and Y"*. For this purpose, a more powerful description language is necessary. Maybe even a full-scale programming language is needed.

In most cases, it is not sufficient to know that an event has occurred. Additional information is needed, such as when the event occurred, where the event occurred or other event specific information. Therefore, the information to be returned when an event has occurred has to be specified along with the predicate.

Filters. A filter can contain additional conditions. They are used to filter out events that should not result in an action, because some additional requirement that is specified in the condition is not fulfilled. The conditions may be based on the information returned by the event or on some other external source like the current time.

Actions. As stated above, the only action that is going to be supported is the sending of event notifications. However, the action does not become void because of that restriction. It contains information for the event notifier component stating where the event notification is to be delivered to and, if there are multiple alternatives, how the event notification is delivered.

2.5.2 Event Algebra

[DITTRICH & GATZIU 2000], p. 25-27, present a relatively simple event algebra for defining complex predicates, which has the following predicate constructors:

- *Disjunction.* The disjunction `P1 | P2` of two predicates `P1` and `P2` describes an event that occurs, if either the event `E1` described by `P1` or the event `E2` described by `P2` occurs. The disjunction constructor is defined primarily for reasons of clarity and convenience, the events could also be defined separately.
- *Sequence.* The sequence `P1;P2` describes an event that occurs, if the events `E1` and `E2` that are described by `P1` and `P2` respectively have occurred in the given sequence.
- *Conjunction.* The event described by the conjunction `P1,P2` occurs, if the corresponding events `E1` and `E2` have occurred in arbitrary order. `(P1,P2)` is therefore equivalent to `(P1;P2) | (P2;P1)`.
- *Negation.* The negative event described by `NOT P WITHIN I` occurs, if event `E` has not occurred within the given interval `I`.
- *Reduction.* With event reduction, it is possible to recognize repeating events only from time to time. The respective predicate constructors are
  `* P` - recognize only one instance of the event described by `P`.
  `* P WITHIN I` - recognize only one instance of the event described by `P` within interval `I`.
  `TIMES (n,P)` - recognize only every *n*th occurrence of the event described by `P`.
- *Relative Time.* It is also possible to describe an event that occurs in a certain temporal relationship relative to another event described by predicate `P`: `P + I` or `P - I`.

This event algebra could be used as a basis for describing some of the complex events that we will discuss later.

2.6 Event Semantics

Event semantics deals with questions regarding what can be said about the occurrence of an event. In principle, the area of event semantics can be divided into the areas of observation semantics and notification semantics. However, since certain issues are relevant in both cases, we do not make this distinction for the remainder of this section.

Some of the important questions regarding event semantics are: Can it be guaranteed that the detected event really occurred, or is it just likely? Can all actual events be detected reliably, or is it possible that event occurrences go undetected? If the occurrence of an event has been observed, can the delivery to all interested clients be guaranteed? Do all clients receive event notifications in the same sequence? etc.

Looking at all these questions, it should be evident that event semantics is a very important area. Unfortunately, because of time restrictions, we can only *touch* it in this thesis. The following should give a short overview of the problems that have to be considered.

As long as events occur within a single unit, e.g. a single computer, and a central component is responsible for observing events, it is relatively easy to answer the questions listed above, e.g. there is a single clock, so events can be ordered by attaching unique timestamps.

If multiple such units are connected together to form a distributed system, and event notifications are exchanged between clients on different units, determining a globally unique sequence of events becomes difficult.

In some cases, it is sufficient to have an ordering that observes the causality between events. For that purpose, logical clocks have been introduced [LAMPORT 1978]. While simple logical clocks allow a global ordering of events, they are not sufficient for fully describing causality, e.g. whether event A is actually causally related to event B. For this purpose vector clocks are needed. Each vector consists of separate clock values, one for each process involved [SCHWARZ & MATTERN 1994]. These vectors are then attached to every message exchanged. It is only possible to use vector clocks if the number of processes involved is a small fixed number, which will not be the case in a globally distributed event management system.

So far, we have only considered the case in which the clients are distributed. The situation becomes even more complicated if the events themselves are distributed, i.e. consist of different subevents that occur in different units. This corresponds to the detection of global predicates, which is discussed in [CHASE & GARG 1998]. They show that the problem of predicate detection in distributed systems for general predicates is NP-complete. The NP-completeness is due to the fact that all possible sequences of partial events have to be considered to determine if "`possibly` φ" or "`definitely` φ" applies, where φ is the global predicate.

In a lot of practical scenarios, however, it may be sufficient to assume that global time is available, so timestamps can be used for ordering events. For this assumption to be reasonable, the local real-time clocks have to be closely synchronized. With protocols like NTP [COULOURIS ET AL. 1994], p. 294, which is used in the Internet, a reasonable clock synchronization can be achieved. However, with this assumption certain guarantees regarding the semantics of events may no longer apply, e.g. concerning the actual sequence and causality of events.

The problem of global time is not the only problem in distributed scenarios. Network connections may have different latencies, so event notifications may overtake each other. What is a reasonable amount of time to wait for event notifications pertaining to events that should have taken place before other events for which event notifications are already available? After a given amount of time, delivering event notifications for the combined event may no longer make sense. This amount of time may depend on the type of event, so it should be specified separately for each type of event.

When using predicate constructors as described in Subsection 2.5.2 for defining complex events, the semantics of the constructors also have to be carefully defined. We will consider the sequence constructor as an example. The event to be detected is described as `(P1;P2)`. What happens, if multiple events that satisfy `P1` occur, before an event satisfying `P2` occurs? The first option is to evaluate in chronological order so the first event counts. The second option is to choose the most recent event. The third option is to accumulate the events satisfying `P1` in some way or another. There is even a fourth option, that for each event satisfying `P1`, an event satisfying `(P1;P2)` is reported.

This section has shown that the questions regarding event semantics have to be thoroughly investigated in the future. The detection of global predicates is a hard problem, but making reasonable assumptions may lead to solutions that are reasonable in practical scenarios.

2.7 Summary

In this chapter we have defined an event terminology and a simple event observation/notification model. On this basis we have discussed the important characteristics of event-based communication. They can be summarized by saying that event-based communication is source-initiated, asynchronous and anonymous, and therefore it is ideally suited for loosely coupling software components.

The next step was to look at how event-based communication is utilized in a wide range of other areas, including programming languages, publish & subscribe services, internet-scale event frameworks and active databases. In each case, the terminology and the underlying conceptual model was contrasted with our definitions in the first part of the chapter. Then, components of a language for describing events were presented. Finally a short overview of event semantics and the related problems was given.

The next chapter will look at aspects of mobile computing and their influences on event-based communication.

Chapter 3

Events and Mobility

Traditionally, computers have been rather large stationary devices. In recent years, computer hardware has become smaller and more powerful, making it possible to put computing hardware into small cases that can be carried around. The first step in that direction were portable computers that could be unplugged on one site, taken somewhere else, and plugged in again on the new site. Mobile computing goes one step further, allowing the computer to be in operation while on the move.

The same applies to being connected to a network. Traditionally, computers had to be plugged into static networks, where nodes are physically connected. With the availability of wireless networks, it is possible to be connected to a network while being on the move. Wireless networks have access points as well, but a change of the access point should be transparent to the user. There should be no disruptions of computing activities. This transparent change of network access while the user is changing his or her location is called *roaming*.

In the given context we are especially interested in a distributed system with a backbone of stationary servers that are connected by a physical network. Mobile computing devices are connected to this backbone through wireless connections. Typical mobile devices in this context are personal digital assistants (PDA), (sub-)notebook computers or wearable computers.

PDAs are about the size of a human hand, and the display covers most of the front part. The PDA may have a couple of buttons, but the main input device is a pen. Text can be entered with the pen in form of stylized characters that are recognized by the PDA. Examples for PDAs are the Palm [PALM 2000] or HP's Jornada [HP 2000].

Notebooks are about the size of an A4 paper with a height of up to five centimeters. Regarding processor, RAM and harddisk they are comparable to desktop computers. As input devices they have a keyboard that is slightly smaller than a regular keyboard, and either a touchpad or a ballpoint that replaces the mouse. As an output device, they have a flat LCD display.

*Subnotebook*s are smaller and lighter than ordinary notebooks. This also means that they have smaller keyboards, smaller screens and often less computing power. An example for a subnotebook is Sony's Vaio [SONY 2000].

Wearable computers are computers that can be worn on the body. They typically have a head-mounted display or a wrist mounted display that allows the operation of the computer while performing other tasks. As input they may utilize speech recognition or special input devices like chording keyboards [BAUER 1998]. An example for a wearable computer is Xybernaut's MA IV [XYBERNAUT 2000].

As the title of this thesis may suggest, the computing devices discussed so far are operated by human users. However, more and more other mobile entities like cars, busses, boats etc. are equipped with computers that may soon have wireless connections. In Nexus, all these mobile entities are modelled as mobile objects.

In the following, we will first look at different aspects of mobility. In the second section, the influence of these aspects on event-based communication will be discussed.

3.1 Aspects of Mobility

There are certain aspects of mobile computing that are inherently different from the stationary case. In [SATYANARAYANAN 1996] mobile computing is characterized by four constraints, three of which are of interest in the context of this thesis:

1. Mobile elements are resource-poor compared to static elements.
2. Mobile connectivity is highly variable in performance and reliability.
3. Mobile elements rely on a finite energy source.

An additional constraint pointed out by the author refers to the fact that mobile computing equipment is more vulnerable to theft, loss or damage, which is not part of our considerations here. Some of the following points are taken from [SATYANARAYANAN 1996], [BADRINATH ET AL. 1993] and [FORMAN & ZAHORJAN 1993].

Wireless Communication. As with stationary networks, wireless networks can be characterized by the technology used, the area covered and the maximum bandwidth that can be achieved. Table 3-1 lists some typical examples to give an idea of what is currently available.

Table 3-1: Wireless Network Technologies

Name	Technology	Type of Network	Bandwidth
IrDA	Infrared (IR)	LAN (1m)	9.6 kbps - 4 Mbps
Bluetooth	(Near Field) Radio	LAN (10m -100m)	< 1 Mbps
802.11b	Radio	LAN	11 Mbps
Metricom	Radio	MAN	28.8 kbps
GSM	Cellular Phone	WAN	9.6 kbps
UMTS	Cellular Phone	WAN	2 Mbps

What all the listed technologies have in common is that the available bandwidth is smaller than that of comparable stationary networks, e.g. Ethernet (10-100 Mbps). Wireless networks are broadcast networks that mostly divide up the total coverage area into cells with one access point each.The actual available bandwidth decreases if the number of mobile devices communicating increases, because the network bandwidth has to be shared among the users in a cell.

In addition, the surrounding environment interacts with the signal, leading to higher error rates and frequent spurious disconnections. This in turn leads to a higher communication latency due to retransmissions, time-out delays and error control protocol processing.

Change of Location. So far, we have looked at wireless communication at a given location. The idea of mobile computing is that the user can change his or her location. Even though changing the access point should be transparent to the user, he or she may experience highly variable network connections depending on the respective location. The user may even leave the area covered by a network, which leads to a long-term disconnection that cannot be hidden from the user.

It is also possible that the user has access to different networks at the same time. These networks can differ in coverage, bandwidth and cost. Since wireless wide area networks are often expensive and have a low bandwidth, an info station concept might make sense. Info stations offer a local wireless connection with a high bandwidth. They are placed at certain strategic locations, so that users have good network access there. Information that may be needed later can be hoarded at the info station. This information is available when the user is passing through areas that otherwise could only be covered by an expensive wireless wide area network with low bandwidth.

Traditional network protocols like IP route packages according to the network address. This concept relies on the fact that the address of a device contains information about the sub network on which it can be found. With mobile devices that change network access points all the time, this is no longer the case. They would need a new address for each access point. On the other hand, higher level protocols like TCP rely on a fixed IP address. Mobile IP (see Appendix B) offers a solution to this problem.

Finite Energy Source. The limited battery life-time is a major problem for all mobile devices. The more computational power a device has, the more energy is needed. Battery life-time for a PDA may be measured in days, but for a notebook computer it is only a couple of hours. Battery technology will improve over time, but energy consumption will remain an important issue. Therefore, hardware and software must be sensitive to power consumption. Unnecessary computational overhead has to be avoided.

Limited Computing Resources. For mobile devices, there are other requirements than for stationary computers. Portability, especially weight, size, robustness and ergonomics have to be taken into consideration. This, together with energy considerations (see above), usually results in lower processor speed, smaller memory size and less hard disk space than in desktop computers. Mobile components improve over time, but they lag behind compared to stationary elements.

Following from the constraints of mobile computing, Satyanarayanan [SATYANARAYANAN 1996] has identified a trade-off between autonomy and interdepence in mobile computing: Because of the resource poverty of mobile computers, they should rely on static servers for complex computations. Because of unreliable and low-performance wireless networks, they should be self-reliant. Since these goals contradict each other, a reasonable balance has to be found, preferably one that adapts to the current conditions.

3.2 Event-Based Communication for Mobile Devices

After having looked at some general characteristics of mobile computing in the last section, we will now focus on mobility in the context of event-based communication. The first part looks at the positive aspects of event-based communication for mobile computing, the second part discusses the related problems.

3.2.1 Advantages of Event-Based Communication

As we have seen in Chapter 2, event-based communication relies on push communication, so the initiative is on the server side and communication only takes place when necessary. In addition, the event notification itself is only a short message, so the limited bandwidth of the wireless connection is adequately considered.

Another interesting point that has not been mentioned so far is that sending on a wireless network consumes more power than receiving. In case of event notifications, the sender is on the server side that does not have the same energy constraints as the client on the mobile device.

In order to save energy, mobile devices often go into a power-save mode when they are not currently in use. In the power-save mode computational activity is reduced to a minimum. Mobile devices can be woken up from the power-save mode by an incoming event notification. Bluetooth is an example for such a technology [BLUETOOTH 2000]. If implemented accordingly, the amount of energy used for being informed about changes can be reduced to a minimum, if it is based on event-based communication.

In the given event model, the complexity regarding the detection of changes is mostly on the side of the Nexus services, so the use of computing and energy resources of the mobile device are also minimized in that respect.

In summary, event-based communication can reduce computational overhead and energy consumption of a mobile device in a scenario of mobile devices connected to a stationary backbone network with servers.

3.2.2 Related Problems and Disadvantages

An unreliable wireless connection can present some serious problems for the delivery of event notifications to mobile clients, but also for the observation of events involving mobile objects.

Observation of Events. The detection of an event can involve attributes of mobile objects, e.g. its location. If the location is determined by a GPS system that is attached to the mobile object itself, current information about the location of an object may not be available in case of a disconnection. Since disconnections are relatively frequent in wireless networks, events may go undetected. If an event cannot be detected reliably, it has an influence on the semantics of the event. It makes a difference if the system can determine whether it can currently observe an event or not. If it can be determined, interested clients can be informed that the observation of the event is temporarily not possible. Otherwise, the system cannot give any guarantees about the observation of this event at all.

Delivery of Event Notifications. One of the most important questions regarding the delivery of event notifications is the desired level of reliability.

If a simple best-effort semantic is sufficient, an unreliable network is not really a problem. This may be the case for some classes of events, especially, if the observation of the event itself cannot be guaranteed. On the other hand, it may not be intended that some mobile objects receive the event information and others do not.

Then, there may be other classes of events, e.g. accident warnings that should be delivered reliably. If reliability is desired, the question remains what should be done in case of long-term disconnections. There may be several attempts to deliver the event notification, but after a given

amount of time it may no longer be useful. Furthermore, event notifications to be delivered should not “pile up”, so that eventually the delivery of event notifications is paralyzed by old event notifications that have to be delivered reliably, but cannot be delivered at the moment.

So, depending on the system to be supported, event classes with different delivery requirements have to be identified. For each of these classes fault semantics have to be defined describing what should be done in case of failures.

The combination of the asynchronous nature of event-based communication and the unreliability of network connections can also be seen as a disadvantage of event-based communication in the context of mobile computing. When querying the system using pull-style communication, the communication is mostly synchronous. The client either receives the information or is aware of a current disconnection. However, in the case of event-based communication, the client registers with the system for event notifications some time in advance. Later, it does not know whether it should have received an event notification during a time of disconnection or not.

Chapter 4

Spatial Events

This chapter will focus on spatial events. In Chapter 2 we have defined an event as an observable change in the state of a model. Such an event can be described in form of predicates expressing relations. An event occurs when the predicate becomes true. Spatial events are events based on the absolute or relative position of objects in space.

4.1 Foundations for the Definition of Spatial Events

In this section we will lay the foundations for describing relevant spatial relations. Important aspects are the dimensionality of the underlying model, a coordinate system and possible general relations between objects.

4.1.1 Dimensions

The concept of space implies the existence of spatial dimensions. The world as we see it has three spatial dimensions, so real objects in its space are always three-dimensional. Time is often considered as an additional dimension. For the purpose of modelling certain aspects of the world, it often makes sense to abstract from the three or four-dimensional case and make a projection into a space of lower dimension.

[STREIT 1999] distinguishes the following object types and their properties:

- 0 - dimensional (0D) objects: points. They have neither length nor surface area.
- 1 - dimensional (1D) objects: lines. They have a length, but no surface area.
- 2 - dimensional (2D) objects: areas. They have a surface area, but no volume.
- 3 - dimensional (3D) objects: solids. They have a volume.

Both 2D and 3D digital models of the world exist. Most maps are two-dimensional projections modelling certain geographical aspects of the real world. It is also possible to have a 2D base model with an additional height information for objects. This is often called a 2.5D model. It is not full 3D, because complex 3D shapes of objects, e.g. spheres, cannot be modelled. A system like Nexus may support all three types of models, whatever data is available.

In principle, the world could also be modelled in a full-scale 4D-model. In this case the history, e.g. the movement of objects in space would become part of the model. The present would always be “on the edge” of the model, because there is no means available to look into the future.

The main practical problem of such a model is the enormous amount of data. Currently, it is only possible to have information about discrete points in time, not the seemingly continuous infor-

mation we experience. All this does not matter much as long as we look at the model from a theoretical perspective. The more important point in this respect is that operations in 4D space may be hard to understand. In the following, we will therefore consider time as a separate concept.

We will abstract from the spatial dimensions wherever possible. Most aspects and relations we will discuss in Subsection 4.1.3 and Subsection 4.1.4 are valid for both the 2D and 3D case. Some of the more specialized relations imply a 3D model. There are also some 2D relations that make sense with a 3D-model, in addition to the 3D relations, e.g for expressing a relation between a mobile object and an area.

4.1.2 Coordinate System

In order to describe the position of objects in space, we need a coordinate system. The most popular type of coordinate system for everyday use is the cartesian coordinate system. In a cartesian coordinate system, pair-wise orthogonal axes intersect at the origin and have equal units of measurement.

Cartesian coordinate systems can be defined in n-dimensional space. An n-tuple of real numbers uniquely describes a point in space. The tuple can also be interpreted as a vector relative to the origin of the coordinate system. In a 2D or 3D coordinate system, the coordinate axes are often labelled x, y and z respectively.

As an alternative to cartesian coordinates, polar or spherical coordinates can be used. In this case a point in space is uniquely defined by giving degrees relative to the coordinate axes and a distance from the origin.

The coordinates for a point on the surface of the earth can either be given in degrees relative to the equator (latitude) and relative to the zero meridian (longitude) or, alternatively, a 3D geocentric coordinate system can be used with the center of the Earth as the origin.

Another important point that has to be taken into consideration in connection with a coordinate system for the planet is that the earth is not a perfect sphere, but a spheroid, where the length of the main axes differ a little.

A three dimensional system that is commonly used for navigation is the global WGS84 coordinate system. WGS84 is also the basis for the Global Positioning System (GPS). It is defined as follows [EUROCONTROL 1998]:

Origin = Earth's center of mass.

Z-Axis = Polar Axis.

X-Axis = Intersection of the zero meridian plane and equator plane.

Y-Axis = Completes a right-handed, Earth Centered, Earth fixed, orthogonal coordinate system.

For maps, the surface of the earth has to be projected onto a 2-dimensional coordinate system. In Germany, the Gauß-Krüger coordinate system is commonly used [STREIT 1999]. The surface of the earth is projected onto the surface of a cylinder. This is done regionally, in sectors of 3° relative to the main meridians. For Germany, the relevant meridians are 6°, 9°, 12° and 15° longitude. In each sector, the Y-axis points north, along the main meridian. The distance values are given relative to the Equator. In order to avoid negative values on the X-Axis, each main meridian gets the value 500,000 (meters). The degree value of the respective main meridian divided

by three is appended to the front of the value, e.g. a coordinate could be X = 3 593,571.20 (relative to the 9˚ meridian) and Y= 5,902,863.21 (relative to the Equator).

The UTM-System (Universal Transversal Mercator Projection) is a similar coordinate system that is used for US and military maps.

The different coordinate systems might be important in the Nexus context, because the GPS system works with WGS84 coordinates, whereas some of the geographical data is given in Gauß-Krüger coordinates. For spatial relations and spatial events in general, it is only important to know that events may be defined relative to coordinate systems.

4.1.3 Position of Objects

The position of an object can be described in one of the three following ways:

a) Position of mobile objects in space (Figure 4-1a)
This is defined relative to a coordinate system.

b) Position of mobile objects relative to a static object (Figure 4-1b)
Since the static object has fixed coordinates, this is basically the same as “position of an object in space”, with an offset to a coordinate system.

c) Position of mobile objects relative to each other (Figure 4-1c)
In this case, a coordinate system may be defined relative to one mobile object.

