Eidgenössische
Technische Hochschule
Zürich, Schweiz

Ecole Suisse Polytechnique Fédérale de Zurich
Politecnico Svizzero Federale di Zurigo
Swiss Federal Institute of Technology at Zurich

Institute of Information Systems
Distributed Systems Group
Prof. Dr. F. Mattern

Web: http://www.inf.ethz.ch/vs/

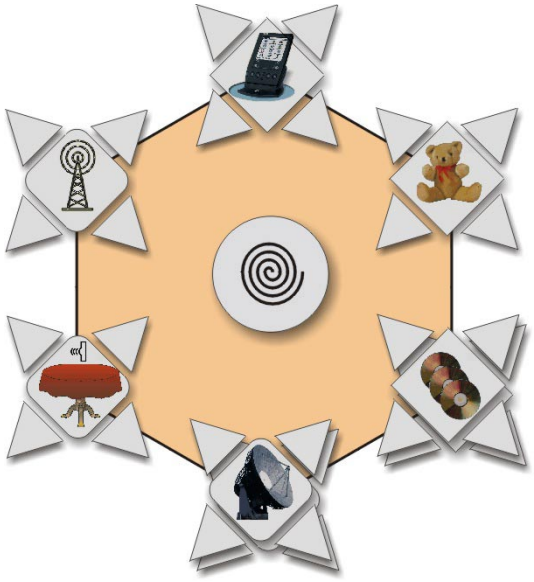
An extensible infrastructure and a representation scheme for distributed smart proxies of real world objects

steps toward a smart environment

Thomas Dübendorfer

Assistant: Kay Römer
Supervisor: Prof. Friedemann Mattern

5th April 2001



An extensible infrastructure and a representation scheme for distributed smart proxies of real world objects

Master's Thesis of Thomas Dübendorfer, thomas@duebendorfer.ch,
ETH Zurich, Switzerland

Document History:

March 2001	First edition
April 2001	Second edition, published as Technical Report[a] **TR 359** of ETH Zurich

[a] `http://www.inf.ethz.ch/publications/`

Contents

Abstract

An extensible infrastructure and a representation scheme for distributed smart virtual proxies of real world objects as presented in this Master's Thesis could be a foundation to make the environment smart by enhancing simple objects with smart virtual counterparts. Design, implementation and deployment issues of concepts such as virtual counterparts (Virtual Objects), proximity (Virtual Location), meta objects (Virtual Meta Objects and Locations), lifecycle of virtual counterparts (Virtual Object Manager) and a queriable persistent Artifact Memory are discussed.

Zusammenfassung[1]:
Eine erweiterbare Infrastruktur und eine Repräsentationsform für verteilte intelligente virtuelle Stellvertreter von realen Objekten wie sie in dieser Diplomarbeit präsentiert werden, könnten den Grundstein legen, um unsere Umgebung intelligent zu machen, indem einfache Gegenstände durch die Intelligenz ihrer Stellvertreter ebenfalls intelligent werden. Es werden Entwurfs-, Implementierungs- und Anwendungsaspekte von Konzepten wie virtuelle Stellvertreter (Virtuelle Objekte), Nähe (Virtuelle Orte), Metaobjekte (Virtuelle Metaobjekte), Lebenszyklus der virtuellen Stellvertreter (Virtueller Objektmanager) und ein abfragbarer persistenter Artefaktspeicher diskutiert.

[1] A german summary is required by the department of computer science at ETH for a Master's Thesis written in English

1 Introduction

1.1 Vision

Although the development of computers shifted society to the information age we are still champions in keeping valuable information inaccessible or not making use of it at all. If information is not recorded at the time of creation then its reconstruction is likely to be cumbersome. Having the right information at the right place and at the right time is often crucial - especially when we are keen on saving time.

A common way of spending productive time is for instance looking for everyday things as they somehow just seem to have vanished. This is especially true if there are many people using them. The specific thing we are looking for certainly knows where it is at the moment, but we don't. We could also say that the location information is inaccessible. If there would be a possibility of asking a hidden object about its current place of presence we would certainly use that means to detect it. The same holds for people we want to reach and cannot find.

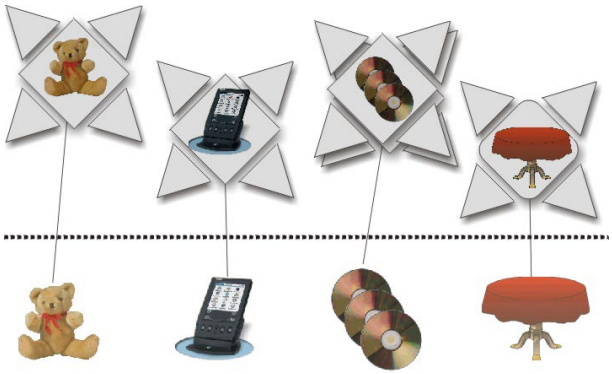


Figure 1: *Each physical object could have a smart virtual counterpart.*

Have we ever thought of what an everyday thing such as a book or a toy could tell us? A book might inform us when a new edition has been published, show the latest list of errata to us when we hold it close to a screen and suggest further reading. It could also tell how long we have spent reading it and which other books have been standing close to it on the shelf. So if once we can't remember the exact name of a book we lent to a friend we can just ask a book that was near to it on the shelf. A personal toy could tell us about when it was on holidays with us, which hotel we visited and which people we met there.

Might it really be possible to enhance everyday things and make them smart? This really sounds incredible for the moment as a book or a simple toy clearly has no input and output devices and so it can neither see nor tell.

Imagine that each thing is uniquely identifiable and that sensors can recognize things close to them. We get information about when a thing is at which location together with which others. The more we know about a specific thing the better we can understand actions in a context and the more information we can deduce from simple recorded events.

Let us suppose that each physical object has a smart virtual counterpart that gets the information about its managed physical object from the sensors and that it will process and store them. This opens up for a broad new field of information providing and introduces new innovative interaction patterns.

As an example, when we hold a paper business card close to an organizer the virtual counterparts of the organizer and the business card could exchange the information printed on it without us having to do any typing at all. The organizer can then connect to its virtual counterpart and retrieve the new address information. In the future it will be sufficient to collect the unique identity of a poster hanging on a wall somewhere in order to retrieve its full information later again without us having to grab for a pen and a notepad. The virtual counterpart knows that we have been at a certain location and has noticed our interest in the specific poster.

When we even go a step further and add some input and output devices to a toy as it is a goal of the Smart-Its[2] project, we could add a microphone, a loudspeaker and a wireless link to a simple teddy bear and enable it to translate spoken phrases into another language by having it send the recorded voice signals to its virtual counterpart and then playing the translation that it receives as the reply by using its loudspeaker. Another toy could sound an alarm tone or even make its virtual counterpart call someone if the room temperature gets too hot and if there has been no one watering the plants in the room for a longer time.

A virtual counterpart tries to collect and make use of all information related to the physical thing which it is associated to. When we take our favorite toy into our car then the toy's virtual counterpart should be able to hop into the car too before we leave. It then travels with us and could tell the whole story of what has happened on the journey, where we have been and who we have met. We can query it on facts of which we cannot remind the

[2] cf. [28]

details anymore and it could tell us the missing pieces. The virtual world where the virtual counterparts are living in is a partitioned one. The small virtual world of the car disconnects from the house's bigger virtual world for the journey and reconnects when it is back again. Virtual counterparts can therefore temporarily move out of reach of each other and return somewhen later.

Today's technology is on the verge of allowing us to make simple things smart by giving each unique identifiable thing a smart virtual counterpart. We need versatile representations for things and locations and an infrastructure has to provide some basic services such as dynamic creation of virtual counterparts, event-based communication, a lookup mechanism to find other virtual counterparts and a way to store information persistently and query on it. In the future as miniaturization of electronic devices has progressed that far such that we can hardly distinguish dust from sensors and wireless communication nodes anymore, we might even talk of smart dust (cf. [29]).

1.2 Background

Several projects in the Distributed Systems Group of Prof. Friedemann Mattern at ETH Zurich are related to current research topics in the field of ubiquitous computing. The main focus lies on generic concepts of a common infrastructure, foundations for device-independent services, new interaction paradigms, as well as on security aspects for communicating and cooperative devices. All of the projects aim at finding the requirements which arise from a proliferation of "smart devices", "spontaneous networking" and "nomadic users". The final goal is to define future steps for a realization of the technical foundations for ubiquitous computing.
As a first step a small infrastructure should be designed and implemented in the Ubiquitous Computing Lab of the Distributed Systems Group, which should demonstrate in a variety of examples the fundamental concepts in the field of ubiquitous computing such as location- and context-awareness and virtual representations of physical objects. This will be a starting point for future investigations and thorough research.

1.3 Scenario

The infrastructure and the representation scheme for distributed smart proxies of real world objects that were designed and implemented in this thesis do enable scenarios like the following one.

The general concept of a virtual representation of everyday things should be demonstrated in a presentation. To each object (e.g., a coffee mug, a book) a unique identity in the form of a radio-frequency identification (RFID)

tag is stuck. Several RFID antennas at different locations can sense these tags. The virtual representations of the locations as well as the virtual counterparts of the objects are capable of receiving and storing the events of tags entering and exiting the specific physically limited place surrounding an antenna.
A tool should enable to query the objects where they are and where they have been and to query the locations which objects are present. Intersecting the sets of visited locations could then reveal that objects have already met before.

1.4 Aspects

The thesis introduces several new terms which are defined in section 3 and it puts emphasis on the following aspects.

- Virtual Object (VO) and Virtual Location (VL) Modelling
- Paradigm of Virtual Meta Objects (VMO) and Virtual Meta Locations (VML)
- Virtual Object Management (VOM)
- Virtual Object Repository (VOR)
- An event-based publish/subscribe model for Virtual Objects and Virtual Locations
- An Artifact Memory (AM) to store and query event histories and state data

1.5 Focus

The core of this Master's Thesis is concerned with building an extensible core software infrastructure for ubiquitous computing and with defining generic concepts in this field as well as with finding a versatile representation scheme for modelling virtual counterparts of physical objects and locations.

2 Related Work

In this section we present some past and current projects related to the vision stated in 1.1 on page 8. We start with discussing object tracking systems which can be seen as the very basis of the more elaborate ubiquitous computing systems and visions that we review subsequently.

2.1 Object Tracking

A basic feature that a software and hardware infrastructure for ubiquitous computing should provide is tracking objects and people in order to tell where they are at the moment or where they have been. As a matter of fact most people tracking systems built so far do not really track people but objects that people carry with them such as badges or tags. The term "object tracking" which we will prefer from now on is more general and moreover does not have the flat taste of an undesirable "Big Brother" that is always watching you and that knows all about you. By the way this fear is heavily related to the fact that most tracking systems are managed by a single centralized authority instead of being truly distributed and allowing for independent objects that can be under full control of the user itself.

Knowing the location of a badge can be used to enable a broad range of applications such as making it easy to locate people in an office environment, open doors automatically with restrictions depending on who is arriving, regulating the heating, switching off lights in rooms where no one is present and letting them on even at night if some people are still working, routing calls to the closest phone and printing on the closest printer, providing a tourist with a location dependent city map with a moving "you are here mark" and suggestions to visit important buildings in the current neighborhood, ticketing for public transport systems and many others still to be conceived.

Active Badge Location System (Olivetti Lab, 1992) The often cited Active Badge Location System (cf. [1]) is one of the first badge tracking systems ever built. It helps locating people in an office environment. Each active badge transmits almost regularly about every 15 seconds its unique code as a short pulse-width modulated infrared (IR) signal that is perceived by a near sensor and then sent to a centralized tracking database through the existing wired network. The tracking database shows the current locations of the staff (including 'AWAY' and probabilities) and supports a simple high-level query and command language to retrieve location histories or among other things to generate a notification signal upon the next sighting of a given badge. Privacy concerns were addressed by emphasizing that badges are

located and not people and by not keeping any long-term location records.

ParcTab Ubiquitous Computing (Xerox PARC, 1995) The ParcTab project (cf. [2]) encompasses as its core the construction of a palm-sized portable device called 'ParcTab' with a touch-sensitive display, three buttons and an infrared transceiver. This small device acts as a thin client that displays server-side generated display data and sends user actions through an infrared uplink to a preestablished non-moving server agent in the local wired network. The communication between agent and infrared gateway is based on SUN®'s Remote Procedure Call (RPC) technique. A beacon sent by the device every 30 seconds helps locating its user. This enables a broad variety of applications that can make use of location information.

Location Based Personal Mobile Computing and Communication (University of Wollongong, 1998) Like a revival of the ideas of the Active Badge Location System and the ParcTab Experiment this project (cf. [3]) brings both together and defines three types of badges. A Dumb Badge simply sends beacons with its unique identity number. A Smart Badge, which was actually built in this project, is a Dumb Badge enhanced by a collection of sensors (orientation, humidity, temperature, light, microphone) and actuators (piezoelectric transducer) as well as by the capability to be reprogrammed. An Intelligent Badge is the culmination in this hierarchy as it additionally provides enhanced I/O capabilities such as a display, audio, video or motion to enable new ways of interaction.

RFID Based Identification and Location Systems
The radio-frequency identification (RFID) technology as described in [9] makes use of the induction that electromagnetic fields cause in a metal wire. This enables passive tags that use only the energy of the electromagnetic sensor field to send the beacon with the unique identification number when they are close to a sensor. The supported distances from the RFID tag to the sensor is used to categorize the various systems. It is differentiated between close coupling (0 - 1 cm), remote coupling (15 cm - 1 m) based on inductive coupling where no battery is needed for the transponder in the RFID tag and long range (1 m - 10 m) systems based on microwave communication that need a separate power support. The identity number transmitted ranges from a simple 1 bit to symbolize the states "present" and "not present" up to about a thousand bytes. More powerful RFID devices possess a microprocessor and some

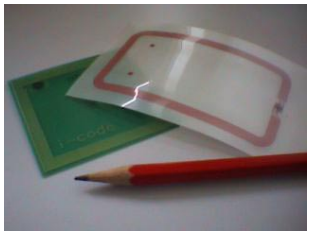


Figure 2: *A transparent RFID label of type remote coupling.*

non-volatile memory. They can encrypt messages and run customized applications. Possible applications of the RFID technology are as different as contactless smartcards, ticketing for public transport systems, animal identification and automatic individual feeding stations, access control, electronic locks, electronic article surveillance, time measurement at sport events and industry automation.

Others There are many other systems that help to locate objects. For instance, a specialized middleware (cf. [4] and [5]) was developed that supports the movement of user interfaces for multimedia teleconferences (based on X) depending on the user's location. Or for instance the MediaCup (cf. [6]) that finds out about what a user is doing with a coffee mug that is enhanced with some sensors and an RFID tag. Our discussion of related work is not to be taken as a final survey.

2.2 Ubiquitous Computing Visions and Systems

The vision of systems that support ubiquitous computing applications is not brand new. There already are some visions and systems that go into that direction. The central idea could be summarized by the phrase "making simple things smart by giving each unique identifiable object a virtual counterpart".

CALAIS (University of Cambridge, 1998) The CALAIS system as described in [10] is an event-based distributed system that consists of event services and applications. Sensors generate events which are matched against templates and pushed to subscribed applications. There are also some not further specified event query mechanisms provided. This distributed system relies on CORBA® as the communication middleware. It seems that the system itself was neither released to the public nor is it in productive use.

Nexus - An Open Global Infrastructure for Spatial-Aware Applications (University of Stuttgart, 1999) The challenging Nexus project (cf. [11]) is currently only a vision of augmented areas that will provide virtual representations of physical objects and links to an information space or service. The user will be able to query and send commands to the virtual representations. However, the first prototype of Nexus is due not before 2002.

Hive: Distributed Agents for Networking Things (MIT, 1999) Hive (cf. [12] and [13]) is an agent-based distributed system that was released publicly in 1999. Agents are installed in a so-called cell that provides a registration service and a weak migration multi-hop capability between

cells. Whereas the early Hive system was based on CORBA®[3] the publicly released system is based on Java^{TM}'s Remote Method Invocation (RMI). So-called shadows that are installed into cells can provide access to resources such as a user interface or a digital camera. The agents are installed and manually connected by the user by using a GUI and can be used to build distributed applications. The system consists of about 24.000 lines of Java^{TM} code.

CoolTown (HP Lab Palo Alto, 2000) The CoolTown project (cf. [14]) aims at giving each person, place and thing a web representation (cf. [15] and [16]) by associating each unique identifiable object a Word Wide Web URL[4]. Based on the common HTTP-protocol the user can interact with the virtual representation. A special device can be used to gather the unique identities and later resolve them into URLs.

Portolano (University of Washington, 2000) The emerging fields of ubiquitous and invisible computing are addressed by the Portolano project (cf. [17]). Invisible and worry-free user interfaces, a new data-centric networking infrastructure (cf. [18]) and distributed services will be developed in this project. As for now only conceptional and visionary papers exist.

Jini^{TM} - Java^{TM} Intelligent Network Infrastructure (SUN®) Jini^{TM} (cf. [35]) is a middleware that is based on the simple view that "everything is a service". The services are distributed in a network and Jini^{TM} offers discovery, join and lookup mechanisms to provide hassle-free access to them. A service can publish a description of itself on a lookup service and provide a proxy that is downloaded by the client and used to access the service functionalities as if the service were local. The Jini^{TM} environment requires no user intervention, it is self-healing (i.e., if leases expire the resources are automatically freed) and consumers of Jini^{TM} services are not obliged to have prior knowledge of the implementation of services they want to use. There is a variety of services that ships with Jini^{TM} such as reggie, the lookup service or mahalo, the transaction manager or mercury, an event mailbox service and some others. Jini^{TM} is still very young; in December 2000 version 1.1 was released.

Others Of course the above list of related visions and systems is not complete. There are some smaller projects as well where some conceptual papers have been published such as the mobile-floating agent scheme (cf. [24]) for wireless distributed computing by the University of Stockholm, 1995.

[3] Voyager 1.0 to be precise
[4] Uniform Resource Locator

3 Concepts

3.1 Location and Proximity

If we want to track objects we need a notion of where the object is at the moment.

3.1.1 Definition

*A **Location** is a physically or logically limited place.*

The following examples illustrate the above definition:

- **Physically limited place**: An object is in a room, lies on the surface of a table, resides on a bookshelf, hides in a toy box, is in a moving car or is inside of a building.
- **Logically limited place**: An object or a person is at a conference, in a dark forest or in vicinity to a good friend.

3.1.2 Categorization

The next step is to find a way to determine the current location of an object, i.e., to get a positional reference.

Position A Position can be *absolute* (e.g., GPS[5]) with respect to a coordinate system or *relative* to another object of which we usually know the exact position especially if it is part of the infrastructure.

Awareness Either the infrastructure or the object or both know the current position of the object.

Activity An object can either actively send beacons to mark its presence or it can remain passive until it is polled by a sensor.

Availability A positioning system can be available only locally (e.g., systems based on infrared or RFID sensors) or globally available. Although a system might be global this does not indicate that it is available everywhere. The global positioning system GPS for instance does no work inside buildings or in tunnels. Systems that learn from sensor input (e.g., from a video stream of a camera mounted on a robot) and are able

[5] GPS is the Global Positioning System, cf. [8]

to recognize the same location later again need a learning phase in advance and are only reliable for places they have already visited.

