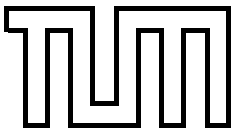
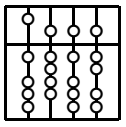


Technische Universität München
Fakultät für Informatik

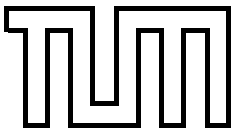


Diplomarbeit

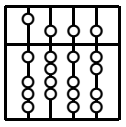
Using Ad Hoc Services for Mobile Augmented Reality Systems

DWARF – Distributed Wearable Augmented Reality Framework

Asa MacWilliams



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Using Ad Hoc Services for Mobile Augmented Reality Systems

DWARF – Distributed Wearable Augmented Reality Framework

Asa MacWilliams

Aufgabensteller:	Prof. Dr. Bernd Brügge
Betreuer:	Dipl.-Inform. Thomas Reicher
Abgabedatum:	15. Februar 2001

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt
und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Februar 2001

Asa MacWilliams

Abstract

Augmented Reality (AR) is a new form of interacting with computers that lets us see otherwise invisible information associated with things in the real world. This can be accomplished using a head-mounted display that projects virtual objects into the user's field of view, making them appear to be part of the real world and letting him interact with them.

In the DWARF project at the Chair for Applied Software Engineering of the Technische Universität München, we have designed a Distributed Wearable Augmented Reality Framework. DWARF is a software framework for building wearable AR systems that lets the user take advantage of services in intelligent environments.

DWARF is designed as a collection of software services that can run on separate hardware components connected by wired or wireless networks. These components can be distributed on the body, e.g. on a belt, or as external devices in intelligent environments. The services discover each other and dynamically cooperate to form a complete system.

This thesis presents a new kind of intelligent *middleware*, software that locates matching services in DWARF and connects them together, letting them *self-assemble* into an ad hoc system.

In designing the middleware, there were three basic challenges. The first was to ensure that the communication between services, once it is set up, it fast enough for AR applications, yet to keep the choice of communication partners flexible. This was accomplished using state-of-the-art communication and service location technology.

The second challenge was to distribute the middleware onto the individual hardware components, so that there is no single central component in the network that could fail. This was accomplished by carefully dividing the middleware into *distributed mediating agents*.

The third challenge involved arranging for communication between services that do not know each other, allowing them to be developed independently. This was solved using an abstract description of services, including types of other services they depend on and quality-of-service information.

A first protoypical implementation of the DWARF middleware covers most of the basic functionality. It has been successfully tested in the first demonstration system built with DWARF, a wearable indoor and outdoor navigation system allowing access to external devices such as printers.

This thesis describes the results of two areas of development: our Augmented Reality framework, DWARF, and the middleware that lets the systems built with DWARF self-assemble.

Zusammenfassung

Augmented Reality (AR, zu deutsch Erweiterte Realität) ist eine neue Art, mit Rechnern umzugehen, die uns ermöglicht, ansonsten unsichtbare Informationen über Dinge der physischen Welt zu sehen. Dies kann durch eine durchsichtige Datenbrille erreicht werden, die virtuelle Objekte ins Gesichtsfeld des Benutzers projiziert und als Teile der wirklichen Welt erscheinen läßt, mit denen man interagieren kann.

Im Projekt DWARF (Distributed Wearable Augmented Reality Framework) des Lehrstuhls für Angewandte Softwaretechnik an der Technischen Universität München haben wir ein Softwaregerüst für tragbare AR-Systeme entwickelt, das den Benutzer Dienste in intelligenten Umgebungen nutzen läßt.

DWARF besteht aus einer Sammlung von Software-Diensten, die auf verschiedenen drahtlos oder fest verbundenen Hardwarekomponenten laufen. Diese Komponenten können am Körper, z.B. an einem Gürtel, verteilt werden, oder als externe Rechner in intelligenten Umgebungen bereitstehen. Die Dienste erkennen sich gegenseitig und kooperieren dynamisch, um ein ganzes System zu bilden.

Diese Diplomarbeit stellt eine neue Art intelligenter *Middleware* vor, eine Software, die passende Dienste in DWARF findet und sie miteinander verbindet, so daß sie sich zu einem Ad-Hoc-System zusammenfügen.

Bei der Entwicklung der Middleware gab es drei grundsätzliche Herausforderungen. Die erste war, sicherzustellen, daß die Kommunikation zwischen den Diensten schnell genug für AR-Anwendungen ist, aber die Wahl der Kommunikationspartner flexibel bleibt. Dies wurde durch den Einsatz moderner Kommunikations- und Service-Location-Technologien erreicht.

Die zweite Herausforderung war es, die Middleware auf verschiedene Hardwarekomponenten zu verteilen, so daß es keine einzelne anfällige zentrale Komponente im Netz gab. Dies wurde durch ein sorgfältiges Aufteilen der Middleware in *distributed mediating agents* erreicht.

Die dritte Herausforderung bestand darin, Kommunikation zwischen Diensten zu ermöglichen, die sich nicht kennen, so daß diese getrennt voneinander entwickelt werden konnten. Dieses Problem wurde durch abstrakte Dienstbeschreibungen gelöst, die Abhängigkeiten von anderen Diensttypen und Dienstgütekriterien beinhalten.

Eine erste prototypische Implementierung der Middleware für DWARF beinhaltet die meisten wesentlichen Funktionalitäten. Sie ist in unserem ersten DWARF-Demonstrationssystem erfolgreich getestet worden, einem tragbaren Navigationssystem für den Einsatz im Gebäude und im Freien, das auch das Nutzen externer Dienste wie Drucker ermöglicht.