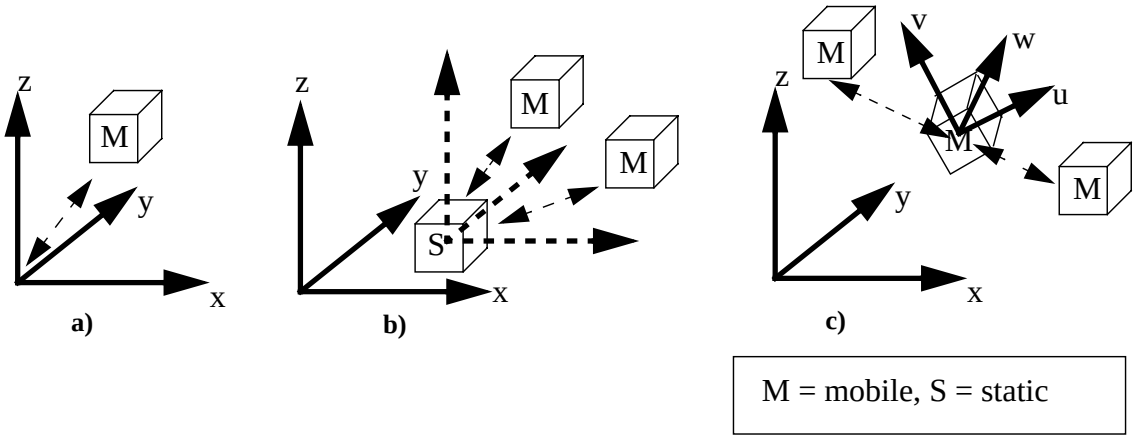


Figure 4-1: Position of Objects
a) mobile object relative to coordinate system
b) mobile objects relative to static object
c) mobile objects relative to mobile objects

4.1.4 Spatial Attributes and General Relations

Given a coordinate system, the positions of objects relative to each other can be defined by the following attributes (Figure 4-2):

a) Distance

b) Direction

c) Orientation of the objects

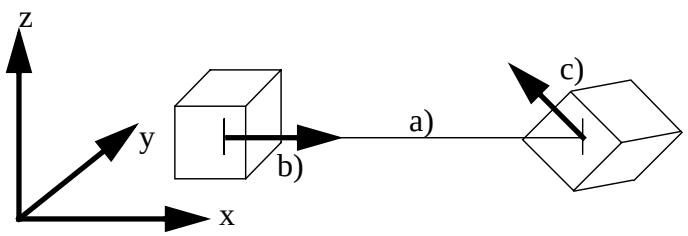


Figure 4-2: Spatial Attributes

These attributes do not necessarily take the extent of an object into account. The position of an object only needs to be given as a single coordinate. This coordinate may be interpreted as the center of the object, whether this coincides with the geometrical center of the object or not. Such an approximation may be reasonable for small objects, but insufficient for larger objects. In those cases the extent has to be taken into account, e.g. it might make sense to measure the distance between objects as the distance between the edges of the objects that are closest to each other.

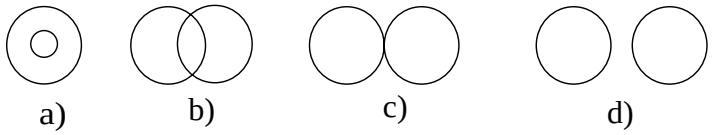


Figure 4-3: General Spatial Relations

The following relations between objects only make sense if the extent is taken into consideration (Figure 4-3):

a) Inclusion: One object is completely within another object.
Example: Jim is in the car.

b) Intersection: two or more objects intersect each other.
Example: A car is halfway in the garage.

c) Tangent: Two object are tangent to each other. This is a special case of intersection, where the points of intersection all lie on a given surface.

d) Disjoint: Two or more objects are disjoint. There is no point of intersection.

In many cases, it is not the extent of the object itself that is of interest, but an extension around the object, i.e. not only the object itself, but also a certain area around the object. This could be represented by a pseudo-object with the respective extent and shape around the given object. For example it might be of interest if somebody comes within five meters of a building, which might be modelled as an intersection or inclusion with the pseudo-object.

If time is taken into account, the movement of objects can also be observed. The related attributes are:

- Change of orientation
- Direction of movement
- Speed
- Change of speed (acceleration/deceleration)

Depending on the granularity and accuracy of the location information, it may be hard or even impossible to determine whether two objects are in a certain relation to each other. This has to be taken into account when defining predicates describing events that can be observed by clients.

4.2 Spatial Relations

In this section we list spatial relations at the most general level, giving an idea of what the interesting relations regarding one or two objects could look like. It is always possible to extend this list with more relations describing the relationship of multiple objects in space or their movement relative to each other. Subsection 4.2.3 gives some examples of more specialized spatial relations that can be derived from the more general relations, but may provide more intuitive concepts. As mentioned above, there is a difference, between an object that is regarded as a single coordinate and an object whose extent is also considered. We look at the second alternative only in exemplary cases.

4.2.1 Relations

- Distance
  - *distance(object$_1$, object$_2$, rel, value)*
    The *distance* relation holds if the distance between *object$_1$* and *object$_2$* is *rel*={smaller than, equal to, greater than,...} a given *value*.
- Direction
  - *direction(object$_1$, object$_2$, vector)*
    The *vector* gives the direction relative to a coordinate system with respect to the location of *object$_1$*. The length of the *vector* can be ignored, because the distance between the objects is not included in the relation. Alternatively the direction can be expressed in degrees relative to the axes of the coordinate system. The relation holds if *object$_2$* can be found in the given direction from *object$_1$*.
- Orientation
  - *orientation(object, vector)*
    The *vector* describes the orientation of the *object*, i.e. the direction in which the *object* is facing relative to a coordinate system. In order to determine the orientation, a normal vector for the object has to be defined.
    This kind of orientation relation is sufficient in many cases, but does not determine the full orientation of an object in three-dimensional space, since the object can also turn around the normal vector axis. For a complete description of the orientation, the normal vector has to be extended to a complete coordinate system for the object.
- Inclusion
  - *in(object$_1$, object$_2$)*
    The *in* relation denotes that *object$_1$* is within *object$_2$*. This can be interpreted in two different ways (Figure 4-4). In the first case, the relation holds if the location (given as a coordinate) is within *object$_2$*. In the second case, the full extent of the object is taken

into account. This information is not available if the location of an object is represented by a single coordinate. We split this scenario into two relations: the *in* relation refers to the first of the above cases, whereas the *in_extent* relation describes the second case.

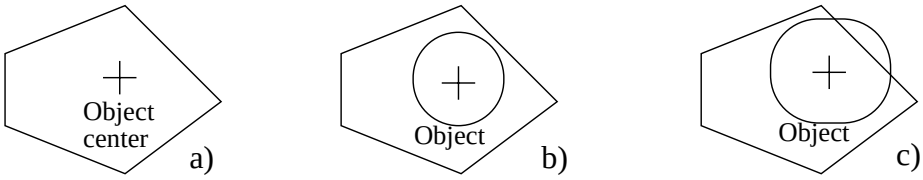


Figure 4-4: in and in_extent Relation
a) in relation holds
b) in_extent relation holds - in relation holds
c) in_extent relation does not hold - in relation holds

- Intersection
  - *intersection(object$_1$, object$_2$)*
    The *intersection* relation holds if the extent of *object$_1$* and the extent of *object$_2$* intersect.
- Tangent
  - *tangent(object$_1$, object$_2$)*
    The *tangent* relation holds if the extents of the two objects are tangent to each other, i.e. they touch each other. To determine if this relation holds, very exact location information or special sensors are needed.
- Disjoint
  - *disjoint(object$_1$, object$_2$)*
    The *disjoint* relation holds if the extents of *object$_1$* and *object$_2$* do not intersect.

The following relations concern the movement of objects. Determining any kind of movement implies a concept of time. As we have discussed in Subsection 4.1.1, time is considered as a concept that is separate from the other three dimensions. In order to determine that an object has moved, the state of an object at a previous time and the state at the current time have to be compared. Therefore, keeping the old state of an object is an implicit basis for those relations.

- Change of Orientation
  - *change_of_orientation(object, degree_value)*
    The *change_of_orientation* relation holds if the degree between the old and the new vector defining the orientation of the *object* becomes equal or greater than the given *value*.
  - *turn_around_axis(object, vector, rel, value)*
    The *turn_around_axis* relation holds if the *object* has turned around a certain axis (the *vector*) by *rel* = {less than, equal to, more than} *value* degrees.
- Direction of movement
  - *direction_of_movement(object, vector)*
    The *direction_of_movement* relation holds if the *object* is moving in the direction indicated by the given *vector.*
  - *direction_of_movement_relative_to_vector(object, vector)*
    The *direction_of_movement_relative_to_vector* relation holds if the vector describing

the direction in which the *object* is moving can be decomposed into one component that is parallel to the *vector* given in the relation and pointing in the same direction, and another component.

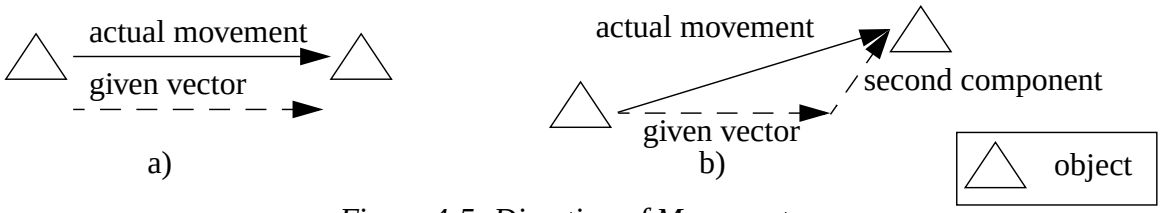


Figure 4-5: Direction of Movement
a) direction_of_movement(object, vector)
b) direction_of_movement_relative_to_vector(object, vector)

Whereas the time component in the previous two relations is implicit, it becomes explicit in the following two relations, e.g. speed is defined as the distance covered in a certain time interval.

- Speed
  - *speed(object, rel, value)*
    The *speed* relation holds if the given object is moving with a speed rel = {less than, equal to, more than} *value* km/h (or another suitable unit).
- Change of speed
  - *change_of_speed(object, rel, value)*
    The change_of_speed relation holds if the change of speed for the given *object* is *rel* = {smaller than, equal to, greater than} the given *value*. A change of speed is either an acceleration or a deceleration. We will look at it again in the context of more specialized relations in Subsection 4.2.3.

4.2.2 Additional Attributes

So far we have defined the relations in a theoretical manner. In order for them to be useful in a real-world scenario, some additional issues have to be addressed.

Accuracy. The accuracy of the sensor information and the implementation of the corresponding observation implicitly determine the minimum accuracy that can be guaranteed when it comes to determining if a certain relation holds. For example, it might only be possible to determine the speed of a moving object with an accuracy of +/- 1 km/h.

This information is very important for the client interested in an event that is based on a certain relation or a combination of relations.

Deviation. For the relations in Subsection 4.2.1 we have implicitly assumed that values such as vectors can be specified precisely. However, we have seen in the previous paragraph that the accuracy of the sensor information may not be as good as we would like. In addition, we often do not want to know, if an object is precisely in the direction given by a vector, but maybe if it is approximately in that direction. The solution for the problem is to specify a deviation.

The direction relation will serve as an example for deviation. The deviation can either be given as a relative deviation as in the first case or as an absolute deviation as in the second case:

- *direction(object$_1$, object$_2$, vector, relative_deviation)*
  The *vector* gives the direction relative to a global coordinate system with respect to the location of *object$_1$*. The *deviation* determines by how many degrees the object can be off the given direction.Together, *vector* and *deviation* define a cone with infinite reach (Figure 4-6). The relation holds if *object$_2$* or the location coordinates of *object$_2$* are within the cone.

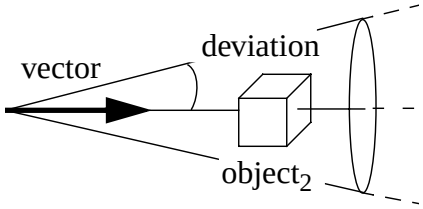


Figure 4-6: Relative Deviation

- *direction(object$_1$, object$_2$, vector, absolute_deviation)*
  Instead of the relative deviation in the *direction* relation, an *absolute deviation* is given. Therefore the *vector* and the *deviation* form a cylinder instead of a cone (Figure 4-7). The relation holds if *object$_2$* or the location coordinates of the *object$_2$* are within the cylinder.

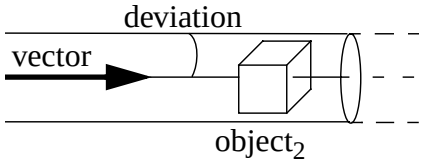


Figure 4-7: Absolute Deviation

A deviation attribute can be added to most of the relations involving direction or orientation. For the other relations an explicit deviation does not make sense. A potential problem with the accuracy of the location information can only be addressed by choosing reasonable values. For example, a meeting of mobile objects could be described by a distance relation, so that a meeting takes place, if the distance between the mobile objects is less than three meters. If the location information is only accurate to +/- one meter, the distance value describing the meeting should be set to four meters instead if all meetings should be covered, expecting that some non-meetings might also be covered by the relation.

Coordinate System. As discussed in Subsection 4.1.2 and Subsection 4.1.3, spatial relations are defined relative to a coordinate system. This can be either a global coordinate system that is defined for the whole model or a local coordinate system that is defined relative to some object.

For some relations, it makes sense to define them either relative to a global coordinate system or relative to a local coordinate system. In this case, the coordinate system can be given as an extra parameter. As an example, we will look at the direction relation (Figure 4-8).

a) *direction(object$_1$, object$_2$, vector, global_coordinate_system, deviation)*
   In this case, the *vector* is given relative to a *global coordinate system* (thick arrows), e.g. approximately north-east as seen in the figure.

b) *direction(object$_1$, object$_2$, vector, local_coordinate_system, deviation)*
   In this case, the *vector* is given relative to a *local coordinate system* that is defined relative to the orientation of *object$_1$* (thin arrows). The *vector* in Figure 4-8b is pointing approximately north-east, relative to the *local coordinates*.

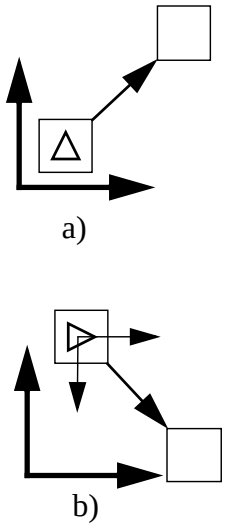


Figure 4-8: Direction Relative to Coordinate Systems

4.2.3 Specialized Relations

The relations we will discuss in this subsection are all specializations of relations listed in Subsection 4.2.1. If the more general relations already cover those cases, why should we bother to look at the specializations at all? The reason is that specializations may offer a simpler concept to the user, which may be more convenient or easier to implement.

- Direction
  - *north(object$_1$,object$_2$), east(object$_1$,object$_2$), south(object$_1$,object$_2$), west(object$_1$,object$_2$), south-west(object$_1$, object$_2$),...*
    Instead of variable vector directions, fixed directions relative to a given coordinate system can be defined, e.g. *north(object$_1$, object$_2$)* means that *object$_2$* is to the north of *object$_1$* relative to a given coordinate system.
  - *below(object$_1$, object$_2$), above(object$_1$, object$_2$)*
    In a 3D model, the additional direction *below* and *above* can be defined relative to a given coordinate system. This is an example where a coordinate system has to be used that is appropriate for the given location. Taking a cartesian coordinate system with the center of the Earth as its origin, the relations may not produce the expected results for most locations on the globe.
- Change of speed
  - *acceleration(object,value)*
    The *acceleration* relation holds if the given *object* is accelerating faster than a given *value*.
  - *deceleration(object, value)*
    The *deceleration* relation holds if the given *object* is slowing down faster than a given *value*.

4.3 Spatial Events

As mentioned before, an event denotes a change. The change can be described in form of a relation that becomes true. So far we have tried to keep a one-to-one correspondence between a relation and a single aspect. Some interesting events, however, may consist of a combination of different aspects.

The name of a predicate describing an event should reflect the character of “change” and express the result of this change, i.e. that a notification is sent. The predicates in Table 4-1 describe spatial events that are the result of the movement of one or more mobile objects.

Table 4-1: Predicates Describing Spatial Events

Predicate	**Description & relations involved**
onMeeting*(object$_1$, object$_2$,..., object$_n$, distance)*	The *onMeeting* predicate describes an event where two or more mobile objects are closer together than the given distance. The main underlying relation is *distance.* This is not sufficient however. For example, objects may be close together in a building, but on different floors. Therefore the height part of the position information has to be considered. This can be expressed with the *tangent* and the *direction* relation. In practical situations, other objects, e.g. walls between the different mobile objects may also have to be considered.
onCrossing*(object$_1$, object$_2$)*	The *onCrossing* predicate describes an event where a mobile object (*object$_1$*) crosses another (pseudo-)object (*object$_2$*), i.e. a line in the 2D case, and a plane in the 3D case. The underlying relation is *intersection*.
onEnter*(object$_1$, object$_2$)*	The *onEnter* predicate describes an event where a mobile object (*object$_1$*) enters another object (*object$_2$*), an area in the 2D case, a solid in the 3D case. The underlying relation is *in*.
onLeave*(object$_1$, object$_2$)*	The *onLeave* predicate describes the inverse event to the event described by the *onEnter* predicate. The underlying relation is *disjoint*.
onOrientation*(object, orientation, coordinate_system, deviation)*	The *onOrientation* predicate describes an event where a mobile object reaches a certain *orientation* given in form of a vector. The vector is relative to the given *coordinate system*. A value for *deviation* defines how far the actual orientation vector can differ from the given vector so that the event is still reported. The underlying relation is *orientation*.
onObjectInDirection*(object$_1$, object$_2$, direction, coordinate_system, deviation, distance)*	The *onObjectInDirection* predicate describes an event where *object$_2$* is located in a certain direction relative to *object$_1$*. The *direction* is given in form of a vector. The vector is relative to the given *coordinate system*. A value for *deviation* defines how far the actual direction vector can differ from the given vector so that the event is still reported. A maximum *distance* can be given, so that only events are reported where *object$_2$* is closer to *object$_1$* than the maximum *distance*. The underlying relations are *direction* and *distance.*
onTouch*(object$_1$, object$_2$)*	The *onTouch* predicate describes an event where *object$_1$* touches *object$_2$*. The underlying relation is *tangent*.

The predicates in Table 4-2 describe spatial events that describe the movement of one mobile object. In order to determine the movement of objects reliably, the granularity and accuracy of the location information must be high. For example, a cell-based indoor-positioning system may not be sufficient to detect some of the events concerning movement.

Most of the predicates in Table 4-2 have an optional *interval* parameter that describes the time interval in which the event has to take place in order to be observed. For example, an event described by *onChangeOfOrientation(OBJECT5,5 ,1h)*occurs only, if the orientation of the object changes by 5 or more degrees within one hour. If the parameter is left out, an infinite maximum time interval is assumed.

Table 4-2: Predicates Describing Events Involving the Movements of Objects

Predicate	**Description & relations involved**
onObjectMovingInDirection*(object, direction, coordinate_system, deviation, time_interval)*	The *onObjectMovingInDirection* predicate describes an event where a mobile *object* is moving in a given *direction*. The *direction* is given in form of a vector relative to the given *coordinate system*. The *deviation* defines how much the direction of movement can differ from the given *direction* so that the event is still reported. The underlying relation is *direction_of_movement*.
onObjectMovingRelativeToVector*(object, direction, coordinate_system, time_interval)*	The *onObjectMovingRelativeToVector* predicate describes an event where a mobile *object* is moving in the *direction* given in form of a vector that can be decomposed into two components, one of which is parallel to the given vector. Both vectors are relative to the given *coordinate system*. The underlying relation is *direction_of_movement_relative_to_vector.*
onChangeOfOrientation*(object, degree, time_interval)*	The *onChangeOfOrientation* predicate describes an event where a mobile *object* changes its orientation by more than the given number of *degrees*. The underlying relation is *change_of_orientation.*
onTurnAroundAxis*(object, vector, coordinate system, rel, degree, time_interval)*	The *onTurnAroundAxis* predicate describes an event where a mobile object turns around a vector by rel = {less than, equal to, more than} the given number of *degrees*. The underlying relation is *turn_around_axis*.
onSpeed*(object, rel, speed)*	The *onSpeed* predicate describes an event where a mobile object is moving rel = {faster than, as fast as, slower than} a given speed. The underlying relation is *speed*.
onChangeOfSpeed*(object, rel, value, time_interval)*	The *onChangeOfSpeed* predicate describes an event where a mobile object changes its speed rel = {less than, equal to, more than} a given value. The underlying relation is *change_of_speed*.

The events described in this section are independent of any actual system. In Chapter 8 we will look at a subset of spatial events that are viable in the Nexus context, and how they can be observed.

Part II: Design for Nexus

Chapter 5

Nexus

Whereas the first part of this thesis was concerned with the general event concept, this second part will present a high-level design for an event service for the Nexus system. It will also discuss architectural issues that have to be addressed in this context. Before going into the details of the Event Service, it is necessary to have a good understanding of the underlying ideas, concepts and proposed components of the Nexus platform. Therefore, this chapter will give a short overview of the Nexus platform.

5.1 Idea

The goal of the Nexus project [HOHL ET AL. 1999] is to examine concepts and methods for supporting location- and spatial-aware applications with mobile users. Ultimately, a global open platform for location- and spatial-aware applications is to be developed. Based on a common world model, the platform should facilitate the cooperation and interaction between different applications. Most current location-aware applications implement their own basic services. Nexus should provide a common middleware on which future location-aware applications can be built.

An important field of application for the Nexus platform will be the realization of so-called situated information spaces. In situated information spaces, information is associated with real-world objects, such as streets, buildings or rooms, as well as with mobile objects such as vehicles or people.

5.2 Example Scenario

Before discussing the general concepts of Nexus, we will describe an example scenario. On the basis of a city information system some of the functionality supported by the Nexus platform will be presented. Having an idea about the potential functionality of the platform will help to understand the underlying concepts.

> “Welcome to Nexusville!”, Paul read on the display of his communication device, just after arriving at the airport. So this was the first city that had opted for the latest city information system, he thought.
>
> On his way to the subway station, the travel information guide for the public transport system appeared on the display. Straight to the city center - that was

> Paul's first destination - and he immediately bought the ticket, online, and the fare deducted from his credit card.
>
> Arriving in downtown Nexusville two hours before his business meeting, Paul decided to go for a walk. An interesting old building caught his eye. Pointing at it with his telefinger, he was informed that in former times this used to be the residence of the Duke of Finkenstein; now it was the headquarter of a champagne producing company. Remembering that he needed to get a present for his fiancée, Paul followed the link to the company's website and ordered a bottle of champagne, to be delivered to his home address.
>
> With the business meeting only half an hour away, Paul activated the navigation system that told him to take bus No. 20 at the next corner, get off two stations down the road, turn right into Church Street and his destination would be the third house on the left.
>
> After the meeting, Paul decided to go shopping in the new Cyber-Mall. Passing a shoe shop, the system reminded him that he wanted to get new shoes. In the old days he had always forgotten about such things. When he entered a department store, the store informed him about a current sale in the video department, all information tailored to his personal profile.
>
> Back to the city center, Paul passed the opera house. Since he was looking for evening entertainment, he approached the nearby virtual advertising column and checked for tonight's performance, finally settling on a ticket for the ballet. Before that, there would be enough time to have dinner, so he asked the system to find him the closest Chinese restaurant...

In this or a similar way, a future Nexus-based city information system might support visitors. In the context of this scenario, events have not been explicitly mentioned, but whenever the mobile communication device proactively informs our user Paul, this could be realized through events. The Nexus Event Service will be covered in detail in the next chapter.

The following other functionalities of the Nexus platform were utilized in the presented scenario:

- passive queries, e.g. to find the Chinese restaurant
- identifying an object by pointing at it, e.g. pointing at the old building
- virtual objects, e.g. the virtual advertising column at the opera house
- accessing external information systems, e.g. accessing the web site to order champagne
- distribution of information according to the location of the user, e.g. when entering the department store
- navigation, e.g. to get to the business meeting

In the following, we will look at the technology, the concepts and finally the components of the Nexus platform that support these functionalities.