Reliability and accuracy It is important to know how reliable and accurate an information about a position is. These requirements vary strongly depending on the field of application such as for instance tracking parcels sent worldwide or objects moving within an office building or recording the movements of a simple pen that is enhanced to a virtual laser pointer by the infrastructure.

It has to be considered that a location can fully or partially overlap other locations.

Besides the proper location ($\vec{x}$) sometimes it might be useful to know also about speed ($\vec{v} = \vec{x}'$), acceleration ($\vec{a} = \vec{x}''$), change in acceleration ($\vec{a}' = \vec{x}'''$) and about orientation ($\vec{p} = [x, y, z]$). These additional measurements allow to predict positions in the near future even though the object might not be in reach of a sensor anymore.

The primary type of positioning systems that the infrastructure which was developed in this thesis will have to support consists of sensors at well-known positions that poll passive RFID tags in a limited place around them. But in fact, any type of positioning sensors that can report sightings at a unique location and generate appropriate entry and exit events can be seamlessly integrated. A later extension to support absolute positioning such as GPS (i.e., many different unique locations which should be processed by one or a few virtual counterparts of locations) is also possible.

3.1.3 Queries

Some common queries on the locations of objects are:

- Where am I? Where is object X?
- Which objects are near me? Which objects are at location X?

The first type of questions returns the current location. The second type returns a list of object identifiers.
It might be reasonable to rely on an already existing query language and only make a few extensions to it rather than invent a completely new one.

3.2 Unique identity

The concept of a unique identity that is assigned to an entity is in widespread use today. Although multiple assignments of unique identities to the same

entity are conceptually not needed, this practice is very common in real life. An interesting psychological fact is that nobody seems to care about the globally unique MAC (medium access control) address of each ethernet card but that the unique processor id in the Intel® Pentium^{TM} III processor was regarded as a big danger for personal privacy and had to be disabled by default.

The following list of examples leads us to a set of important criteria and shows how astonishingly widely accepted the (mostly unaware) use of unique identities in real life is.

- **Internet and Communication**: Domain name, IP address, MAC address, e-mail address, session id of a website, subscriber identity module (SIM) in a mobile phone
- **Computer Systems**: Login name, process id, software licence key, processor id, fully qualified file name, disk drive letter
- **Society**: Postal address, social security number (AHV/SSN), passport number, card of identity number, car license number, key to a lock
- **Finance**: credit card number, number on a bank note, account number, cheque number
- **Business and Commerce**: Employee number, order number, confirmation number, ISBN[6], EAN[7] code
- **Biometry**: Iris scan, fingerprint (16 characteristics are sufficient), DNS sequence
- **Others**: RFID tag

The term "unique" is not precisely defined in real life. We always have to consider two aspects: **Time** (temporary or permanent uniqueness) and **context** (in which an identity is unique). A session id on a website might only be unique for a couple of months and then be used again. A process id might be reused after the computer system has been restarted. A number on a bank note is only unique in the context of a currency and not worldwide for all bank notes. Instead of 'context' we could also say that an identity is only unique within a **domain** or namespace. Prepending a worldwide unique domain name[8] to an identity which is only locally unique results in

[6] International Standard Book Number
[7] European Article Number
[8] E.g., a registered Internet domain name such as "duebendorfer.ch".

a fully qualified name for the entity which is worldwide unique.

The reason why most identification numbers are only *temporarily unique* is that we tend to restrict the length of the identification representation for convenience of use and because of technical limitations. Nobody would like to have a 30 digit credit card number or car license number on his or her plate just to make sure that no one in the far future might ever get the same number again. On the other side for instance the available permanent memory of cheap RFID tags is very limited. As another example the 32 bit wide IPv4 network numbers are getting scarce and therefore certain sets of number assignments are only temporarily unique.

The use of *multiple assignments* of different unique identities for the same entity is so common because this practice makes the different assignment authorities fully independent of each other. It also reduces administration overhead as local assignment authorities do not have to be controlled by a worldwide master assignment organization. Furthermore innovative systems can establish their own proprietary identification rules that best suit their technical and organizational needs which a one-suits-all standardized solution can never offer. A person might even wish to get a different customer identity number for each service used in order to conceal his or her personal habits and get some guarantee for privacy. However, a unique identity does suffice to link all kinds of rights for services to an entity such as paying in a shopping mall, using it as a passport, using it as a key to open the doors, using it as a link to the cinema tickets bought by phone and many more. It would be best if the identity is not only assigned but inherent to the entity (e.g., the unique DNA sequence of a human).

As nice[9] as it would be to demand that each physical object, that will be represented in our infrastructure, has to provide its unique identity in form of a worldwide unique hierarchically constructed URI[10] this unfortunately would exclude the use of cheap RFID tags which use a few dozen bytes as the identification number. Therefore we cannot rely on a consistent naming scheme but we at least have to make sure that there are never two entities in the system at the same time with the same identification label.

A representation of a unique identity should allow for a reasonably large set of different identities to guarantee uniqueness over a longer time period and at the same time the representation should have a sensibly short length to remain convenient in use and in order not to set unnecessarily high technical

[9] URI-style names are short, legible object keys that can easily be written down, entered by hand, or exchanged over the phone and there is no cryptic deciphering needed.

[10] Universal Resource Identifier (URI). Example (as in CORBA's IIOP, cf. [38]): iioploc://host:1234/ImplName/ABC/DEF/ObjName

requirements. It would be advantageous if the identity is human readable and not only machine readable which excludes long hex codes and the like. Depending on the application domain it might be a very good idea to have a checksum included especially if humans have to enter the identity manually.

The representation of a unique identity that we will simply call *objectID* and which is used by the infrastructure presented in this thesis is a *system-wide unique unicode character sequence of variable length* with no enforced structure. This enables us to support a wide range of naming systems and allows for a hybrid naming scheme. The user group of the system can make its own set of conventional rules for a flat (e.g., random numbers) or hierarchical (e.g., URLs, IPv6 address, EAN code, etc.) naming scheme that is most appropriate for the intended application domain.

3.3 Smart Virtual Counterparts

The vision in 1.1 introduced the idea to enhance simple everyday objects with smart virtual counterparts as shown in figure 1. These counterparts live in a virtual world and can make use of a common communication infrastructure independent of which object they actually represent. We differentiate between *virtual objects* that stand for things including people and *virtual locations* that represent locations as defined in 3.1.1 on page 16.

3.3.1 Virtual Object

Each unique identifiable physical object becomes smart through a virtual counterpart that receives, processes and stores all events that are related to this identity. This Virtual Object can be regarded as a proxy for the physical object and is able to interact with other Virtual Objects. As the term "proxy" is heavily used in all kinds of contexts we refrain from using that term to denote the virtual counterpart and simply call it *Virtual Object* (VO).

Figure 3: *A Virtual Object.*

3.3.2 Virtual Location

Each physical (e.g., an RFID sensor) or logical (e.g., being at a conference) location (as defined in 3.1.1) has an associated virtual counterpart, the *Virtual Location*, where objects can subscribe with event templates for event notifications. A Virtual Object can be at more than one Virtual Location at the same time (e.g., on the table and

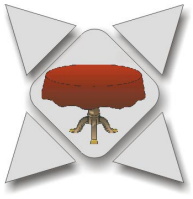


Figure 4: *A Virtual Location.*

in room F1).
Sightings of objects at a certain sensor automatically associate it with that location. A Virtual Location can be regarded as a specialization of the concept of a Virtual Object.

3.3.3 Lifecycle of Virtual Counterparts

Before a virtual counterpart can come to life, its representation has to be created and the infrastructure must know which physical object or which location it is associated to. When a tagged object is sighted by a component of the infrastructure the virtual counterpart corresponding to the unique identity which has been reported is dynamically loaded and activated. It receives all events related to its associated physical object or location as long as it remains active and it can communicate with arbitrary other virtual counterparts.
As a rule of thumb a virtual counterpart of a physical object should terminate itself some reasonable time after it is no longer associated with any Virtual Location. Upon termination it is unloaded by the infrastructure. It will not be reloaded and activated again until it is next sighted. The counterpart can then access the persistent data which it might has stored earlier. Removing the association to a unique identity will prevent the virtual counterpart from being reloaded again as the infrastructure does no longer know which representation it should load and activate. However, a default representation that would be taken as the virtual counterpart of a physical object or location with a unique identity that was not previously associated to a specific virtual counterpart could be used in this case.

3.4 Meta Objects

When we think of unique identifiable objects such as playing cards, various issues of a newspaper or food products it might be more reasonable to handle many similar physical objects by only a single virtual representation. All events related to objects managed by a single Virtual Object instance that we call *Meta Object* are routed to it instead of creating a separate Virtual Object for each playing card or each newspaper issue. This concept can dramatically reduce resource requirements and application development complexity.

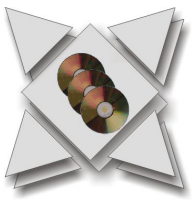


Figure 5: *A Virtual Meta Object managing a collection of compact disks.*

3.4.1 Virtual Meta Object

A collection of usually similar objects can be managed by a single virtual representation which internally differentiates the events received. We call

this special instance of a Virtual Object a Virtual Meta Object (VMO).

3.4.2 Virtual Meta Location

Analogous to the concept of Virtual Meta Objects similar locations can be managed by a single Virtual Meta Location. For instance in a museum, various sensors at different locations could be managed by a single Virtual Meta Location. Furthermore this approach is particularly useful when dealing with absolute positioning data (e.g., GPS) that would otherwise create a separate Virtual Location for each unique identifiable position. All locations of the form GPS(x,y,$height$) could be managed by one or a few Virtual Meta Locations.

Figure 6: *A Meta Location managing GPS positions.*

3.5 Virtual World

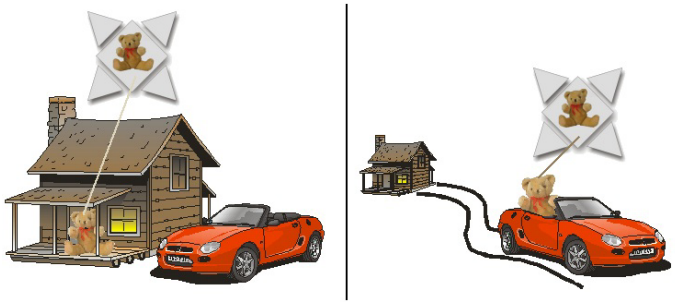


Figure 7: *When the teddy bear leaves the house for a journey, it is still accompanied by its virtual counterpart.*

The virtual counterparts live in a virtual world which is created by a software and hardware infrastructure. It is unrealistic to assume that this infrastructure will span the whole world and that some global services will be accessible from all places. The structure of the virtual world is partitioned and there will be smaller and bigger islands of virtual worlds where the virtual counterparts can live in. For instance as is shown in figure 7 a teddy bear might move out of his home for a journey. When the physical teddy leaves the house, its virtual counterpart should accompany it by hopping into the car too. As the car moves away from the house some services and other virtual counterparts will get out of reach as the communication link to the house's virtual world disconnects. For the journey, the teddy bear will live in the car's much smaller virtual world. But when the car returns at night and the teddy enters the house again then its virtual counterpart

can move back to the house's reconnected virtual world and make use of the full infrastructure again. This example shows a need for the migration of virtual counterparts.

3.6 Communication Paradigms

The distributed Virtual Objects must support a communication paradigm that is simple but flexible and extensible at the same time. We give a short overview of various concepts related to communication. In our infrastructure the objects can exchange messages that we will call *events*.

There are two basic communication patterns in use:

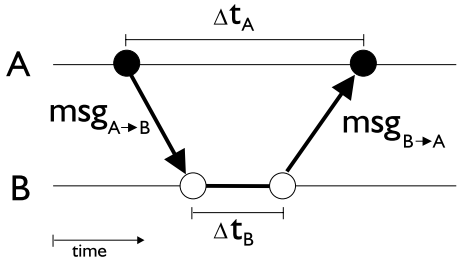


Figure 8: *Illustration of a message exchange between two entities.*

- **synchronous**: The sending entity A has to wait during the time interval $\triangle t_A$ (cf. figure 8) until the other party B has processed the message and has sent a reply. We can say that synchronous message exchanges have a blocking behavior.
- **asynchronous**: The sending party A can work during $\triangle t_A$ on new tasks after the message delivery. The receiving party B will process the message and send the reply back by executing a callback at A to inform it about the reply. We can say that asynchronous message exchanges have a non-blocking behavior.

When defining a message protocol there are usually two different types of messages:

- **notification**: The message sent does not require a reply[11] (unless an optional acknowledgement on the transport layer that the message has arrived).
- **request-reply pair**: The message sent is a request that always requires a reply[12].

[11] E.g., time broadcast messages do not require a reply.
[12] E.g., the GET message of the HTTP protocol always expects a reply.

Request-reply pairs can be imitated by two notification messages that are somehow linked together; one is sent from the sender to the receiver and the other one back. This procedure involves establishing two different network connections[13] and the parties involved would have to match the reply to the correct request. However, this is not as efficient as sending a reply on the same already established network connection (e.g., TCP/IP socket). In our infrastructure we will support both types of messages.

Messages delivery can be categorized into two different groups:

- **acknowledged**: The sender requires an acknowledgement that the message was successfully received at the remote object.
- **unacknowledged**: The sender only makes sure that the message was delivered into the network but does not know whether the remote object finally received it.

In our infrastructure we will rely only on acknowledged messages.

Case Study: Remote Method Invocation (RMI) Java^{TM}'s Remote Method Invocation, which allows to call methods of objects on a remote node in the network, is synchronous by design. When we assume the network bandwidth to be reasonably high and the delay to be short then the processing time at the receiver becomes the crucial issue. If the receiver is processing messages fast (i.e., $\triangle t_B$ is small) we can live with synchronous calls as $\triangle t_B$ is short and the sender is not blocked for a longer time. However, the processing of some messages can last longer than we are willing to wait. In this case we can use some work-arounds:

- The *sender* can start a separate thread that does the synchronous call. The thread can be either polled for the result by the main process or it can make an upcall to the main process when the answer from the remote object was received.
- The *receiver* can immediately return an answer and start a separate thread to do the actual message processing. Note that this is useful only for notification messages (i.e., the answer is of type void) but it helps to keep the sender simple.

We decided to support the server-side thread in our core infrastructure as the first work-around makes communication for the sender heavily complex. Nevertheless objects are free to implement the first solution too, which can show especially useful for messages of the request-reply type where there are no server-side threads to cut down the time during which the sender is blocked.

[13] In Jini^{TM} this procedure would additionally require a lookup and the download of the receiver's stub (at least for the first message exchange).

3.7 Persistence

In an environment where objects are created and unloaded dynamically and where objects are not guaranteed to be reestablished at a predefined location it is not a good idea to store persistent data on the current local node. Furthermore it would be especially useful if objects can be queried about current and past events.

3.7.1 Artifact Memory

Virtual Objects need a place to store event histories and stateful information. We decided to provide a service, called *Artifact Memory*, to store those data persistently. As a special feature, stored data can be queried with a high-level query language. Each Virtual Object has access only to its own data repository and can store events and other arbitrary key/value pairs in it.

3.8 Privacy or Defeating the Fear of a 'Big Brother'

Although privacy is important it was not the primary focus of this thesis. Thorough further investigations will have to be done on this issue before the infrastructure described should be applied in a public environment. For the moment it is merely a research prototype that does neither provide strong authentication nor encryption.

Achieving good usability whilst guaranteeing security and privacy in the field of spontaneous networking and ubiquitous computing will be a major research field for the next decades to come.

Our infrastructure for ubiquitous computing does allow for independent distributed virtual counterparts that have as much control over themselves as possible and neither the objects itself nor their event exchanges are managed by a single centralized master. The objects are free to exchange arbitrary events without being obliged to send them trough a predefined relay instance and they can use their own encryption for the events if the receiving object is able to interpret it. Besides, the fact that this infrastructure is primarily intended to support an enhancement of everyday things and that its main purpose is not tracking people, where the fear of a 'Big Brother' looms large, mitigates the situation to some extent. Nevertheless a few important aspects which are discussed below will have to be considered for future extensions of the infrastructure.

Access to object specific data might be restricted depending on:

- which object is asking
- what the object in question represents

- when the question is asked
- which other objects are near it
- how the conditions of the environment look like

In order to detect misuse each access to sensitive data should be logged and analyzed.

If we want to protect an object's privacy, then we should:

- protect the object's identity
- protect personal details (i.e., events sent and received)
- protect the object's habits (i.e., where it is and how long)
- prevent impersonation (i.e., an object should not be able to masquerade as another object unless it is privileged to)
- prevent misuse of an object by a third party (e.g., when a key is stolen it might refuse any service)

A severe problem lies in protecting an object's unique identity. The unique identity is usually broadcasted unencryptedly to a sensor. Furthermore the infrastructure relies on the fact that every object can find every other object and send an event to it. Therefore it seems almost impossible to protect the identity and to prevent impersonation. One approach is to demand a proof of identity together with each event sent. This could be done by using digital signatures and certificates.
If we installed a centralized communication master (which would contradict to our distributed infrastructure) we could implement an even simpler solution by making use of *one-time identities*. An object would send a new identity in each encrypted event to the central master that has a list of the current identities of all active objects. Only the master and the object itself know about the new identity. This can also prevent the detection of an object's habits which could else be deduced by analyzing data logged at a sensor. The one-time identities make it very hard to find out which different identities belong to the same object as long as the one-time identities are well chosen.

4 A Virtual Object eXtensible Infrastructure

Now that we have discussed the basic concepts of locations, unique identities, smart virtual counterparts, communication paradigms, persistence and privacy in section 3 we can start with the design of an extensible infrastructure for virtual counterparts. To make it clear which infrastructure we are discussing we will use the acronym **VOXI** (**V**irtual **O**bject e**X**tensible **I**nfrastructure) for future reference.

4.1 Design Goals

VOXI should enable to make simple things smart by giving each unique identifiable object a smart virtual counterpart.