Diese Diplomarbeit beschreibt die Ergebnisse zweier Entwicklungsgebiete: unseres AR-Frameworks, DWARF, und der Middleware, die den Systemen, die mit DWARF gebaut werden, ermöglicht, sich spontan zusammenzufügen.

Preface

Purpose of This Document This thesis was written as a Diplomarbeit, akin to a Master's thesis, at the Technische Universität München's Chair for Applied Software Engineering. From June through December 2000, six other Computer Science Master's students and I designed, developed and tested the first version of DWARF, a Distributed Wearable Augmented Reality Framework. In this thesis, I would like to explain the ideas behind my work, document its results, and show its future implications.

Target Audience This thesis addresses several different audiences. Here, I would like to give you, the reader, an overview.

Augmented Reality researchers and other computer scientists should read Chapter 2 for the philosophy behind our framework, DWARF, and to see what we're up to in Munich. Chapters 3 and 5 deal with middleware issues that extend beyond Augmented Reality, and chapter 7 shows what our framework can do.

Future developers should read Chapter 2 to understand the framework's architecture and Chapters 3, 4, 5 and 6 to see how to use and extend the middleware. Chapter 7 shows how to build systems with DWARF, and chapter 8 points to interesting future work.

General readers unfamiliar with Augmented Reality should read Chapter 1, which briefly shows why this is an exciting field and reviews the concepts I worked with in this thesis. Chapter 2 is understandable and interesting for non-computer scientists, explaining why we developed DWARF. Chapter 7 shows what our first demonstration system can do.

Even if some of this thesis may be unintelligible, I hope the rest is worth the read!

The DWARF team and management which is already familiar with the concepts behind our framework in chapter 2, should read Chapters 3 and 5, in which I describe the design of the middleware. Chapter 7 shows how DWARF can be used to build different types of systems, and Chapter 8 takes a look at the future possibilities of DWARF.

Reading in Pictures If you have little time, but would still like to get an idea of DWARF and its middleware, I recommend reading Chapter 1 and then glancing through the Figures in Chapters 2, 3, 5, and 7. In these, I refine what starts as a vague concept into a full system design, using various diagrams in the Universal Modeling Language (UML).

Timeline The structure of this thesis parallels the timeline of my involvement in the DWARF project.

Chapter 1 gives a brief introduction to the field of Augmented Reality and is based on discussions we had in a summer school in 1999.

Chapter 2 discusses different architectures for mobile AR systems, and explains why we decided, in June 2000, to build DWARF as a framework of distributed services.

In Chapters 3, 4, 5 and 6, I describe the main focus of my work: building the middleware necessary for such a framework. This encompasses the time span from July to November 2000.

Chapter 7 comes from the final stages in before we built our demonstration system. It describes how we used the various DWARF services and the "glue" of the middleware to assemble a working prototype of an Augmented Reality outdoor and indoor navigation system. I also describe how another possible scenario could be implemented.

Chapter 8 concludes the thesis with an evaluation of the project's results and suggestions for future work in the area.

Electronic Version If you are a developer wishing to use the DWARF middleware and you are currently reading a paper version of this thesis, make sure to get hold of the electronic version, in Portable Document Format. This provides cross-references for class and interface names which are helpful when looking up a particular feature.

Acknowledgments This thesis would never have been possible without the tireless effort of many people.

First, I would like to thank the rest of the DWARF team—Martin Bauer, Florian Michahelles, Christian Sandor, Stefan Riß, Martin Wagner and Bernhard Zaun, and our advisors, Thomas Reicher and Christoph Vilsmeier. None of us knew, when the project started, just how much work it would be, nor that it would be so successful.

Second, I would like to thank Bernd Brügge and Gudrun Klinker, who, through their cooperation across different fields, have opened a new area of research, Augmented Reality in intelligent environments. Thanks especially to Bernd Brügge for his help in finding concise names for new ideas.

Special thanks goes to Martin and Martin for their help in describing the DWARF architecture, to Christian for last-minute proofreading, and to Thomas for many architectural discussions and his knack of finding the right literature.

Finally, I would like to thank my family, who worked their way through pages and pages of computer science terminology, and above all my wife Sabine, who provided unlimited support in everything.

The Author I would very much like to hear any comments or suggestions you may have regarding the work I did for this thesis or the DWARF project as a whole. This especially includes any questions you may have if you wish to use DWARF. Please do not hesitate to contact me at *Asa@MacWilliams.de*.

Overview

Contents

List of Figures

1 Introduction

Introduction to Augmented Reality, frameworks, ad hoc services, middleware, and the DWARF project. Goals of my thesis.

In this chapter, I would like to introduce the concepts this thesis is based on to readers who may not yet be familiar with them.

In my thesis, I developed the *middleware* for *DWARF*, a *framework* for *Augmented Reality* systems, built from *wearable computers* that can use *ad hoc services* in *intelligent environments*. Each of these concepts is explained in the following sections.

I also briefly explain the goals I set out with for the DWARF project and when writing this thesis.

1.1 An Introduction to Augmented Reality

Bringing Things and Information Together The term Augmented Reality (AR) stands for a new form of interaction between people and computers. It is based upon the assumption that most useful things that we would like to do deal with real people and things and with information, *in that order*.

We have our own mental representation of things, which we access naturally when physically dealing with them. But there are many other forms of information associated with things, such as other people's knowledge of them, their history, or—if the "things" are other people—their own knowledge. For many kinds of human endeavor, accessing that "external" information is crucial. For example, in medicine, we need to know the patient's history; in machine maintenance, the machine's history; in architecture, the plans of a future building; and in diplomacy, a diplomat's political attitude.