5.3 Technology

A mobile Nexus user needs a mobile device in order to access the location-specific information provided by the Nexus platform. In the scenario above, we have called it communication device. The Nexus communication device could be a personal digital assistant (PDA), a notebook or

subnotebook computer, or even a wearable computer (see Chapter 3). In the future, other devices may become available, e.g. a combination of PDA and cellular phone.

The communication device needs a wireless connection to communicate with the server infrastructure that holds the information pertaining to the Nexus model. The available wireless technology has already been discussed in Chapter 3, so we do not go into details here. The communication device also has to have access to information about its current location, which can be delivered by tracking and positioning systems.

Tracking and Positioning Systems. There are two types of systems that can provide the current position of a mobile device. Positioning systems allow the mobile device to determine its own position with the help of one or more sensors. Tracking systems are systems that monitor a certain area and determine the location of objects within that area.

Another distinction regarding tracking and positioning systems is whether they are cellular or non-cellular systems. A cellular system determines the position of an object by determining which cell the mobile device is currently in. Therefore, the granularity of the location information is restricted to the size of a given cell. Non-cellular systems determine the position of an object directly.

The third practical distinction is whether the systems are used indoors or outdoors. Outdoors, a Global Positioning System sensor can be used. GPS is a satellite-based non-cellular positioning system. The GPS sensor determines its own position by analyzing the signals it receives from a number of satellites. The accuracy of the location information is within 20 meters of the actual location. A differential GPS can increase the accuracy of the location information to within a few meters or even centimeters. For a differential GPS, an additional terrestrial sender has to be within a certain distance.

Indoors, there is no generally available tracking or positioning infrastructures. In principle, systems based on infrared, such as the Active Badge system [WANT ET AL. 1992], and systems based on radio-frequency, such as RADAR [BAHL & PADMANABHAN 1999] exist, but installing them on a large scale is still too expensive. Indoor systems can be either positioning or tracking systems. Inertial sensors can help to improve the accuracy of the location information. A digital compass may be used for implementing the telefinger mentioned in the scenario above.

In summary, the basic technology for implementing the Nexus system is already available and the technological progress will help to improve the infrastructure for spatial-aware mobile systems like Nexus even further. The Nexus system should be designed in such a way that new technologies can be dynamically integrated when they become available.

5.4 Nexus Model

The Nexus platform is based on a model of regions of the physical world, which can be augmented with virtual objects [HOHL ET AL. 1999]. Of course, the virtual elements of such an *Augmented Area* can only be experienced through a Nexus application running on a mobile device. The internal representation of an Augmented Area in Nexus is called *Augmented Area Model* (Figure 5-1). An Augmented Area is spatially limited. Typical examples of Augmented Areas are cities, university campuses or buildings.

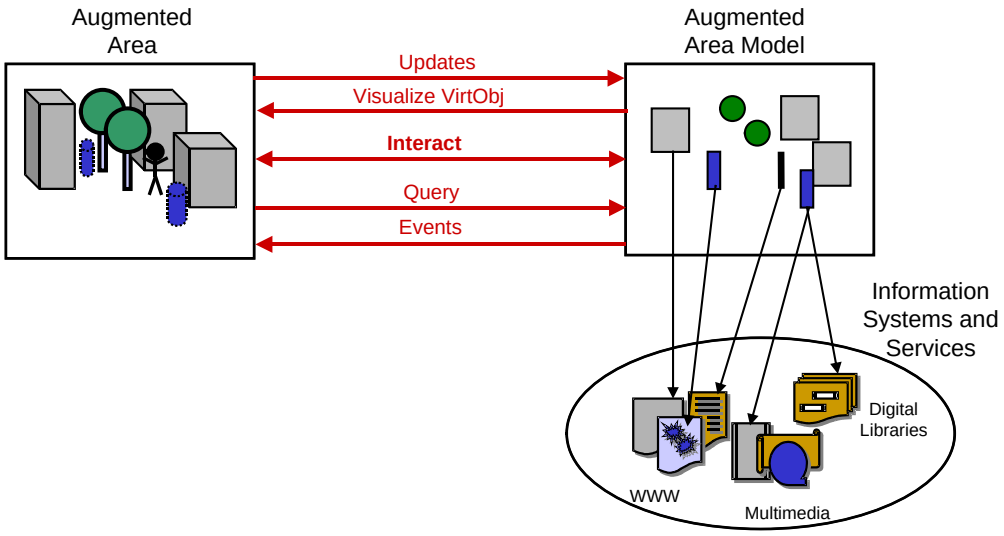


Figure 5-1: Interaction between Augmented Area and Augmented Area Model (taken from [COSCHURBA ET AL. 2000])

Augmented Areas are not regarded as isolated units. They can overlap and include each other, offering different levels of detail. Together they form an *Augmented World* (Figure 5-2). Since the Augmented Area Models are described in a uniform way, applications can switch between them easily, e.g. from the Augmented Area Model describing the university campus to the model describing the university library when the Nexus user is entering the library.

The physical objects that exist in the Augmented Area are modelled in the Augmented Area Model. The model includes static objects, i.e. objects whose location changes never or rarely, such as streets, buildings, rooms and walls, but also mobile objects like cars, trains and people.

Virtual objects in Augmented Areas have no equivalent in the real world. They serve as anchors in the Augmented Area to access external systems and services, e.g. a web server with information that is relevant at the given location.

Virtual objects can either be associated with a fixed location or with another object. In order to make it easier for the user to understand and handle virtual objects including their associated functionality, metaphors are used [BAUMANN ET AL. 2000]. Examples for such metaphors are virtual advertising columns and virtual Post-its. A virtual advertising column has a fixed location and a certain range of visibility, just

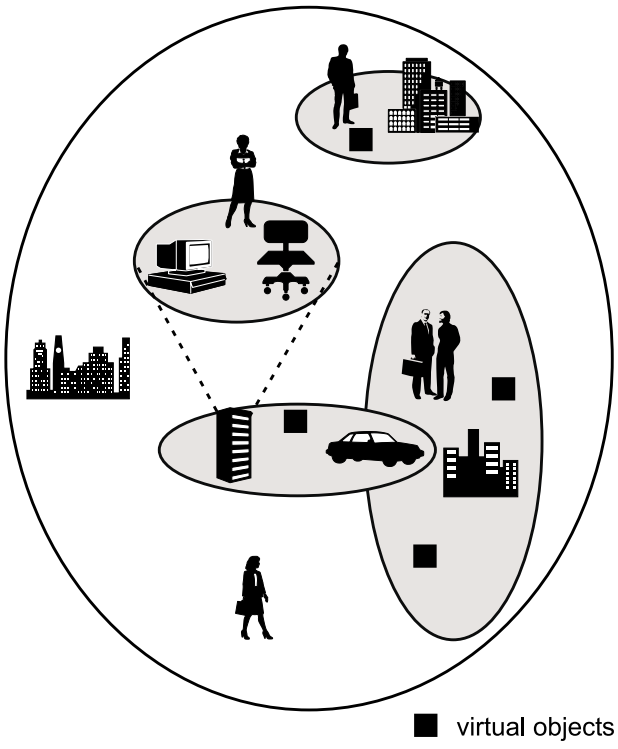


Figure 5-2: Augmented Areas (taken from [HOHL ET AL. 1999])

like a real-world advertising column. Information items can be attached to the virtual advertising column, and going beyond the functions of the real-world equivalent, external information services can be accessed. Virtual Post-its can be attached to other objects, static or mobile. Conforming to the metaphor, virtual Post-its carry only simple pieces of information.

Ideally, Augmented Area Models are “living models”, meaning that changes in the Augmented Area are automatically propagated to the model. The opposite case could also be supported, so that changes in the Augmented Area Model lead to changes in the Augmented Area. For this purpose, controllable actor elements have to be available.

5.5 General Design Objectives

The Nexus platform is intended as a global and open platform. This entails some general design objectives, which we will discuss in this section. The objectives have to be considered when designing the system as a whole, but also when designing the architecture of every single component. We will come back to this point, when describing the design for the Event Service in the following sections.

The most obvious objective is scalability. If Nexus is to be employed on a global scale, it has to accommodate any number of clients and huge amounts of data. This means any centralized components have to be strictly avoided, a distributed architecture is a necessity. Also, the system must be easily extensible, so that the infrastructure can be improved at any time, without disrupting the service. Ideally, the Nexus platform should be able to adapt to the current situation, e.g. by balancing the load between different servers.

A global platform basically implies a heterogeneous environment. Different technologies will be involved including all kinds of mobile devices, communication infrastructures, location and positioning systems etc. The interoperability between different technologies has to be accomplished. Another important aspect is the heterogeneity of the information sources the Nexus platform has to deal with. For different Augmented Areas, different types of data will be available with different levels of detail. Therefore, a uniform way to access heterogeneous data sources has to be defined.

Fault tolerance is another important objective, especially in an environment with wireless connections that are not as reliable as physical connections. Also Nexus components may have to be replicated to avoid single points of failure.

In the context of Nexus, security and privacy aspects are very important. The user should have control over who can access what kind of information about him. This is especially true for information regarding the location of mobile users. If privacy aspects are neglected, Nexus could easily be used as a basis for a “Big Brother” [ORWELL 1949] type of application.

The success of an open platform is often related to providing a simple, but powerful tool that everybody can use. This is exemplified by the success of the World Wide Web (WWW). Everybody can offer information. This should also be considered when designing Nexus components. In order for a platform to be accepted by the user, the efficiency of key mechanisms is extremely important.

5.6 Components

A coarse-grained architecture of the Nexus platform with its main components is shown in Figure 5-3, see also [COSCHURBA ET AL. 2000].

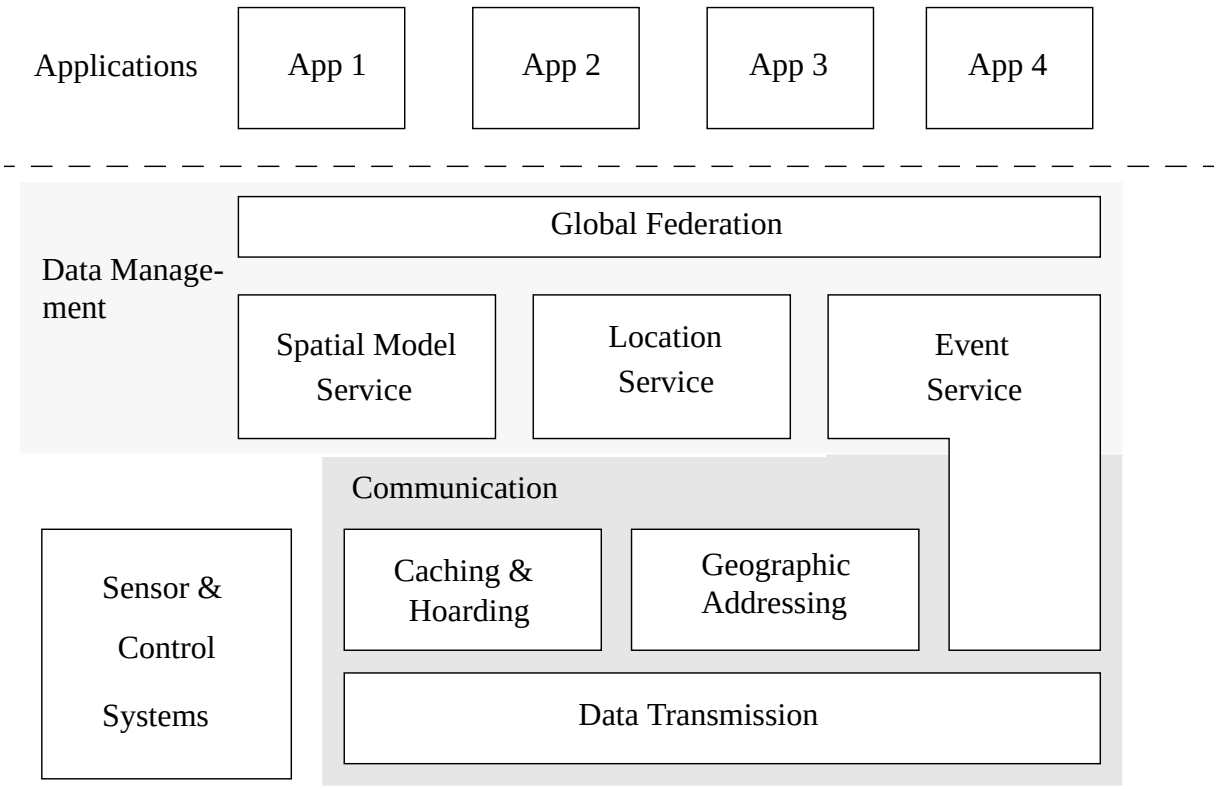


Figure 5-3: Nexus Architecture

The main components in this architecture fall into the areas of data management, communication and sensor & positioning systems.

The components in the area of data management store the augmented area models described in Section 5.4. The location data of mobile objects is highly dynamic. This entails different storage requirements compared to less volatile data. For example, the index structures that are typically used for efficient database access do not make sense in this case. The data changes so often that updating the index structures becomes inefficient. Therefore location data is managed by a special component, the Location Service.

All other objects, whose data is more static, are managed by the Spatial Model Service. This includes the data pertaining to physical objects, e.g. buildings or rooms, but also virtual objects like virtual advertising columns or virtual Post-its, including their links to external information sources.

The distinction between mobile and static objects is made transparent to the user by a global federation. It partitions queries involving both kinds of objects into separate queries for the Location Service and the Spatial Model Service, and integrates the partial results to return the combined result.

The communication between the different components is handled by a communication platform. The basis of this communication platform is the data transmission component. It integrates different wireless communication technologies by providing a single interface to them, allowing a seamless transition between different communication media. In addition, a certain quality-of-service level can be requested from the data transmission component. Renegotiations take place if this service can no longer be provided, e.g. because there was a switch from a high-bandwidth to a low-bandwidth communication medium.

A hoarding mechanism allows the Nexus client to download information in advance that will most likely be needed in the future. In this way temporary network disconnections can be overcome.

The geographic addressing component makes it possible to send messages to mobile objects in a given area. This means geographic addressing allows location-aware communication, which is a true extension to the conventional addressing schemes unicast, multicast and broadcast.

The Event Service is both a communication component and a data management component. It is a data management component, because observing complex events involves keeping state, which is data in a wider sense. In addition, it provides registered client applications with event information, so in that sense it is a data source. Obviously, it is also a communication component, because it is responsible for delivering event notifications.

The sensor and control systems components provide access to the real world. The sensors supply information about the current state of the world. This information can be used to update the respective Augmented Area Models. The most important information in the Nexus context is the location information of mobile objects, which is processed by the Location Service. Through controls, certain changes in an Augmented Area Model can be propagated to the real world, e.g. by changing a setting in the Augmented Area Model, a light can be switched on in the respective Augmented Area.

After this short overview of the components, we will look at some of the components in more detail in the following subsections. The Event Service will be covered in detail in the next chapter.

5.6.1 Location Service

As mentioned above, the Location Service manages the location information of mobile objects [LEONHARDI & KUBACH 1999]. For a location- and spatial-aware platform like Nexus, a location service is an essential component, since its information is the basis for a large part of the functionality of the platform.

The location information comes directly from location sensors. However, the location data has to be independent of any given sensor system. The heterogeneous environment of the Nexus system requires that the Location Service works with GPS data, but also with data from any indoor positioning system. Because of this property and because the Location Service is independent from the applications it is used by, we also talk about a *universal location service*.

For scalability reasons and in order to provide efficient management of the location data, the Location Service has to be distributed. The Location Service consists of location servers that hold the actual location information and registers that we will look at later. Each location server holds the location data for a designated area.

In order to optimize the performance of the Location Service as a whole, multiple copies of the location information with varying accuracy can be kept. There is always one location server that holds the primary copy of the location information. This location server has a direct connection to the sensor system, so its information is always the most accurate. In order to improve performance and reliability, secondary copies of the location information can be stored on other location servers. Their information will be less accurate than that of the primary server, but it may be closer to the application interested in the data.

There are primarily two types of queries the Location Service has to answer, *location-of-object* queries and *objects-at-location* queries. Location-of-object queries return the current position of the specified object. Objects-at-location queries returns all objects that are currently at the specified location or, more precisely, in the specified area. Another related query is the *nearest-object* query that returns the position of the nearest object that satisfies certain constraints.

When a location server is queried for a location of an object, it first checks whether it has a copy of the location information of the desired accuracy. If this is the case, the location information is returned (Figure 5-4). Otherwise, the location server queries the Object Register, which stores the list of location servers that hold copies of the location information for each object. The Object Register returns a list of location servers that hold the location information with the desired accuracy. One of these location servers is then queried and the location information is returned to the client.

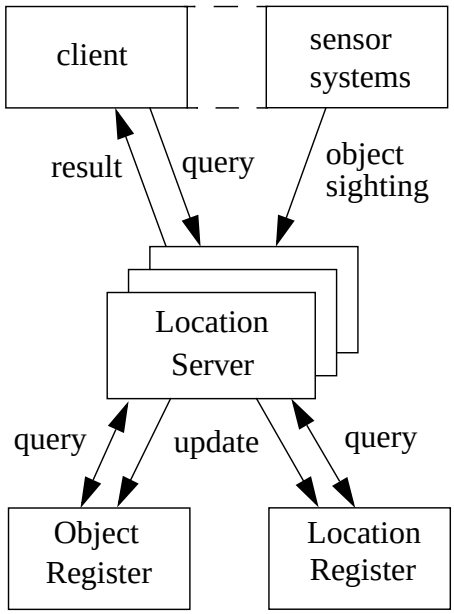


Figure 5-4: Location Service

When a location server is queried for objects at a given location, it checks if it covers the specified area. If this is the case, it returns a list of objects that are within the specified area. Otherwise, it queries the Location Register for location servers that cover the specified area. The Location Register has entries for all location servers and the areas they cover. So the Location Register returns a list of location servers. One of these location servers is then queried and a list of objects that are within the specified area is returned to the querying client.

It has become apparent that other Nexus services also need a register with a similar functionality as the Location Register, i.e. they need a register that can be queried for the instance of the service that is responsible for a given geographic region. Therefore, it is planned to implement a more general register called Area Service Register that is also used by the Location Service in place of the Location Register. For reasons of scalability, the Area Service Register has to be distributed. It will be replicated and organized in a hierarchical fashion.

As we will see in Chapter 8, the Location Service will be the most important event source for spatial events in Nexus. The way events can be supported by the Location Service will be discussed there.

5.6.2 Spatial Model Service

The Spatial Model Service manages the data pertaining to the objects in the Augmented Area Models that have more static characteristics. This includes both physical objects like rooms and buildings, but also virtual objects like virtual advertising columns and virtual Post-its.

Like all the other Nexus services, the Spatial Model Service has to be distributed. It consists of multiple Spatial Model Servers. The actual data will be stored in a database. A major research issue is how to map the data onto database schemes, so that spatial queries can be answered efficiently. At the time of writing, object-relational databases [ORACLE 2000] and database extensions for geographical information systems (GIS) [ESRI 2000] allow the efficient storage of two-dimensional data. Three -dimensional data is not fully supported yet, in most cases height information is stored as an additional attribute, so the presentation is actually 2.5D (see Chapter 4, Subsection 4.1.1).

An important research issue is the integration of heterogeneous information. Data from different sources may be available and multiple representations of the same data have to be integrated so that they can be accessed in a uniform way. In addition, different applications running on different kinds of mobile devices may need spatial information in different levels of detail.

The Spatial Model Service will also be an important information source for the Event Service. This will be discussed in Chapter 8.

5.6.3 Geographic Addressing

The idea of geographic addressing is to send messages to recipients in a certain geographical region [COSCHURBA 2000]. Since networks based on Internet technology are not organized geographically, this is not a straight-forward task. The routing of messages according to geographic location is also known as *geocast* [IMIELINSKI & NAVAS 1997].

Geographical regions can be addressed by explicitly specifying geographical coordinates, e.g. in form of a polygon or a circle. Alternatively, objects can be specified explicitly, e.g. lecture room V20.01. For this purpose objects need to have a unique ID or name. Possibly a hierarchically organized geographical naming system can be used. If objects are specified, the system has to map the object name to the geographical region covered by the object. The case is even more complicated if moving objects, e.g. cars or trains, are used as geographic destinations.

Geographic addressing could be used to deliver location-dependent event notifications. In Chapter 6 this possibility will be discussed in more detail.

5.6.4 Hoarding

The idea of hoarding is to download information that is likely to be accessed in the near future in advance. Hoarding is especially useful for mobile devices with a wireless connection. Data is downloaded when a wireless connection with high-bandwidth is available. This helps to improve performance when the user subsequently enters an area where only low-bandwidth connections are available. With hoarding, it may also be possible to make short network disconnections transparent [KUBACH 1999][COSCHURBA ET AL. 2000].

The Nexus platform may utilize an info-station infrastructure, with a number of high-bandwidth info stations that provide wireless LAN connections with a limited range. The rest of the area would be covered by a low-bandwidth wireless WAN.

When a user arrives at an info-station, the hoarding component tries to hoard as much information as possible on the user's mobile device. The problem with hoarding is how to predict what information the user will need in the near future. In a spatial-aware context, it can be assumed that the information the user will access is related to his or her current location. More refined strategies can also take into account user profiles, the time of day etc.

Events and hoarding do not have much in common, since events by nature cannot be hoarded, however, entering the area covered by an info-station may be defined as an event, and an event notification can be sent to inform the mobile device that a high-bandwidth wireless LAN connection is now available. The same may apply when the mobile device is leaving the area again.

5.7 Summary

This chapter has given an overview of the Nexus system. The discussion of the general design objectives for the Nexus platform will serve as a basis for the design of the Nexus Event Service, which will be presented in the following chapters. Since the Event Service will interact closely with other parts of the Nexus system, a high-level architecture of the Nexus system was introduced, showing the connections between the different components. Some of the components were presented in more detail, because of their relevance for the Event Service. The scenario introduced the desired functionality of the Nexus platform, highlighting the importance of spatial events in Nexus.

Chapter 6

The Event Service

The Event Service will be a central component within the Nexus platform. As seen in the example scenario in Section 5.2, many client applications depend on being informed whenever changes in the model occur. In most cases these changes regard the current location of the mobile device on which the location is running. In principle, the information could be obtained by constantly querying, i.e. polling the Nexus platform. However, a more efficient and also convenient way to achieve this is to utilize event-based communication. The Event Service facilitates this kind of communication. Clients can register for events they are interested in, and receive event notifications when the events actually occur.

This chapter will give a high-level overview of an architecture for the Nexus Event Service. The tasks of the different components and the interactions between the components will be discussed here. Some of the more important architectural issues that are relevant for the Event Service as a whole will be discussed in Chapter 7. The topics there include the interaction of the Event Service with the other Nexus Services and the distribution of the Event Service.

6.1 Outside View

This section will give a short overview of the functionality of the Event Service when seen from outside, as if it were a black box. The Event Service is the connection between event producers and clients (Figure 6-1).