We first specify a few fundamental design principles that we try to meet:

- **simple (to use)**: It must be simple to create new virtual counterparts and install them in the infrastructure. Complex tasks such as discovering services of the infrastructure or communicating with other remote counterparts should be encapsulated into powerful high-level commands, which abstract from unimportant details involved in the completion of the task.
- **extensible**: The core of VOXI must guarantee to be extensible for services which are added later and which should seamlessly integrate.
- **distributed**: Locations and physical objects in the real world are distributed by their very nature and so we require the virtual counterparts to be. The infrastructure has to provide communication facilities between remote counterparts and allow for the distribution of its core services. Virtual counterparts should get a means to migrate to other nodes in the virtual world. Some nodes, which form a small part of a bigger virtual world, might even get disconnected for some time which should not make any virtual counterpart die.
- **dynamic**: Virtual counterparts are dynamically created upon sightings and should not have to be manually activated (unless explicitly wished) before they can be used. The virtual world should reflect the dynamically changing relations of objects to locations in the real world. Services and objects can appear and vanish without the system having to be restarted.
- **scalable**: As the number of physical objects which are enhanced by smart virtual counterparts grows the infrastructure will have to be enlarged. It is important that core services are scalable by design.
- *nice to have:* secure, efficient, robust

4.2 System Architecture

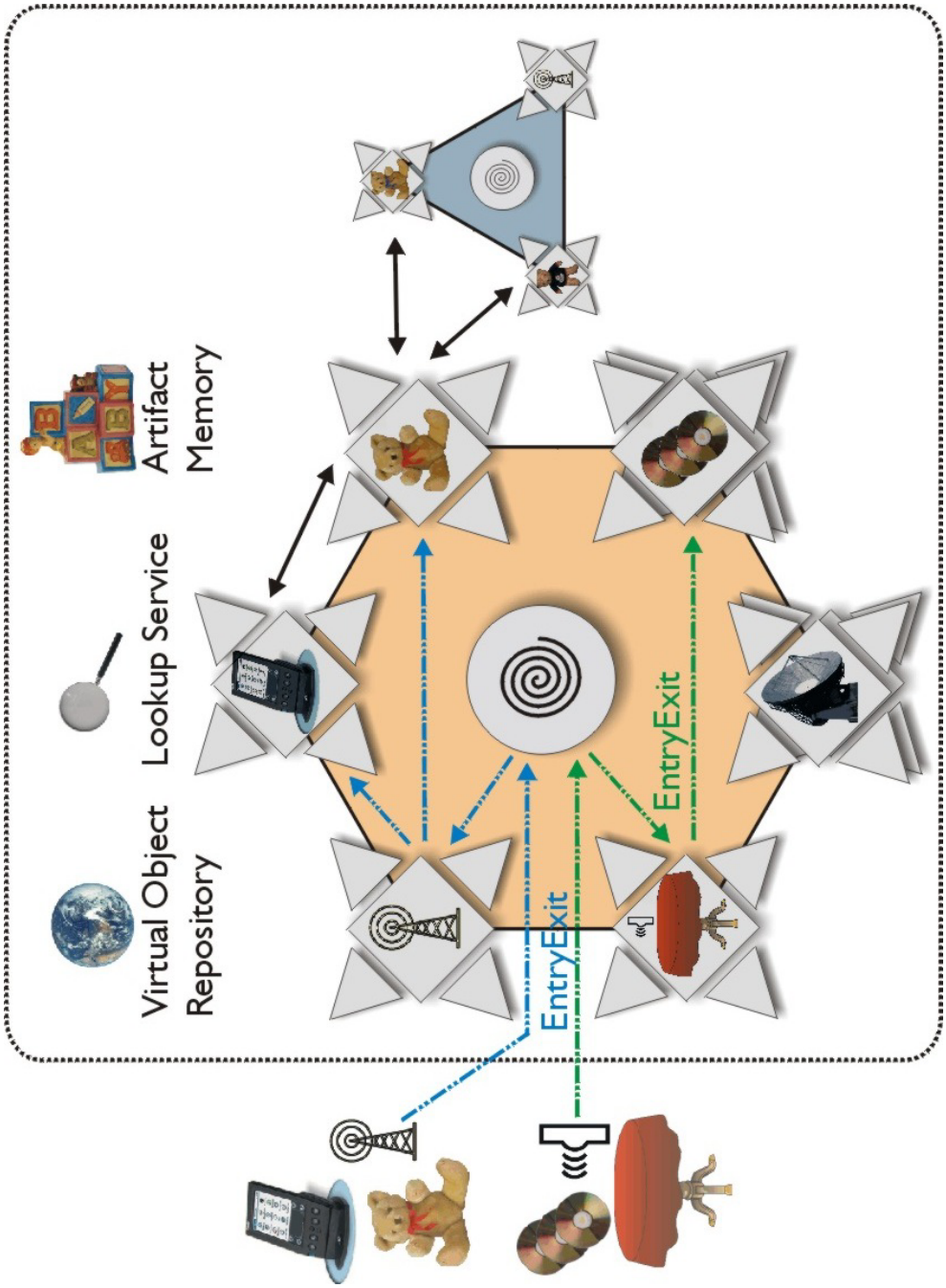


Figure 9: *System Architecture of VOXI - the Virtual Object eXtensible Infrastructure.*

The design of VOXI's system architecture is shown in figure 9. A few everyday objects such as a teddy bear, an organizer and a heap of compact discs are brought into the range of one of the two sensors at different locations. Upon object sightings VOXI dynamically creates the virtual counterparts for the objects and locations. For each sighting the sensor creates an Entry Event and sends it to a predefined Virtual Object Manager which is responsible to download the Virtual Location named *locationID* and the Virtual Object named *objectID*[14] from the Virtual Object Repository and create them if they are not already there. Then the Virtual Object Manager registers the new Virtual Location and the new Virtual Object at the Lookup Service and finally forwards the Entry Event to the Virtual Location which in turn forwards it to the Virtual Object.
Now the Virtual Object can subscribe itself at the Virtual Location for events and it can start communicating directly without going through its Virtual Object Manager. It can communicate with every other active component in VOXI including the Artifact Memory, where it can store events and other data persistently. A Virtual Object can not only communicate and collect information, it could also display a GUI[15] on the screen of the machine that its Virtual Object Manager resides on or send events to a special service that can control an actuator in the real world.
When a physical object leaves a location by exiting the range of a sensor then an Exit Event is created and sent to the predefined Virtual Object Manager which forwards it to the corresponding Virtual Location which in turn forwards it to the Virtual Object. If the Virtual Object decides to terminate then the Virtual Object Manager which has created it will be automatically informed and unregisters it from the Lookup Service.
VOXI has three core services: A Virtual Object Repository, a Lookup Service and at least one Virtual Object Manager. As soon as those services are running, an Event Source can send Entry Events and herewith make the virtual counterparts appear. The system dynamically grows and shrinks - without being restarted - as new services and Virtual Object Managers join. The Artifact Memory, which provides persistent storage for data of virtual counterparts, is not a core service but rather a nice add-on.

4.3 Events

VOXI's communication relies on events as the message entities that can be exchanged between two arbitrary virtual entities. The event format must be reasonably efficient, simple to use, serializable and versatile. Specialized Virtual Objects or Locations should be free to introduce their own proprietary event types. The Artifact Memory should be able to store the events and to provide a query facility on the data fields of the events stored.

[14]The two identities *locationID* and *objectID* are contained in the Entry Event.
[15]Graphical User Interface

4.3.1 Definition of Event

*An **event** has a unique type which determines a set of compulsory key/value pairs as its properties.*

Please note that the definition does not require an inheritance hierarchy of the different events. The type of an event is always a unicode string restricted to ASCII[16] letters and digits to assure that we can display the type properly on any computer. For the sake of simplicity we conventionally restrict the data type of a key to a unicode string[17] and the type of a value to a unicode string, a long[18] number or a double[19] number. Events that will not be stored at the Artifact Memory[20] can have an arbitrary data type for the values including complex nested types such as a list or an array of events. Please note that new proprietary data types must support the equality operator[21].

As events are the communication backbone of VOXI and therefore very important we now define a couple of common operators on events.

4.3.2 Definition of Equality of Events

*Two **events are equal** if they are of the same event type and if they have exactly the same key/value pairs as their properties in respect to data type and contents.*

If you need a more flexible way to compare events then you can use template matching, which is heavily used in the subscription mechanism for events at a Virtual Location.

4.3.3 Definition of Event Matching

*An **event template M** of type T_{tmp} having the properties key_{t_1}/val_{t_1}, key_{t_2}/val_{t_2}, ..., key_{t_n}/val_{t_n} **matches the event E** of type T having the properties key_1/val_1, key_2/val_2, ..., key_m/val_m if at least one of the following rules holds true:*

1. $M =$ `null`
2. $((T_{tmp} =$`""`$) \vee (T_{tmp} = T)) \wedge$*the template has no properties*

[16] American Standard Code for Information Interchange
[17] Type 'string' is a sequence of 16 bit unicode 1.1.5 characters.
[18] Type 'long' is a 64 bit signed integer number.
[19] Type 'double' is a 64 Bit IEEE 754-1985 floating point number.
[20] We bear in mind that the Artifact Memory will rely on an SQL database in order to support a powerful query language for free. A future version of the Artifact Memory could rely on an object-oriented database and accept any data types.
[21] In Java^{TM} a new data type must support the `equals` method.

3. $((T_{tmp} =$"") $\vee (T_{tmp} = T)) \wedge$ *the set of template properties* $\{key_{t_1}/val_{t_1}$, key_{t_2}/val_{t_2}, ..., $key_{t_n}/val_{t_n}\}$ *is a subset of the set of event properties* $\{key_1/val_1$, key_2/val_2, ..., $key_m/val_m\}$ *whereas a* val_x *of value* `null` *in the template matches any other corresponding value in the event*

If the event we try to match has a key in his properties where a list of events is associated then any `null` values in this "nested" events are not taken as a wildcard but rather matched for equality with the corresponding values in the nested events of the template.

4.3.4 Definition of Strict Event Matching

An ***event template M strictly matches the event E*** *if the template M matches (cf. 4.3.3) the event E and if the set of E's property keys* E_{key} *minus the set of M's property keys* M_{key} *is the empty set* ($E_{key} \backslash M_{key} = \emptyset$).

4.3.5 Predefined Event Types: EntryEvent and ExitEvent

Two core event types are predefined in VOXI:

EntryEvent A sighting of an object *objectID* at location *locationID* at time *timestamp* causes the sensor (or it's gateway driver in VOXI to be exact) to create an event of type "EntryEvent" with the properties objectID=*objectID*, locationID=*locationID* and timestamp=*timestamp*.

ExitEvent An object named *objectID* leaving the range of a sensor at location *locationID* at time *timestamp* causes the sensor the create an event of type "ExitEvent" with the properties objectID=*objectID*, locationID=*locationID* and timestamp=*timestamp*.

There is one predefined logical location called "**Virtopia**" that allows to create an object which does not have to be at a physical location at the moment. The reason we define it here is to give the Virtual Objects a chance to react appropriately to this fact (e.g., by not storing such events in the Artifact Memory). This location is also used when migrating Virtual Objects.
Please note that some services such as the Artifact Memory (cf. 4.10) or the publish/subscribe mechanism of Virtual Locations (cf. 4.6.1) introduce further predefined event types.

4.3.6 Meta Information About Events

It will show very useful to provide a means to supply separate meta information about an event. The Artifact Memory, among others, will make use of that feature. Furthermore this enables us to easily introduce encrypted or authenticated events as an extension. The meta information could hold information about the cryptographic algorithms used or proofs of identity.

Meta information is simply a set of arbitrary key/value pairs whereby the data type of the key is a unicode string.

4.4 Communication Infrastructure

4.4.1 Event Handling

We decided that each entity in VOXI has to support two methods for event processing. A method **notify** for notifications and a method **request** for request-reply pairs as defined in 3.6. The second one greatly improves performance as was discussed on page 24. We call the sender of an event the Event Source.

4.4.2 Remote Calls

As virtual entities in VOXI can either be local to an event source (e.g., a Virtual Object Manager forwarding an event to a newly created Virtual Location) or on a remote node, we need a paradigm that supports both.
Event sources such as sensor gateways in VOXI should be as simple and efficient as possible. Therefore we require the Virtual Object Manager to return immediately from the remote call to its *notify* method upon Entry and Exit events. A suitable solution by using a receiver thread was already discussed in 3.6.

4.5 Virtual Objects

4.5.1 Features

We support the conceptual ideas of a Virtual Object as discussed in 3.3.1 by requiring the following list of features from a Virtual Object:

event handling Application specific event handlers `notify` and `request` that accept at least the event types EntryEvent and ExitEvent.

self-description A name, an up-to-date RDF/XML[22] description of the object and an iconic image must be supported. The description can contain semantic and status information and is explicitly allowed to change over time. This makes it much more powerful than simple reflection of a Virtual Object class.

location-awareness The Virtual Object should preferably store the locationIDs of all locations it was recently sighted at. This information is contained in the Entry- and ExitEvents.

[22] Resource Description Framework (RDF) that relies on the eXtensible Markup Language (XML), cf. [42]

lifetime management An object can be asked about the last time it was in use (the so-called LRU[23] timestamp) and it has to provide a method `pleaseExit` that asks it to exit as soon as possible.

autonomy Virtual Objects, once created by a Virtual Object Manager (VOM), can act and communicate independently of the VOM and other objects in VOXI. The outgoing events are sent directly to the receiving object and are not controlled by a local or remote Virtual Object Manager unless they are Entry- or ExitEvents. The Virtual Objects can also be active without any need to continuously receive events.

security The `pleaseExit` method requires a *security token* that the Virtual Object Manager tells the Virtual Object when it is created. A call of this method with a wrong token can be ignored. The token can also be used to require special services in a future extension of VOXI.

resource thrift Virtual Objects are run in a thread of the Virtual Object Manager and should use as few resources (e.g., memory, communication bandwidth, processing power) as possible because many Virtual Objects are active concurrently.

4.5.2 Case Study: Process or Thread?

A Virtual Object Manager (VOM) could create new Virtual Objects (VO) and Locations (VL) either in a new thread or in a separate process (in a new Virtual Machine). The comparison in table 1 lists some advantages and disadvantages of the two solutions.

criteria	process	thread
memory	$\approx$ 10 Megabyte/VM[a]	considerably less than 1 MByte
creation	slow	fast
security	high (policy file for VM)	medium[b]
parameter passing	command line parsing; String	constructor args; Java^{TM} types
communication	inefficient (remote call)	efficient (local method call)
independence	high	low
garbage collection	upon exit of VO/VL's VM	not under control of VOM[c]

[a] VM stands for Java^{TM}'s Virtual Machine
[b] ClassLoader policy possible but any thread can exit the VM of the VOM.
[c] Effect of Java^{TM}'s garbage collector call `System.gc()` is implementation specific.

Table 1: Comparison of process versus thread to run a Virtual Object or Location in.

[23] Least Recently Used (LRU)

The list is not final of course. We decided to use threads to wrap our virtual counterparts in VOXI although we have to accept a somewhat lower security. Using processes instead of threads would not scale at all. Nevertheless it is still possible to explicitly install a Virtual Object Manager with only one Virtual Object and Location running in it which is comparable to a fully user controlled Virtual Object in a separate process.

4.5.3 Activatable and Permanent Virtual Objects

We can imagine two different types of Virtual Objects. The first one is created once and then remains permanently active. This might be a good solution for Virtual Objects that have to initialize a big heap of data which causes a long initialization phase or Virtual Objects that provide a service in VOXI. The second one is created upon a sighting and can be regarded as an activatable object. The Virtual Object Manager has the role of a *mediator*[24] that intercepts (Entry) events while the object is off-line, creates and runs the object on demand, and then forwards the event when the object is ready to receive it.
Both solutions are supported by VOXI. A permanent Virtual Object can easily be created by issuing a "virtual" sighting at location "Virtopia" (cf. 4.3.5).

4.5.4 Location-Awareness

Virtual objects are encouraged to implement the following high-level query capability which is similar to the one used in the Active Badge System (cf. [1]):

- **where?** - where are you? (returns an array[25] of *locationID*s)
- **with?** - who is with you? (returns an array of *objectID*s)
- **history?** *or* **history(***timestamp***)**? - get a list of recently visited locations since the optional time *timestamp* or else up to three[26] most recent locations (returns an array of *locationID*s)

These queries are embedded in the following protocol. Please note that it is an extension to the protocol which virtual components use when interacting with the Artifact Memory (cf. 4.10.3).

[24] E.g., in Jini^{TM} the RMI daemon (rmid) is the *mediator* where activatable objects such as the Lookup Service can register themselves.
[25] An object can reside at multiple locations concurrently (cf. 3.1.1).
[26] The restriction to a default value of three is arbitrary.

[request-reply]

request:				
meta:	"`sender`" =	*objectID*		
event:	-type:	"`QueryRequest`"		
	-properties:	"`queryLanguage`"	=	"`VOXIQL`"
		"`query`"	=	"`where?`" or "`with?`" or "`history?`" or "`history(`*timestamp*`)?`"
reply:				
meta:	*empty*			
event:	-type:	"`QueryReply`"		
	-properties:	"`errorCode`"	=	"`success`" if successful. On error anything else such as "`error`","`not found`", ...
		"`errorDetails`"	=	additional textual information about the error (if any)
		"`queryLanguage`"	=	"`VOXIQL`"
		"`result`"	=	an array of type String that holds a copy of all locationIDs or objectIDs that were selected by the query

4.6 Virtual Locations

Virtual Locations as defined in 3.3.2 model proximity of an object to a location (cf. 3.1.1). A Virtual Location is very similar to a Virtual Object. Additionally it supports template-based subscriptions for and publishing of events.

4.6.1 Template-Based Event Subscriptions

A Virtual Object can subscribe for events at a Virtual Location by simply sending an event template and setting the meta key/value pairs "`metaType`" = "`SubscriptionEvent`" and "`subscriber`" = *objectID*. The Virtual Object *objectID* will then be notified[27] by the Virtual Location about all future events that match[28] the registered template(s).
In order to unsubscribe the Virtual Object sends and event template and sets the two meta information pairs "`metaType`" = "`SubscriptionRemovalEvent`" as well as "`subscriber`" = *objectID*. There are two different unsubscription possibilities:

[27] The subscription mechanism of the Virtual Location remembers only the *objectID* of the subscriber and not a (VM dependent) reference to the Virtual Object. This indirection allows that events are automagically forwarded to a Virtual Object after it has migrated.

[28] Please note that we do not use 'strict' matching here.

1. A template with a type of `""` and no properties (i.e., an event template that matches all other events) removes all registered subscriptions of the component *objectID*.

2. A template that contains an earlier specified template removes only the specified template. If the specified template was never registered then the removal request is ignored.

4.6.2 Event Publishing and Distribution

When publishing events of arbitrary types a Virtual Object sends the event that it wants to publish to interested subscribers to a Virtual Location and sets the meta information "`metaType`" = "`PublishEvent`". The Virtual Location then sends the event to all matching local subscribers (but only once if it matches several templates of the same subscriber). A Virtual "Master" Location could subscribe to all Virtual Locations that appear at the Lookup Service and hereby establish a central event manager which could trigger composite events (cf. [26]).

4.6.3 Location-Awareness

A Virtual Location is encouraged to support the VOXIQL[29] query "`with?`" (cf. 4.5.4) and return a list of objectIDs of objects that are currently at that location. The other two queries mentioned in section 4.5.4 cannot be applied to Virtual Locations.

4.7 Meta Objects

The paper [27] presents a case study of a program called 'RFID Chef' that suggests meals upon sightings of ingredients that it recognizes by their RFID tags. The 'RFID Chef' processes Entry- and ExitEvents for a variety of grocery items. Another comparable example would be a program that monitors a card game and that can give useful hints on how the players can improve their skills. Every card would be tagged with an RFID tag. A third example is a virtual music box that downloads audio[30] files corresponding to small physical music tokens that act as proof of ownership for the downloaded songs.
In these three examples our design of an autonomous Virtual Object that represents exactly

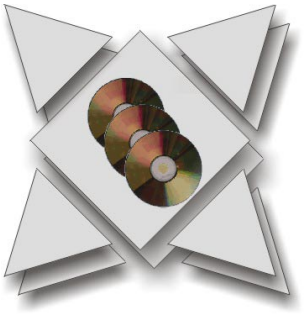


Figure 10: *A collection of compact discs managed by a single Virtual Meta Object.*

[29] VOXI's Query Language
[30] A very common audio file format would be MP3.

one physical counterpart seems not to prove efficient. We would rather need a tightly coupled federation of objects.