We have tried to bring information and things closer together by using various tools: a maintenance manual, a patient's chart or a a diplomat's aide.

Traditionally, dealing with things and with their associated information have been separate processes. For example, fixing a machine might involve reading the (paper) maintenance manual, physically adjusting screws and writing in a (paper) maintenance log.

The introduction of computers into workplaces such as factory floors or hospital wards, in an attempt to streamline the processing of information, has often had a negative side effect: information has moved farther away from the things (or people) it belongs to. Entering patient information into a database system down the hall is harder than making a mark on a chart.

As new technologies become available, this problem has spawned a variety of new ideas in computer science, among which are AR and Ubiquitous Computing.

These ideas take different approaches towards bringing information and things together.

Figure 1.1: A wearable Augmented Reality system.
The position of the user's head is calculated by analyzing the image from the video camera on the helmet, and three-dimensional images are projected on his see-through display. Note that the term *wearable* is still relative.

Ubiquitous Computing aims to make things smart by adding microchips to them. For instance, a doorknob could recognize your fingerprints and let you enter a room, while tuning the radio to your favorite station.

AR follows a different approach. The idea is to extend the capabilities of our own senses. We can easily gain information as to a thing's color, weight, smell, sound or flavor by looking, touching, smelling, listening or tasting. We cannot—yet—easily see the shape of a planned building simply by looking at the empty building site. AR aims to fix this.

Computer systems can easily access information, e.g. a model of a future building, using *their* "senses", networks. So, all we have to do is to give computers a way to provide the information to our own senses, *when and where we need it.* This means not having to look at a blueprint, but seeing an image of the future building in front of us, projected onto our field of vision—for example—by optics in a pair of regular glasses.

The trick is to make this work in such a way that it actually helps, rather than encumbers us.

Making it Work The term AR was coined to set it off from Virtual Reality (VR), a technology where computer-generated three-dimensional models are shown on data goggles, immersing the user in a virtual world. In a sense, VR only lets the user deal with information, since he is prevented from seeing the real world around him.

The first attempts at AR used modified VR goggles that the wearer could use to see both the real world and the virtual world simultaneously. Using various forms of tracking devices to figure out the user's head position, the image in the goggles could be adjusted to match objects in the real world. This way, repair instructions could appear right on the parts to be repaired. Today, see-through head-mounted displays are available commercially for the

purpose of AR. Other forms of augmentation, using sound, touch or smell, are still in the research stage.

Making AR systems useful involves solving many difficult problems: establishing the user's position accurately and quickly, understanding what information the user needs when, and displaying it in high quality. To make matters worse, all of the necessary equipment must be small and lightweight enough so the user can carry it around with him, without it interfering with his work in the physical world.

Several such AR systems have been built as prototypes, but actual use of them for real work is still limited [17]. An example of an AR system, which we built as a demonstration during the DWARF project, is shown in Figure 1.1 on the preceding page. The necessary technology is here today, however, and commercially viable AR systems are expected to be "just around the corner."[46] For an overview of current AR technology, see [2], [27] and [26].

1.2 Mobile and Wearable Computers

One challenge in building AR systems is making them comfortable for the user. Virtual Reality goggles are large and cumbersome, and for AR, you often need even larger headsets with built-in video cameras so that the computer can see what the user is seeing. These are heavy, awkward, not very attractive, and have wires attached all over the place.

Even worse, AR requires a lot of computing power to figure out where the user is looking and to draw virtual objects convincingly. This has to go somewhere, and many systems require the user to carry around a backpack full of computer equipment.

One main area of AR research thus involves using *wearable computers*, small devices that can be attached to a belt or put in vest pockets, much like cellular phones or hand-held organizers. Since AR requires vastly more computing power than transmitting a telephone conversation, this is no easy task.

1.3 Intelligent Environments

Where can you use Augmented Reality systems? Many potential environments have been investigated, ranging from wearable mobile systems for outdoor use [1] to systems that are permanently installed onto a desk [60].

A promising field for Augmented Reality systems lies within the context of *intelligent environments* and *cooperative buildings* [8]. These are rooms, buildings, or even entire campuses that intelligently adapt themselves to their users' needs. They adjust the environmental conditions such as lighting and heating, and provide services such as intelligent meeting or video-conferencing rooms.

By combining wearable AR systems with extra services such as external cameras and projection screens, systems can be built that provide a convincing and useful AR experience: mobile systems can display virtual objects that need to be positioned precisely in three dimensions, and intelligent rooms can display information on the walls, provide extra computing power so that the mobile systems can stay small and lightweight, or use external cameras to track the user's position. [35]

1.4 Ad Hoc Services

If we envision a future in which many people carry small wearable AR systems around with them, much as cellular phones today, a new challenge emerges. As they enter new cooperative buildings and intelligent environments, people will want to be able to use the services there quickly and easily, without having to reconfigure their computer systems. In fact, they will not even want to think of their AR systems as computers, but as appliances that simply do what is expected of them. This is a very difficult problem. For example, what ought to be the simple task of printing a document from a notebook computer on a printer when visiting somebody else's office is usually quite complicated. There are technologies emerging to facilitate the use of *ad hoc services*, but many of them only address simple fields, such as home networks. For future AR applications, more powerful systems will be necessary.

1.5 Applications and Frameworks

Most AR systems that have been built up to now are single-purpose systems. This means they are geared towards solving one specific problem, with no intention of solving others.