Figure 6-1: Event Service Interface

The term event producer refers to an active event source. In the Nexus system the typical event producers are Nexus services, such as the Location Service or the Spatial Model Service. They are active in the sense that they can invoke methods of the Event Service if necessary. So, the event producer is the system entity in which the event takes place, or more precisely, where it becomes visible, so that it can be observed. The client is the entity interested in receiving event notifications when the event has actually occurred

The event producer can register its observable events with the Event Service, telling it how these events can be observed. There may be other ways to make events available for observation, which we will discuss later.

The clients register with the Event Service to receive event notifications for the events they are interested in. The Event Server then observes the events and notifies the client when the event occurs.

In the Nexus System, the most likely event producers are the Location Service, which is responsible for providing information about mobile objects, and the Spatial Model Service in which the objects with more static properties are stored. Often, information from both services is needed to find out whether an event has occurred. An example would be the event "Jim and Anne are in the same house". The Location Service only knows about the mobile objects (Jim and Anne) and the Spatial Model Service knows about static objects (houses). The Event Service should not be restricted to those two services, but open for input from other potential event producers.

A client can be an application running on a mobile user's device, but it is also possible that Nexus services themselves are interested in events. Even the Event Service can be its own client, registering for events that are the basis for higher-level events.

In order to handle events, both clients and servers need to implement certain interfaces. The client interface has to allow the Event Service to "push" event notifications to it. An event handler then has to make sure that the event is delivered to the right application(s). The event producer interface has to provide access to the Event Service so that the event can be observed.

6.2 Internal Structure

The purpose of this section is to give an overview of the different components of the Event Service (Figure 6-2). The service is partitioned into three components: Predicate Management, Observation Service and Notification Service. This is done for conceptual reasons, to allow a separation of concerns. It is not meant as a preliminary decision regarding the implementation of the Event Service.

The name Event Service was chosen in analogy to the other Nexus services. This definition does not coincide with the definition of the *CORBA Event Service*, which is more similar to the Notification Service component. In fact, it would be an option to use the CORBA Event Service as an instance of the Notification Service.

We have introduced a component for all the tasks that are related to "managing" event descriptions, which are nothing but predicates. Therefore we chose the name "Predicate Management". The Predicate Management component manages the information about events that are currently being observed and "advertises" predicates describing events that can be observed throughout the system.This is done in form of templates. When registering for an event notification, these templates have to be instantiated with values, e.g. by specifying a certain area.

The Predicate Management component also provides the service interface to the rest of the system. Firstly, events that are available for observation are registered and unregistered with the Predicate Management component. Registering an event means informing the Event Service about the observability and availability of an event. The registration can be done by the service offering the event, e.g the Location Service, or by some other entity. It is also possible to have pre-registered events that are always available.

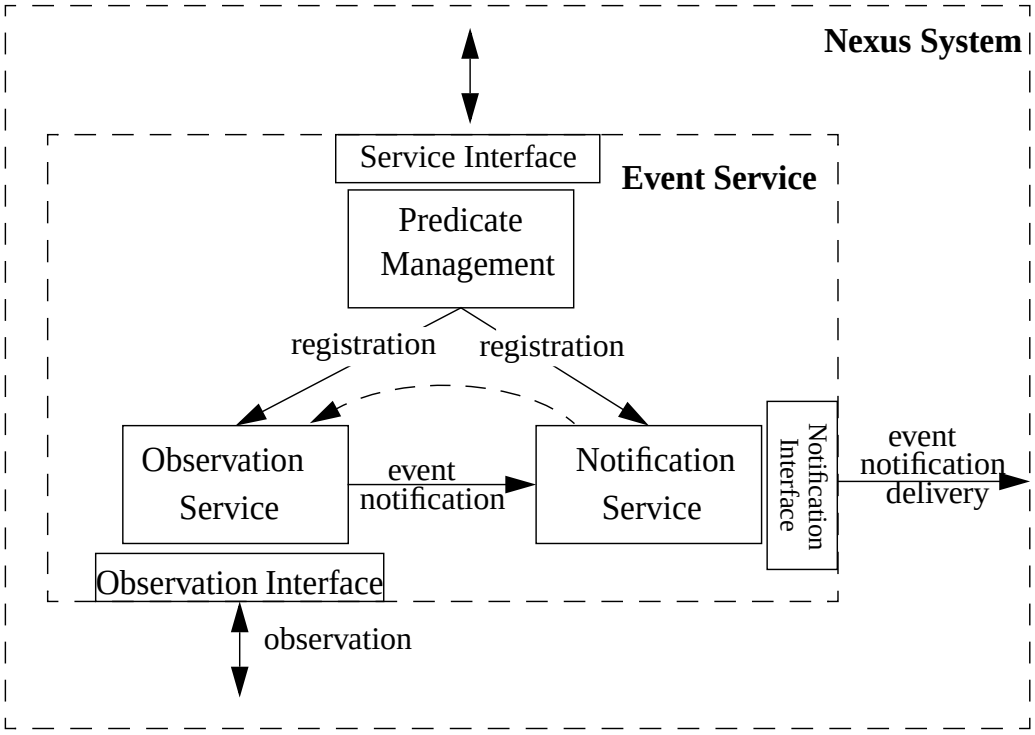


Figure 6-2: Internal Structure of the Event Service

Secondly, clients can register and unregister for event notifications, for themselves and on behalf of others. The Predicate Management in turn registers the event with the Observation Service, so that the event can actually be observed, if this is not already the case. It is also registered with the Notification Service, so that the event notifications can be delivered to the client(s) on whose behalf the registration was performed. A more detailed description of the Predicate Management component can be found in Section 6.4.

The task of the Observation Service is to observe an event. When an event occurs, the Observation Service has to hand over an event notification message to the Notification Service. A destination, e.g. in form of an address, name or channel, is given as a parameter in order to identify the registered clients as recipients, so the Notification Service can deliver the event notification to them.

A more detailed description on how events can be registered for observation, how events can be observed and how event observation can be distributed is given in Section 6.5 and Section 7.2, respectively.

The task of the Notification Service is to deliver event notifications to all clients that are currently registered for a given event. Depending on the type of event, the frequency in which it occurs, the number of recipients and certain quality of service requirements, there are a large number of possible Notification Service instances. The different alternatives are discussed in Section 6.6.

6.3 Describing Events

As we have seen in Section 2.5, any system handling events needs to have a suitable language for describing these events. This is essential for detecting the occurrence of events, returning the important information about the event, e.g. when the event occurred and where it occurred, and delivering an event notification containing this information to interested clients.

6.3.1 Language Elements

In Subsection 2.5.1, we have presented a language structure with three components - *predicates*, *filters* and *actions*. Predicates are used to describe the actual events. If an event occurs, the predicate becomes true and this can result in an action. A filter can be used to specify further conditions that decide if the action is actually executed. The only action supported here is the sending of event notifications.

Predicates. For the description of an event itself, only the predicate component is needed. The other parts are used when the client registers for an event notification. The Observation Service uses that information for the observation of events.

In Nexus the following types of predicates may be relevant:

- *Basic Predicates*
  Basic predicates are those predicates that are directly offered by the event producers, e.g. the different services, without any modifications. For example, the Location Service may offer a predicate describing an *onEnter* event, which triggers an event notification when an *object of type X enters a defined area Y.*
- *Composite Predicates*
  Composite predicates are combinations of other predicates. The language has certain connector elements, e.g. the ones mentioned as part of the event algebra presented in Subsection 2.5.2. These connector elements can be available to the clients, so that the clients can register for more complicated events, e.g. consisting of multiple basic predicates. For example, a client wants to know if *an object of type X enters area Y OR area Z.*
- *Complex Predicates*
  Complex predicates can also depend on a number of other predicates. In addition, they can have state and include complicated calculations. A simple example for such an event would be "*more than N objects of type X in area Y*".

In comparison to the types of predicates presented in Subsection 2.5.1, we have added another type of predicate, the composite predicate. We make a distinction between composite predicates and complex predicates, which were both labelled as complex predicates there. We do this for practical rather than conceptual reasons.

The underlying idea is the following: The language for defining a composite predicate is relatively simple and can be made available directly to the users, allowing them to combine predicates with the available constructors.

A language that supports more complex predicates, e.g. predicates that have state and depend on other predicates in a non-trivial way, has to be more powerful and is therefore more complicated. So for observing an event described by a complex predicate, program code has to be written. This may be too complex for the typical Nexus user. Apart from that, it might be a policy

decision to let ordinary users utilize available complex predicates, but not write any of their own.

Complex predicates may be integrated into the system in form of plug-in triggers. A plug-in trigger is a piece of code that can be plugged into another service. The plug-in trigger observes events that take place inside the other service and informs the Observation Service, whenever such an event has occurred. Plug-in triggers will be discussed in more detail in Subsection 7.1.1.2.

Filter. A client can register for an event that has to pass through a filter before an event notification is sent. The filter can consist of a query and/or may depend on event-related values, if such values exist. When describing events, there may be a trade-off between what should be observed as an event and what should be filtered out later. If the balance is more on the event side, there are less occurrences to be observed, but the observation structure is more complicated. If the balance is more on the filter side, there are more event observations, a lot of which will be filtered out, but the observation of the event itself is less complicated. An example might be: *"Send me an event notification if I enter the department and there are more than five people in the meeting room" (e.g. because I do not want to miss an important meeting...).* The event to be observed could be: *I enter the department*, and a filter in form of a query could be used to determine if *more than five people are in the meeting room*. Alternatively, the whole event could be observed with a complex predicate keeping track of the number of people currently in the meeting room. What is more efficient for the system as a whole cannot be determined yet. It may even make sense to change the observation at runtime, e.g. depending on network conditions.

Action. The only action that is going to be supported is the sending of event notifications. The reason for this restriction is that the Event Service should be kept as simple and general as possible. Additional action functionality in the Nexus Platform can be realized by having entities that can perform the action register for the respective event notifications. The action part of the structure is going to be used for information about how and where the event notification is going to be delivered.

We do not present a real language syntax for describing events in the Nexus system at this point, because this language will probably be influenced by the query language employed for querying the Nexus platform. The query language in turn will be strongly influenced by the query language of the underlying database system. Since no decisions in that regard have been made at the time of writing, it does not really make sense to present a full language syntax for describing events here.

6.3.2 Registering Events

The Nexus Event Service does not have a predefined, fixed set of events that are being offered for observation. Instead, new events can be integrated dynamically, and the availability of events may differ between objects and areas.

When describing possible events, event producers may register *predicate templates* instead of fully instantiated predicates, e.g. the exact location or area can remain uninstantiated. When a client registers for such an event, a predicate instance is created from this predicate template. The registration process is described in detail in Section 6.4.

6.3.3 Naming of Predicates and Events

Events and predicates have to be uniquely identified within the Nexus system. This is true for predicates and predicate templates describing events that are being offered, but also for events that are actually being observed. For example, it must be possible for the client to uniquely identify an event, so that the event notification can be passed on to the right application(s). The naming should be consistent with the way other entities in the Nexus system are being named.

Components that may be part of such a name are:

- EventID, e.g. a unique ID for the event *onEnter,* offered by the Location Service
- ID of the server on which the event occurred, e.g. IP address
- ID of the observation server on which the event is currently being observed, e.g. IP address
- a counter that is increased once an observation server is rebooted, to distinguish between different instantiations of an event
- a local counter for each observation server to distinguish between locally observed events
- the creation time

It is not so important which actual components are part of the name, but it is important that predicates, events and event notification can be uniquely identified in the Nexus system and that this is done in a globally consistent way.

6.4 Predicate Management

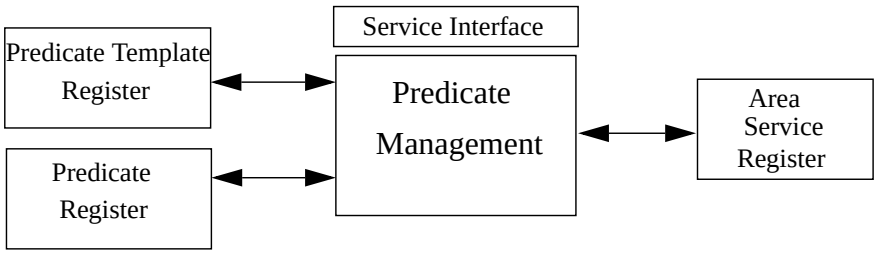


Figure 6-3: Predicate Management Component

The Predicate Management component “manages” the information about events, event producers and clients. The information is stored in two registers (Figure 6-3), the Predicate Template Register (Section 6.4.1) containing information about events that can be observed within the system, and the Predicate Register (Section 6.4.2) containing information about the events that are currently being observed within the system. For scalability reasons, the registers and the Predicate Management component have to be distributed as well (Section 7.2).

The Predicate Management component also serves as the Service Interface over which the other entities can interact with the Event Service. In order to find out about the Event Server/Predicate Management component that is responsible for a given area, a register mapping the area to the responsible Event Server is needed. A register with a similar purpose is needed for the Location

Service. Therefore, a general Area Service Register for Nexus has been proposed that maps an area and a server type to the server that provides the desired service in the given area.

When an event producer registers an event that can be observed, the necessary information is stored in form of an event template in the Predicate Template Register. The event template describes an event whose variables have to be instantiated before it can actually be observed. An example would be an *onEnter* event that could be offered by the Location Service. It is not known beforehand for which area a client may want to register the *onEnter* event. Therefore, the Location Service registers it using a variable for the area. However, the event producer can set a constraint by giving a range for such variables, e.g. the maximum area.

The Predicate Management component provides access to the Predicate Template Register for the clients, so that they can find out about available events. Alternatively, they can just try to register for event notifications. If this is not possible, an error message is returned.

When a client registers for an event notification, the predicate template variables are instantiated with actual values. The Predicate Management component checks the Predicate Register if the event is already being observed. If not, the event producer is notified (if necessary), so the event can actually be observed and the predicate describing the event is registered in the Predicate Register. Otherwise, the entry in the Predicate Register is updated with the client as an additional receiver for the event notifications.

Since all event-related registrations go through the Predicate Management component, authentication should take place there, and security parameters should be set, so that only authorized clients can register for event notifications or even find out about the existence of events. On the other hand, clients should be able to trust the event producer's identity. Thus the producer as well as the client should authenticate themselves.

6.4.1 Predicate Template Register

As mentioned above, every service registers the predicates describing its events with the Predicate Template Register in the form of predicate templates. This way, the distributed Predicate Template Register is used for "advertising" events to interested clients. This is important if the events being offered can vary from area to area or from server to server. If there is a fixed set of events offered for observation throughout the Nexus system, it is unnecessary to have an Predicate Template Register. If there is a subset of events that are offered throughout the Nexus system, they do not have to be advertised, since their availability as standard services can be assumed. Another alternative is to let the Event Service preregister such services. Of course events may also be registered by other entities if the service does not register them itself.

The following information about an event should be stored in the Predicate Template Register:

- Event name (= predicate name)
- The Service offering the event
- Servers on which the event is available for observation
- Variables
    - Area (restriction: Maximum Area)
    - Object ID(s) of object(s) involved (restriction: type of objects)

  - etc.
- Information about how the event can be observed
  - How to inform the service that the event is going to be observed (if necessary, e.g. push)
  - How to observe the event (push/pull, query, function call), see Chapter 7
  - If the event involves state, how to instantiate the state
  - What to do if the event is no longer going to be observed
- Event Description (to inform users about the event)

An important question is, whether events are registered by the services on the level of single servers or on the service level. This question is discussed in detail in Subsection 7.1.2.1.

In principle, other forms of advertising events are possible. The services could be queried directly, but this option is inconsistent with the idea of a global federation that hides the distribution into different services. A trader could be used to try to match the client's requirements with the events being offered, but this approach is difficult or even infeasible. Therefore we have chosen to have a Predicate Template Register.

6.4.2 Predicate Register

The Predicate Register contains the predicates or, in other words, the description of the events to which clients are currently subscribed. The Predicate Register contains all the information in form of the [event - filter - action] triple as described in Section 6.3. In addition, it contains some meta information about the event. This includes information about the duration for which events have to be observed, which could be given as once, n times, for a fixed interval or until an event is explicitly unregistered. Furthermore, it includes information about the current delivery mechanism and about quality of service requirements specified by the client. If multiple clients subscribe for the same event, this information may change over time.

The Predicate Register is used by the Predicate Management component to check whether events should still be observed (duration) and to choose the right delivery mechanism, depending on

- the number of subscribers
- the frequency of event occurrence
- the quality of service/client specification.

and to adapt to changes of the given parameters by changing the delivery mechanism to the one best suited to the current situation. Possible delivery mechanisms are unicast and different variants of multicast and geocast.

6.5 Observation Service

The Observation Service is one of the key components of the Nexus Event Service. Event Observation is by its nature very much dependent on the underlying system, where the events take place or become visible. The most important characteristic of event observation is the strong dependence on the type of event source and on the interface being offered by this event source.

Since these issues have an influence on the Event Service as a whole, most issues related to the Observation Service will be discussed in Chapter 7. The first part of Chapter 7 covers the interaction between the Event Service and the other Nexus services that serve as event sources. Totally passive sources have to be queried, whereas active sources may provide the information proactively.

Another important issue regarding the Observation Service is its distribution, which will be covered in the second part of Chapter 7. In order to observe some of the more complex events, the Observation Service itself has to be notified about the occurrence of "lower level" events, and in that way it is its own client. For an efficient evaluation of complex events, an optimal distribution of the event observation is essential.

6.6 Notification Service

The task of the Notification Service is to deliver event notifications to all clients that are currently registered for a given event. The Notification Service has an interface for accepting event notifications from event producers. The event producer is either the Observation Service, or another service producing basic events, if the Notification Service is also used as a communication mechanism for delivering basic events from the actual event producer to the Observation Service (see Subsection 7.1.1.1).

There is a wide range of different kinds of events. From the point of view of the Notification Service they differ in the frequency of occurrence, the number and kind of clients registered and the quality of service required for the delivery. Because of these differences, it is difficult, maybe even impossible to use a single communication mechanism for delivering event notifications to the clients. Therefore, multiple Notification Service instances will be employed to fulfil the different requirements (Figure 6-4). A common interface hides the internal complexity of the Notification Service from the clients.

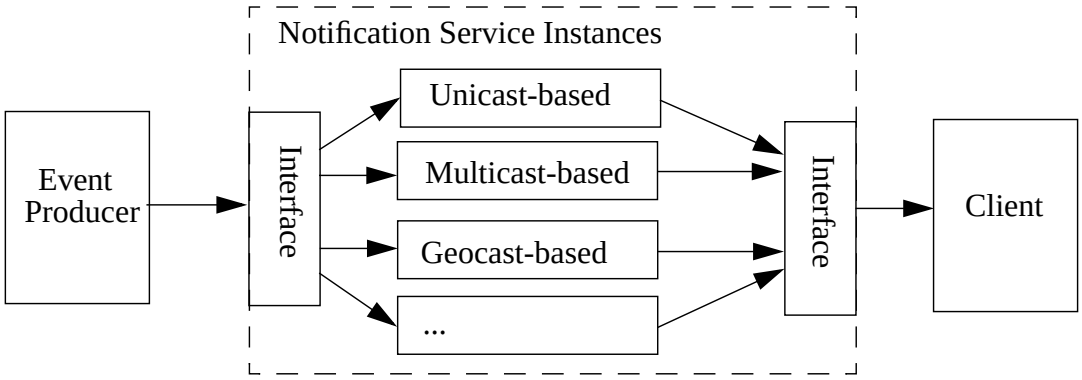


Figure 6-4: Notification Service

In the following, we will discuss certain event characteristics and the communication mechanisms that are most appropriate for handling the delivery of the event notifications to the clients

- For events with only few registered clients and for events that occur only once or sporadically, unicast-based event delivery is most appropriate, since the overhead of building and maintaining a multicast delivery tree is too high.
- For frequent events with many registered clients, multicast communication is most efficient.
- For event notifications addressed to a geographic region, communication based on geocast, which is being developed within the Nexus project, has to be used.

In addition to these basic communication mechanisms, the Notification Service instances can differ in their service characteristics. The range goes from “best effort” to different levels of “reliable” event notification delivery. A mid-level reliability would be that the service tries to deliver the event notification for 60 seconds before giving up, as TIB/Rendezvous from TIBCO Inc. does in its “reliable delivery mode” [TIBCO 1999].

The underlying communication infrastructure for the Nexus platform is will be very heterogeneous, in particular because mobile clients will be connected to the network via different types of wireless connections. In this context, it is especially difficult to provide service characteristics that go beyond “best effort” semantics.

Since wireless connections are far less reliable than other network connections, it is possible that mobile users are temporarily disconnected. How should a “reliable” event notification delivery react to this? What should happen if the mobile user is disconnected for a longer period of time, e.g. because wireless services are not available in the area that is currently being visited? In these cases, reasonable fault semantics have to be defined.

Another question concerning the semantics of event notification delivery is which clients receive the event notification message - only clients that have been registered at the time the event notification was sent, or potentially also clients that register while the event notification is being delivered. It might also be possible to offer multiple semantics.

6.6.1 Representation of Event Notifications

Event notifications have to contain information about the event. This may include some or all of the following:

- Event Type (see below)
- EventID
- ID of the event origin (server, object)
- Time stamp
- Duration for which the event is “valid”
- Other event specific information

The most important issue is that the client has enough information about the event to map it to the applications interested and pass it on to them. This may involve the first three attributes.

The question is whether event notifications should be typed or not. The type information could be used for filtering, e.g. all clients should listen to emergency event notifications that may be distributed using geographic addressing, but they may not want to receive any other event notifications.

6.7 Summary

This chapter presented an overview of the logical components of the Nexus Event Service.We identified three logical components: The Predicate Management component manages all the information about predicates, events and subscribers that is necessary for successfully observing events and notifying interested clients. The Observation Service does the actual observation of events. It informs the Notification Service whenever it has observed an event. The Notification Service then delivers an event notification to interested clients.

Chapter 7

Architectural Issues

In this chapter we will discuss important issues regarding the architecture of the Event Service as a whole. These issues pertain to two areas, the interaction of the Event Service with other Nexus components and the distribution of the Event Service. Different alternatives will be presented and evaluated regarding their usefulness. In a lot of cases, there are strong interdependencies between the possible solutions to different issues.

For our discussion, we will assume the following: The observation of events should take place close to their origin, in order to minimize the network traffic. The origin of an event is defined as the location, where the occurrence of an event becomes visible to the rest of the system. Network traffic is an important criterion, because the other Nexus services, e.g. the Location Service and the Spatial Model Service, are also going to be distributed, for scalability reasons (see Section 5.5).

The question that naturally arises here is how "closeness" is defined. This could be described in form of a cost function that returns the cost of communication. Placing the observation as close as possible to the event origin means minimizing that cost function. The cost function could include the following weighted parameters: delay, bandwidth, current load, etc. In a first approximation an optimization for shortest delay could be assumed.