This idea led to the design of Virtual Meta Objects that can manage a set of several "primitive" physical objects instead of merely a single one. The only feature we need to add is a mapping facility that maps all *objectID*s of such a set of "primitive" objects to a single Virtual Meta Object. We decided that the Virtual Object Repository (cf. 4.9) has to provide such a mapping facility. The Virtual Object Manager has to register all "primitive" objects at the Lookup Service to make them transparently accessible for other Virtual Objects and to make them look like autonomous Virtual Objects. Please note that a Virtual Meta Object is only a single component in the system and does not necessarily host a collection of inner Virtual Objects which are autonomous and could leave the Virtual Meta Object on demand.
For short, VOXI provides Virtual Meta Objects that allow for central management of a couple of physical objects which results in less complex applications as we have less inter-object communication, simpler application code as it is less distributed, easier coding and debugging as the relevant code is more monolithic but at the same time we hazard the consequences of a bottleneck application, a single point of failure and central monitoring which could raise concerns about loss of privacy.

Analogously VOXI also provides Virtual Meta Locations that manage a set of physical or logical locations.

4.8 Virtual Object Manager

4.8.1 Core Tasks

Entry- and ExitEvents need a destination in VOXI where an event source can send them to upon sightings. The Virtual Object Manager represents a preestablished instance which takes care of the creation and destruction of Virtual (Meta) Objects and Virtual (Meta) Locations. After creation it registers the new components at the Lookup Service.

The following list shows the central tasks of a Virtual Object Manager.

- Accept Entry-/ExitEvents
- Dynamically create virtual counterparts upon EntryEvents:
  - Dynamically download code and resources for Virtual (Meta) Objects and Virtual (Meta) Locations from at least one predefined Virtual Object Repository

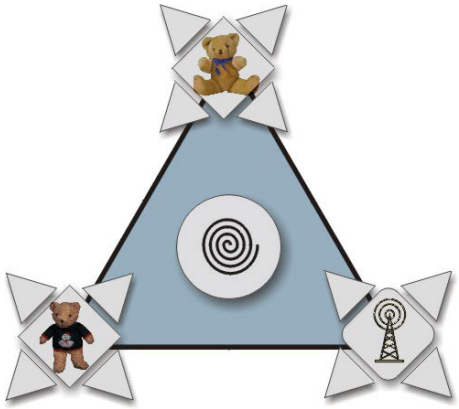


Figure 11: *A Virtual Object Manager with two Virtual Objects and one Virtual Location.*

  - Allow for Virtual Meta Objects/Locations by supporting the mapping facility of the Virtual Object Repository
  - Execute the downloaded code
  - Register new Virtual Objects/Locations at the Lookup Service
  - Renew the leases for registered Virtual Objects/Locations at the Lookup Service (background thread)

- Destroy virtual counterparts:
  - Forward ExitEvents to the corresponding Virtual Location that forwards it to the correct Virtual Object
  - Unregister and destroy Virtual Objects upon their termination
  - Force termination of Virtual Objects/Locations when the Virtual Object Manager shuts down

- further:
  - Manage local resources and reclaim them from the least recently used Virtual Objects when scarce
  - Migrate Virtual Objects upon migration requests

4.8.2 Creation of Virtual Objects and Locations

A simple EntryEvent sent by an event source to a Virtual Object Manager causes a whole 'chain reaction' of creation steps. If a sensor sights a tagged teddy bear named *objectID* at its location *locationID* then it sends an `EntryEvent(`*objectID*`,` *locationID*`,` *timestamp*`)` to its predefined Virtual Object Manager (VOM). The VOM then checks whether it has to create

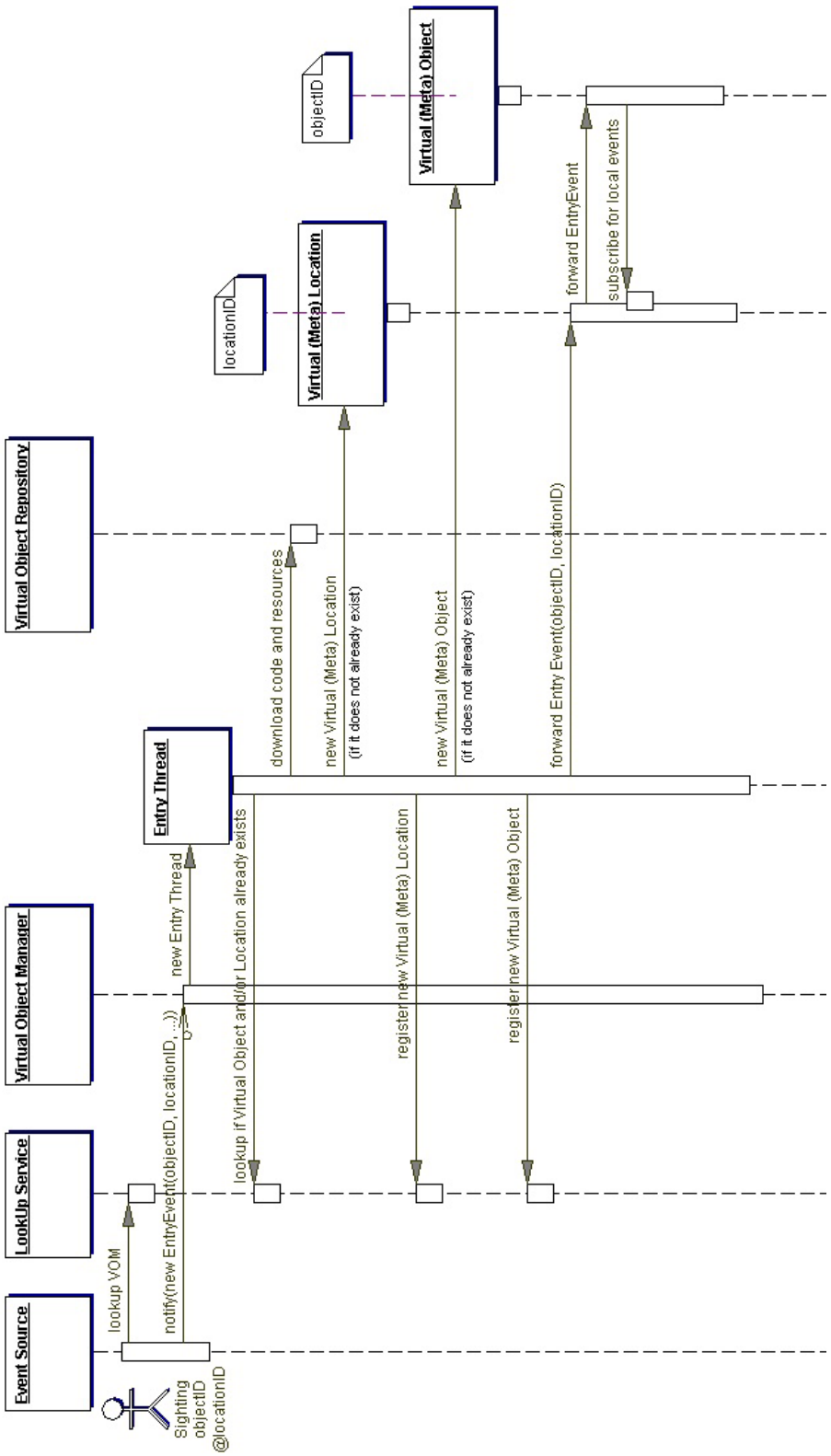


Figure 12: *Sequence diagram that illustrates a high-level view of the processing of an EntryEvent by components in VOXI.*

any virtual counterparts or if some of them are already there and he only has to register new counterparts or simply forward the event to a remote VOM[31]. The sequence diagram in figure 12 shows the high-level view of the components and steps involved in the creation of virtual counterparts. Please note that this illustration does not show the necessary checks that are performed to determine whether there actually is a need for creating new object instances or if mere registering would be sufficient. Those checks are shown in the flowchart in figure 13.

The Virtual Object Manager implements the concept of a server-side thread (cf. 3.6) in order to return immediately from a remote call to its event notification method *notify*. For each arriving EntryEvent the Virtual Object Manager spawns a new EntryThread that takes care of the processing and terminates itself upon completion of this task. The flowchart in figure 13 illustrates the processing details. The Virtual Object Manager has to be thread-safe by design, which will prove to be challenging for the implementation, as many EntryEvents can arrive in parallel and have to be processed as fast as possible without being put in a designated waiting queue.

4.8.3 Case study: Should a Virtual Location Incorporate the Features of a Virtual Object Manager?

An alternative to distinct Virtual Object Managers would be that the Virtual (Meta) Locations would take over the role of a Virtual Object Manager and be responsible for the creation of new Virtual (Meta) Objects. This would imply that a Virtual Location has to be manually pre-established before an event source at the corresponding physical or logical location can send Entry- and ExitEvents to it. Furthermore an event source would have to deal with the mapping mechanism if it is moving between various locations (e.g., GPS positions) that are managed by a single Virtual Meta Location. Another conceptual problem arises from the fact that a Virtual Object can reside at more than one Virtual Location concurrently (e.g., on a table *and* at a conference *and* in a room). This somehow contradicts to the close relation with only a single Virtual Location where it is managed. Load balancing for Virtual Object creation and destruction in case of many objects entering and leaving a certain Virtual Location can hardly be supported as a location is a single instance only. The Virtual Location does not only have to deal with bookkeeping of which objects are present and of event subscriptions but also with the creation and destruction of Virtual Objects. A last thought that influenced our decision to design the Virtual Object Ma-

[31] We can find out about on which other Virtual Object Manager an already existing Virtual Object resides by retrieving the attribute "ObjectManager" from the Virtual Object's registration entry at the Lookup Service.

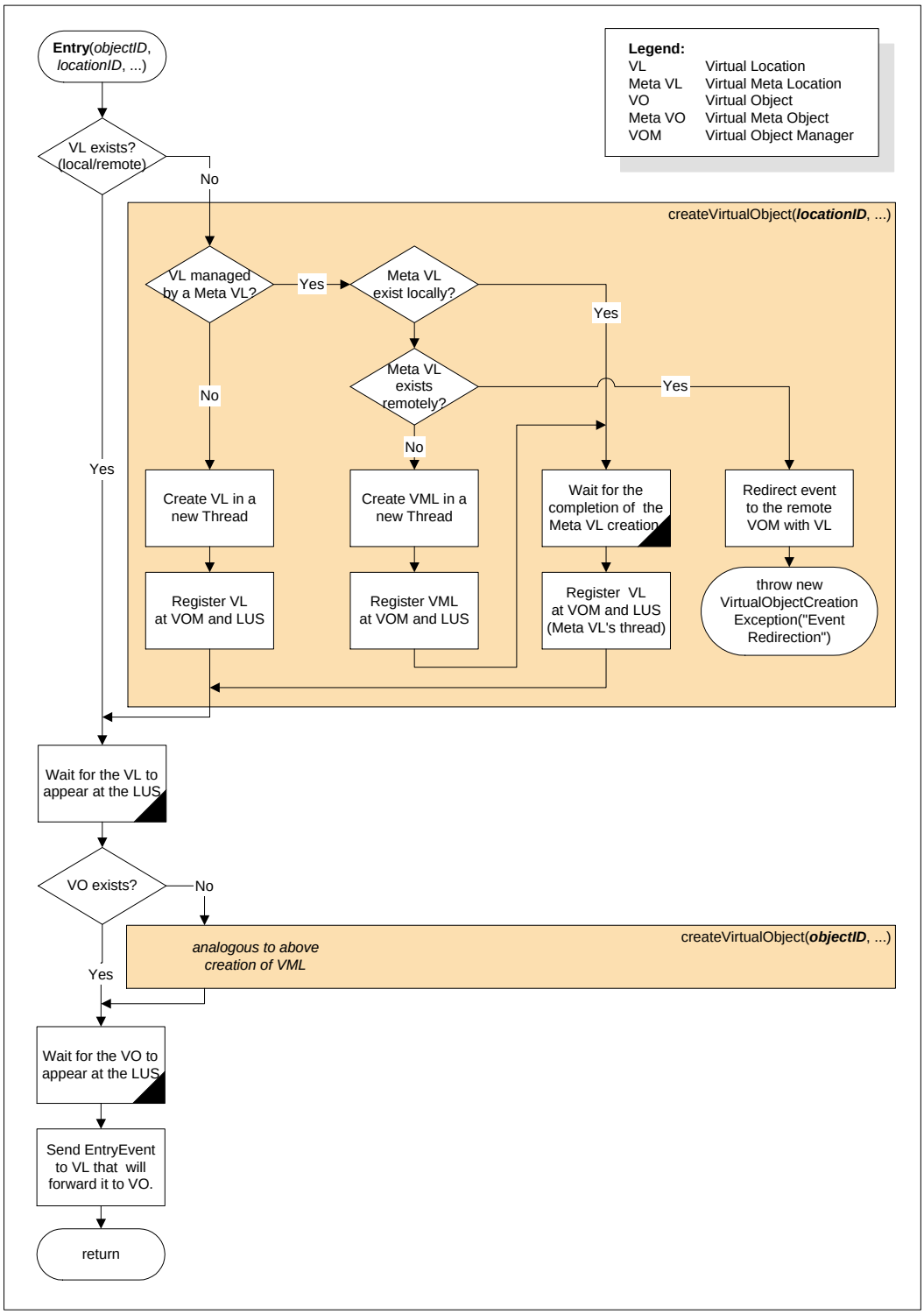


Figure 13: *Flowchart that illustrates the detailed processing of an EntryEvent by the EntryThread.*

nager as a distinct component was to separate conceptually different tasks in our design. We think that there will be far more Virtual Locations than Virtual Object Managers. In this case our design decision also leverages resource requirements as distinct components are run in separate Virtual Machines.

4.8.4 Unloading Virtual Objects and Locations

Figure 14 shows a high-level view of the actions at various components that are performed when a sensor recognizes that an object has left its location. The illustration hides the possible variations. A Virtual Object that is associated with a Virtual Location different from the *locationID* in the ExitEvent certainly will not terminate. It is important to recognize that the Virtual Object finally decides by itself whether it wants to exit by leaving its main[32] method.
The Virtual Object Manager takes care of ExitEvents in a separate ExitThread which is spawned to enable an immediate return (also known as 'non-blocking' call) of the remote call to its notification method *notify* and in order not to let the event source wait. The flowchart in figure 15 shows the details of the actions performed by the ExitThread when processing an ExitEvent.

4.8.5 Migrating Virtual Objects

The scenario with the teddy bear leaving the house and going on a journey on page 22 has indicated a need for migration in the virtual world which was shown to be partitioned. Virtual counterparts should be able to follow the associated physical object by moving between parts of the virtual world which temporarily can disconnect from each other. Therefore a Virtual Object Manager should provide a method for Virtual Objects to help them migrate to a remote Virtual Object Manager. VOXI's Virtual Object Manager does support a bare means for relocating Virtual Objects by providing a method `migrate`(*objectID*, *remoteVirtualObjectManagerID*). However, this helper function does neither fulfill the requirements of weak migration (i.e., data and code but no execution state) as it is supported in Hive by the implementation of a multi-hop facility (cf. [12]) nor the ones of strong migration (i.e., data, code and execution state). The method `migrate` is much simpler. All it does is remember to issue an `EntryEvent(`*objectID*`,` *locationID*=‘`Virtopia`’,*current timestamp*`)` to the remote Virtual Object Manager after the Virtual Object *objectID* has terminated itself and after it might have written all its stateful data to a persistent storage (e.g., to the Artifact Memory (cf. 4.10)). The EntryEvent will recreate it at the remote

[32]The main method of a Virutal Object will be defined to be `doMain()`.

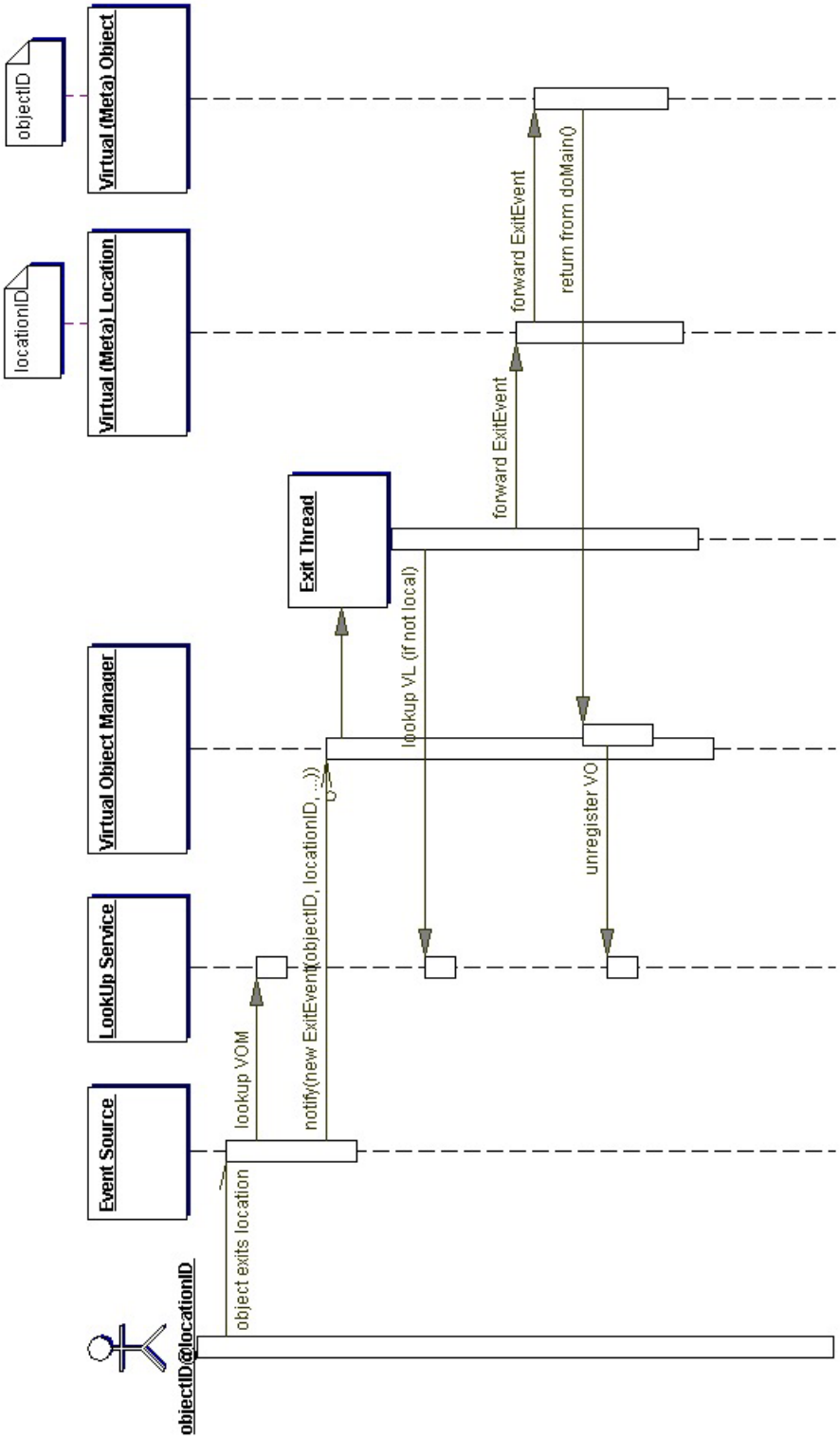


Figure 14: *Sequence diagram that illustrates the processing of an ExitEvent by components in VOXI.*