This approach is fine as long as one's goal is to solve specific difficult technical problems, such as estimating a user's position in three dimensions or drawing convincing three-dimensional images of virtual objects. From a software engineer's standpoint, this is unfortunate, since it makes developing different AR systems and applications much harder.

One possible solution to the problem of quickly building many different, but related, applications, is the use of *frameworks*. A framework is a set of components providing a general solution that can be refined to provide a specific application [7]. For AR, such a framework can provide components for position tracking, modeling three-dimensional objects, navigating through tasks for the user to perform, recognizing spoken commands, and so on. These components can then be assembled with a small amount of application logic to build a specific AR application in much less time than it would take to build a system from scratch.

1.6 The DWARF Project

In the summer of 1999, the Technische Universität München and the Friedrichs-Alexander-Universität Erlangen held a summer school on the topic of AR. There, a group of the participants decided to build a new framework for AR systems.

We called it DWARF, for Distributed Wearable Augmented Reality Framework.

The main idea behind DWARF was to consolidate research results in the field of AR and combine them with modern software engineering methods to build a framework that can be used to develop many different AR applications.

This way, as the hardware matures so that powerful wearable systems can actually be built, the software infrastructure necessary to write useful applications will already be in place. As the systems become acceptable to users, ideas for new applications will emerge, and with the right software framework, they can be implemented quickly.

One of the goals in designing DWARF was to consistently use ad hoc services, so that the system is essentially self-assembling. This makes it possible to distribute different computing tasks onto different hardware devices which the user can then simply plug together to make a working system. It also means that separate mobile systems and intelligent environments

using the same framework will be able to cooperate easily—indeed, they will spontaneously self-assemble into a larger system.

The DWARF project started in earnest in the spring of 2000, and has meanwhile resulted in six Master's Theses, a first prototypical implementation of the framework, and one model system that has been built and tested.

1.7 Middleware

In developing DWARF, we designed several different components—for tracking the user's position, displaying objects, navigating through a complex flow of tasks, etc. One of the core requirements of DWARF is the ability to spontaneously self-assemble out of these distributed components. Thus, we needed something to go "in between" the functional components, letting them find each other and communicate with one another.

This kind of task, arranging the communication between components, is generally handled by *middleware*. Middleware conceptually sits "in the middle", and the other components use it to communicate. There are many kinds of middleware for distributed systems on the market today. Most of them are designed for building large-scale computer systems out of many networked computers, e.g. integrating web servers with databases to make on-line shopping services.

For DWARF, we needed a new kind of middleware, one that would support both spontaneous self-assembly and fast and powerful communication mechanisms. This is exactly what I developed during the project.

1.8 Goals

My personal involvement in the DWARF project covered three main areas: architectural ideas for building AR systems with ad hoc services, designing the necessary middleware so the various DWARF components could cooperate, and assembling the components into a working prototype which we demonstrated in December 2000. In this thesis, I describe how I addressed these goals.

A major consideration was the balance between design and implementation. I developed a design for the DWARF middleware which can last through several iterations of extension, porting and even re-implementation, and at the same time provided a stable basic implementation of the features we urgently needed. This way, enough middleware is available right now to continue work on various components of DWARF and to add new components.

For major extensions of DWARF, however, additional parts of the middleware will have to be implemented. I have already designed some of these parts, and I present them in this thesis as well.

2 Software Architectures for Mobile Augmented Reality Systems

Mobile Augmented Reality systems can be modeled in different ways. We believe that a framework of self-assembling distributed services is the best, and have designed such a framework, DWARF.

This chapter addresses a general audience, and should basically be understandable to everyone. First, I discuss ways of building AR systems from three different points of view: the user, the software architect, and the project manager.

Then, I present the DWARF architecture, which combines the most powerful architectural components from all three standpoints: it is a framework of self-assembling distributed Services that supports AR in intelligent environments.

2.1 Augmented Reality Systems From the User's Point of View

There are different philosophies as to how AR systems can be modeled. These models basically differ in the user's view of the system, and how information and things are brought together.

2.1.1 Task-Centered Augmented Reality

The simplest model of an AR application is closely focused on a task the user wants to perform. Examples are maintenance and repair of machinery, or navigation in unknown territory. For these tasks, we often already have paper-based instruction manuals or directions, or even computer-based expert systems. If these guides are formalized into *Taskflows* [57], an AR system can use the Taskflows to guide the user through the task. The user's input possibilities are often limited to simple commands like "next step", "go back" or "part 3 is broken".

This kind of AR application is probably the most commercially promising one for the next few years [46]. It is comparatively easy to build, since the environment is, by definition, known before the system is deployed, and the system can be optimized for a specific task. For maintenance tasks, the system often does not even have to be completely wearable, but can rely on infrastructure in the factory hall. Many research teams have focused on this type of application, and for the demonstration of the DWARF framework (Section 7.1), we chose a scenario that included this kind of navigation along a predetermined path, as well.

Of course, with this kind of AR, the user's flexibility is quite limited, as he can only perform the task programmed into the application.

2.1.2 Ubiquitous Computing

Sometimes considered the opposite of AR, Ubiquitous Computing puts many different kinds of computers into the world around us. These computers offer various services, such as printing, movie viewing, or ordering a pizza. There is no predetermined Taskflow—the user moves around and uses whatever computers he wants to.

Reality can be "augmented" with Ubiquitous Computing by making things themselves smart. Rather than having a virtual map projected onto the wall by a head-mounted display, the wall could have a built-in rear-projection display.

This kind of environment is useful for fairly unstructured work, such as business trips, where the user has a clear idea of how to perform his task, and only wants specific help at specific times. Like task-centered AR, it does not have the same kind of requirements towards mobility, as small computing devices can be permanently installed into rooms and buildings. Disadvantages are that it is hard to display different information to different users at the same time, and that the real-world objects themselves have to be changed in order to bring information to them.