7.1 Interaction with Other Nexus Components

There are several aspects regarding the interaction of the Event Service with other Nexus components, primarily with the Location Service and the Spatial Model Service. "Coupling" is concerned with the question of how closely the Nexus service and the other services should be integrated and how they should communicate with each other. The next question is the abstraction level on which the Event Service and the other services should interact. The abstraction level refers to the federation layers that may hide some of the underlying complexity from the higher levels. Finally, there is the question of what level of abstraction the Event Service should offer to its clients.

7.1.1 Coupling

An important issue for the Event Service is how closely it should be coupled with the other Nexus services that serve as event producers, or, in other words, how closely the Event Service is integrated into the other services. The alternatives range from a loose coupling, so that the other services do not even know about the existence of the Event Service, to a strong coupling, where

parts of the Event Service, i.e. the Observation Service, are integrated into the other services in order to observe events. Another alternative is that the Event Service can integrate plug-in triggers (pieces of code) into the other services, using a standard API. This option will be discussed in more detail in Subsection 7.1.1.2.

Strongly intertwined with the coupling issue is the question of where the complexity of the observation is placed. The range goes from the observation being fully on the side of the Observation Service to more observation inside the services themselves.

Also closely related to the question of coupling is the question of communication style. Both the pull and the push communication style (Subsection 2.3) can be used for the communication pertaining to the observation of events. The different options for the communication between the Event Service and the others services are discussed separately in Subsection 7.1.1.1.

The following list of options shows the range of alternatives regarding the coupling of the Event Service with the other services:

- *Other services are not aware of the existence of an Event Service.*
  This is the loosest form of coupling. Of course, this imposes a lot of restrictions on the Event Service. It needs to know the interfaces of the other services, so it can query the service or call functions that provide the necessary information. This event information can then be translated into event form and offered to the clients. The communication between the Event Service and the other services is only of the *pull* type and consists of a regular polling. In order to make this option feasible, the polling has to be done close to or on the node where the respective information is located. Otherwise the amount of network traffic is prohibitive. This, in turn, requires that the Event Service has a means of finding out where the necessary information is located, which may be related to the level of interaction with other services (Section 7.1.2).The polling rate determines the frequency with which the occurrence of an event is checked, thereby limiting the ability to detect events with a short duration. In other words, it has an effect on the semantics of an event, since some simple events may go undetected.
- *A loose coupling between the Event Service and other services.*
  The other services know about the existence of an Event Service and provide (simple) events. The complexity of the events that are being offered can vary considerably. As an example, see the discussion about possible events that can be offered by the Location Service. This ranges from raw events, e.g. the new coordinates of an object after a movement, to very complex events, e.g. more than N people in room Y. We will discuss this issue in detail in Subsection 8.2.4. The communication between the Location Service and the Event Service can be either of the *pull* or the *push* style (Section 7.1.1.1). In this case, the frequency of the polling for the *pull* type may not be as crucial as in the previous case. The service can determine the occurrence of the event itself and deliver the information to the Event Service when a pull is executed. However, the polling rate influences the time interval between the occurrence of the event and the delivery of the event notification. In case of the *push* type, it is one option to let the service offering the event hand over an event notification directly to the regular Notification Service. The Observation Service can register for such an event and use it for the observation of more complex events. Alternatively, the Observation Service and the event producer can communicate via a special “Notification Service” that is optimized for this purpose, maybe realized by a simple call-back. In the case of a loose coupling, it makes sense

to let the services register the events being offered with the Event Management themselves (Section 6.4).

- *A strong coupling between the Event Service and other services.*
  The closest form of coupling is to fully integrate parts of the Observation Service into all the other services. In this way, the Observation Service can be given access to all the internals of the other services. The observation of events itself involves only local communication. All other communication is either between Observation Service components in order to observe more complex events involving multiple services, or in the form of event notifications that are directly handed over to the Notification Service component.
  This alternative allows a close observation of all possible events and there is no unnecessary communication overhead. However, from a Software Engineering perspective, this is a very problematic solution. Whenever a new version of the service is developed, the observation component has to be integrated and adjusted to the new environment. It is also contrary to the design principle that advocates a separation of concerns and a modular component structure.
- *Services allow plug-in triggers on server/database level.*
  A relatively close coupling between the Event Service and the other services without some of the disadvantages of the previous option can be achieved by offering a standardized interface that allows the Event Service to integrate plug-in triggers on the level of servers or databases in the offering service itself (see Subsection 7.1.1.2 for more details). This suggests an interaction on the server level (Subsection 7.1.2). On the one hand, this approach is relatively flexible and allows complex events to be observed directly in the particular service, in which the event occurs. Since the observation is closest to the event origin, it can be observed most efficiently and the use of network resources is minimized. On the other hand, a relatively complex interface for the plug-in triggers has to be implemented in the other services. A description language for the triggers has to be offered. Such a language may be given in the case of an underlying active database. However, such a language is very implementation-dependent and may differ among the services, which is a problem if a wide variety of services is to be supported. This may also be a problem for a consistent advertising of events.

Discussion of the alternatives. Due to all its shortcomings, the first alternative, where the other services do not know about the Event Service, is suitable only for existing systems to which an Event Service has to be added with minimal changes to the existing services. The second alternative, with a loose coupling between the Event Service and the other services appears to be reasonable, especially since it comprises a number of options. Therefore, this alternative should be considered when implementing the Event Service. Unless stated otherwise, the loose coupling using push communication with basic events will be assumed as the basis for event observation for the remainder of this thesis. The third alternative is out of question, because of the conceptual problems regarding software development. The fourth alternative, the use of plug-in triggers, may be more complex regarding its implementation than the second alternative. However, it has a greater flexibility. Therefore, it should be considered if the number of services to be supported is limited, which is currently the case in Nexus, and the architecture permits an implementation of the plug-in trigger API with a reasonable effort.

7.1.1.1 Push and/or Pull Communication?

As we have seen in the previous discussion, the style of communication between the Event Service and the other services regarding the observation of events is an important issue. Therefore we will look at the coupling issue once more, but this time from the perspective of communication style.

The idea of the Event Service as a whole is to offer push style communication. The question is whether this is already supported on the level of event observation. This would mean that the services offering events do so using push style, or, in other words, provide their own (basic) events to the Observation Service.

The following alternatives should give an idea of the range of possibilities. Examples for each of these alternatives can be found in Subsection 8.2.4.

- *Query-based/calling functions (pull)*
  In this case, the event observation is almost totally on the side of the Observation Service. In the extreme case, the event producing service does not even know about the Event Service. The event is observed by regularly querying the service or calling certain functions, in other words the Observation Service is polling the service. A possible problem regarding polling is that if the rate is too high, it may have a huge effect on the load of the event producing service. If the rate is too low and the event offering service is not actively providing events to the Observation Service in the pull style, the event may be missed.
- *Raw events (push and pull)*
  The event producing service is offering raw "change-of-state" events that may have an influence on the occurrence of the event that is to be observed. The Observation Service registers for all "changes of state" that might make the predicate become true. When it receives such a notification, it queries the event producing service to see if the predicate actually became true. Of course, this approach is only feasible if events are not extremely short-lived, so that they can still be detected in the query phase.
  A possible advantage of this approach is that the complexity for observing events inside the event producing service is lowered. This is especially the case if the communication inside the service is based on events as well. The clear disadvantage is the communication overhead resulting from a lot of notifications that have to be sent, most of them unnecessarily.
- *Basic events (push)*
  The event producing service notifies the Observation Service about basic events. These basic events may be more complicated than the raw events mentioned in the previous paragraph, so the communication style is "push" only. This will result in fewer notifications, which are all relevant, so the communication overhead is reduced. On the other hand, the event producing service has to observe the basic events itself, which results in more complexity there.
- *Complex events (push)*
  In this case, the event producing service notifies the Observation Service when a complex predicate as a whole has become true. This means that either the event producing service has to support a wide range of complex events itself, or it has to offer support for plug-in triggers. The result is that only as many event notifications as absolutely necessary are passed on to the Observation Service, minimizing the communication overhead. Also, the observation inside the service can be much more efficient than an observation from the outside. The disadvantage can be that a lot of the complexity of the Observation Service is moved into the event

producing services, especially if the observation of complex events is not realized through plug-in triggers.

When using push communication between the event producing service and the Observation Service, it is possible to use the Notification Service directly, which has the additional advantage that clients can register for those events directly without going through the Observation Service, or a separate means of communication could be introduced, which might make that communication more efficient.

Discussion of the Alternatives. The Observation Service should rely mostly on push style basic events, because this reduces the communication overhead. Complex events can be supported in form of plug-in triggers. In order to support legacy components that may be integrated into Nexus, the additional support of pull communication for event observation might be considered.

7.1.1.2 Plug-In Triggers

As we have seen above, the use of plug-in triggers for event observation is worth to be considered. This subsection deals with the question of where the plug-in triggers should be placed, and what their structure could be like.

Trigger Placement. There are three different locations, where the plug-in triggers could be placed:

- *Observation Service*
  Using plug-in triggers within the Observation Service itself is one alternative of how complex events can be observed. The observation of such an event can take place on the Observation Server that is closest to the sources of the events involved. In spite of this placement, the network traffic may still be considerable. The big advantage is that no changes to the other services are necessary. To determine the value of the complex predicate, events from multiple different sources may be used as input. Also, the language can be chosen independent of any requirements that may be imposed by the other services.
  Of course, as far as the Observation Service is concerned, there may be other internal mechanisms for evaluating complex events that do not need to take the form of plug-in triggers, but it is an alternative that should be considered.
- *Server*
  The advantage of placing the plug-in triggers in the event producing server is that the event observation can be very efficient and only the minimum amount of network traffic is created. But there are also some disadvantages. The complex event is restricted to one single service. It cannot combine basic events from different services. A standard interface and a standard description language for plug-in triggers has to be defined and implemented for all services, which may involve considerable effort.
- *Database*
  As discussed in Subsection 2.4.5, active databases allow the definition of internal events. If the server utilizes an active database for storing its data, the database event mechanism may be used as an event source, producing basic events for the Nexus Event Service. Using internal database events allows a very efficient event observation. However, some of the same restrictions as in the Server case apply. In addition, different databases have different capabilities regarding the definition of events, and this would also apply for the triggers defined on

that basis. If different kinds of active databases are used, different definition languages have to be supported, which is a disadvantage with respect to the portability of the Nexus platform.

Structure of a Plug-In Trigger. When plug-in code is used, the following steps have to be performed to install it as a trigger in the system.

- *Parameterization*
  Certain parameters have to be set. Example: The plug-in code should trigger an event notification if more than five people are in a certain room. The code is available in form of a predicate template, *observing X people in room/area Y.* So the parameters X and Y have to be set.
- *Installation*
  The parameterized code has to be installed at the ideal location for observing the basic events it is based on.
- *Registration for events*
  The trigger has to register for the events that are the basis for its event observation. In the example above, it could be *onEnter* and *onLeave* for the given room/area.
- *Initialization of state (if any)*
  Before the event observation can begin, the current state has to be initialized. Only after this is done, the event notifications for which the trigger has registered in the previous step are evaluated. In the example, the trigger would query for the number of people currently in the room. From then on, the *onLeave* and *onEnter* notifications would be used to increase and decrease the number of people in the room.

Who Defines Templates for Plug-In Triggers? The question is whether the templates for plug-in triggers should be predefined by the system, so that the clients can use them, or whether the clients should be able to install new trigger templates themselves. It is not clear yet if this is mainly a policy question, in which case it should not be discussed here, or a question of having to implement special mechanisms to support this, e.g. security mechanisms akin to those of a Java applet execution environment. In this case it may have an effect on the architecture and therefore should be considered when designing mechanisms for plug-in triggers in more detail than we can do it here.

7.1.2 Interaction Level

Another important issue is the level on which the Event Service should interact with the other services. The envisioned Nexus architecture has several layers that hide the complexity of the underlying services from its clients. The service level federation hides the distributed nature of the service and the global federation hides the fact that multiple services may be involved in replying to a query (Figure 7-1). In this situation, the question arises on which level the Event Service should interact with the other services that offer events.

As stated in the introduction to this chapter, the entity that observes the event has to be located as close as possible to the event origin. Therefore it must be possible to find out about the location(s) where the event has to be observed. Another related point is the service level on which the clients can interact with the Event Service (Subsection 7.1.3).

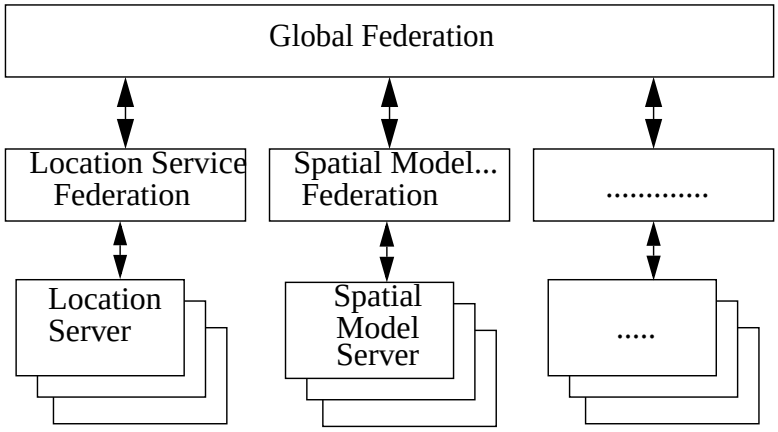


Figure 7-1: Service Levels

So there are three levels on which the Event Service could interact with the other services:

- *Global Federation*
  The advantage of interacting with the Nexus system on the abstraction level of the Global Federation is that clients do not need to know about the fact that different types of objects, e.g mobile, static and virtual objects, are handled by different services. This complexity is hidden by the global federation.
  If the Event Service wanted to take advantage of this, it would not have to care about the type of the underlying services. Since events exist that involve different services, e.g. mobile and static objects, it would be the task of the federation to break these up into suitable components and combine event parts before making the combined event visible to the Observation Service. By doing this, some of the complexity of the Observation Service has to be replicated in or moved to the Global Federation layer. In addition, the Global Federation has to determine a suitable location where the event can be observed by the Observation Service. To do this, the Global Federation needs to know about the location of the servers involved, which is exactly what is hidden by the service federation layer.
  There is no need to go further into the details of this option to see that it does not make sense to have the Event Service cooperate with the other services on the level of the Global Federation. This option either runs contrary to the idea of having a multi-level federation, since it needs information that is hidden by the second level of the federation, or the assumption has to be given up that the observing entity should be placed close to the origin of the event.
- *Service Level Federation*
  When interacting with the Nexus system on the service federation level, it is necessary to know about the different types of objects (mobile, static), but not about the distribution of the servers.
  In this case, the partitioning of the event into sub-units (not necessarily separate events, e.g. it could be a query and an event) according to the types of objects involved, remains the task of the Event Service. Still, the same argument as in the previous case applies here, when it

comes to placing the observing entity close to the origin of the event. Therefore, the option is not considered any further.

- *Server Level*
  When interacting with the Nexus system on the server level, the Event Service needs to know about the fact that different servers are handling the different object types and the distribution of the relevant objects on the respective servers.
  Unlike in the previous cases, the Event Service knows about the location of the server(s) that are potential event origins, and it can place the observation entity accordingly. The information about the servers responsible for a given area will be available in the Area Service Register (Subsection 5.6.1).

The assumption that the entity that observes the event should be located as close as possible to the possible event origin has a strong influence on the decision regarding the level on which the Event Service interacts with the other services that offer events. Therefore, the only conceptual level that appears to be reasonable for the interaction is the server level.

7.1.2.1 Registering Predicate Templates

As we have seen in Subsection 6.4.1, it is an important question on what level predicate templates are registered by the services: either on the level of single servers or on the service level. In the last subsection we have discussed on what level the Event Service interacts with the other services. Here we will discuss how the other services present themselves to the Event Service. There are two alternatives, either as a uniform service or as a collection of independent servers.

The advantage of registering events separately for each server of the event producing service is that different events can be offered in different areas, e.g. different versions of the Location Service could offer different events. On the other hand, it makes registering for event notifications more difficult. It cannot just be assumed that the registration is possible. Since some services will only be offered locally (e.g. temperature), it probably makes sense to register events on the server level. Another problem arises when a mobile object moves from an area for which the event is offered to an area for which it is not offered.

This leads to the next question: At what level should the event be registered in the Event Service hierarchy, especially if the distribution of the Event Service does not coincide with the distribution of the registering service (Section 7.2)? In order to find the predicate describing the event in the register when needed, the respective predicates should be registered with the distributed Predicate Template Registers in all those areas, where the maximum event area is not completely part of the area served by a lower level server. Recursively, it also has to be registered with all servers for which the intersection of the maximum event area and the area that is served by the server is not empty.

What happens when a client wants to register for an event in an area that is larger than that of a single server? It has to be checked whether this can be achieved by registering the event with multiple servers. For this purpose, the Predicate Template Register serving the whole area is checked first. If the event is not registered there, it is checked for all the registers serving subareas; this goes on recursively. If it is possible to cover the whole area, the event can be registered. Another alternative would be to combine events offered for different areas to form an event offered for the combined area on a higher level, so that clients can directly register for it.

Still, this approach may not make sense for all types of events, and in some cases the Observation Service has to take care that the semantics do not change. Examples of the distributed observation of such events will be discussed in Section 8.3.

7.1.3 Level of Abstraction Offered to the Clients

As seen in the previous subsection, the Nexus architecture hides some of the complexity of the Nexus services from the clients by introducing two federation layers. In the following, we will discuss how the Event Service could be integrated into this architecture from the point of view of the clients (Figure 7-2).

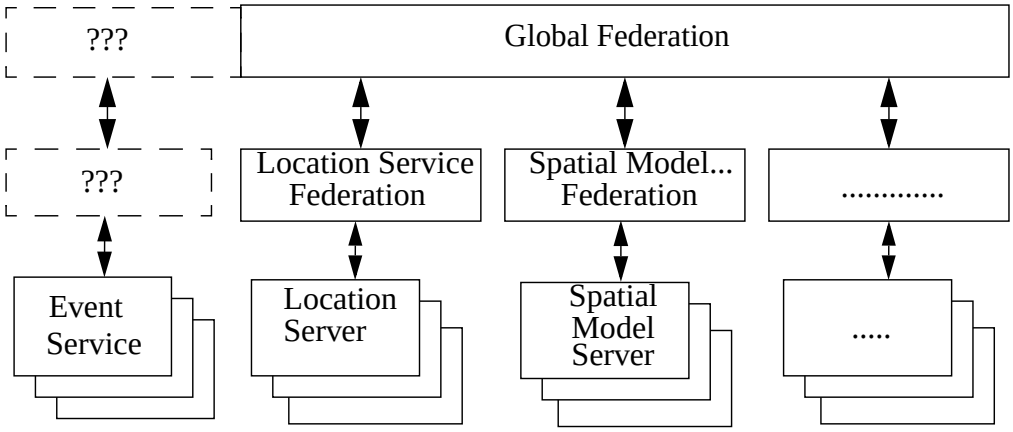


Figure 7-2: Level on which Clients Interact with the Event Service

- *Global Federation*
  The Global Federation hides the underlying complexity of the different services and provides a consistent view of the Nexus platform as a whole. Integrating the Event Service into the Global Federation allows the clients to interact with the Event Service on a similar abstraction level as with the other services.
- *Service Level Federation*
  Depending on how the question of the Global Federation is handled for the Event Service, it might make sense or might even be necessary to have an Event Service Federation, hiding the distributed nature of the service. It has yet to be determined whether it makes sense to give clients access to this level.
- *Server Level*
  Since the Event Service may be its own client, it might be a good idea to give at least this “special” client access to this level, potentially making communication more efficient. It is also possible to define special events that are visible on this level, but not on the federation level.

It is clear that clients should interact with the Event Service on an abstraction level similar to that of the other services. A straight forward way to achieve this is to integrate the Event Service into the Global Federation, so that clients have consistent interfaces for registering for event notifications and for querying the Nexus platform.

However, as we have seen in the previous section, it may not be possible to hide certain aspects of the service distribution in all cases, e.g. if events are available in certain areas, but not in others. On the other hand, that is not really the purpose of the federation layer. The purpose of the federation is to provide an abstraction layer, so the clients do not have to know about the underlying complexity in order to query the service. That does not mean that the layer can hide restrictions that are the result of the underlying structure.

7.2 Distribution

As we have seen in Section 5.5, all Nexus services have to be distributed for scalability reasons. This also applies to the Event Service. The Event Service consists of event servers and the corresponding registers. Each event server is responsible for a certain region.

In principle, other distribution strategies like the distribution by type of event are possible. For the Location Service, two alternatives, the distribution by object and the distribution by location, have already been evaluated [KUBACH ET AL. 1999]. Under certain assumptions, which appear to be reasonable in the context of the Nexus system, the authors found that a distribution by location is to be preferred for the Location Service. Since our assumption is that the event observation should take place as close as possible to the event origin, and the Location Service is an important event source, it seems to be reasonable to distribute the Event Service in the same fashion.

Let us take the consideration a bit further. Most events involve multiple objects. If the information about these objects is distributed by location, an Event Service that is also distributed by location can place the observation on an event server that is close to the possible event origin. In this case, most of the communication that is necessary for observing the event is local communication, e.g. on a LAN. However, if the Event Service was distributed by event type, the responsible server would most likely not be available in the same LAN as the information about the objects. This means all communication would be remote communication going over a WAN, which incurs higher costs, e.g. the delay would be greater.

If the information about the objects was distributed e.g. by object type, the data about the objects involved would most likely be kept on different servers, so the communication regarding the event observation would be remote again. In that case the placement of the event observation is almost arbitrary. Therefore, distributing event servers by location, i.e. making an event server responsible for a certain area, is a reasonable approach.

However, there are still several alternatives for the actual placement of the Event Service components (i.e. the event servers) in the Nexus system, ranging from placing them into clusters to placing them into the areas they are serving. Of course this also depends on the distribution of other services, which will be discussed later in this chapter. In the following, we also describe a centralized version to which we can compare the other solutions.

- *Centralized*
  Having one central Event Service component has a number of advantages. It makes the management of events much easier. For example, no handovers in the Event Service are necessary when an object leaves a certain area and the question of finding the best location for observing an event does not even arise. There is one consistent view on the whole system, which may have a positive effect when it comes to defining event semantics. There is almost no latency resulting from communication within the Event Service, which is a big advantage

when looking at events that consist of multiple sub-events. On the other hand, the latency between other services and the Event Service is high. The main problem with this approach is the lack of scalability. A central component, and especially the corresponding network, can only handle a finite amount of communication. Therefore a centralized approach does not fulfil the scalability requirements of the Nexus platform.