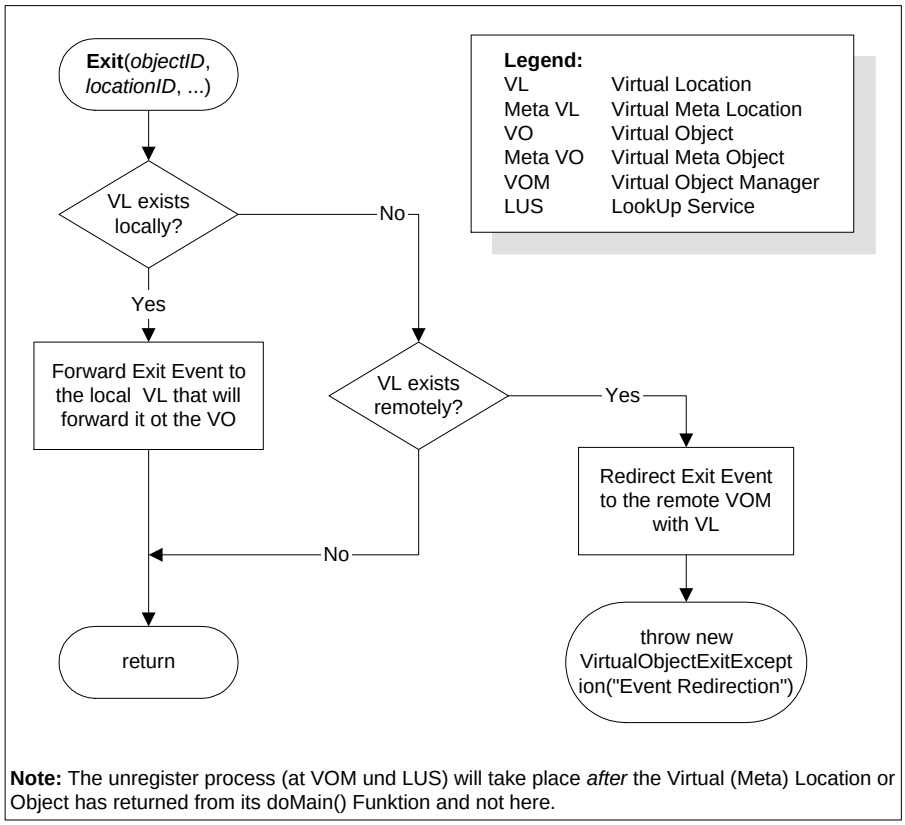


Figure 15: *Flowchart that illustrates the processing of an Exit Event by the ExitThread.*

Virtual Object Manager where it can load the data from the persistent storage and reinitialize its state if needed before the "smaller virtual world" it just migrated to will disconnect. As the Virtual Object Manager caches[33] the resource and code of the newly migrated Virtual Object it is sufficient to only provide a single Virtual Object Manager and a Lookup Service in the "smaller virtual world". During the journey the teddy bear might not have access to an Artifact Memory but when it returns home the virtual counterpart of the teddy gets notified about its proximity to the house and it can use the Lookup Service to find the Artifact Memory, which is now accessible again.
If future scenarios show any need for a more powerful support for migration then the design of the Virtual Object Manager could be extended with an additional `migrateWeak` and/or `migrateStrong` method.

4.9 Virtual Object Repository

The virtual representations of real world objects are stored as executable code and resources (e.g., iconic image, initial configuration, description files). The Virtual Object Manager dynamically loads the code and resources of a Virtual Object or Location from the Virtual Object Repository upon a sighting of a specific object at a location. Storing code and resources at each Virtual Object Manager would involve a cumbersome procedure to update, add or delete that data, especially if not all Virtual Object Managers are reachable at a time. This consideration led us to the design of a federation of Virtual Object Repositories.

Figure 16: *The Virtual Object Repository.*

The Virtual Object Repository (VOR) fulfills two main tasks:

- **repository**: Store executable code and resources of Virtual Objects and Locations that can be dynamically downloaded and instantiated by a Virtual Object Manager upon sightings.
- **mapping**: Map unique identifiable object identities (*objectID*s) to the *objectID* of an item stored in the repository.

The mapping capability enables the use of Virtual Meta Objects and Virtual Meta Locations as several objects can be mapped to a single object or location that manages the others. It is important that the mapping mechanism does only provide *objectID*s and does not directly deliver items stored in the repository. This is needed as a Virtual Object Manager will create a Virtual

[33] This applies only if the virtual counterpart together with its resource files is provided in archived format (as a JAR file) at the Virtual Object Repository.

Meta Object only upon the sighting of a first object that is managed by a specific Virtual Meta Object and therefore needs the code and resources only once. To reduce network bandwidth the mapping mechanism should preferably be cached at a Virtual Object Manager.

The implementation should allow to distribute this facility to different distant nodes in the network in order to guarantee for a good availability and performance. Meanwhile it should remain comfortable to frequently update code and resources by having a single master repository for the update management. This can be achieved by using well understood replication techniques such as mirroring the contents of a filesystem. As the Virtual Object Manager caches the downloaded resources and code for a Virtual Object it has created, it is possible to disconnect from the Virtual Object Repository as long as there are no new virtual counterparts created. However, the ideal solution would be if the physical object itself could directly provide the code and resources of its virtual counterpart implementation. This might become true in the future when for example cheap writable RFID tags can store bigger amounts of data than just an identification number.
Two last requirements that we have is that the Virtual Object Repository should rely on common network transport mechanisms and that it should be open for further extensions related the the objects stored (e.g., inventory listing or a search capability for items by specifying properties instead of an objectID).

4.10 Artifact Memory

4.10.1 Overview

The need for a service that allows for persistent storage of data arises from the fact that the lifetime of Virtual Objects is limited. They are dynamically created and and destroyed. If a later instance of a Virtual Object wants to find out about past events that happened during the lifetime of its predecessor instance (with the same *objectID*) it can access the Artifact Memory where it can retrieve the data required. Besides from storing events the Artifact Memory provides a means to store key/value pairs that can be used much like environment variables also known from operating systems (e.g., UNIX) with the difference that the data is stored persistently. We call this kind of data "**persistent state data**".

Figure 17: *A persistent Artifact Memory.*

Some fields of application for Ubiquitous Computing (e.g., remind our scenario in 1.3) require a powerful query facility for past and current events. Queries on events that were collected by different virtual counterparts might be of interest too. Such a query service is provided by our Artifact Memory.

4.10.2 Core Tasks

We summarize and specify the core tasks of the Artifact Memory in the following list:

- store persistent state data as key/value pairs persistently
- retrieve the value of persistent state data for a given key
- store events of an arbitrary event type persistently
- accept queries on event histories and return a set of events as the answer

All information stored is bound to an owner specified by a unique identity (i.e., an *objectID*, a *locationID* or an identity of another component such as a Virtual Object Manager).

4.10.3 Protocol Specification

Any object in VOXI with a unique ID can make use of the Artifact Memory by sending events that comply to the following protocol specification. Storing events persistently is a very common task and so it was designed to be as simple as possible.

a) storing persistent state data [notification]
meta: "`sender`" = *objectID*
event: -type: "`ContextStorageEvent`"
-properties: key/value pairs to store

Note that values of existing keys are replaced. We restrict keys and values for context data to the Java^{TM} type String.

b) storing events [notification]
meta: "`metaType`" = "`StorageEvent`", "`sender`" = *objectID*
event: the event to store

This is very convenient as a received event can be persistently stored with only adding a meta information and not having to change the event itself. We restrict the property keys to the Java^{TM} type String and the values to the types String, long and double (cf. 4.3.1).

c) retrieving persistent state data [request-reply]

request:

meta:	"`sender`" =	*objectID*
event:	-type:	"`ContextDataRequest`"
	-properties:	keys of which we want to retrieve the associated values; values supplied are gracefully ignored

reply:

meta:	*empty*	
event:	-type:	"`ContextDataReply`"
	-properties:	key/value pairs for all requested keys; keys that were not found in the storage have an associated value of null

d) queries on event histories [request-reply]

request:

meta:	"`sender`" =	*objectID*		
event:	-type:	"`QueryRequest`"		
	-properties:	"`queryLanguage`"	=	"`SQL92`"
		"`query`"	=	query in the appropriate query language (highly implementation specific)

reply:

meta:	*empty*			
event:	-type:	"`QueryReply`"		
	-properties:	"`errorCode`"	=	"`success`" if successful. On error anything else such as "`error`","`not found`", ...
		"`errorDetails`"	=	additional textual information about the error (if any)
		"`queryLanguage`"	=	"`SQL92`"
		"`result`"	=	an array of type Event that holds a copy of all events that were selected by the query

Note: Large result sets can be avoided by making use of an the appropriate SQL extension[34].

The events in a) and b) are sent to the Artifact Memory's *notify* method, the requests in c) and d) are sent to the *request* method as we expect a reply.

[34] In PostgreSQL and MySQL you can use the keyword `LIMIT` *firstRowNr*, *lastRowNr*.

4.10.4 Privacy and Availability Issues

The Artifact Memory raises serious privacy concerns if it is realized as a single centralized database. The administrator of the database has full access to all persistent information such as events and state, which the virtual counterparts have stored. There are at least two possible solutions to deal with this problem. Either we require the virtual counterparts to encrypt the contents of the events and state information which they store at the Artifact Memory or we store the persistent data at distributed locations. A good place would be on a writable tag of the associated physical object.
The other problem which arises from a centralized solution and the fact that our virtual world is a distributed one is that we need a globally available connection to the single Artifact Memory or else the virtual counterpart might not be able to use persistency.
As VOXI is a research prototype we decided to prefer usability and a powerful query facility to a secure Artifact Memory. However, it would be a good idea to extend VOXI by a more secure persistence and query facility which can guarantee better privacy.

4.10.5 Suggested Extensions

When we have a complex scenario with many objects involved such as a business meeting we might want to record what has happened during a certain time frame which we call *session*. We might want to be able to replay all events of an important situation. The Artifact Memory could support this scenario by accepting *sessionIDs* that can bind events of different objects to a unique session. This feature could be used to replay certain object interactions. A session table could consist of a series of references to events stored in the event tables.

4.11 Lookup Service

Components in the Virtual Object eXtensible Infrastructure (VOXI) need a way to find other components such as Virtual (Meta) Objects, Virtual (Meta) Locations, Virtual Object Managers or the Artifact Memory. A lookup service enables virtual components to be registered and lookup up. A federation of such servers guarantees for better availability and reliability. The use of leases for the registrations that have to be renewed in certain time intervals and that automatically delete the registration if a renewal is not accomplished makes this service especially robust.

Figure 18: *The Lookup Service.*

Virtual components can not only provide their objectID but also a set of attributes. This can be used to hold up-to-date state information and to

show details about which object is managed by which Virtual Meta Object and by which Virtual Object Manager. Such information could be used to build a GUI that shows the current state of components in VOXI.
The lookup service should support a (multicast) discovery protocol that obviates the need to predefine a network address where components can find a Lookup Service as it can be discovered if necessary.
A subscription mechanism for notification of changes in the registry based on matching service templates greatly simplifies the interaction with the Lookup Service[35].

4.12 Event Source

Event sources send events to components in VOXI. The origin of an event could be a sighting of a physical object by a sensor or an organizer which connects by infrared to a gateway and sends it's unique identity. Of course, events can also be generated without using any hardware at all. VOXI supplies a simple application called **EventSource** which allows to send arbitrary events to any virtual component in VOXI. It allows to set the key/value pairs in the meta information, the event type and its properties. Then it creates the event and sends it to the virtual component specified by its unique identity and prints the reply if any.

4.13 VoxiWatch GUI

A GUI as mentioned in 4.11 would greatly help when developing applications with VOXI. As it enables to watch VOXI at work we will call it **VoxiWatch**. It should graphically show active Virtual Object Managers together with the managed Virtual Meta Objects, Virtual Objects, Virtual Meta Locations and Virtual Locations. The hierarchy of which component manages which other can be read from the attributes in the Lookup Service that are registered together with the *objectID*.
When selecting a component, available information about that object should be shown. VoxiWatch could be extended to integrate the features of the EventSource application and it could allow to send queries to components and show the replies.

It is important to understand that VoxiWatch is quite restricted in its possibilites to monitor VOXI. The events are not sent through a central instance and so VoxiWatch cannot trace them. It can neither know at which Virtual Locations a Virtual Object resides currently as only the Virtual Object itself knows that. VoxiWatch gets all the information displayed from the Lookup Service and the attributes registered together with the *objectID*[36] or from a

[35] The implementation will choose Jini^{TM}'s Lookup Service `reggie`.
[36] cf. 5.12

component itself upon inquiries. This shows that installing a central monitoring and control authority in VOXI is almost impossible which was one of our considerations when designing the system and by deciding to give the Virtual Objects as much independence and autonomy as possible. However, especially the current prototype implementation of the Artifact Memory[37] definitely has to be improved if privacy is a major concern.

The screenshot in figure 19 shows the VoxiWatch GUI. There are two active Virtual Object Managers shown: MyVOM and MyCarVOM. A collection of compact discs lies on a table. The discs are all managed by the Virtual Meta Object MyVirtualMusicBox. This fact is shown in the status bar when the mouse is pointed to a disc. Two teddies are sitting in the car and are waiting for the green teddy which is about to migrate to the MyCarVOM. The Artifact Memory is also shown. The location "Virtopia" was created when the first teddy moved to the new Virtual Object Manager MyCarVOM.

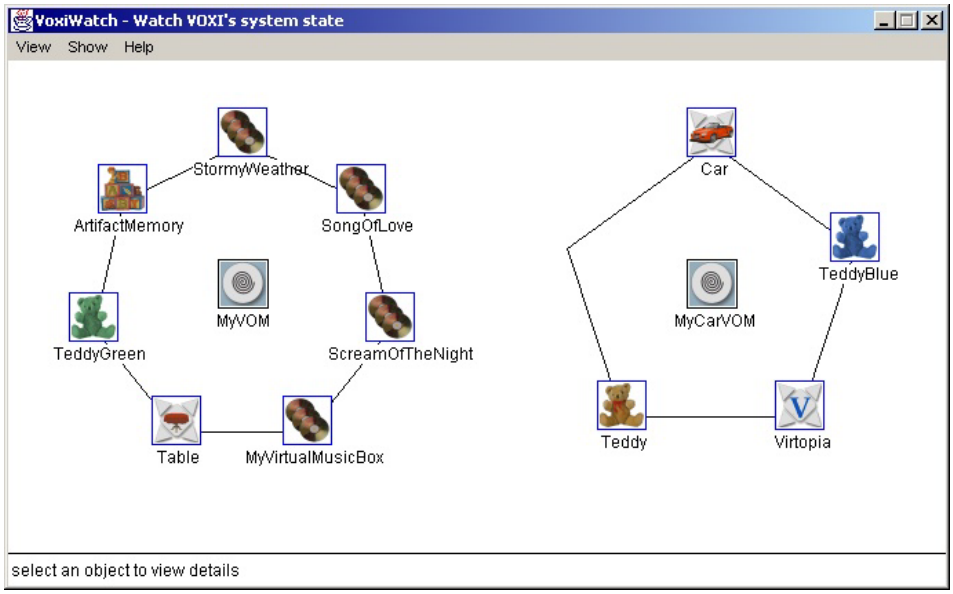


Figure 19: *Screenshot of the VoxiWatch GUI.*

4.14 Security and Privacy

As already mentioned in section 3.8 VOXI is open for future extensions concerning security and privacy. There are two key points which allow to embed encryption and authentication.

[37] Refer also to the discussion in 4.10.4.

1. The meta information of an event could carry information about an encryption scheme for the event. It could also hold information related to authentication.

2. A Virtual Object Manager communicates a security token to all Virtual Components that it creates (by providing it as a parameter of the component's constructor when creating the virtual counterpart locally which then gets wrapped in a new thread). This security token can be used for privileged services of the Virtual Object Manager (although the component might already have migrated to a distant location) and it is used as a privilege to actively ask a component to terminate (cf. the `pleaseExit` method of a Virtual Object).

4.15 Further services

An infrastructure is never complete as new requirements arise with new applications. VOXI is highly extensible for new services. Any component that can register itself at the Lookup Services and that supports VOXI's event communication paradigm can be seamlessly introduced without restarting the system and act as a service for components in VOXI. It might be clever to wrap a new service in a Virtual Object that is created at the location of 'Virtopia' (cf. 4.3.5) and managed by a Virtual Object Manager. In this case the developer can fully concentrate on the new service and let the Virtual Object Manger do all the tasks to correctly integrate a new service component into VOXI.
New events, which new services will need for their operation, do not have to be understood by all components in the system. Only components in VOXI such as for example virtual counterparts which want to use the new service must understand the new event-based protocol of the newly provided service.

The following list of suggested further services or extensions is by no means final:

Infrastructure Information This service can tell how far apart two locations $locationID_1$ and $locationID_2$ are and which other location is close to a given one. The positioning information has to be collected (probably manually) in advance before such a system is possible.

GPS support The problem with absolute positioning data is that we do not want to create a Virtual Location for each unique absolute position such as GPS(23,30,400) or GPS(23,33,405). The only thing we have to enhance is the mapping capability provided by the Virtual Object Repository in order to support pattern matching (e.g., regular expressions) of unique location identities. The location pattern "GPS(x,y,$height$)" could then be mapped to a single or a few Virtual GPS Meta Locations.

5 Implementation

5.1 General

The implementation is based on young but promising technologies as VOXI is intended to be a research prototype rather than a world wide productive infrastructure where people and vital services depend on. The implementation of VOXI provides a simple but extensible core infrastructure.

We tried to meet the following implementation guidelines:

- **Java**^{TM} as the programming language: Object-oriented and platform-independent code, support for Remote Method Invocation (RMI), Java^{TM} is type-safe, garbage collector, security is supported by policies for classloaders and the virtual machine's (VM) sandbox, inherent support for multi-threading
- **Jini**^{TM}[38] as the core infrastructure: Useful services for a distributed infrastructure such as a LookUp Service (LUS) or distributed events are provided
- **Simple**: Small required interfaces; heavy code reuse (inheritance, voxi package); event-based communication paradigm
- **Extensible**: OO design, standards (TCP/IP, SQL, RDF/XML, Jini), generic events and interfaces

5.2 Challenges

The experience gained during the implementation phase made several development challenges apparent. VOXI is a system which can be described as being:

- distributed
- asynchronous
- dynamic
- multi-threaded
- scalable
- extensible

[38] Jini^{TM} is an acronym that stands for Java^{TM} Intelligent Network Infrastructure.

However, the above properties are not even complete as we should possibly build a system that is at the same time robust, secure, highly available and not too resource demanding.

To make it even more challenging, the fact that we always have to deal with time synchronization problems in a distributed environment has to be considered when evaluating timestamps of for example Entry- and ExitEvents.

5.3 Overview

The result of the implementation efforts is a collection of Java classes and some resource files that were all put in a hierarchial package named voxi. A description can only illustrate the important ideas behind the implementation. When developing applications for any system the unwritten law that "the source is the documentation" will still hold true. A lot of time was spent on writing thorough comments directly into the source[39] code files.