2.1.3 AR-Ready Intelligent Environments

Somewhere between task-centered AR and Ubiquitous Computing is the idea of AR-ready intelligent environments. Here, services in the environment interact seamlessly with a mobile AR system the user is carrying or wearing [35]. For example, video cameras installed in the corners of an "intelligent room" could accurately track the pose of many users' heads simultaneously by using workstations in the room. This position information would then be used by the users' head-mounted displays to accurately align the virtual scenes they are showing in three dimensions.

Such a scenario allows for the same kind of powerful AR as task-centered systems, but adds flexibility. As the user moves in and out of different rooms, he can take advantage of different services provided there, following the Ubiquitous Computing metaphor. This can either happen automatically, such as for tracking systems, or manually, such as when the user wants to show a presentation on a particular video beamer. It also allows mobility (since the user has a small AR system with him), but takes advantage of additional resources when they are available.

Obviously, this model is the most challenging to implement. Both mobility and the spontaneous use of services need to be addressed. However, systems using such intelligent AR-enabled environments are very powerful, as they can accommodate predetermined Taskflows as well as the user's spontaneous desires.

2.2 Development Processes

Developing one AR system is challenging, whichever type of system you choose. Developing many different AR systems is even harder. In this section, I will briefly describe different types of development processes that make it easier to develop single or multiple AR systems.

2.2.1 Single Systems

The most obvious way of tackling the design of an AR system is to proceed from the problem statement through the classical phases of requirements analysis, system design, object design

to the implementation, and develop one single system that satisfies the requirements as best as possible.

Advantages This method is the fastest for any given application—you begin at the beginning, continue until you reach the end, and stop. The resulting system is designed to solve exactly the problem that you started out with.

Disadvantages The obvious disadvantage of such a development process is that when you want to develop a second AR system, you have to start over again. Reusing code from the first system is not easy, as it is tailored to one specific application. All you can generally reuse is the developers' experience.

2.2.2 Systems with Component Reuse

One solution to this problem is to build AR systems out of components that can be reused in other systems. For example, an optical tracking component could be used for an AR videoconferencing system or for maintenance work.

In this development process, the system is divided into components or subsystems with as general functionality as possible, in the hope that they can be reused later on. This is a method that is often used in building software systems today.

Advantages This approach allows new systems to leverage the effort put into previous systems. From a business standpoint, it works towards economies of scale: the more systems you build, the more components you have, and the faster you can build new systems.

Disadvantages Even with component reuse, building a new AR system can be quite difficult. The components of several older systems might not fit together well, and many new components will still be needed for any given new application.

Also, you never know, while developing a component, whether it actually will ever be reused, so the extra effort of modularizing it may be wasted.

2.2.3 Frameworks

The last approach I would like to describe is to develop a framework for AR systems. Here, you try to design an architecture that can accommodate all sorts of AR systems, build components that fit into this architecture, and then build applications quickly by "filling in the blanks". A framework consists not only of the bare components, but also of a basic pattern in which the components fit together, and communication infrastructure allowing them to communicate.

Frameworks are often used for user interfaces, and are also becoming available for enterprise applications.

Advantages This approach allows new systems to be built very quickly. Given a specific problem to be solved with AR and a good framework, developing the system itself is almost easy. Again, you can use the economies of scale: once effort has been invested in the framework, developing individual systems becomes cheap.

Disadvantages Developing a framework produces no immediate result, as it does not actually solve a given problem. Thus, the initial design of a framework can be unprofitable from a business standpoint. Also, the design is very difficult, since it is impossible to anticipate all requirements of all future AR systems.

Frameworks and Applications A solution to this disadvantage is to pursue parallel tracks, developing both the framework and applications simultaneously (with two different teams) or alternatingly. Thus, an initial version of the framework would be used to implement the first two (hopefully very different) systems, and the experiences learned can be recycled into the next version of the framework. [58]

This process allows you to keep the framework's development "in sync" with reality, by solving real problems, and still allows for the growth of a framework that can solve AR in a general fashion.

2.3 Hardware and Software Architectures

AR applications have quite stringent requirements of the systems that they run on. To complicate the matter, many of these requirements are contradictory—for example, a high-end graphics workstation would provide wonderful image quality, but would be far to heavy for a wearable system [36]. This fact has prompted many AR research teams to focus only on specific design goals, sacrificing others for the time being. This is analyzed in some detail in [42].

In this section, I will describe several possible hardware and software architectures that tackle this balancing act between the various design goals (mobility, range, performance, and so on) in different fashions. These vary widely in ease of implementation and in the power and flexibility of the resulting systems.

2.3.1 Monolithic Systems

The simplest type of AR system runs on a single computer. This computer has many different peripheral devices attached, in order to track the user's position, display output, and so on. See Figure 2.1 on the next page.

Often, this type of system also has only a single program running on the computer [36]. For example, the program could repeat the following 30 times a second: read an image frame from the video camera, calculate the user's position, update the model to display, and render the three-dimensional model onto the head-mounted display.

Examples Almost all early AR systems were monolithic, and many still are. For examples, see [36] or [55].

Advantages This type of system is quite efficient at displaying a fixed model or an animation, registered in three dimensions.

The monolithic architecture is easy to design—there is no need to deal with problems of network communication, or coordinating and synchronizing concurrent programs. It also can be very fast, since there is no lag in communication between different computers.