- *Servers are located in the area they serve*
  The other extreme regarding the distribution of the Event Service is to place the servers into the areas they serve. Of course, the size of the area can vary, resulting in different granularities.
  Placing servers into the area they serve has the advantage that information can be updated very fast, since latency is low. Again, this is valid under the assumption that the other services, e.g. the Location Service, are distributed in a similar way. The disadvantage is that the latency of communication between servers is rather high, which has a negative impact on the observation of events that are distributed among servers in different areas. The main advantage of this approach is that it is more likely to be scalable.
- *Clusters of servers serving certain areas*
  The cluster approach falls somewhere between the two previous approaches. The Event Servers are placed in one or more clusters. They still have a designated area they serve (which might change dynamically), but they are not located in that area. Instead they are collocated, which means that the latency between servers is low, but there is a high latency between clients and servers. In other words, the communication properties are more similar to the centralized approach, but a cluster approach might scale a little better (e.g. add new servers when necessary), and it might be easier to achieve fault tolerance through replication and dynamic reconfiguration.

The intention behind Nexus is to develop an open global platform for spatial-aware applications. This also entails having different providers that may wish to cooperate. Therefore, a centralized approach for the Nexus Event Service is not feasible. This leaves the spectrum between the other approaches, which needs further investigation.

7.2.1 Distribution Structure

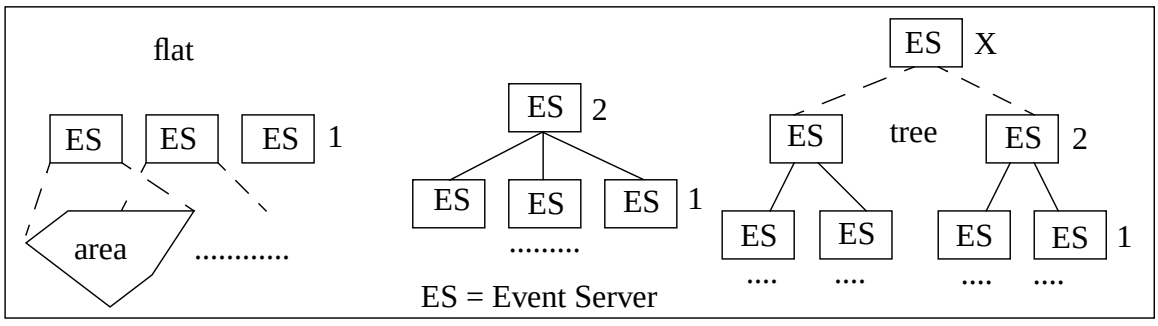


Figure 7-3: Distribution Structure

Another important question regarding the distribution of the Event Service is how many layers it should have. The spectrum ranges from a flat distribution to multilevel trees (Figure 7-3). On the lowest level, each “Event Server” serves a certain area. In Figure 7-3 most of the areas are

depicted by dots only. The server on the next higher level combines these areas to form a larger area and so on.

The distribution structure is important when it comes to placing the entity that does the actual observation on a server. This is especially important, if an event is composed of (sub-) events that are observed by different event servers. It might or might not make sense to try to place the observation for the combined event on an event server between the two original event servers. Of course, the importance of this question depends on the placement of servers - in their respective areas or in clusters. The distribution structure of the other services, which is discussed in the following subsection, is also an important issue.

7.2.2 Relation to the Distribution of Other Nexus Services

In order to place the event observation close to the event origin, it is necessary to know the location of the server(s) (e.g. a spatial model server) where an event origin could be located. Therefore, it is essential to be able to find out which servers are responsible for the area in which the event might be observed. This information can be found in the planned Area Service Register (Subsection 5.6.1).

But this information might not be enough, especially if it is not possible to place the event observing entity on the same server as the potential event origin. In this case, the closest server on which the observing entity could be placed has to be determined. For this purpose some information about the network structure has to be known to the system. The modeling of networks within Nexus has already been considered as part of the planned implementation of geographic addressing [COSCHURBA 2000]. The minimum information might be the address of the subnetwork, under the assumption that there is an event observing entity on every subnetwork.

Another alternative may be to “bundle” an event-observing entity with every kind of server. This means that an event observing entity is running on every node on which a Nexus related server is running. Above this low level structure, there could be a hierarchy of observation servers where events could be observed that are composed of distributed events observed on a lower level. Again, the ideal placement of that “higher-level-observation” has to be determined.

7.2.3 Handover of Event Observation

In this context, handovers should also be considered. Mobile objects can move around, so they may leave the area served by one location server and enter the area served by another location server. This means, a handover from one location server to the other has to take place. This may entail another handover on the event server level, if an event is associated with the mobile object, because the place where the observation takes place may no longer be optimal.

> An example for such an event is the following: “*Send an event notification when a mobile object comes close to an object of type computer shop*.” The event is observed close to the location server that has the current position of the mobile object. If the location server changes, we are in the situation described in the previous paragraph.

The Location Service has to ensure that the basic events are still delivered to the Observation Service after the handover has taken place. The question is if it is worth to hand over the obser-

vation to another observation server. This depends on a trade-off between the overhead that is caused by a handover and the communication overhead caused by the increased “distance” from the event origin. The complexity of the actual handover protocol has a strong effect on this trade-off. Again, a cost function could be defined that may serve as a basis for the decision.

A more difficult problem arises if the mobile object leaves one area, but does not immediately appear in another known area. This situation may also be due to an interruption of the wireless connection. Reasonable fault semantics for events registered for this mobile object have to be defined.

7.2.4 Distribution of Event Service Components

Should Event Management, Event Observation and Event Notification be distributed in a combined fashion, so that each “event node” has all of them (Figure 7-4)? This could make the communication between the different components more efficient, but could also lead to unnecessary overhead, e.g. the access to the Event Management may be infrequent compared to the observation of simple events. The alternative would be to distribute the components separately, which leads to more administrative overhead, e.g. the components need to know how (and where) to access each other.

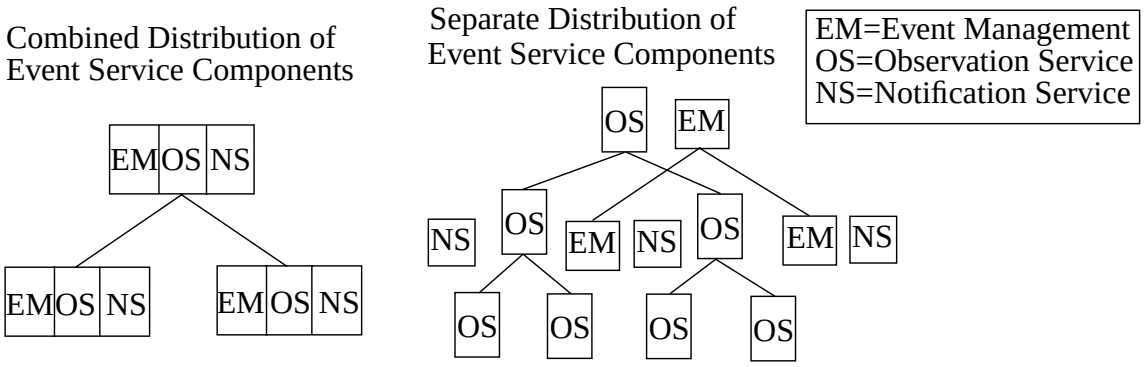


Figure 7-4: Distribution of Event Service Components

7.3 Summary

In this chapter we have discussed issues related to the interaction of the Event Service with other Nexus components and the distribution of the Event Service.

As far as the interaction between the services is concerned, a loose coupling with *push*-style basic events seems to be reasonable, because it reduces the communication overhead. *Pull* communication may be added for supporting the observation of events in legacy components. Plug-in triggers offer greater flexibility, especially regarding the observation of complex events close to the event origin. Therefore, plug-in triggers should be supported if a plug-in trigger API for the Nexus services can be implemented with reasonable effort.

An interaction of the Event Service with the other services on the server level allows the location of the observation to be close to the possible event origin. The clients, on the other hand, should interact with the Event Service on a similar abstraction level as with the other services. There-

fore, the Event Service should be integrated into the Global Federation, so that clients have consistent interfaces for registering for event notifications and for querying the Nexus platform.

It is clear that the Nexus Event Service has to be distributed and that the distribution should preferably be by location. Apart from that, there are still a lot of open questions regarding the distribution of the Event Service. Most of them are related to the fact that the distribution of the Event Service depends very much on the distribution of the other Nexus Services and this distribution is not totally clear yet. However, in this chapter we have given an overview of some of the parameters that are relevant for the definition of a distributed Event Service architecture, so it should be easy to derive a distributed architecture for the Event Service once the distribution of the other services is clear.

Chapter 8

Spatial Events in Nexus

In Chapter 4 we have looked at spatial events in general. Now we will discuss the subset of spatial events that will be supported by the Nexus system. The reason why not the complete range of spatial events will be implemented has to do with what information of the real world will be available in the Nexus model, and how accurate that information will be. The discussion of simple spatial events will also help to derive requirements for the Event Service. Nexus will also support other kinds of events. A selection of such events can be found in Appendix C.

Since the Location Service is responsible for tracking the location of mobile objects, its information will be most important for determining whether an event has taken place. For quite a number of spatial events, additional information, e.g. about static objects, is needed as well. This may be provided by the Spatial Model Service or by other Nexus services.

In this chapter we assume that the spatial events will be provided as basic events, i.e. they will be provided directly by the Nexus services. As we have discussed in the previous chapters, an alternative (or additional) approach is to integrate special plug-in triggers into the Nexus services that observe the event and send the respective event notifications. The plug-in trigger approach offers more flexibility, especially with respect to keeping state information. For the observation of simple events, i.e. events without such state information, it does not make much difference if the observation is done by plug-in triggers or by the services themselves.

In the following section, we will describe the spatial events that will be available in Nexus and discuss why other types of spatial events cannot be observed in the currently envisioned Nexus model. For a better understanding and as a basis for the following sections, a list of real-world event examples is given in both natural language and in a formalized version.

The second section is concerned with the observation of spatial events. In the first part, the real-world examples are classified according to the types of objects involved. This is taken as the basis for determining what kinds of services are involved in the observation of the events and what their respective roles are. In the second part, different sets of basic predicates that could be offered by the Location Service are discussed, and also how the spatial predicates defined in Section 2 could be implemented with these basic predicates.

Finally, the third section looks at how the observation of some selected predicates could be distributed across Location Server boundaries.

8.1 Event Descriptions

Table 8-1 presents the subset of predicates from Table 4-1 and Table 4-2 that are actually offered by the Nexus system. The main differences lie in the implementation-specific details that have to be taken into account.

Here, the use of the term *object* has to conform to the object concept of Nexus. Since the Nexus project is still in the modelling phase at the time of writing, and the object concept has not been fixed yet, we assume the following:

An *object* is an entity in Nexus that can be addressed with a unique name and has some spatial properties. At least an attribute containing the current location of the object is needed. An object does not have to have a physical equivalent in the real world, it can be completely *virtual*. If the location of an object is fixed, it is called *static object*, otherwise *mobile object*.

In order to describe a specific event, the parameters of the predicates have to be instantiated. For this purpose, an object can either be specified explicitly, i.e. by giving its unique name, or by using an object selector. An *object selector* describes an object or a group of objects by specifying certain object attributes, while leaving others unspecified.

Some of the predicate parameters have to be specified explicitly, while others can alternatively be specified by object selectors. The reason for this distinction has to do with the observability of an event. If the event description covers too many objects, an efficient observation is no longer possible. We will discuss this problem in Section 8.2.

In the following we will assume the availability of a full 3D model, not just a 2.5D model that assigns a height information attribute to objects in an otherwise 2D model of the world. In this case it makes sense to differentiate between the *onEnterArea* and the *onEnterObject* predicate. An *area* is a section of the surface of the earth, which does not have to coincide with an object. Since the height information is ignored, an area is a 2D element. It can, for example, be defined in form of a polygon. In a 2.5D model the *onEnterObject* predicate can be mapped onto the *onEnterArea* predicate, because there is no other 3D information than the (maximum) object height available. This is insufficient for reliably detecting a true *onEnterObject* event.

Another 2D element is the plane. Unlike an area, which is “parallel” to the surface of the earth, a plane can have any orientation in space. Whereas the area is an everyday concept, the plane is an auxiliary construct that allows the definition of certain events.

Table 8-1: Predicates Describing Spatial Events in Nexus

Predicate	**Description & relations involved**
onMeeting*(mobile_object$_1$, mobile_object$_2$, ... , mobile_object$_N$, distance)*	The *onMeeting* predicate describes an event where two or more *mobile objects* are closer together than the given *distance*. *Mobile object$_1$* has to be specified explicitly, the others can be specified explicitly or through object selectors.
onCrossing*(mobile_object, plane)*	The *onCrossing* predicate describes an event where a *mobile object* crosses a *plane* in space.
onEnterArea*(mobile_object, area)*	The *onEnterArea* predicate describes an event where a *mobile object* enters a given *area*. The *mobile object* can be specified explicitly or through an object selector. How the *area* is specified will depend on the means offered by the Nexus platform.

Table 8-1: Predicates Describing Spatial Events in Nexus

Predicate	**Description & relations involved**
onEnterObject*(mobile_object, object)*	The *onEnterObject* predicate describes an event where a *mobile object* enters another *object*, which can be mobile or static. Either object can be specified explicitly or through an object selector, but the case that both objects are specified through an object selector will be excluded.
onLeaveArea*(mobile_object, area)*	The *onLeaveArea* predicate describes the inverse event to the event described by the *onEnterArea* predicate. The *mobile object* can be specified explicitly or through an object selector. How the *area* is specified will depend on the means offered by the Nexus platform.
onLeaveObject*(mobile_object, object)*	The *onLeaveObject* predicate describes the inverse event to the event described by the *onEnterObject* predicate. The objects can be specified explicitly or through an object selector, but the case that both objects are specified through an object selector will be excluded.
onObjectInDirection*($object_1$, $object_2$, direction, coordinate_system, deviation, maximum_distance)*	The *onObjectInDirection* predicate describes an event where $object_2$ is located in a certain direction relative to $object_1$. $Object_1$ has to be specified explicitly, $object_2$ can be specified either explicitly or through an object selector. The *direction* is given in form of a vector. The vector is relative to the given *coordinate system*. A value for *deviation* defines how far the actual direction vector can differ from the given vector so that the event is still reported. A *maximum distance* can be given, so that only events are reported where $object_2$ is closer to $object_1$ than the *maximum distance*.

In order to determine the movement of objects reliably, both granularity and accuracy of the location information must be high. For example, a cell-based indoor-positioning system may not be sufficient to detect some of the events concerning movement reliably.

Table 8-2: Predicates Describing Nexus Events Involving the Movements of Objects

Predicate	**Description & relations involved**
onObjectMovingInDirection*(mobile_object, direction, coordinate_system, deviation, interval)*	The *onObjectMovingInDirection* predicate describes an event where a *mobile object* is moving in a given direction. The *mobile object* has to be specified explicitly. The *direction* is given in form of a vector relative to the given *coordinate system*. The *deviation* defines how much the direction of movement can differ from the given direction so that the event is still reported. The optional parameter *interval* describes the maximum time interval in which the event has to take place in order to be observed.
onObjectMovingRelativeToVector*(mobile_object, vector, coordinate_system, interval)*	The *onObjectMovingRelativeToVector* predicate describes an event where a *mobile object* is moving in the direction of a *vector* that can be decomposed into two components, one of which is parallel to the given vector. Both vectors are relative to the given coordinate system. The optional parameter *interval* describes the maximum time interval in which the event has to take place in order to be observed. The *mobile object* has to be specified explicitly.

Table 8-2: Predicates Describing Nexus Events Involving the Movements of Objects

Predicate	**Description & relations involved**
onSpeed*(mobile_object, rel, speed)*	The *onSpeed* predicate describes an event where a *mobile object* is moving *rel* = {faster than, as fast as, slower than} a given *speed*.

In Table 8-3, the predicates are listed that will probably not be implemented in the Nexus system. The reason for that is given in the second column.

Table 8-3: Spatial Events Not Supported in Nexus

Predicate	**Reason why predicates will not be implemented in Nexus**
onOrientation*(object$_1$, orientation, coordinate_system, deviation)*	Extra sensors are needed to determine the orientation of an object, e.g. there are tilt sensors available for 3COM's Palm. Most likely only location information will be available for the majority of mobile objects.
onChangeOfOrientation*(object, degree, interval)*	same as above
onTurnAroundAxis*(object, vector, coordinate_system, rel, value, interval)*	same as above
onTouch*(object$_1$, object$_2$)*	Precise information about the exact position of *object$_1$* and *object$_2$* including their extension relative to each other, or special sensors are necessary to reliably detect such an event. Most likely, such information will not be available.
onChangeOfSpeed*(object,rel, value, interval)*	Precise information about the speed of the *object*, plus a sufficient history of the speed information, or, alternatively, special sensors are needed to determine the change of speed. Most likely that information will not be available.

8.1.1 Examples of Spatial Events

For the following discussion, we will concentrate on the predicates *onEnterArea*, *onEnterObject* and *onMeeting*. In order to find out more about the underlying characteristics of the respective spatial events, we list a couple of examples. The examples are given in natural language and in a formalized language including the predicates defined above. The natural language examples remain imprecise as it is usually the case in natural conversation, so *meet* might imply that objects get closer than 5 meters to each other, which is information that can be used to describe an appropriate event

Table 8-4: Event Example

	onMeeting, onEnterArea (2D)	**onEnterObject (3D)**
1)	Jim and Anne meet. onMeeting(JIM145, ANNE123, distance(<=5m))	Jim gets into his car. onEnterObject(JIM145, CAR234)
2)	I meet a professor. on Meeting(MARTIN185,person(X, attribute: professor), distance(<=2m))	I get into a car. onEnterObject(MARTIN185, car(Y))
3)	I'm near the CS department. onEnterArea(MARTIN185, area(CS-DEPARTMENT14,radius(100m)))	I enter the CS department. onEnterObject(MARTIN185, CS-DEPARTMENT14)
4)	I'm near a university building. onEnterArea(MARTIN185, area(building(Z, type=university),radius(100m)))	I enter a university building. onEnterObject(MARTIN185, building(Z, type= university))
5)	Somebody is close to my front door. onEnterArea(person(V),area(DOOR039, radius(5m)))	Somebody enters my office onEnterObject(person(W),OFFICE2.172)
6)	Two professors meet. onMeeting(person(X, attribute: professor),person(Y, type=professor), distance(<=5m))	A professor gets into a car. onEnterObject(person(S, attribute: professor), car(T))
7)	A professor is close to a shop. onEnterArea(person(R,attribute: professor), area(shop(U), radius(50m)))	A man enters a shop. onEnterObject(person(S, attribute: man), shop(U))

8.2 Observation of Spatial Events

8.2.1 Classification of Spatial Events

The examples (Table 8-4) have been chosen in such a way that they can easily be classified according to the combination of different types of objects involved (Table 8-5), and they also cover the classification space. We distinguish between static and mobile objects and between "specified" objects, which are identified by their unique object name, and "unspecified" objects, which are described in form of object selectors. The numbers in the table refer to the given examples. *Not possible* means that such an event cannot be observed, i.e. because two static object cannot change their position relative to each other. Finally, *unreasonable* means that the event cannot be observed with a reasonable (= scalable) effort.

Table 8-5: Event Classification

	mobile/specified	**mobile/ unspecified**	**static/specified**	**static/ unspecified**
mobile/specified	1)	2)	3)	4)
mobile/unspecified		6) unreasonable	5)	7) unreasonable
static/specified			not possible	not possible
static/unspecified				not possible

Summarizing the information in the previous paragraphs, the description for location-dependent predicates can take the following abstract forms (* = zero or more times, | = or):

- EventName(specified mobile object, (object | object selector | area), extra attributes*)
- EventName(specified static object, (mobile object | object selector), extra attributes*)

In other words, the first parameter of a predicate has to be instantiated with an explicitly specified object, regardless of its type. Additional objects can be specified by an object selector.

In contrast, the examples 6) and 7) are instances of the combination

- EventName(mobile object selector, object selector, extra attributes*)

This implies that the event has to be observed for all objects described by the object selector. Theoretically, this is possible, since the number of mobile objects within Nexus is always finite. The problem is that the number of objects may be very large. If the observation of the event involves all professors of the computer science department in Stuttgart, it may be feasible. If the observation involves all professors in Germany, it will become infeasible. If this type of event should be considered in some way - contrary to what we have advocated so far - an idea might be to define a cost function and have the system decide according to this function, if it is feasible to observe an event or not.

There cannot be any events in the lower part of the table, because there are no event characteristics with two static objects. An exception might be the addition of a new static object, if this is defined as an event. Another kind of event is the attachment of a virtual object, e.g. a Post-It note, to an object. These kinds of events are not considered any further at this point, but some more examples are listed in Appendix C.

8.2.2 Services Involved in the Observation of Spatial Events

As seen in the previous subsection, an efficient event observation is based on either a specified mobile object or a specified static object. Therefore, the observation of the event has to be “attached” to one of the specified objects in the predicate.

Now, we want to look at how the events can be observed, or in other words, what services are involved in the observation of the events, depending on the types of objects involved. The Location Service itself handles only the information about mobile objects. So predicates involving both mobile and static objects cannot be handled by the Location Service alone. Table 8-6 shows how the observation of events involving different pairs of objects can be handled by the Location Service and the Spatial Model Service.

Table 8-6: Observation of Events Involving Different Kinds of Object Pairs

	specified mobile object	**specified static object**
specified mobile object	This case can be handled by the Location Service alone.	In this case, the Spatial Model Service has to be queried for the location and extent of the static object. The rest is handled by the Location Service.
unspecified mobile object	This case can be handled by the Location Service alone.	In this case, the Spatial Model Service has to be queried for the location and extent of the static object. The rest is handled by the Location Service.
specified static object	If there is a choice to "attach" an event to a specified static object or a specified mobile object, it is better to attach it to the specified static object, because in that case it can be handled (mostly) by the Location Service. The Location Service has the most accurate location information about the mobile objects, which can be easily compared with the location information of the static object which has to be queried from the Spatial Model Service only once. See *specified mobile object(row)/ specified static object(column)*.	not possible
unspecified static object	This is the only case that cannot be observed by the Location Service alone. Either the observation has to be done as a direct cooperation between the Location Service and the Spatial Model Service, or it has to be (partially) handled by the Observation Service. This will be further discussed in Subsection 8.2.3.	not possible

The essence of Table 8-6 is that the observation of events differs depending on the parameters of the predicates. So internally, there is not just one implementation handling the observation of

an event defined by *onEnterObject* for example. Instead there are multiple implementations for the different kinds of object pairs.

8.2.3 Observing Events Involving more than one Nexus Service

In the specified mobile object/unspecified static object case, two Nexus Services are involved in determining if an event has taken place. The Location Service has to keep track of the current location of the specified object and the Spatial Model Service has to determine if there is a static object relative to that location that makes the given predicate true.

There are two principle ways to observe such an event: A direct cooperation between the Location Service and the Spatial Model Service or the Observation Service has to coordinate the observation. In the following we will discuss both options.

Cooperation between Location Service and Spatial Model Service.