5.3.1 The VOXI Package

The structure of the VOXI package looks like this:

Package prefix	Description of package contents
voxi.core.*	The very core of VOXI. All services and components such as the Virtual Object Manager, Artifact Memory, the superclass for Virtual Objects and Locations, Event defintion and others
voxi.util.*	Utilities that can also be used outside of VOXI, e.g., a central log module or an image loader
voxi.apps.*	In this package you will put your own Virtual Objects, Virtual Locations, Meta Objects or application specific services.

5.4 Communication Infrastructure

It is tempting to use RMI[40] as we decided to implement VOXI in Java^{TM}. But RMI is known to suffer performance problems (cf. [33]) and so we require that Virtual Objects are free to use other mechanisms (e.g., as in CORBA® [37]) where reasonable. This can be achieved by decoupling the `VirtualObjectEventListener` interface from the `java.rmi.Remote` interface, which would mark it as a remote interface that relies on RMI. To assure compatibility with an RMI implementation we have to declare each remotely accessible method to throw `java.rmi.RemoteException`. Please note that

[39] There is also a HTML reference for the voxi package which was generated by using the javadoc utility.

[40] Java^{TM}'s Remote Method Invocation

this exception does not require RMI at all even though it resides in the `java.rmi` package. If a Virtual Object decides to use its own communication paradigm instead of RMI then it must not implement `java.rmi.Remote`. When a Virtual Object wants to communicate with another component then it needs the stub before it can make an RMI call. As VOXI is open for new Virtual Object types and as we do not want to distribute new RMI stubs to all possible communication partners in advance, we use Jini^{TM}'s proxies. A Virtual Object registers a reference to the codebase of its proxy (which in fact can be a RMI stub being stored a the Virtual Object Repository) at the Lookup Service and each Virtual Object which wants to communicate can simply download the proxy on demand. In VOXI all Virtual Objects which do not provide additional remote methods can simply use the proxy of `VirtualObjectImpl` which is supplied in the voxi package and which is already available locally at each Virtual Object Manager. A Jini^{TM} proxy which does not implement `java.rmi.Remote` will be downloaded as if it were a simple stub and herewith is able to use proprietary communication to its "backend" that resides somewhere in the same network.

However, RMI, which is seamlessly integrated into Java^{TM} and which uses a transport mechanism[41] with acknowledgments, is used in VOXI for the implementation of the abstract Virtual Object superclass[42] that most application-specific Virtual Objects will inherit from. Unfortunately RMI[43] is not clever enough to handle calls to local instances without overhead by simply doing a local call. The reason might be that this would alter the calling semantic as references are passed differently in local method calls in comparison to remote method calls. The Virtual Object Manager therefore uses local calls to forward Entry- and ExitEvents where possible to improve performance. Objects that are using RMI can receive many simultaneous calls to their remote methods which causes the RMI receiver thread to spawn a new sub-thread for each call. The developer therefore has to make sure that the remote methods are thread-safe.

5.5 Events

5.5.1 Generic Event Type

Events as defined in 4.3 are the backbone of VOXI's communication. It is crucial that the implementation is open for new event types and that sending and handling events is efficient. It must be possible to introduce new event types while the system is already running.
We have chosen a very generic event type that stores the event properties

[41] TCP/IP
[42] Note that RMI is not part of the core interfaces that a Virtual Object must implement.
[43] RMI as of JDK 1.2

in the form of key/value pairs in a HashMap[44]. The type of the event cannot be changed after the event was created but the key/value pairs still can be edited. Analogously to other Java^{TM} types we also provide an `equals` method to compare two events for (commutative) equality as defined in 4.3.2. The final variable `serialVersionUID` makes it more efficient to serialize an event upon sending it to a remote receiver as the number does not have to be recalculated upon each serialization. By the way we chose a `HashMap` because it can hold named keys, null values and because it is unsynchronized (in contrast to the similar Hashtable) which makes it extremely fast.

VOXI's core event type `voxi.core.event.Event` is important enough to be presented in almost full source code here:

```
public final class Event implements Serializable {
 static final long serialVersionUID = 7958819389897872161L;

 /**
   * The unique type identifying label for this event.
   */
 public final String type;

 /**
  * The properties specific to each event type defined
  * only by convention.
  */
 public HashMap properties = new HashMap();

 /**
  * Constructs a new event with the specified type.
  *
  * @param  type the label which identifies this event's "class"
  * @throws IllegalArgumentException if the type parameter is null
  */
 public Event(String type) {
   if (type == null)
     throw new IllegalArgumentException();
   this.type = type;
 }

 /**
  * Compares this event to the specified event
  * (for commutative equality).
```

[44] `java.util.HashMap`

```
   *
   * @param  obj   the event to compare with
   * @return true if both events are equal
   */
  public boolean equals(Object obj) {
   ...
  }
}
```

5.5.2 Case Study: Event Subclasses Versus HashMaps

One could argue that a HashMap cannot guarantee proper type checking for the compulsory key/value pairs depending on the event type and this is fully true. Nevertheless we refrained from using event subclassing because this would be quite inefficient as we will show now: When delivering an event to a remote component we download the stub of the receiver once and call its notify or request method. The stub[45] has to serialize the event and move it to the remote component where it is deserialized into a new event instance. The new instance can only be created if the class definition for the new event subclass is available. RMI provides a reference to the corresponding class file which has to be downloaded from a web server. This involves a huge overhead for new previously unknown event types that each receiver will have to perform just to state in a later phase that it actually cannot process this new type of event. When we take 10 new event subclasses that are sent to 100 virtual counterparts all of them will have to download the new class code which causes 1000 downloads from the Virtual Object Repository. And if the repository might be unavailable for some time then new events cannot be processed at all. The implementation of a `HashMap` in contrast is available locally and the only thing we have to do when introducing new events by convention is making sure that the event type names are unique. Furthermore it is much easier for an even handler to investigate new event types. It does not have to use complicated (but powerful) mechanisms such as reflection[46] to find out about the event properties. It can simply iterate over the entry set of the `HashMap` that contains all key/value pairs.

5.5.3 Event Wrapper

We want to embed our generic event format into a distributed event of Jini^{TM} in order to make the event receiving methods of a virtual component compatible to subscribers for distributed events and to allow for the use of event mailboxes and future event services of Jini^{TM}. Moreover we need

[45] In most cases this will be an RMI stub but VOXI allows proprietary transport protocols too.

[46] Reflection is also known as introspection.

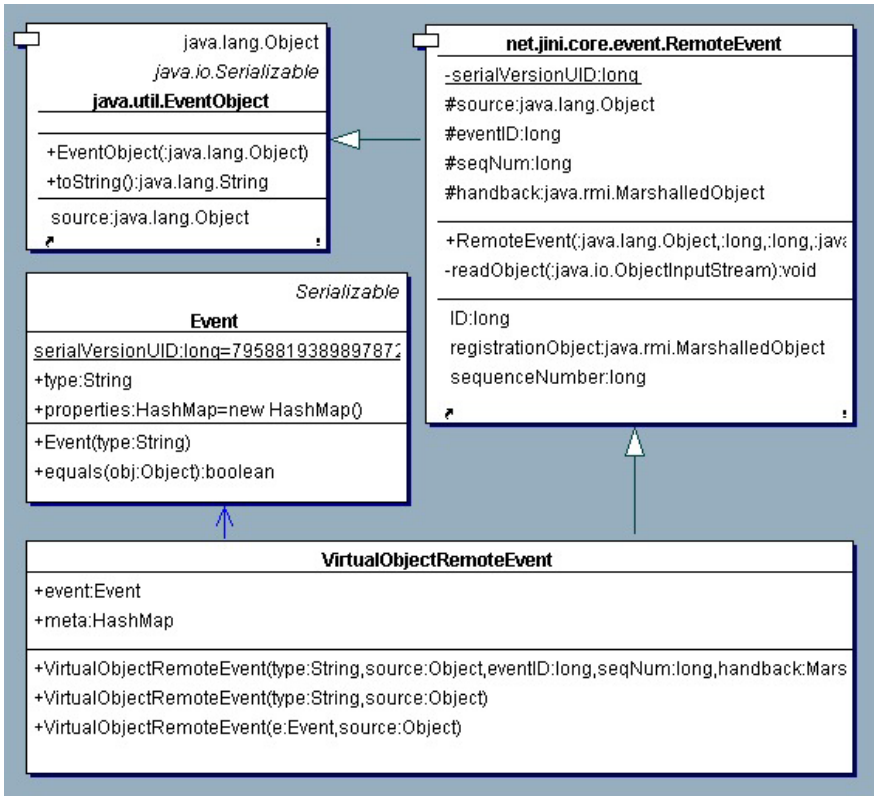


Figure 20: *UML class diagram of the event Wrapper VirtualObjectRemoteEvent which embeds VOXI's core event and meta information.*

a place to put the meta information. Both issues can be solved with an event wrapper that subclasses a distributed event (cf. figure 20) and by an interface that subclasses the RemoteEventListener interface (cf. figure 21). This design shows how seamlessly the VOXI components fit into Jini^{TM}.

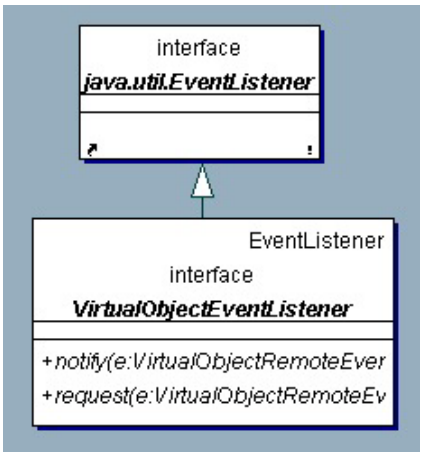


Figure 21: *UML class diagram of the event listener interface extension VirtualObjectEventListener.*

5.5.4 Event Delivery

For ease of use VOXI provides an **EventDeliveryManager**[47] that sends an event to a local or remote object named *objectID*. The complicated mechanisms of finding the receiver object, downloading its stub and handling minor errors are hidden from the user.

5.5.5 Event Matching

The class **EventTemplateMatching**[48] implements the template matching for events and is compliant to our definitions of "matches" (cf. 4.3.3) and "strictly matches" (cf. 4.3.4).

5.5.6 Timestamps

A timestamp can be represented in a huge variety of formats. We have chosen to implement it as the number of milliseconds that have passed since Jan 1, 1970, 00:00:00 GMT. The timestamp is represented as a 64-bit signed integer value of type long. This format is restricted to the range $[-2^{63} \ldots + (2^{63} - 1)]$ which corresponds to the years -292 million years B.C.

[47] `voxi.core.event.EventDeliveryManager`
[48] `voxi.core.event.EventTemplateMatching`

. . . +292 million years A.D., which hopefully will be sufficient.

The reason why we have not chosen a variable length text representation that could represent every possible date in the very far future is that the handling (i.e., sorting, comparing, retrieving from the realtime clock[49], storing in a database) of numeric timestamps is much more efficient.
VOXI itself does not rely on timestamps at all. However, the Entry- and ExitEvent were defined to have a timestamp included by the creator of the event. If we want to evaluate timestamps from different event sources (e.g., events stored in the Artifact Memory) than we need a possibility to synchronize time at the event sources (e.g., by using the Network Time Protocol NTP).

5.6 Virtual Objects

5.6.1 Features

The list of the features of Virtual Objects as shown in the description of the design (cf. 4.5.1) is rather long. The core implementation tries to provide as much functionality as possible such that the developer of a new Virtual Object can make new Virtual Objects inherit all the features from the abstract `VirtualObjectImpl` reference implementation superclass and then focus on the event handler.

The class diagram in figure 22 shows which basic features a Virtual Object supports by design.

event handling By extending the `VirtualObjectEventListener`[50] interface two event handlers, `notify` for event notifications that require no reply and `request` that always sends an event as the reply, have to be implemented. The reference implementation in `VirtualObjectImpl`[51] simplifies both entry points to a single method `eventHandler` that processes both sorts of events. The core event types `EntryEvent` and `ExitEvent` are accepted. The `ExitEvent` calls the `pleaseExit` method of the object by default. However, a developer has full control to change the default behavior.

self-description The interface `Describable`[52] enforces that each Virtual Object has to supply a description (a String with information compliant to RDF/XML), a name (a String) and

[49] The method System.currentTimeMillis() in Java^{TM} returns a timestamp that is fully compliant to our timestamp definition.

[50] `voxi.core.event.VirtualObjectEventListener`

[51] `voxi.core.virtualobject.VirtualObjectImpl`

[52] `voxi.core.description.Describable`

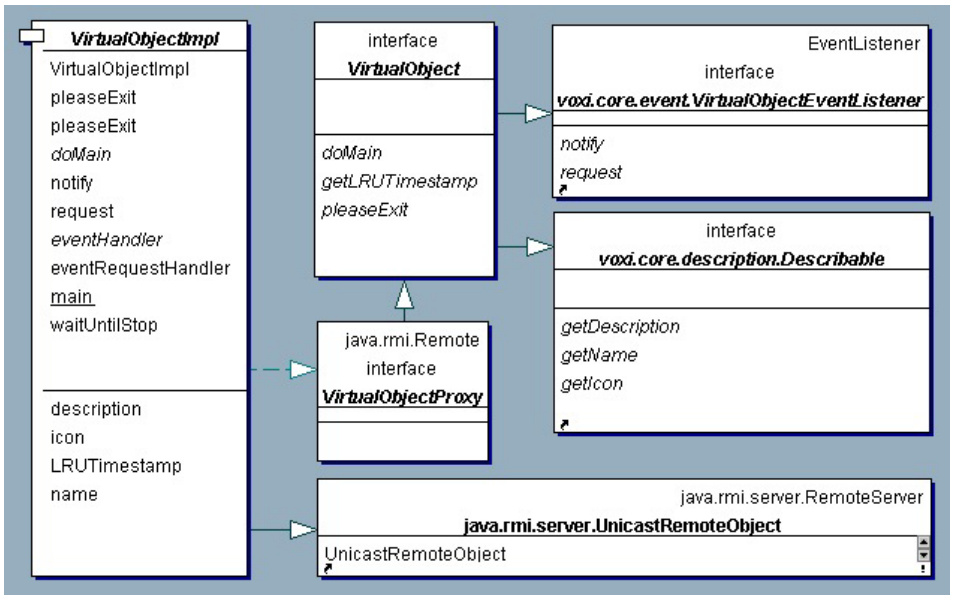


Figure 22: *UML class diagram that shows the important interface* `VirtualObject`*, which is common to all Virtual Objects, and the reference implementation* `VirtualObjectImpl`*.*

an iconic image (a java.awt.image, preferably of the size 32x32 pixels).

lifetime management The interface `VirtualObject`, which is common to all Virtual Objects independent of whether they use RMI (as the reference implementation `VirtualObjectImpl` does), introduces three methods related to lifetime management.

- The method `doMain` is called after the Virtual Object has been created by a Virtual Object Manager. When a Virtual Object returns from that method then it terminates. Subsequently it is unregistered and unloaded by the Virtual Object Manager. Instead of busy waiting to prevent an early return we recommend to use the `waitUntilStop` method in the reference implementation which simply lets the main thread of an object wait until `pleaseExit` is called. The method `waitUntilStop` uses a blocking technique which does neither prevent nor slow down the reception and processing of events in the mean time.
- The `getLRUTimestamp` method returns the timestamp (cf. 5.5.6) of the last event reception by default and

can be used by the Virtual Object Manager to find out which Virtual Object was not in use for a long time and unload it if resources get scarce.

- Finally the method `pleaseExit` gracefully asks the Virtual Object to terminate as soon as possible by returning from its `doMain` method.

The constructor of `VirtualObjectImpl`[53] is passed a reference to the Virtual Object Manager that has created this instance, the unique *objectID* and a security token that the Virtual Object Manager will reuse when calling the `pleaseExit` method.

5.6.2 Implementing a New Virtual Object "MyObject"

Writing a new Virtual Object should be as simple as possible. We therefore made the Virtual Object Manager take over most of the complicated tasks such as asynchronously registering a Virtual Object at the Lookup Service or renewing the leases. An abstract superclass for Virtual Objects reduces the programming effort to a bare minimum. Service classes in the voxi package provide many high-level abstractions from the cumbersome asynchronous communication and interaction patterns and for common tasks such as logging[54].

A new Virtual Object "`MyObject`" usually[55] inherits from the reference implementation `VirtualObjectImpl` and has to implement the event handler `eventHandler` and the main method `doMain`. It usually sets its own description, name and iconic image during initialization. Application specific or user-supplied new Virtual Objects such as "`MyObject`" reside in the package `voxi.apps.virtualobject.`*MyObject*. Before an `EntryEvent(`objectID = "MyObject"`,` *locationID*`,` *timestamp*`)` can activate the new Virtual Object its class files, resources and a mapping file have to be copied to the Virtual Object Repository (cf. 5.10) where it will be downloaded from by a Virtual Object Manager upon creation.

5.6.3 Some Notes on RMI

The Remote Method Invocation (RMI) in the Java^{TM} Development Kit (JDK) v. 1.2 no longer requires a RMI skeleton for the server. A RMI stub on the client side now is sufficient as the Java^{TM} Virtual Machine was

[53] The Virtual Object Manager uses reflection to find a compatible constructor and **does not require a subclass of VirtualObjectImpl** (as this simple-minded enforcement would restrict VOXI to RMI compliant Virtual Objects).

[54] `voxi.util.Log`

[55] Don't forget that our infrastructure is also open for Virtual Objects that use e.g., CORBA® instead of RMI.

extended to handle remote calls itself. As Jini^{TM} requires JDK v. 1.2 (i.e., Java 2) we no longer need to generate a RMI skeleton.
If no new remote methods are added to an application-specific implementation of a Virtual Object then there is not even a need to generate a client RMI stub as the one of `VirtualObjectImpl`, which is already provided, will be used. This feature can be seen as a generic RMI stub.
Virtual Objects provide remotely accessible methods, which are called using RMI (in the reference implementation). As RMI starts a new thread for each call and as this allows for multiple parallel remote calls a Virtual Object has to be thread-safe. The programmer has to protect internal data structures from simultaneous edit operations by different threads which else would cause havoc.

5.6.4 Internal State of a Virtual Object

The `VirtualObjectImpl` uses two internal boolean flags: `ready` and `stop`. Both are set to false upon construction of the Virtual Object. The `ready` flag is set to true after the initialization has been done. The `stop` flag is set to true when `pleaseExit` is called. The `waitUntilStop` method, which many Virtual Objects will use to prevent themselves to return from `doMain`, will return after the `stop` flag has become true.

As `VirtualObjectImpl` extends the class `UnicastRemoteObject` a Virtual Object will actually not cease from receiving events after having returned from the `doMain` method as long as the wrapping thread (cf. 5.9) has not exited. An event handler therefore might want to take care of the state of the flag `stop` before processing an event and refuse events in a state when `stop` is set to `true`. However, the wrapping thread will issue a non-blocking[56] call to the method `unregister` of the Virtual Object Manager after `doMain` was left and before exiting.

5.7 Virtual Locations

A Virtual Location is quite similar to a Virtual Object. Additionally it provides an event subscription and publishing mechanism which is implemented in the default event handler of the abstract reference implementation `VirtualLocationImpl`[57]. Additionally a Virtual Location has to forward `Entry-` and `ExitEvents` to the corresponding Virtual Objects as illustrated in figures 12 no page 39 and 14 on page 43.