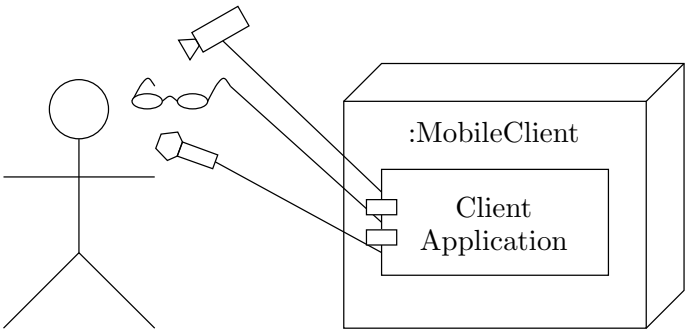


Figure 2.1: A monolithic Augmented Reality system.
The mobile AR system has a video camera for determining the user's position, a head-mounted display, and a microphone for voice input attached.

Disadvantages There are some disadvantages, however. Since all the functionality is contained in a single computer, this can become large and heavy, making it hard to carry around. Also, there is a limit to the processing power that can fit in a single (portable) system, and tasks like image analysis are computation-intensive. This means that adding additional functionality such as voice recognition to such a system can be impossible because not enough processing power is available. There is another problem: some peripheral devices (such as cameras and displays) will only run when attached to a specific hardware platform and only have drivers for certain operating systems. This can make it impossible to use two devices attached to the same computer—for example, if the camera driver only runs on Windows 98 and the voice input system needs Windows NT.

Single-system architectures can be highly optimized to fit a specific task, and perform that task very well. However, adapting such a system to a different application can be difficult, especially if the software is also designed in a monolithic fashion.

2.3.2 Client-Server Systems

Client-server AR systems aim to reduce the inefficiency of single systems by moving some functionality off the mobile computer, the *client*, onto a larger stationary computer, the *server*. This type of architecture is found in the internet, where Web browsers (clients) access the Web sites' servers. See Figure 2.2 on the following page.

Examples A system for airplane maintenance has to store instructions for many different types of repairs, most of which occur very infrequently. By storing only a small number of frequently needed instructions on the mobile computer, this can do without a hard disk and can be kept small and lightweight. When an unusual repair task shows up, the client loads the instructions from the server using a wireless network.

Another application of the client-server architecture offloads processing power for image analysis, three-dimensional rendering etc. onto the server, allowing the mobile clients to have smaller and less power-hungry processors.

An interesting example of a client-server AR system is described in [54], where soldiers wear AR systems in the field that are wirelessly connected to a command and control center.

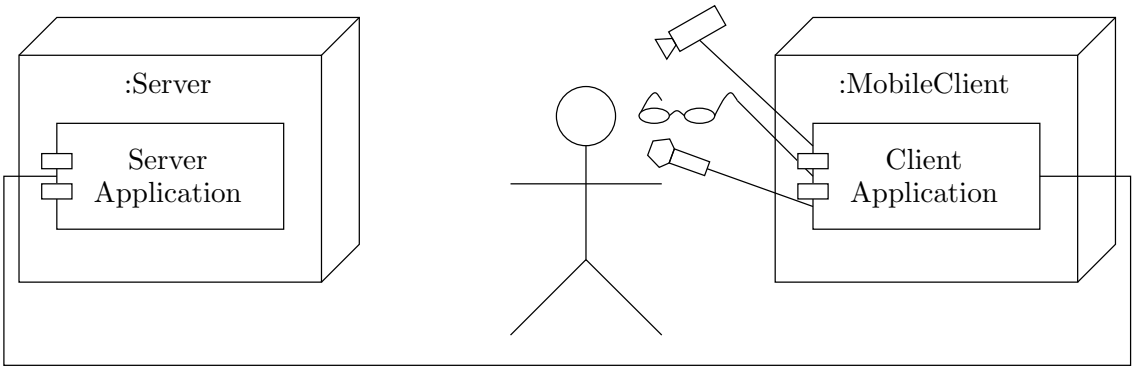


Figure 2.2: A client-server Augmented Reality system.
The mobile system uses the same devices as before, but the server is used for computation and data storage.

Advantages The client-server approach supports multiple users well: the same server can supply instruction data to many mobile users. It also allows users to collaborate in an augmented world—descriptions of virtual objects are stored on the central server, and the individual clients display them from the viewpoint of their users. This is also called a *repository architecture* [7].

Client-server AR systems have obvious advantages in factory floor or office environments. Here, it is easy to set up a server and a wireless network for communication with the clients, and client-server systems can be built for multiple users at a fraction of the cost of multiple individual systems.

Disadvantages Building a client-server system is harder than building a monolithic system, since you have to deal with network communication hardware and software. Also, there are concurrency and synchronization issues: client and server have to stay "in sync", which leads to more complex communication protocols.

Offloading processing onto the server may actually cost more power than it saves, since this requires a high-bandwidth wireless network.

In addition, the mobile clients are not much use without their server, making the range of such systems limited.

2.3.3 Cooperating Components

AR systems based on cooperating components also address problems of monolithic systems, but differently than the client-server approach. Here, the user carries all of the functionality around with him, just as with monolithic systems. Instead of having one computer that all peripheral devices are attached to, however, the system consists of several networked computers, with only a few devices attached to each. See Figure 2.3 on the next page.

Examples There are not many examples of such systems yet. One is MIThril [45], described in Section 2.4.2.