In a cooperation model, the Location Service has to query the Spatial Model Service regularly, asking for the location of the nearest object in the area that has the attributes defined by the object selector. With that information, the Location Service can observe the event. It has to query the Spatial Model Service often enough to guarantee that the Spatial Model Service is queried after the mobile object has covered the distance between its old position and the previously nearest object, regardless of the direction in which the object is actually moving (Figure 8-1a). In this way, the detection of an event occurrence can be guaranteed, because up to that point, the mobile object cannot have come into contact with an object that fits the object selector. If the event only involves getting closer than a certain distance to the object fitting the object selector, this distance has to be substracted from the original distance first. As an optimization, all objects in a given area that fit the object selector can be queried. In this case, the point in time for the next query can depend on the direction in which the mobile object is moving (Figure 8-1b).

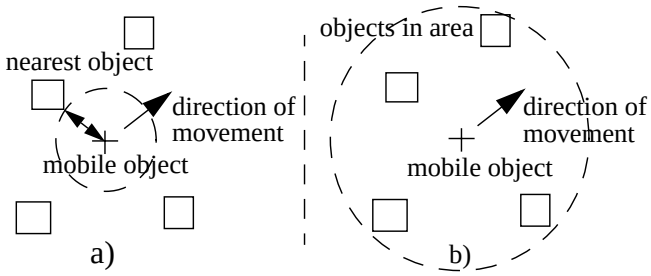


Figure 8-1: Query for Static Object

The advantage of the cooperation approach is that the event can be observed by the Location Server that has the most accurate information about the current location of the mobile object. Also, the number of services involved is minimal. The main disadvantage is a conceptual point: The Location Service has to know about the existence of the Spatial Model Service. That contradicts the concept underlying the Nexus Architecture that the services are independent of each other and that the information is only integrated for the clients on a federation layer above the actual services.

Observation Service handles Event. In this case, the Observation Service itself has to keep track of the mobile object and, in that respect, take over the role of a Location Server for the specified mobile object. With the information about the current location of the object, the Observation server can query the responsible Spatial Model Server in the same way as the Location Service in the first option.

The advantage of this approach is that the concept of the Nexus Architecture can be maintained. The Location Service does not need to know anything about the Spatial Model Service and the observation of a more complex event is the task of the Observation Service. The disadvantage is the replication of the location information on the Observation Service, which will be less accurate than the information on the Location Server, and the involvement of three Nexus Services instead of two.

If plug-in triggers are used, there is a third option that could solve the problem. A plug-in trigger could be plugged into the Location Service that observes the event. In this case, this plug-in trigger has to query the Spatial Model Server, which means the Location Server itself does not need to know anything about the Spatial Model Service. The plug-in trigger could be conceptually considered as part of the Observation Service, so there is no contradiction to the concept behind the Nexus Architecture, but the advantages of the first option remain.

Let us sum up our discussion: An event involving a specified mobile object and an unspecified static object can be observed. For conceptual reasons, the second option is to be preferred. Alternatively, if plug-in triggers are available, the third option should be considered.

8.2.4 Basic Events Offered by the Location Service

So far, we have assumed that the observation of most events defined by the predicates is supported directly by the Location Service, but this does not have to be the case. In this subsection, we will discuss how a given set of spatial events can be observed, depending on the set of basic events offered by the Location Service. This includes continuous event notifications, e.g. for tracking the location of objects, that have not been discussed so far. Additionally, more complex predicates with state are included, e.g. *X mobile objects are in area Y.* These predicates have been omitted so far, because they cannot be described with the simple relations discussed in Section 4.2.

Set of events.

The following set of events will be the basis for the discussion about basic events that are to be offered directly by the Location Service. The order of the events in the event set approximately reflects their complexity.

1. tracking location of object of interest (continuous)
2. tracking location of all objects in the area of interest (continuous)
3. *onCrossing(object, plane)*
4. a) *onEnterArea(object, area)*
   b) *onEnterObject(mobile_object, object)*
5. a) *onLeaveArea(object, area)*
   b) *onLeaveObject(mobile_object, object)*
6. *onMeeting (mobile_object, mobile_object_filter*, distance)*
7. a) *X mobile objects are in area Y*
   b) *X mobile objects inside object Z*
8. *onSpeed(object, faster_than, speed)*

Basic Events offered by the Location Service.

Events not directly supported by the Location Service have to be observed by the Observation Service, based on lower level events that are offered directly by the Location Service. In the following, we will look at a range of scenarios in which the complexity of events directly supported by the Location Service is rising step by step.

1. Query (no special observation inside the location service)
   If the Location Service does not provide any basic events, all event-related information has to be gathered by regularly querying the Location Service. The querying corresponds to pull communication and is nothing but regular polling. The more accurate the location information has to be, the shorter the interval between queries, and the higher the network load.
   It is necessary to keep state for observing the events (3) to (8), because the previous position of the objects has to be known in order to determine whether an event has occurred.

2. Raw events
   In this case, the Location Service offers the following basic events:
   - tracking location of object of interest
   - tracking location of all objects in an area of interest

   This means that the Observation Service can register for event notifications for the location of all objects it is interested in. The communication can be seen analogous to updating secondary copies of location information. It corresponds to a change from pull to push communication. As in the previous case, the question of accuracy of the location information can be a problem. The more accurate the location information should be, the more communication is necessary.
   The state for events (3) to (8) has to be kept by the Observation Service.

3. More complex events
   In addition to the raw events in the previous case, the following events could be offered by the Location Service:

   a.
   - *onCrossing*
   - *onMeeting (mobile_object, mobile_object_filter*, distance)*

   This means that the events (3) and (6) are directly supported by the Location Service, which has the necessary state available automatically, e.g. the old position of the object and the new one which is currently replacing the old one.
   The events (4) and (5) can be observed by the Observation Service by subscribing to *onCrossing* events for all the sides of the area or object. Then, the Observation Service only has to check if the mobile object actually changed its location from outside the object or area to inside, or the other way round. For that purpose the observation service has to maintain state information. With the observation of events (4) and (5) it is possible to observe event (7) in the same way as will be described in b.

   b.
   - *onEnterArea, onEnterObject*
   - *onLeaveArea, onLeaveObject*

In this case, the events (4) and (5) are also directly supported by the Location Service, in addition to events (3) and (6). Event (7) can be observed by initializing the state through a query (number of objects in/inside Y) and then the Observation Service can register for the events *onEnterArea*/*onEnterObject* and *onLeaveArea*/*onLeaveObject* respectively. For event (8) the location still has to be tracked so that the speed with which an object is moving can be calculated.

4. Complex events, example: *X mobile objects are in area Y*
   In this case, event (7a) could be observed directly inside the Location Service. Only events involving more than one service would have to be observed by the Observation Service.

From the point of view of the Observation Service, the third and fourth alternatives appear to be the best, because they reduce the communication load. For this reason, the first and second alternatives are less attractive, although the basic events introduced by the second alternative may be needed to implement events similar to event (8). As far as the third alternative is concerned, it is doubtful that alternative a. is easier to implement than alternative b., so alternative a. is only of theoretical interest. The fourth alternative moves a lot of the complexity for observing events into the Location Service.

The next step is to look at the alternatives from the point of view of the Location Service. The first alternative involves no changes to the Location Service, but leads to a high communication load. The events in the second alternative correspond to Location Service functions that have to be offered anyway in order to update secondary copies of the location information on other location servers, so offering them in form of basic events should not be a problem. The communication load is also high. The third alternative involves the implementation of some event observation functionality in the Location Service. On the other hand, the amount of communication is reduced considerably. Finally, the fourth alternative moves most of the complexity of observing events to the Location Service, even though event observation is not really its task. However, if the observation can be implemented in form of plug-in triggers, the Location Service only has to provide a standard interface, therefore this option should be looked into.

For the remainder of this thesis it is assumed that the third alternative has been selected, because it seems to be a reasonable trade-off between the communication load and the complexity of observation that has to be taken over by the Location Service.

An important problem, which is relevant for the other components of the Nexus system as well, is the accuracy of location information. The accuracy of location information can be restricted by the system used for determining the location, e.g. a badge system or another cell-based system can only provide location information with the granularity of a given cell. This may be too imprecise to define certain events, e.g. event (8) from the previous example. Apart from the system-immanent accuracy restrictions, it is also possible that a user does not release accurate location information for privacy reasons.

8.3 Distributed Observation of Spatial Events

In this section, we are going to look at a few selected events and at what happens when their observation has to be distributed among different servers. Ideally, the distribution should be transparent to the client. It is the responsibility of the event component of the federation layer to achieve this transparency.

8.3.1 onEnterArea

A simple example of a predicate describing an event whose observation needs to be distributed is an *onEnterArea* predicate that covers an area for which different location servers are responsible, e.g. *Area X* in Figure 8-2.

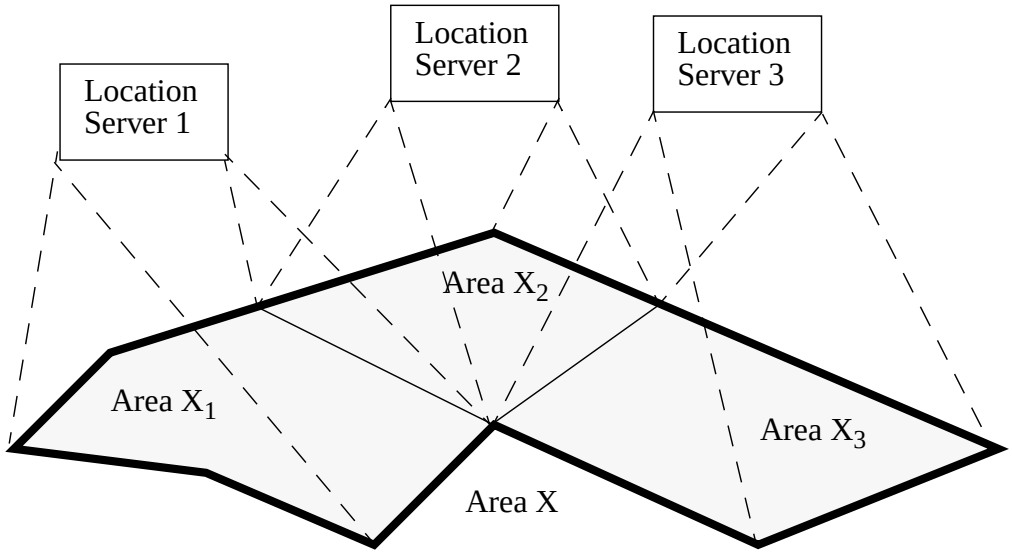


Figure 8-2: Distributed onEnterArea

Location Server 1 is responsible for [sub-]*Area* X_1, Location Server 2 is responsible for *Area* X_2 and Location Server 3 is responsible for *Area* X_3.

If the Nexus platform should support this case, the event component of the federation layer has to split up the observation so that the event is observed by all three location servers, e.g. the predicate *onEnterArea(X, OBJECT123)* could be split up into *onEnterArea(*X_1*, OBJECT123) OR onEnterArea(*X_2*, OBJECT123) OR onEnterArea(*X_3*, OBJECT123)*. It is easy to see that this simple split will not produce the desired result. If *OBJECT123* is in *Area* X_1 or X_3 and enters *Area* X_2, an event is observed, even though no event has taken place regarding *Area X* as a whole.

Therefore, in order to determine whether an event regarding *Area X* has taken place, the simple *onEnterArea* predicate is not sufficient, additional information is necessary. There are different solutions for this problem.

The first solution would be to include the previous position of *OBJECT123*, when an *onEnterArea* event is reported to the Observation Service. This allows the Observation Service to check whether the previous position was outside *Area X*. In this case, the event is valid and an event notification is sent, otherwise the event is discarded. In other words, *onEnterArea* events are filtered on a higher level to eliminate “false” events.

The filtering requires an efficient algorithm to find out whether a position is inside a given area. The location service has to be able to deliver the previous position of an object, not just the current one. This is also the case when a handover between different location servers takes place. Therefore the previous position of an object has to be part of the handover.

The second solution would be to give the location server the whole area (2D) as a parameter, including the parts for which it is not responsible. In this case, the location server could check whether the previous position handed over by the previous location server was already within the area. In other words, the filtering would take place on a lower level, before the event is even reported. Thereby any unnecessary communication could be eliminated.

The third solution is for the Observation Service to realize the original *onEnterArea* predicate by registering *onCrossing* predicates with the Location Service. The respective planes have to model the borders of the original area. Whenever an *onCrossing* event occurs, the Observation Service has to check whether the mobile object entered or left the area and send an event notification if appropriate.

Since the *onLeaveArea* predicate is the inverse of the *onEnterArea*, the discussion above applies to the *onLeaveArea* predicate in the same way.

8.3.2 onEnterObject

In the previous subsection, we have discussed the distribution of the two-dimensional *onEnterArea* predicate. Now, the same needs to be done for the three-dimensional *onEnterObject* predicate (Figure 8-3).

As an example, take a house for which two different location servers are responsible. If the event to be observed is described as *onEnterObject(OBJECT123, HouseX)*, it has to be split up into two parts, e.g. *onEnterObject(OBJECT123, $HouseX_1$) OR onEnterObject(OBJECT123, $HouseX_2$)*.

The same solutions as in the previous subsection can be applied, except that everything has to be mapped into three-dimensional space here: Instead of checking whether the previous location of the mobile object was already within an area, it has to be checked whether it already was inside a given space.

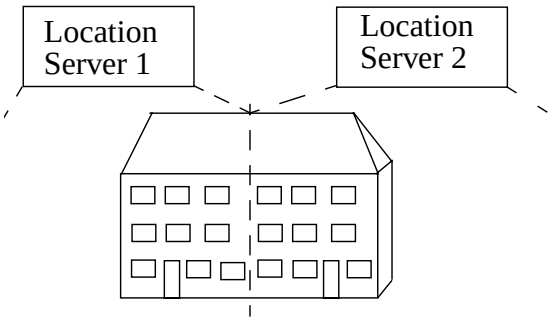


Figure 8-3: Distributed onEnterObject

8.3.3 onMeeting

An *onMeeting* event is always being observed by the location server that is currently handling the specified object in the onMeeting predicate. For example, if the predicate is *onMeeting(Jim, Computer Science professor, distance(<50m))*, the location server responsible for the area in which Jim is currently staying is responsible for the observation.

A problem occurs if Jim is closer to the edge of the area than the distance specified in the *onMeeting* predicate. In this case, a CS professor may be closer to Jim than the distance specified in the predicate, but the location server cannot find out, because the professor is not within its area (Figure 8-4).

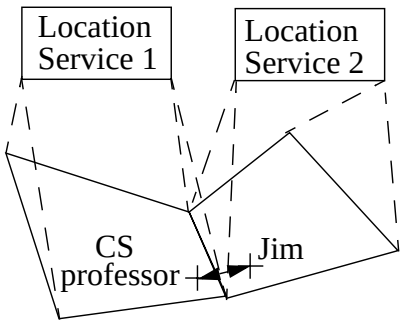


Figure 8-4: Distributed onMeeting

There are the following options to deal with this problem:

1. Do not allow *onMeeting* across areas covered by different location servers.
2. Let the areas of the location servers overlap and limit the maximum distance for *onMeeting* to the overlapping distance.
3. Let some other instance, e.g. the Observation Service, track the location of objects in the relevant areas.The area to be tracked changes dynamically with the movement of the specified object.
4. Let the location servers cooperate, so that one location server tracks objects in the relevant area covered by the other location server. Again, this area is changing dynamically.

Since *onMeeting* is an important predicate, it should be supported across location server boundaries. Therefore the third and fourth option should be investigated further.

8.4 Summary

In this chapter we have shown what kind of spatial events can be offered by the Nexus platform. The selection of possible events depends on the availability, but also on the accuracy of the underlying model data. In Nexus, different services are responsible for providing the location information about mobile and static objects. Therefore, the observation of some basic events involves both the Location Service and the Spatial Model Service and has to be coordinated by the Observation Service. The distributed observation of basic events across the boundaries of single servers is not trivial, but, as we have seen by the examples in Section 8.3, it is feasible. Altogether, we have demonstrated how spatial events can be supported by the Nexus platform.

Chapter 9

Conclusion

In this thesis we have investigated the area of Internet-scale event management with a special focus on the requirements of mobile users. The first step was to define the terminology for the terms that are relevant in this context, such as *event*, *predicate*, *observation* and *event notification*. This was important, because the terminology is not used consistently throughout the literature. We also introduced a simple event observation/notification model that helped in classifying related work.

After analyzing the relevant concepts and properties of event-based communication, mobility and spatial events, we introduced the Nexus project. From its key concepts we derived a design for the high-level architecture of the Nexus Event Service.

We identified three logical components, the Observation Service for observing events, the Notification Service for delivering event notifications to interested clients and a Predicate Management component that is responsible for managing the information pertaining to the events.

We investigated important architectural issues and discussed different design alternatives regarding the Event Service, especially the interaction of the Event Service with the other Nexus services and the distribution of the Event Service. The observation of events should take place as close as possible to the event origin. In order to place the event observation close to the event origin, the Event Service needs to know about the distribution of the other services. A global federation hides certain aspects of the distribution of the other Nexus services. Therefore, the Event Service should be integrated into the global federation and offer the same level of abstraction to its clients.

Finally, we discussed which of the identified spatial events can be implemented in the Nexus context and how they can be observed. For a couple of representative examples we showed how the observation of spatial events can be distributed.

Altogether, we have shown that developing an Event Service for Nexus is both desirable and feasible. This thesis has structured the wide field of event management and has laid the foundation for future work in this area.

9.1 Future Work

This section gives an outlook on future work based on the results of this thesis. First, we will present a number of questions and ideas that could not be covered in detail in this thesis due to the limited amount of time that was available. Then, we put the thesis into the larger context of a possible Ph.D. research project and finally we outline what the next steps could be.

So far we have assumed that the predicates themselves contain information about their scope, e.g. the location and the time for which they are valid. Alternatively, scope could be implemented as a separate concept. For example, a client might be interested in *onEnter* events for all offices in the computer science department. Instead of registering predicates for each event separately, the client could register for an *onEnter* event and give the scope as all rooms in the computer science department. It could also use a scope for the time period for which it is interested in the event, e.g. from 9:00 to 17:00 on weekdays.

Whereas this kind of scope is mainly of interest to the client, the Nexus services could also restrict the scope in order to ensure that clients can register only for events it can handle. For example, it may be feasible to observe an event like two professors meet within a department building, but it may not be feasible to observe it for the whole of Germany. So this scope has to do with policy issues, which we will discuss in the following.

In this thesis we have primarily looked at mechanisms for event management, and we have mostly left out policy issues. For an event mechanism to be successfully used in a commercial environment, some thoughts have to be given to those issues, especially in view of the following questions:

Who bears the costs, both computationally and financially, for event observation and notification? This is important regarding the question of why somebody would want to provide an Event Service. There has to be some incentive, either directly, because fees can be collected, or indirectly, because the added value makes it worth for the provider to offer the service. Of course these issues are relevant for the Nexus platform as a whole.

As far as the costs are concerned, the placement of event observations is of special importance. Can the observation be placed at the home location of the inquirer? If this is not possible or desirable, who will accept the event observation? Some locations may be more popular than others and therefore carry a higher load.

What kinds of predicates can be accepted for observation and who may define them? For example, registering a predicate that results in 10,000 events per second may overload the system. So predicates have to be checked to determine how “reasonable” they are, and they must be dropped when a certain limit is reached. The question is what instance determines this, and how it is determined.

Are there different classes of clients that can register for different classes of events? For example, simple clients may only register for basic predicates with a limited scope, whereas privileged clients may introduce their own plug-in triggers with unlimited scope.

As a result of our investigation into the field of event management, we have identified four main research areas that may provide the basis for a Ph.D. research project. For each of the four areas, we have formulated a research objective:

1. Communication Mechanism

   A suitable communication mechanism for delivering event notifications has to be developed that takes into account the characteristics of events and mobile recipients, but also quality of service requirements.

2. Global Predicates

   An efficient mechanism for evaluating complex global predicates in a distributed fashion has to be developed.

3. Mobility

   The special characteristics of mobile clients regarding the observation of events and the delivery of event notifications has to be investigated. Suitable fault semantics have to be found in case of disconnections.

4. Distribution

   Different strategies for distributing the Observation Service have to be evaluated.

In the following we will list possible first steps that may bring us closer to the realization of the research objectives:

- Analysis of a typical scenario to derive requirements regarding the frequency and distribution of spatial events.
- Implementation of a suitable communication mechanism based on these requirements.
- Comparison of different methods to represent and evaluate complex events.
- Design of an interface for plug-in triggers.
- Simulation of different distribution strategies for the Event Service.
- Implementation of an Event Service prototype.

Altogether, this thesis can be seen as a first step into an interesting and promising research area. We hope that many more steps will follow.

Part III: Appendix

A Commercial Notification Services

This Appendix gives an overview of two specifications for notification services in distributed environments, the CORBA Event Service [OMG 1997] and the Java Distributed Event Specification [SUN 1998] and two commercial products, Orbix Talk [IONA 1996] and TIB Rendezvous [TIBCO 1999]. All these services concentrate solely on delivering event notifications to groups of clients, there is no event observation component.

A.1 CORBA: Event Service Specification

Underlying model: :

Figure A-1: CORBA Event Channel

The underlying idea of the CORBA Event service is to decouple the communication between CORBA objects. This is achieved by introducing an event channel between supplier and consumer objects (Figure A-1). The suppliers produce event data, the consumers process event data. The event channel is a CORBA object that allows multiple suppliers to communicate with multiple consumers asynchronously. It is both consumer and supplier of events.

The CORBA event service supports two different communication models, the *push* and the *pull* model. In the push model, the supplier takes the initiative to transfer event data to the consumer, in the pull model, the initiative is on the side of the consumer, who requests the event data from the supplier. It is also possible to have push suppliers and pull consumers at the same time. In this case the event channel buffers the event data.

The actual communication can be either generic or typed. In the generic case, the push and pull operations take a single parameter of the type *any,* into which all the data has to be packed. In the typed case, the operations have to be defined in OMG IDL.

Design Principles:

- distributed environment -> no global, critical or centralized service (possible disconnections)
- multiple consumers and multiple suppliers
- consumers can request events or be notified of events
- single supplier request to communicate event data to all consumers at once
- suppliers do not need to know the identity of consumers and vice versa
- different qualities of service possible (e.g. levels of reliability, semantics: at-most-once, at-least-once, exactly-once)

For complex events, it is possible to construct a notification tree of event consumers/suppliers checking for successively more specific event predicates.

A.2 Java: Distributed Event Specification

Definitions:

Event: Something that happens in a Java object. This corresponds to some change in the abstract state of the object.