[56] The internal unregister operations are fast. It is meant that the call does not block until the object is unregistered from the somewhat slower Lookup Service.

[57] `voxi.core.virtuallocation.VirtualLocationImpl`

5.8 Meta Objects

Actually there is no formal difference between the implementation of a Virtual Meta Object or Meta Location and the implementation of a Virtual Object or Location. Of course a Meta Object will get `EntryEvents` for several other objects which it will manage internally. The only thing that is really different is that the mapping file for the Virtual Object Repository will not contain a self-reference but more than one unique identity and has to be copied to all the different names of the identities contained in it to make the mapping work. Please consult the description of the mapping mechanism in 5.10 for further details.

5.9 Virtual Object Manager

The Virtual Object Manager is at the core of VOXI. The implementation was especially challenging as it is heavily multi-threaded. Much work has been invested to make it thread-safe without unnecessarily blocking any parallel executable operations and to avoid memory leaks. The design of the Virtual Object Manager in section 4.8 contains a comprehensive description of its internal processing, which will not be repeated here.

5.9.1 Virtual Object Creation Upon EntryEvents

The flowchart in figure 12 on page 39 has illustrated the processing of an `EntryEvent` by the Virtual Object Manager. An extended illustration that is closer to the implementation is shown in figure 23.

The creation of a new Virtual Object works like this: The *objectID* contained in the `EntryEvent` is used to retrieve the mapping to a Virtual Meta Object (or to itself if it manages itself). A local mapping cache speeds up this operation and a check guarantees that the Virtual Object does not already run locally or remotely. Then the code and resources of the new Virtual Object are downloaded from the Virtual Object Repository (cf. 5.10) by using a new `URLClassLoader`[58]. The code basis is either a single JAR file or a base URL for several distinct files. The new Virtual Object is instantiated and handed to a newly spawned thread of type `VirtualObjectThread`, which has its name set to the *objectID*. The thread's `run` method calls the `doMain` method of the Virtual Object to make it initialize itself. Then the Virtual Object Manager registers the new Virtual Object in its data structures and at the Lookup Service. The thread which embeds it will automatically unregister it when it will return from the `doMain` method.
Virtual Objects and Locations that are managed by an already running Virtual Meta Object or Meta Location are only registered internally and at the

[58] A class loader also allows to restrict policies if needed.

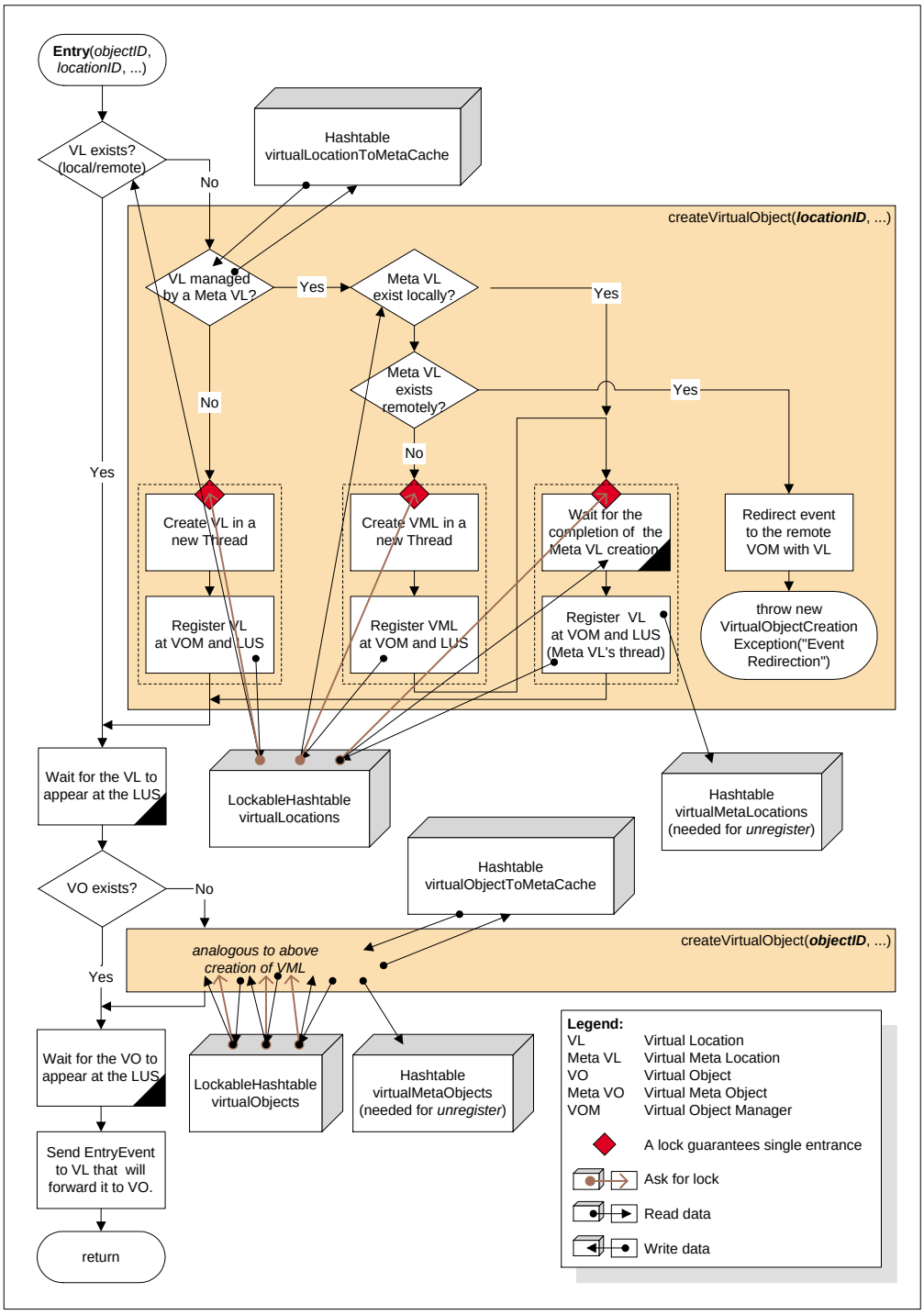


Figure 23: *Flowchart that illustrates the processing of an Entry Event by the Entry Thread.*

Lookup Service but no new thread is created.

The creation of Virtual Locations is analogous to the above procedure and the creation of Virtual Meta Objects and Meta Locations differs only in that they are indirectly created by `EntryEvent`s with an *objectID* or *locationID* that maps to the identity of the Meta Object or Meta Location.

5.9.2 Data Structures

The Virtual Object Manager uses three distinct data structures for the bookkeeping of Virtual Objects:

- The field `virtualObjects` (of type `LockableHashMap`) contains the mappings from *objectID* to the thread (of type `VirtualObjectThread`) which wraps the associated Virtual Object with that *objectID*. The *objectID* belongs to a Virtual Meta Object or to a Virtual Object which either manages itself or is managed by a Virtual Meta Object. This implies that several different *objectID*s map to the same thread if they belong to the same Virtual Meta Object.
- The field `virtualMetaObjects` (of type `Hashtable`) simplifies the management of Virtual Meta Objects, especially if a Virtual Meta Object returns from its `doMain` method which causes **all** managed Virtual Objects to be unregistered as a consequence. The field maps an *objectID* of a Virtual Meta Object to a linked list[59] with all the *objectID*s of the managed Virtual Objects. The first item in the list is by convention the *objectID* of the Virtual Meta Object itself.
- The field `virtualObjectToMetaCache` (of type `Hashtable`) implements a simple caching mechanism that maps an *objectID* to the meta-*objectID* of an associated Virtual Meta Object which manages this Virtual Object or to the *objectID* of the Virtual Object itself if it manages itself.

The bookkeeping of Virtual Locations is done in separate fields which use the same three types of data structures as explained above. The separate fields are named `virtualLocations` ($\doteq$ `virtualObjects`), `virtualMetaLocations` ($\doteq$ `virtualMetaObjects`) and the field for the local cache of Virtual Meta Locations is `virtualLocationToMetaCache` ($\doteq$ `virtualObjectToMetaCache`). The functionality is exactly the same.

5.9.3 Multi-Threading Issues

A thorough look at the flow chart in figure 23 that shows the processing of the `EntryEvent` reveals some interesting scenarios related to multi-

[59] `java.util.LinkedList`

threading. You can recognize three locks in the colored box which is named `createVirtualObject(`*locationID*`, ...)`. There are three other similar locks in the second colored box below which is only partly drawn as it it fully analogous. There is exactly one lock for each *locationID* that is going to be stored in the field `virtualLocations`[60] which is of our own type `LockableHashMap`. Only a thread which can obtain the lock is allowed to execute the operations in the dotted box below the lock (i.e., register and/or create the corresponding Virtual Object). A thread which does not get a lock simply omits the dotted box and goes on beyond[61]. These special locks are crucial as we will see in the following scenarios:

Preventing Multiple Creations of the Same Counterpart: Two compact disks named $diskID_1$ and $diskID_2$ are sighted almost simultaneously by a sensor at a location named *locationID*. This causes two `EntryEvent`s to arrive almost concurrently at the Virtual Object Manager. As the Virtual Location *locationID* of the sensor does not already exist, both threads that process the different `EntryEvent`s would decide to create the Virtual Location and register[62] it which would cause two Virtual Locations with the same name *locationID* to be active. This would be an inconsistent state of VOXI as it violates the concept of unique identities what we try to prevent. This scenario has explained the leftmost lock.
The lock in the second column of the flowchart is used to assure that a Virtual Meta Location is created only once although there might be multiple threads processing `EntryEvent`s with *locationID*s that map to the same meta-*locationID*. Only the one that gets the lock on the meta-*locationID* can enter the dotted box. The others will omit the dotted box and try to get the lock for their *locationID* as shown in the third column in the flowchart. This lock assures that the Virtual Location *locationID* which is managed by a Virtual Meta Object is registered only once. The action "wait for the completion of the Virtual Meta Location's creation" in the box following that lock is also crucial. It assures that a Virtual Location is not registered at the Lookup Service prior to its Virtual Meta Location. Without the wait block (which relies on Java^{TM}'s wait/notify mechanism that was implemented in the `waitForKey` method of our `LockableHashtable`) a first thread could still be creating the Virtual Meta Location whereas an almost parallel second thread would omit the creation box and try to register its Virtual Location at the Lookup Service with a reference to the not yet existing Virtual Meta Location, which would actually cause a null reference to be used for the registration if the new thread for the Virtual Meta Location was not already created.

[60] The case for *objectID* and `virtualObjects` is fully analogous.
[61] Note that threads do not wait on these locks.
[62] Note that the `register` operation is rather slow which makes this race condition more likely to happen.

5.9.4 Processing ExitEvents

The processing of `ExitEvent`s is straight forward and is almost entirely explained by figure 14 which is shown in section 4.8 where the design of the Virtual Object Manager is described.

The unregister process will not take place until a Virtual Object returns from its `doMain` method.

5.9.5 Migration Support

The bare migration support provided by method `migrate`(*objectID*, *remoteVirtualObjectManagerID*) does update an internal memo which will set a trigger for an `EntryEvent(`*objectID*`,` *locationID*=‘Virtopia’`,`*current timestamp*`)` and send it to the specified remote Virtual Object Manager that is named *remoteVirtualObjectManagerID*. The trigger is fired after a Virtual Object has returned from its `doMain` method and after it was unregistered internally and from the Lookup Service.

There is no special support for data synchronization between the two Virtual Object Instances. A Virtual Object is advised to write all important data to the Artifact Memory before quitting. The “migrated” instance could than read the Artifact Memory (e.g., persistent state data) and load an old state. Subscriptions to Virtual Locations are not automatically deleted when a Virtual Object has migrated. The Virtual Location is designed to send the events to the new network address automagically without having to be notified about the migration.

5.10 Virtual Object Repository

An HTTP[63] compliant web server was chosen as the basis of the Virtual Object Repository. An FTP compliant server would also have been a common solution for mere data downloads but FTP-only servers are not as versatile as HTTP ones because they usually lack a server-side scripting capability. We decided to use the Apache (cf. [40]) web server as it is free, stable and powerful.
The Apache web server processes HTTP which is transported over the common network transport protocol stack TCP/IP[64]. This makes it easy to reach the Virtual Object Repository from any host in a network based on Internet’s technology. HTTP communication could easily be encrypted with

[63] Hypertext Transfer Protocol (cf. [39])

[64] TCP/IP is the Internet’s Trasmission Control Protocol (TCP) in the transport layer that relies on the Internet Protocol (IP) in the network layer.

SSL/TLS[65] if needed in a future release of VOXI. Digital certificates could be used together with SSL/TLS to provide state-of-the-art authentication.

5.10.1 Repository

Contents stored on a web server can easily be mirrored by other web servers to achieve a good performance and availability. This leads to a federation of Virtual Object Repositories with a single master repository that holds the primary copy of the code and resource data of all Virtual Objects and Locations available. We refrained from an update everywhere replication scheme as this would require exact distributed time-stamps or at least the use of a versioning scheme to enable the service to resolve conflicts of different code and resource versions when synchronizing repositories. The primary copy strategy also enables a hassle-free control over the data in the various repositories.

A Virtual Object Manager must know at least one URL[66] reference of a nearby Virtual Object Repository. A list of such URLs at the Virtual Object Manager can increase the availability of the repository service.

The repository has to store executable code in the form of Java^{TM} class files and resources such as for instance images or text files. As Virtual Objects are written in Java^{TM} they can make use of the hierarchical packaging[67] mechanism. In this case the code and resources must be stored in a hierarchical directory structure that matches exactly on the package structure.
The Virtual Object Repository provides two different ways to store files related to a Virtual Object named *objectID* respectively files related to a Virtual Location named *locationID*:

- **normal**: All files are stored in a folder of the name *objectID* resp. *locationID* and its subfolders.
- **archived**: All files are archived in a JAR[68] file of the name *objectID*`.jar` resp. *locationID*`.jar`.

An application specific **Virtual Object** named *objectID* is expected to be part of the package `voxi.apps.virtualobject.`<*objectID* in lower case> and therefore a Virtual Object "`MyTeddyWinnie`" is stored in the folder `voxi/apps/virtualobject/myteddywinnie` as a Java^{TM} class file of the

[65] SSL stands for Secure Sockets Layer which is a channel encryption scheme than can be used to encrypt HTTP traffic. The Transport Layer Security (TLS) scheme is superseding the SSL standard.
[66] Uniform Resource Locator (cf. [39])
[67] Refer to the keyword `package` of Java^{TM}.
[68] Java^{TM} ARchive; the Java^{TM} SDK supplies a tool to build JAR files.

name `MyTeddyWinnie.class` or in the archive `MyTeddyWinnie.jar` in the repository's root folder.
Likewise an application specific **Virtual Location** named *locationID* is expected to be part of the package `voxi.apps.virtuallocation.`<*locationID* in lower case> and therefore a Virtual Location "`MyOffice`" is stored in the folder `voxi/apps/virtuallocation/myoffice/` as a Java^{TM} class file of the name `MyOffice.class` or in the archive `MyOffice.jar` in the repository's root folder.
A Virtual Object Manager first looks for the jar file and if that cannot be found then it looks for the class file in the appropriate folder. If this action fails too, then the Virtual Object or Location cannot be created. The current mapping facility does not provide a default virtual counterpart for unknown unique identities.

In order to make configuration changes easier the constant package prefix names, the root folder of the repository as well as other global settings in VOXI are set in the Java^{TM} source file `voxi.core.Global.java`.

5.10.2 Mapping Facility

The mapping facility maps *objectIDs* to the *objectID* of a Virtual Meta Object and *locationIDs* to the *locationID* of a Virtual Meta Object. If the *objectID* or *locationID* is not associated with a Virtual Meta Object or Location then the same *ID* is returned.

```
# A music box that manages a couple of cd tokens
# Here goes the Virtual Meta Object
MyVirtualMusicBox
# And here go all the Virtual Objects we want to
# manage by our virtual music box
SongOfLove
ScreamOfTheNight
StormyWeather
```

Figure 24: *A sample mapping file for the Virtual Meta Object MyVirtualMusicBox.*

In order to keep things simple we implemented this lookup mechanism with static text files. However, if there is any need to make the lookup more powerful, then a server-side script or e.g., servlet can be used without having to change anything at the Virtual Object Managers that are using this service. When the Virtual Object Repository serves a request of the form GET http://<host and port of Virtual Object Repository>/mapping/*objectID*[69]

[69]In a mapping file for a Virtual Meta Location the *locationID* is used in the URL

then it returns a mapping text file.

Syntactical structure of a mapping file `M` specified in EBNF[70]:

```
M              = MetaIDLine {IDLine} {CommentLine}.
MetaIDLine     = {CommentLine} objectID NewLine.
IDLine         = {CommentLine} objectID NewLine.
CommentLine    = "#" Unicode NewLine.
NewLine        = [CR] LF.
objectID[a]    = Symbol {Symbol}.
Symbol         = (Letter | Digit) Letter | Digit.
Letter         = "A" | "B" | ...  | "Z" | "a" | "b" | ...  |
                 "z".
Digit          = "0" | "1" | ...  | "9".
Unicode        = <16 bit unicode>.
CR             = "0x000D".
LF             = "0x000A".
```

[a]The *objectID* is replaced by the *locationID* in mapping files for Virtual Locations.

In short the definition means that the first uncommented line is the *objectID* of the repository item that all the other *objectIDs* in the later uncommented lines of the file are mapped to. To make the lookup mechanism work the sample file shown in figure 24 must be copied and renamed to all the *objectIDs* that the mapping file contains.

An *objectID* that is mapped to itself has a mapping file with only one uncommented line where its *objectID* is contained.

Although Java^{TM} supports unicode[71] we restrict unique *objectIDs* or *locationIDs* to a sequence of ASCII letters and digits.

5.10.3 Future Extensions

The architecture chosen for the Virtual Object Repository is open for future extensions such as a lookup by object properties instead of by *objectID* (e.g., by using a server-side scripting language such as PHP[72] or by using a Java^{TM} servlet that searches the repository) or for a more powerful mapping mechanism that supports pattern matching to map locations such as GPS(x,y,*height*) on a single *objectID*. It might also support a default mapping for unknown unique identities.

instead of the *objectID*.

[70]EBNF is the Extended Backus-Naur Form that can be used to specify a syntax.

[71]Java^{TM} supports Unicode 1.1.5. See http://www.unicode.org/ for details

[72]http://www.php.org/

5.11 Artifact Memory

The Artifact Memory implements facilities to store events and context data persistently and for querying on event histories.

It is implemented as a Virtual Object with the objectID **ArtifactMemory** that once has to be installed at an arbitrary[73] Virtual Object Manager by issuing an `EntryEvent` for it (preferably with *locationID*="`Virtopia`"). The Artifact Memory opens a JDBC[74] connection the an SQL database[75] to store and retrieve data and events. For good performance the database should be run on the same host as the Artifact Memory itself.
The protocol defined in 4.10.3 is implemented by parsing the header information and translating the commands to SQL queries and filling the results into events.

All events of the same type (i.e., events that have the same type name and property keys) are stored in the same table. Tables are dynamically created when new event types arrive. The data types of the values associated with keys in the event's `HashMap` define the SQL data type used in the table. The Artifact Memory assumes that only the restricted set of data types as stated in 4.3.1 is used or else an error is reported.