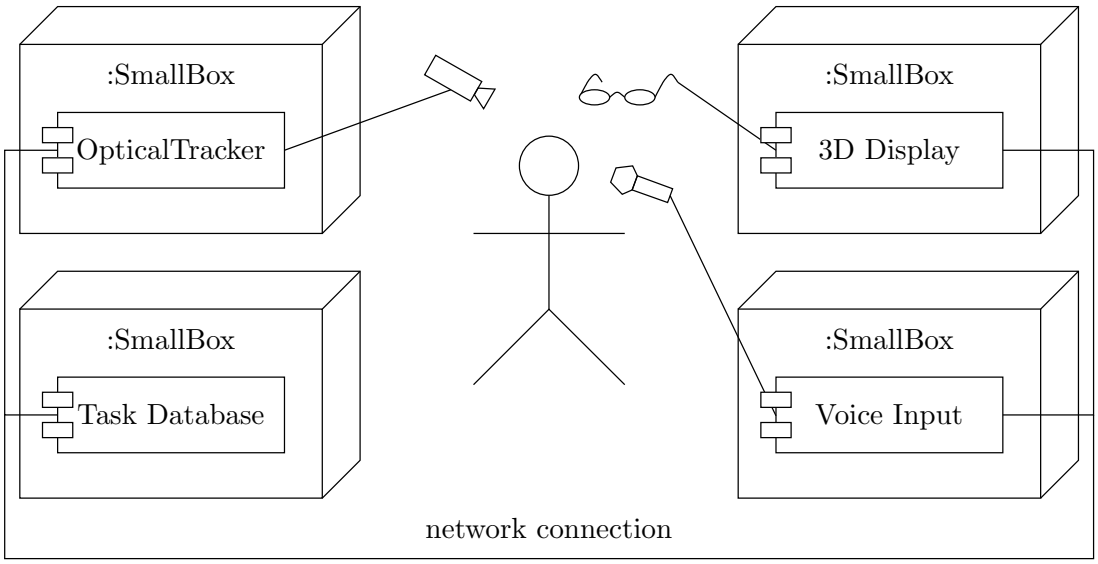


Figure 2.3: An Augmented Reality system based on cooperating components.
The user carries several small hardware components with him, each with its own devices attached. They are configured to cooperate.

Advantages Using several smaller networked computers instead of one large one can make a mobile system easier to carry. The components can be distributed evenly around the body (e.g. in vest pockets or on a belt), rather than putting one large system in a backpack. This type of system also allows different processor architectures and operating systems to be combined, taking advantages of their respective strengths. For example, an optical tracking component could use an enhanced floating-point processor and a real-time operating system for image analysis, and an airplane repair manual could be stored in a system with a large flash memory card. This way, the system can be built using just the hardware one actually needs, and there are far fewer driver incompatibility problems.

Systems built from cooperating components do not suffer from the extensibility limits that monolithic systems have. Adding a voice recognition system simply means adding another small computer with its own microphone and voice recognition software.

Disadvantages Obviously, this type of system is harder to build than a monolithic system. The components need to communicate using some sort of network. This means additional hardware: each component needs a network card, some network technologies need hubs, and unless using a wireless network, you have connecting wires, which may tangle. It also means extra software: the components must communicate using some sort of network protocol, and you have to deal with even more synchronization and concurrency issues than in a client-server system.

There is also a lot more configuration work to do: since the software obviously cannot run in a single process, the various programs need to be configured to find each other. This means that some form of identifier, e.g. host names and port numbers, needs to be assigned to each software component, and the components must know the identifiers of their communication partners.

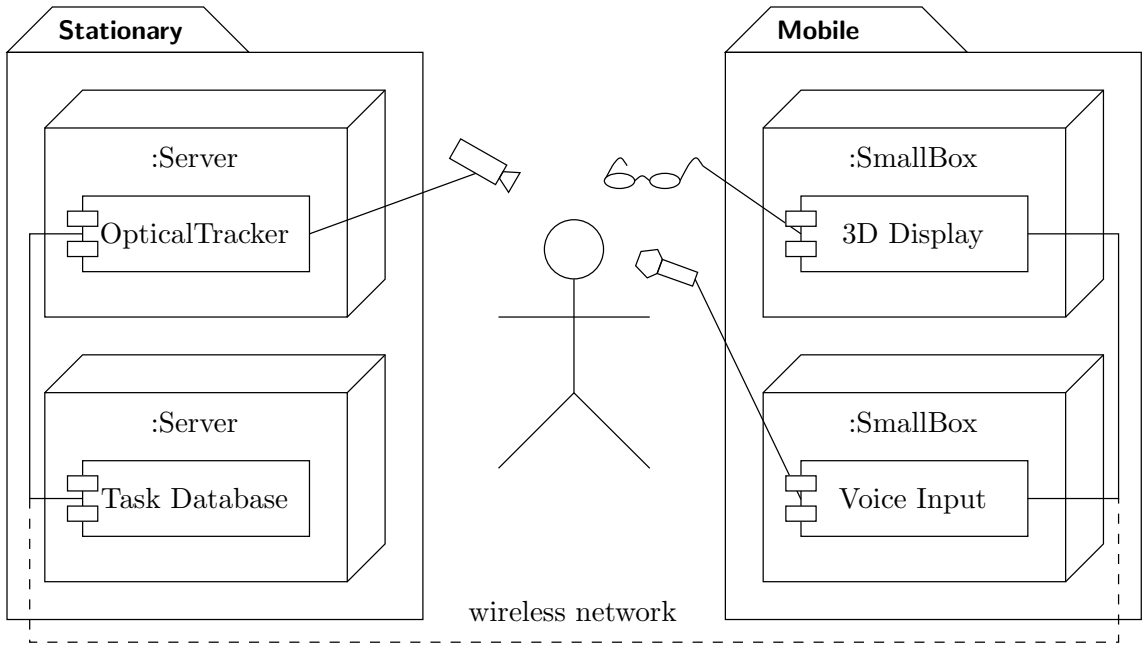


Figure 2.4: An Augmented Reality system based on self-assembling cooperating Services. The user is carrying two small wearable computers with him, and these cooperate with stationary ones in the room he is in. This figure represents one possible configuration of the system. As the user walks on, the mobile systems could disconnect from the database and connect to other tracking devices.