Underlying model:

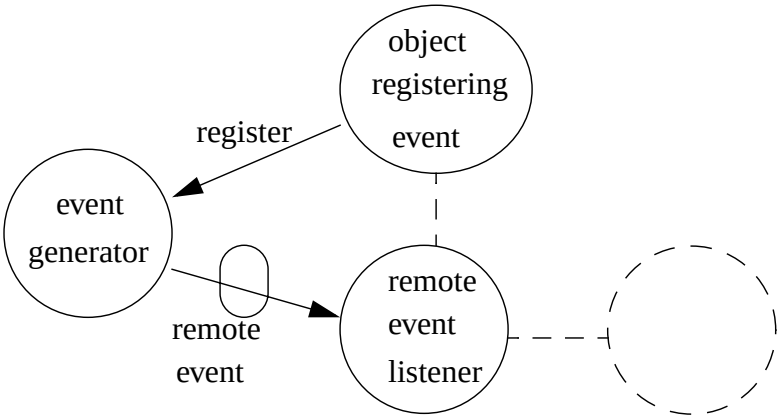


Figure A-2: Java Distributed Event Model

The Java distributed event service allows an object running on one Java Virtual Machine to register interest in the occurrence of some event occurring in an object on some other Java Virtual Machine.

The event generator is responsible for identifying the kind of events that can occur, allowing other objects to register interest in the occurrence of such events.

The object registering for event notifications can be different from the object that will be receiving the notifications (Figure A-2). The intention behind this architecture is to allow the placing of third party objects between the object generating the notification and the party receiving the notification. Examples of such third party objects are store-and-forward agents, notification fil-

ters and notification mailboxes. The third party objects may be able to offer certain delivery guarantees.

When an event occurs, the event generator creates “Remote Event” objects that are sent as a notification to the objects that have registered interest in the occurrence of such events. The registration for event notification is limited to a certain time (lease). The lease can be renewed. In this model, the event generator needs to know about all its listeners.

A.3 Orbix Talk

Definitions:

Message: Piece of information that is transferred via some (possibly decoupled) communication mechanism from a point of origination to a potentially large number of interested parties. The information arises unpredictably in response to real-world *events*.

Messaging System: Enable processes to register interest in specific events, and make them aware of these events as they unfold.

Underlying model: :

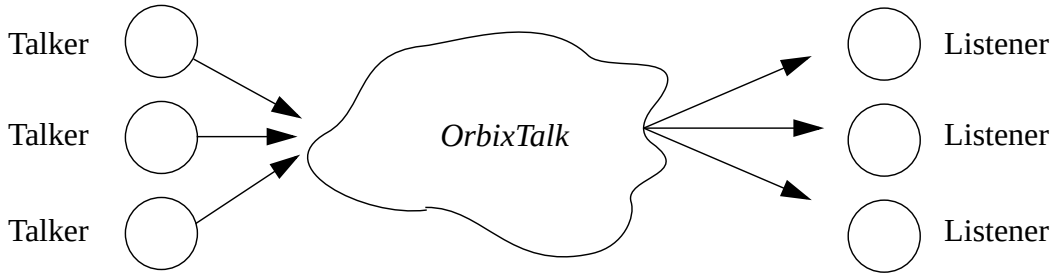


Figure A-3: OrbixTalk Model

Iona presents Orbix Talk as a “scalable messaging support for distributed object systems.” The communicating entities are called Talkers and Listeners (Figure A-3).The messaging is decoupled and unidirectional. Any number of talkers can issue messages to all listeners without knowing anything about them. The information that is shared through the messages is organized into a hierarchical structure of topics. The topics are identified by topic names, so a listener can inform OrbixTalk which topics it is interested in using those names.

Implementation Details:

Orbix Talk uses its own reliable multicast protocols: OrbixTalk Reliable Multicast Protocol (otrmb) and OrbixTalk TalkStore Protocol, store and forward mode (otsfp) and store-only mode (otrsop). These protocols are implemented on top of IP Multicast. OrbixTalk translates Topic Names to IP Multicast Groups. More than one topic can be multiplexed on the same group. The other side has to do the necessary filtering to extract the topics it is interested in.

The reliable multicast protocol uses sequence numbers and it automatically rerequests missing packets. Periodical messages indicate the last sequence number, so it can be detected if the last message is missing.

For the persistent modes, an OrbixTalk TalkStore is used that writes messages persistently on disk, in a sequential log. In the store-and-forward mode the messages are delivered to the listeners after they have been stored. In the store-only mode the message is delivered to the listeners while the message is written to persistent storage. In case of a failure, a listener can ask the TalkStore for needed messages after restarting.

A.4 TIBCO, TIB/Rendezvous

Underlying Model:

TIB/Rendezvous applications communicate by exchanging messages over the TIBCO information bus. Messages contain “self-describing data” - data structures that contain information about their data types, sizes and names.

There are three different interaction styles:

- publish/subscribe
  The model is event-driven rather than demand-driven. Publishing events are independent of subscriptions. For publishing, only one “broadcast message” is used. Broadcast messages can use IP broadcast or multicast addressing, depending on the availability of multicast capable hardware and software (Figure A-4).
- request/reply interactions (point-to-point)
- broadcast request/reply interactions
  In this case multiple servers get the request, which is useful for load balancing and the distribution of subtasks.

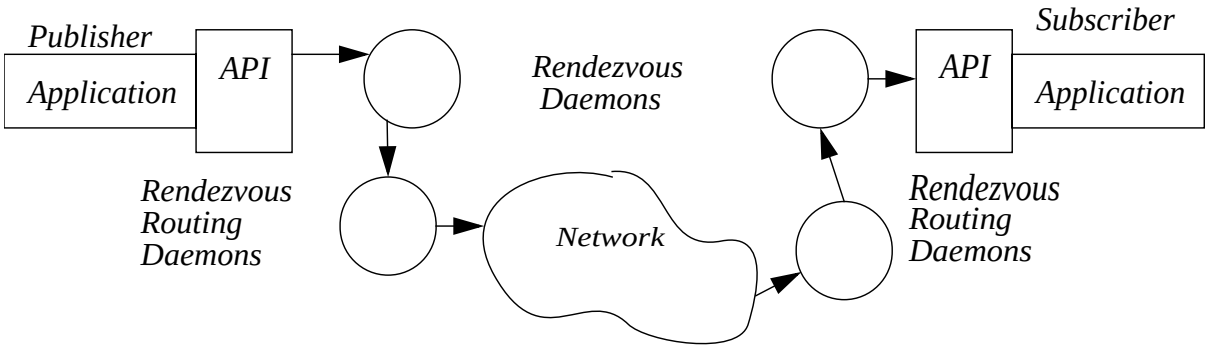


Figure A-4: TIBCO’s Publish & Subscribe Model

TIBCO’s patented “subject-based addressing” [TEKNEKRON 1993] is used for delivering a message to the destination(s). When a subscriber listens to a subject, the subject name is associated with a callback function that is dispatched by matching subject names.

TIB/Rendezvous offers two levels of reliability:

- reliable message delivery
- certified message delivery (explicit confirmation to sender for each registered recipient.)

A Recent Message Cache keeps the most recent message for the benefit of new listeners, but in general only applications that are running at the time the message is sent will receive the message.

Implementation Details:

Applications call the TIB/Rendezvous API functions to send messages, listen for messages etc. Behind the scenes the TIB/Rendezvous Daemons direct the flow of messages, divide and reassemble messages and guarantee reliability on top of unreliable broadcast/multicast protocols. TIB/Rendezvous Routing Daemons deliver messages between networks. For fault tolerance, the use of redundant processes is suggested.

B Mobile IP

The Internet Protocol (IP) is designed to allow routing in a static network. For this purpose, every device is assigned a fixed IP address. In order to save space in routing tables, a sub network address can be extracted by masking the low-order bits of the address, so that distant routers only need to know where to send packets for the whole network and do not need entries for every single device. This implies that the IP address of each device has to include the IP prefix of the sub-network it is connected to. Therefore, a mobile device connecting to different sub-networks would need a new IP address every time it changes sub-networks. On the other hand, connection oriented higher level protocols like TCP rely on a fixed IP address to maintain the connection.

The purpose of Mobile IP is to allow mobile devices to change networks, but to make this change transparent to applications and higher level protocols like TCP that need IP as a basis. The following description is based on [PERKINS 1997].

In Mobile IP for IP version 4, two IP addresses are assigned to each mobile device: a fixed home address and a care-of address. The home address is a valid address of the mobile device's home network. This address is used in TCP connections. The care-of address is an address of the network the mobile device is currently connected to.

When a mobile device connects to a new network, it has to register with the foreign agent and is assigned a care-of address (Figure B-1). The foreign agent has to run on some computer within the foreign network.

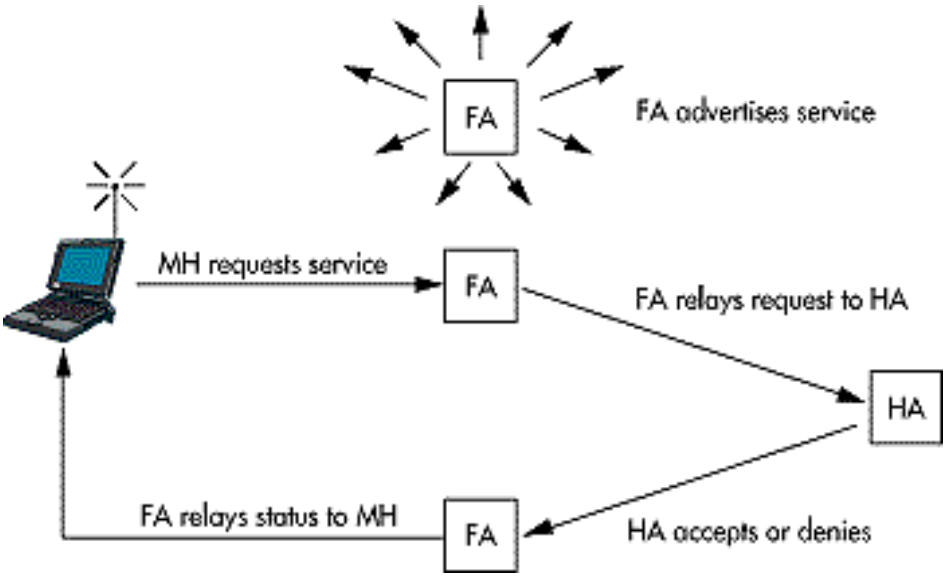


Figure B-1: Registering with a Foreign Agent
(taken from [ROTHERMEL 1997])

There are different ways for the mobile device to find the address of a foreign agent, e.g. by sending a broadcast to the network or by listening to the regular service announcements of the foreign agent. After receiving a care-of address, the mobile device generates a registration request and sends it via the foreign agent to the home agent. The request contains a hash (MD5) for the purpose of authentication.The home agent checks the hash and registers the care-of-address, if the hash is correct. It then sends a reply back to the foreign agent, which evaluates the reply and sends it on to to the mobile device. The mobile device in turn checks the authenticity of the reply.

A packet sent to the mobile device (via the home address) is always routed to the home agent (Figure B-2). The home agent looks up the current care-of address, encapsulates the IP packet by adding a new IP header and sends the encapsulated packet to the foreign agent. The sending of an encapsulated packet is also called tunnelling. The foreign agent unpacks the encapsulated packet and hands the original packet over to the mobile device.

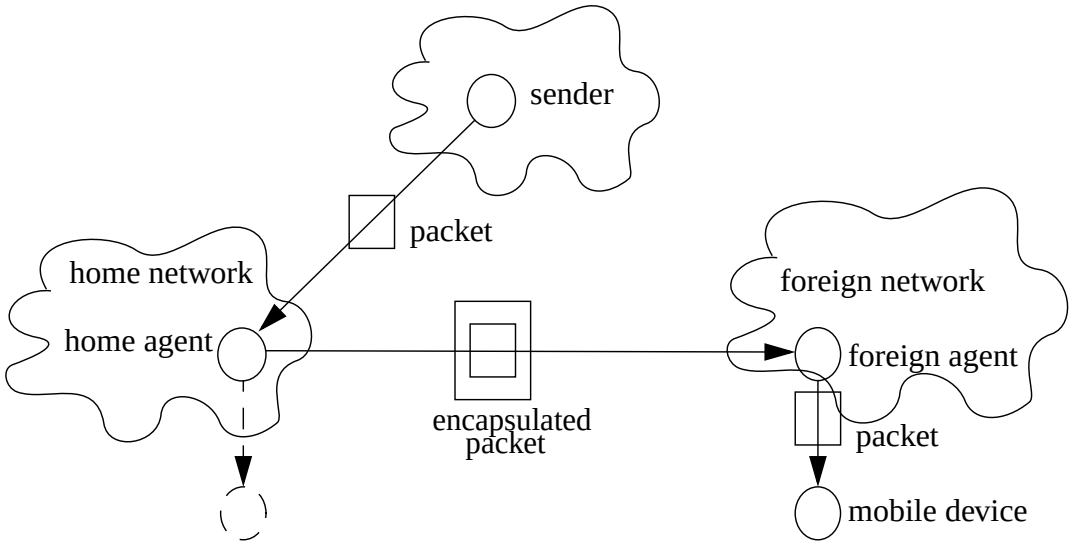


Figure B-2: Mobile IP

Mobile IP as presented here is a solution to the problem of allowing mobile devices in an IP network, but there are also a number of points that could be improved. The main problem is that communication between the sender and the receiver is always indirect. The packet has to be tunnelled between the home network and the foreign network. As an optimization, the mobile device can notify the sender to send packets directly to the care-of address of the mobile device. There is also a strong dependency on a functioning home agent and a functioning foreign agent.

C More Nexus Events

In this thesis we have mainly been concerned with simple spatial events. However, in the Nexus context, there are a number of other classes of events that should be supported. In the following we will give a short summary of those classes.

- Complex Spatial Events involving multiple objects plus state information
  - N mobile objects in area X
  - N mobile objects in room Y
  - object X has been at location Y M times
- Events involving time
  - event notification at a certain point in time (possibly further filtering)
  - no event has occurred within a certain interval of time
- Events involving other sensor input
  - temperature > or < X at a certain location
  - temperature > or < X with respect to a certain mobile object
  - the sun comes out/ it starts raining
  - wind speeds > X
  - door is opened/closed
- Events involving user input
  - new tea is ready
- Creation of new static objects
  - a new static object of type X has been added to the model
  - a new static object has been added to the model within area Y
- Events involving virtual objects
  - a Post-It has been attached to an object owned by X
  - a Post-It has been attached to an object in area Y

D References

[BADRINATH ET AL. 1993]

Badrinath, B. R.; Acharya, Arup and Imielinski, Tomasz
Impact of Mobility on Distributed Computations
Operating Systems Review, 27(2), April 1993
ftp://paul.rutgers.edu/pub/badri/sigop.ps.Z

[BAHL & PADMANABHAN 1999]

Bahl, Paramvir and Padmanabhan, Venkata N.
RADAR: An In-Building RF-based User Location and Tracking System
Technical Report MSR-TR-99-12, Microsoft Research, 1999
http://www.research.microsoft.com/~padmanab/papers/msr-tr-99-12.pdf

[BARRET ET AL. 1996]

Barrett, Daniel J.; Clarke, Lori A.; Tarr, Peri L. and Wise, Alexander E.
A Framework for Event-Based Software Integration
ACM Transactions on Software Engineering and Methodology, 5(4), ACM, October 1996

[BAUER 1998]

Bauer, Martin
A Collaborative Wearable System with Remote Pointing
Master Thesis, Department of Computer and Information Science, University of Oregon, USA, 1998

[BAUMANN ET AL. 2000]

Baumann, Joachim; Coschurba, Peter; Kubach, Uwe and Leonhardi, Alexander
Metaphors for Context-Aware Information Access
CHI 2000 Workshop, Den Haag, 2000

[BLUETOOTH 2000]

http://www.bluetooth.com

[CHASE & GARG 1998]

Chase, Craig M. and Garg, Vijay K.
Detection of Global Predicates: Techniques and Their Limitations
Distributed Computing, 11(4), Springer, 1998
http://link.springer.de/service/journals/00446/papers/8011004/80110191.pdf

[COSCHURBA 2000]

Coschurba, Peter

Personal communication, 2000

[COSCHURBA ET AL. 2000]

Coschurba, Peter; Kubach, Uwe and Leonhardi, Alexander

Research Issues in Developing a Platform for Spatial-Aware Applications

SIGOPS European Workshop, 2000

[COULOURIS ET AL. 1994]

Coulouris, George; Dollimore, Jean and Kindberg, Tim

Distributed Systems - Concepts and Design

Second Edition, Addison-Wesley, 1994
ISBN 0-201-62433-8

[DITTRICH & GATZIU 2000]

Ditrrich, Klaus R. and Gatziu, Stella

Aktive Datenbanksysteme - Konzepte und Mechanismen

Zweite Auflage, dpunkt.verlag, 2000
ISBN 3-932588-19-3

[ESRI 2000]

ESRI GIS and Mapping Software

http://www.esri.com

[EUROCONTROL 1998]

European Organization for the Safety of Air Navigation and Institute of Geodesy and Navigation, University FAF Munich

WGS84 Implementation Manual

Version 2.4, February 1998

[FORMAN & ZAHORJAN 1993]

Forman, George H. and Zahorjan, John

The Challenges of Mobile Computing

Technical Report 93-11-03, Computer Science and Engineering, University of Washington, November 1993

[HOHL ET AL. 1999]

Hohl, Fritz; Kubach, Uwe; Leonhardi, Alexander; Rothermel, Kurt and Schwehm, Markus

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, Washington, USA, 1999

http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1999-hohlEA-01.ps.gz

[HP 2000]

Hewlett Packard

Jornada Handheld PC

http://www.hp.com/jornada/

[IMIELINSKI & NAVAS 1997]

Imielinski, Tomasz and Navas, Julio C.

Geographic Addressing, Routing, and Resource Discovery with the Global Positioning System

Communications of the ACM, 1997

http://www.cs.rutgers.edu/~navas/dataman/papers/cacm99.ps.gz

[IONA 1996]

IONA Technologies

OrbixTalk (TM) White Paper

IONA Technologies Limited, April 1996

[KUBACH 1999]

Kubach, Uwe

A Universal, Location-Aware Hoarding Mechanism

Proceedings of the International Symposium on Handheld and Ubiquitous Computing (HUC ’99), Karlsruhe, Germany, 1999

[KUBACH ET AL. 1999]

Kubach, Uwe; Leonhardi, Alexander; Rothermel, Kurt and Schwehm, Markus

Analysis of Distribution Schemes for the Management of Location Information

Technical Report TR-1999-01, IPVR, University of Stuttgart, 1999

http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1999-kubach-01.ps.gz

[LAMPORT 1978]

Lamport, Leslie

Time, Clocks and the Ordering of Events in Distributed Systems

Communications of the ACM, 21(7), July 1978

[LEONHARDI & KUBACH 1999]

Leonhardi, Alexander and Kubach, Uwe

An Architecture for a Distributed Universal Location Service

Proceedings of the European Wireless ’99 Conference, Munich, Germany, 1999

http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1999-leonhardi-01.ps.gz

[MARIMBA 1997]

Marimba

Castanet White Paper

Marimba Inc., 440 Clyde Avenue, Mountain View, CA 94043, USA, 1997

[OMG 1997]

Object Management Group (OMG)

OMG CORBA Services, Chapter 4 - Event Service Specification

Framingham Corporate Center Object Management Group, Inc. 492 Old Connecticut Path Framingham, MA 01701, USA, December 1997

http://www.omg.org/cgi-bin/doc?formal/97-12-11.ps

[ORACLE 2000]

Oracle

Oracle8i Release 2 New Features Summary - Features Overview

Oracle Corporation, 2000

http://www.oracle.com/database/
documents o8i_r2_new_features_summary_fo.pdf

[ORWELL 1949]

Orwell, George

Nineteen eighty-four

Penguin Books, Middlesex, England, printed 1964, first published 1949

[PALM 2000]

Palm Inc.

http://www.palm.com

[PATON & DÍAZ 1999]

Paton, Norman W. and Díaz, Oscar

Active Database Systems

ACM Computing Surveys, 31(1), ACM, March 1999

http://www.acm.org/pubs/citations/journals/surveys/1999-31-1/p63-paton/

[PERKINS 1997]

Perkins, Charles E.

Mobile Networking Through Mobile IP

Tutorial, Internet Computing online, IEEE, 1997

http://computer.org/internet/v2n1/perkins.htM

[RIFKIN & KHARE 1998]

Rifkin, Adam and Khare, Rohit

The Evolution of Internet-Scale Event Notification Services

http://www.cs.caltech.edu/~adam/isen/wacc/

[ROSENBLUM & WOLF 1997]

Rosenblum, D.S. and Wolf, A.L.

A Design Framework for Internet-Scale Event Observation and Notification

Proc. Sixth European Software Engineering Conf./ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering, Zurich, Switzerland, September 1997

http://www.ics.uci.edu/~dsr/

[ROTHERMEL 1997]

Rothermel, Kurt

Rechnernetze I

Lecture Notes (German), IPVR, University of Stuttgart, Germany, 1997

[SATYANARAYANAN 1996]

Satyanarayanan, Mahadev

Fundamental Challenges in Mobile Computing

Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, May 1996

http://www.acm.org/pubs/citations/proceedings/podc/248052/p1-satyanarayanan/

[SCHWARZ & MATTERN 1994]

Schwarz, Reinhard and Mattern, Friedemann

Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail.

Distributed Computing, Springer, 7(3), 1994

http://www.inf.ethz.ch/department/IS/vs/publ/holygrail.pdf

[SONY 2000]

Sony

Vaio Notebook Computers

Sony Electronics Inc., 2000

http://www.ita.sel.sony.com/products/

[STREIT 1999]

Streit, Ulrich

Einführung in die Geoinformatik

Lecture Notes (German), Institut für Geoinformatik, University of Münster, Germany, 1999

http://castafiore.uni-muenster.de/vorlesungen/geoinformatik/frames/fsteuer.htm

[SUN 1998]

Sun

Distributed Event Specification

SUN Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043-1100, USA, March 1998

[TEKNEKRON 1993]

Technekron Software Systems

Apparatus and Method for Providing Decoupling of Data Exchange Details for Providing High Performance Communication between Software Processes

U.S. Patent Document, Patent No. 5,187,787, February 1993

[TIBCO 1999]

TIBCO

TIB/Rendezvous - White Paper

TIBCO, PALO ALTO World Headquarters TIBCO Software Inc. 3165 Porter Drive Palo Alto, CA 94304 USA, 1999

[WANT ET AL. 1992]

Want, Roy; Hopper, Andy; Falcão, Veronica and Gibbons, Jonathan

The Active Badge Location System

ACM Transactions on Information Systems, 10(1), 1992

http://www.acm.org/pubs/citations/journals/tois/1992-10-1/p91-want/

[XYBERNAUT 2000]

Xybernaut

Xybernaut Mobile Assistant IV

Xybernaut Corporation, 12701 Fair Lakes Circle, Suite 550, Fairfax, Virginia 22033, USA

http://www.xybernaut.com/product/prod_serv.htm

Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst
und nur die angegebenen Hilfsmittel verwendet habe.

_____

Martin Bauer