5.12 Lookup Service

All the features of the Lookup Service as discussed in 4.11 are provided by **reggie** which is the lookup service implementation that ships together with the Jini^{TM} environment. Reggie can be run in multiple instances at different nodes in the network in order to form a federation of lookup services. The multicast discovery protocol supports such a federation and will return a list of all Lookup Services found upon discovery.

Java^{TM}'s RMI implementation does provide the **rmiregistry** which is a simplified lookup service that does not support attributes and that has to be run on each host in a network as for security reasons only local applications can bind to it (cf. [34]). A lookup, in contrast, can be done from any host. We decided to prefer reggie.
When we register the same object reference at reggie with a different set of attributes then the previous entry in the Lookup Service will be replaced. This behavior jibs at our concept of Meta Objects where a single reference is registered for many different *objectID*s[76]. Fortunately, reggie uses unique

[73] Depending on the database policy you might have to reconfigure access to the database from the host where the Artifact Memory is running on.
[74] Java^{TM} DataBase Connectivity
[75] PostgreSQL was used in the implementation
[76] The *objectID*s are placed in the attribute "Name" of the registered `serviceItem`.

*serviceID*s internally for each `serviceItem` that gets registered, which we can assign ourselves. This method allows us to implement Virtual Meta Objects and Virtual Meta Locations.
The Virtual Object Manager registers some default attributes for the virtual counterparts which it creates.

- **Name**: The unique identity[77] of a virtual counterpart[78].
- **MetaObject**: The unique identity of the virtual meta counterpart which it is managed by *or* its own unique identity if it manages itself.
- **ObjectGroup**: The unique name of the group[79] it belongs to such as "`VirtualObject`", "`VirtualLocation`", "`VirtualMetaObject`" or "`VirtualMetaLocation`" as defined in the global settings[80].
- **ObjectManager**: The unique identity of the Virtual Object Manager which manages the virtual counterpart of this registration entry.

A virtual counterpart, which wants to add some proprietary attributes to its lookup service entry, can get its associated join manager[81] from the Virtual Object Manager[82].

5.13 Event Source

The implementation of the Event Source is quite simple as it can make use of the various high-level services (e.g., `EventDeliveryManager`) in the voxi package. `EventSource` is a command line application that scans its arguments and creates an arbitrary event with the meta and event information supplied. Then it sends the event as a notification or as a request (as specified by the user) to a virtual component that implements the `VirtualObjectEventListener` interface and which is registered in the Lookup Service.

Gateways for sensors to VOXI can easily make use of the `EventSource` to send their own `Entry-` and `ExitEvent`s. All they have to do is call an external Java^{TM} program and pass some command line arguments. The usage of `EventSource` is displayed if no arguments are supplied. The `EventSource` merely acts as stub for real event sources such as a sensor. It has an interactive mode (parameter option `-i`) that allows to send multiple events without restarting it for each event.

[77] I.e., *objectID* or *locationID*.
[78] I.e., Virtual Object, Virtual Meta Object, Virtual Location or Virtual Meta Location.
[79] The Virtual Object Manager itself belongs to the group "`VirtualObjectManager`".
[80] `voxi.core.Global.java`
[81] `net.jini.lookup.JoinManager`
[82] Use `getJoinManager(String` *uniqueID*`)`.

6 Deployment

We wand to list some ideas of how our infrastructure could be deployed in various scenarios. The list shows that we can make some steps toward a smart environment with VOXI but it also makes clear that we are still far away from a smart environment. Nevertheless we can already start right now to dream about a world where even dust[83] will get smart.

6.1 Scenarios

First Steps

- **Object and people tracking**:
  Objects that are tagged with a unique identity or a badge that regularly broadcasts a beacon with its identity can be recognized by a sensor and easily be tracked with VOXI. The virtual counterparts are dynamically created upon sightings and get all the location information from the sensors in the form of entry and exit events. The Virtual Objects can be queried about the locations where they have been recently sighted and a Virtual Location can be queried about which objects are there at the moment. The Artifact Memory enables sophisticated queries on past events.

- **Business card exchange**:
  Imagine a paper business card with a thin RFID label stuck on its back. When the card is put close to an organizer the sensor recognizes that both objects are in its vicinity and creates the corresponding `EntryEvents`. The smart virtual counterpart of the business card can then find out that there is an organizer at the same location and transfer the information which is written on the card to the virtual counterpart of the organizer by simply sending a special business card data event.

- **Virtual jukebox - new human computer interaction**:
  Tiny compact disc tokens are labelled with an RFID tag. When tokens are put on a table then a virtual jukebox puts all the associated music tracks in its play list, downloads the streaming audio data from the Internet and starts playing the music. The virtual jukebox is implemented as a Virtual Meta Object.

- **Card game advisor**:
  A Virtual Meta Object that manages all playing cards used in a game could display the current set of cards on the table on a nearby screen and advice the players which action to perform next and later show a replay of the game and explain how the players can improve their

[83] Not to be taken too seriously.

skills. Similarly, an 'RFID Chef'[84] could be realized as a Virtual Meta Object.

Not Too Far Away

- **Smart household**:
  The fridge could do some bookkeeping on his stock and trigger new deliveries if some vital grocery items are missing or if they are no longer fresh.

- **Virtual nanny**:
  A virtual nanny could control a baby in a room and make sure that it does not play (i.e., comes close to) with dangerous objects and trigger some actions such as flashing some lights or calling a person to prevent injuries of the unattended baby.

- **Smart universal remote control**:
  A palm-sized remote control device with a touch screen could dynamically download the appropriate user-interfaces[85] of devices which have a unique identity. The smart remote control could be used as the only control for video recorders, washing machines, TV, DVD, heating systems, etc., which no longer would have to provide their own awkward and proprietary buttons and tiny keyboards as a physical user interface.

- **Smart toys**:
  Toys such as Smart-Its[86] can open new fields of interaction for amusement and education. A doll could change its mood depending on the weather outside which it retrieves from the Internet. A multilingual teddy could translate spoken words by recording them with a microphone, sending it to its virtual counterpart and playing the reply on its loud speaker. Such smart toys could help to provide a pioneering new learning and playing environment for young people.

[84] cf. [27]
[85] Similar to the idea of Jini^{TM}'s proxy approach
[86] cf. [28]

7 Conclusions

7.1 Summary and Outlook

The effort made to build the core of VOXI - a Virtual Object eXtensible Infrastructure - was worth while. We could show that it is possible to build an architecture that relies on cutting-edge but young technologies such as Jini^{TM} whilst staying extensible. This Master's Thesis formulated many fundamental principles and concepts in a clear way which could be a basis for further research. The representation scheme chosen to represent real world objects and physical as well as logical locations allows to use it for many different applications in the wide field of Ubiquitous Computing.

We hope that VOXI will contribute as a research prototype for further investigations in the field of Ubiquitous and Invisible Computing and that some of the concepts will be seized in related work.

7.2 Future Work

However, there remains a lot of future work to be done by people interested in enhancing VOXI. Some ideas are listed below.

- Making use of rewritable RFID tags (e.g., to allow the physical object to take a tiny bit of state information with it when physically moving).
- Design and implementation of security and privacy mechanisms. Among others authentication for the access to the Artifact Memory might be a first task.
- Extension of the VoxiWatch GUI by providing a way to issue "`where?`" and "`with?`" queries to virtual counterparts and additional functionalities as described in section 4.13.
- Design and implementation of a more powerful context-oriented query and trigger language for the Artifact Memory.
- Enhancement of the mapping mechanism of the Virtual Object Repository to support pattern matching (e.g., mapping "GPS(x,y,*height*)" to a GPS meta object).
- Enhancement of the bare migration support to a weak or even strong multi-hop migration capability.
- Finding a way to give each Virtual Object and Location its own interactive web page (based on the `getDescription` method and RDF/XML).
- Distribute and decentralize the Artifact Memory service.
- *Others: See the various remarks about extensions in this document.*

Acknowledgements

I want to conclude with saying thank you to Prof. Friedemann Mattern, his research group and especially Kay Römer who supported my interesting work in the Distributed Systems Group at the Department of Computer Science at the Swiss Federal Institute of Technology ETH Zurich, and gave many useful suggestions and ideas concerning the Virtual Object eXtensible Infrastructure.

A Bibliography

References

A.1 Object Tracking Systems

[1] R. Want, A. Hopper, V. Falcao, and J. Gibbons **The active badge location system**, ACM Trans. Info. Syst. 10, 1, Jan 1992, 91-102
One of the earliest badge tracking systems.

[2] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis and Mark Weiser, **An Overview of the ParcTab Ubiquitous Computing Experiment**, IEEE Pers. Comm. 2, 6 (Dec 1995), 28-34, source: [Citeseer]
http://sandbox.xerox.com/parctab/
Mobile thin client devices linked by infrared gateways to a network that provide location information and support interaction with server-side applications such as reading e-mail.

[3] H. W. Peter Beadle, G. Q. Maguire Jr., M. T. Smith, **Location Based Personal Mobile Computing and Communication**
http://www.it.kth.se/~maguire/LocationAware/ieee_lan_98/
Development of a Smart Badge.

[4] John Bates, David Halls, Jean Bacon, **A Framwork to Support Mobile Users of Multimedia Applications**,University of Cambridge UK, Computer Laboratory, 1996, published in: ACM Mobile Networks and Nomadic Applications
A Framework to keep user connected to multimedia teleconferencing sessions when they are moving to another computer.

[5] John Bates, David Halls, Jean Bacon, **Middleware Support for Mobile Multimedia Applications**,University of Cambridge UK, Computer Laboratory, 1997
A Middleware to move user interfaces based on X depending on the user's location.

[6] Hans-W. Gellersen, Michael Beigl, and Holger Krull, **The MediaCup: Awareness Technology Embedded in an Everyday Object**, Uni Karlsruhe, 1999, published in: Handheld and Ubiquitous Computing, HUC'99, Springer, ISBN 9-783540-665502, pp. 308-310
Experiments with a simple coffee mug enhanced by a couple of sensors.

[7] R. Borovoy, M. McDonald, F. Martin, and M. Resnick, **Things that blink: Computationally augmented name tags**
IBM Systems Journa, Vol. 35, Nos. 3 & 4, 1996 - MIT Media Lab

http://www.research.ibm.com/journal/sj/mit/sectionc/borovoy.html
http://www.research.ibm.com/journal/sj/mit/sectionc/borovoy.pdf

[8] N. Ackroyd, R. Lorimer, **Global Naviation: A GPS User's Guide**, Lloyd's of London, 2nd edition, 1994.

A.2 Radiofrequency Identification (RFID)

[9] Klaus Finkenzeller, **RFID-Handbuch**, 2. Auflage, 1999, Hanser Verlag, ISBN 3-446-21278-7
Foundations and deployment scenarios of inductive radio systems, transponders and contactless chipcards (written in German)

A.3 Ubiquitous Computing

[10] Giles John Nelson, **Context-Aware and Location Systems**, PhD. Thesis, 1998, Clare College
Comprehensive description of the general foundations of context-aware and location systems and of an event-based distributed system called CALAIS.

[11] Fritz Hohl, Uwe Kubach, Alexander Leonhardi, Kurt Rothermel, Markus Schwehm, **Next Century Challenges: Nexus - An Open Global Infrastructure for Spatial-Aware Applications** University of Stuttgart, published in: Mobicom 1999 Seattle Washington USA, pp. 249-255
http://www.nexus.uni-stuttgart.de/
Vision of augmented areas that provide virtual representations of physical objects and links to an information space. The first prototype is due not before 2002.

[12] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Mae, **Hive: Distributed Agents for Networking Things**
http://www.hivecell.net/hive-asama.html
http://nelson.www.media.mit.edu/people/nelson/research/hive-asama99/hive-asama99.ps
An infrastructure for agents that can be used to build distributed applications.

[13] Nelson Minar, **Designing an Ecology of Distributed Agents**, Master's Thesis, MIT Media Lab, May 1999
Foundations for Hive.

[14] **CoolTown project**, HP Labs, Palo Alto, USA
http://www.cooltown.hp.com/

Infrastructure that aims at giving each person, place and thing a web representation.

[15] Deborah Cashwell, Philippe Debaty, **Creating Web Representations for Places**
http://www.cooltown.hp.com/papers/placeman/placesHUC2000.pdf
This work is part of to the CoolTown project.

[16] Tim Kindberg, et al., **People, Places, Things: Web Presence for the Real World**, HP Laboratories Technical Report, HPL-2000-16
This work is part of to the CoolTown project.

[17] **Portolano**
http://www.cs.washington.edu/research/portolano/
Visions and concepts about ubiquitous and invisible computing.

[18] M. Esler, J. Hightower, T. Anderson, and G. Borriello, **Next Century Challenges: Data-Centric Networking for Invisible Computing: The Portolano Project**, University of Washington, Mobicom 1999
http://portolano.cs.washington.edu/papers/mobicom99/mobicom.ps
A vision that is part of the Portolano project.

[19] H.W. Peter Beadle, B. Harper, G. Q. Maguire Jr., J. Judge, **Location Aware Mobile Computing**
Proc. IEEE/IEE International Conference on Telecommunications, (ICT'97), Melbourne, April, 1997
http://www.it.kth.se/~maguire/LocationAware/ICT97/ict97.htm

[20] M. Weiser, R. Gold, J.S. Brown, **The origins of ubiquitous computing research at PARC in the late 1980s**
IBM Systems Journal, Vol 38, No. 4 - Pervasive Computing
http://www.research.ibm.com/journal/sj/384/weiser.pdf

[21] C. Schmandt, N. Marmasse, S. Marti, N. Sawhney, and S. Wheeler, **Everywhere messaging**
IBM Systems Journal, Vol. 39, Nos. 3 & 4 - MIT Media Laboratory
http://www.research.ibm.com/journal/sj/393/part1/schmandt.pdf

[22] T. Selker, W. Burleson, **Context-Aware Design and Interaction in computer systems**
IBM Systems Journal, Vol. 39, Nos. 3 & 4 - MIT Media Laboratory
http://www.research.ibm.com/journal/sj/393/part3/selker.pdf

[23] H. Lieberman and T. Selker, **Out of context: Computer systems that adapt to, and learn from, context**
IBM Systems Journal, Vol. 39, Nos. 3 & 4 - MIT Media Laboratory

http://www.research.ibm.com/journal/sj/393/part1/lieberman.html
http://www.research.ibm.com/journal/sj/393/part1/lieberman.pdf

[24] George Liu and Gerald Q. Maguire Jr., **A Virtual Distributed System Architecture for Supporting Global-distributed Mobile Computing**
TRITA-IT R 95:01, Dec. 94, source: [Citeseer]

[25] George Y. Liu et al., **A Mobile-Floating Agent Scheme for Wireless Distributed Computing (1995)**, source: [Citeseer]

[26] **Eine Ereignis-Architektur für kontextsensitive Anwendungen in verteilten Systemen**, Diploma Thesis of Jürg Senn, Distributed Systems Group of Prof. F. Mattern, ETH Zurich, March 2001
The thesis (written in German) describes among other things the implementation of a compact specification language that is used to define composite events that are triggered upon logical and temporal conditions.

[27] Marc Langheinrich, Friedemann Mattern, Kay Römer, Harald Vogt, **First Steps Towards an Event-Based Infrastructure for Smart Things**, Distributed Systems Group, Departement of Computer Science, Swiss Federal Institute of Technology, ETH Zurich, 8092 Zurich, Switzerland
http://www.inf.ethz.ch/vs/publ/papers/firststeps.ps
This paper discusses why there is a need for an event-based infrastructure and presents a nice case study of an program called 'RFID Chef' that suggests meals upon sightings of ingredients that it recognizes by their RFID tags.

[28] **Smart-Its**, European research project
http://www.inf.ethz.ch/vs/research/proj/smartits.html
Joint project of the Distributed Systems Group, Perceptual Computing and Computer Vision Group at ETH, TecO (Germany), PLAY (Sweden), and VTT (Finland)
The Smart-Its project is one of 16 projects conducted under the European Union's Disappearing Computer initiative and aims at making everyday things smart. The project has been started in January 2001 and will continue till Juli 2003.

[29] **Smart Dust**
Robotics and Intelligent Machines Laboratory, Department of Electrical Engineering and Computer Sciences, Uni of California at Berkeley
http://robotics.eecs.berkeley.edu/ pister/SmartDust/
The long-term project Smart Dust was started in June 1998. It's aim is to enable autonomous sensing and communication in a cubic millimeter.

A.4 Java^{TM}

[30] David Flanagan, **The Java^{TM} Programming Language**, 2000, 3rd ed., ISBN 0-201-70433-1, Addison-Wesley

[31] David Flanagan, **Java^{TM} Foundation Classes in a Nutshell**, Nov. 1999, 3rd ed., ISBN 9-781565-924871, O'Reilly & Associates
http://www.oreilly.com/catalog/javanut3/
Guide and reference book for JFC

[32] David Flanagan, **Java^{TM} in a Nutshell**, Sept. 1999, 1st ed., ISBN 9-781565-924888, O'Reilly & Associates
http://www.oreilly.com/catalog/jfcnut/
Guide and reference book for Java

[33] Mark Bull, Scott Telford, **Programming Models for Parallel Java Applications**
http://www.ukhec.ac.uk/publications/reports/paralleljava.pdf
file: paralleljava.pdf
Performance comparision of RMI and other inter-process communications paradigms

[34] **Getting Started using RMI**
http://java.sun.com/j2se/1.3/docs/guide/rmi/getstart.doc.html
Comprehensive introduction to RMI.

A.5 Jini^{TM}

[35] Scott Oaks, Henry Wong, **Jini^{TM} in a Nutshell**, March 2000, 1st ed., ISBN 9-781565-927599, O'Reilly & Associates
http://www.oreilly.com/catalog/jininut/
Guide and reference book for Jini

[36] W. Keith Edwards, Brian Murph, **Core Jini^{TM}**, December 2000, 2nd edition, ISBN: 0-130-89408-7, Prentice Hall
Guide for Jini^{TM} with many code examples

A.6 CORBA®

[37] **CORBA®**
http://www.corba.org/
A comprehensive middleware for distributed systems.

[38] Michi Henning, Steve Vinoski, **Advanced CORBA® Programming with C++**, ISBN 9-780201379273, Addison-Wesley

A.7 Internet

[39] **The World Wide Web Consortium**
http://www.w3.org/
Publications of many standards and drafts related the the Internet and World Wide Web.

[40] **The Apache Software Foundation**
http://www.apache.org/
Publications of many standards and drafts related the the Internet and World Wide Web.

[41] **Transport Layer Security (TLS)**
http://www.ietf.org/html.charters/tls-charter.html
A channel encryption scheme for Internet traffic.

A.8 XML

[42] **Resource Description Framework (RDF) - Model and Syntax Specification**
http://www.w3.org/TR/1999/PR-rdf-syntax-19990105
RDF provides an extensible model and a syntax to describe properties of a resource.

A.9 Online Archives

[Citeseer] ResearchIndex - The NECI Scientific Literature Digital Library, **http://citeseer.nj.nec.com/cs**
Huge free archive of research papers.

[ACM] ACM Digital Library, **http://www.acm.org/dl**
Archive of ACM research papers.

[Hypatia] Hypatia Electronic Library, **http://hypatia.dcs.qmw.ac.uk/**
Research papers in Pure Mathematics and Computer Science.

B List of Figures

List of Figures

C List of Tables

List of Tables