2.3.4 Self-Assembling Cooperating Services

With self-assembling cooperating Services, we take the idea of cooperating components one step further. As above, the mobile AR system consists of multiple computers networked together, each with their own peripheral devices. The new idea is that the components find each other dynamically, rather than being configured to know each other's locations.

To accomplish this, we use the concept of *Services*. A Service is a piece of software (and hardware) which provides certain functionality to the user or to other Services. Services might need other Services to run—for example, the head-mounted display needs position information from a tracking Service. However, it does not care which tracking Service it gets this position information from, as long as the position is accurate. The Services cooperate in a *peer-to-peer* fashion.

This model allows us to write Services that only know what *kind* of Services they depend on, rather than configuring components with the addresses of other components. Also, we can take advantage of external Services provided by servers in an intelligent environment, such as a tracking Service with a fixed camera in a room. See Figure 2.4.

So far, the only system we know of that uses such an architecture is our own DWARF system, described in Section 2.4.

Advantages Systems that assemble themselves out of cooperating Services have all the advantages that systems built from cooperating components have, as well as three new ones.

The first advantage is that it eliminates the additional configuration work required with distributed components, since the Services find each other dynamically.

Second, this allows the system to reconfigure itself dynamically, taking advantage of additional external Services, as the user moves around within intelligent environments. Thus, the system can adapt itself to environments with varying degrees of infrastructure, ranging from no infrastructure (outdoors, where it can compete with monolithic systems) to a dense infrastructure (on a factory floor, where it can compete with client-server systems).

Third, such a system has an extra measure of fault tolerance. When one Service fails, the rest of the system can fall back to another Service providing the same functionality (though perhaps with a lower quality). This is especially important in mobile applications, where network connectivity is shaky and navigation systems can fail.

Disadvantages Of course, this architecture has disadvantages, as well. First of all, it introduces an extra level of specification, since Service types and the functionalities they provide have to be formalized somehow. Second, it necessitates a powerful middleware system, which allows the Services to find and communicate with each other dynamically, without knowing each other beforehand.

Middleware The middleware in an AR system consisting of self-assembling cooperating Services has a large bill to fill. It needs to provide a mechanism for Services that do not know each other to find each other and connect themselves together.

One obvious approach here is to register all Services with a central middleware server. This checks to see which fit together and then connects them. Unfortunately, this approach makes the idea of self-assembling Services much less useful: there is now a central component that everything else relies on.

A more powerful concept requires the middleware to act like a medium in biological or chemical reactions, in which molecules float around and eventually bond together because their receptors match each other. See Figure 2.5 for such a model of Services finding each other.

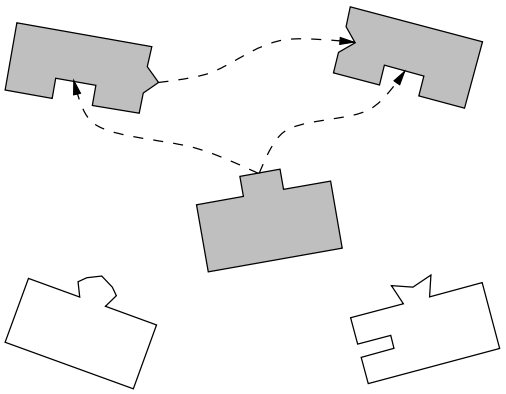


Figure 2.5: A cloud of self-assembling Services.
Note that some of the Services fit together and some do not. The middleware must find the matching Services and connect them together.

The main focus of my work in has been developing such a middleware system, which is described in detail in Chapters 3, 5 and 6.

2.4 The DWARF Architecture

In DWARF, our Distributed Wearable Augmented Reality Framework, we combine what we consider to be the most advanced aspects discussed in the previous section:

- For the user, systems built with DWARF will enable *intelligent AR-ready environments*, as described on page 7.
- For the project manager, DWARF provides a *framework* that lets new AR systems be developed quickly (page 8).
- From the system architect's standpoint, DWARF systems consist of *self-assembling co-operating Services* (page 13).

The rest of this section describes the requirements we had in designing DWARF, the overall system design and the functionality the framework provides. Extensive parts of this section were written jointly and published simultaneously with Martin Bauer and Martin Wagner in [3] and [77].

2.4.1 Requirements Analysis

This section summarizes the main requirements we established for DWARF at the beginning of the project. A more extensive description can be found in [4].

Purpose of the System

The purpose of DWARF is to provide a framework for the development of general Augmented Reality systems.

In our opinion and the context of a general framework, AR is any augmentation—not only visual—of the user's experience of the real world by means of computing devices ranging from WAP-enabled cell phones to high-end graphic workstations. These augmentations may be textual descriptions of walking directions on a mobile phone or virtual documents that appear to be attached to real objects.

Since DWARF is a framework, we only provide functionality that can be used by many AR systems, such as tracking and user interface Services.

If a user of the framework wants to write a working AR application, he has to provide the overall application functionality himself. However, this will be based on the framework's Services, which have high-level interfaces facilitating rapid development.

Functional Requirements

The framework must provide different types of functionality that are needed in building AR systems, as shown in Figure 2.6 on the following page.

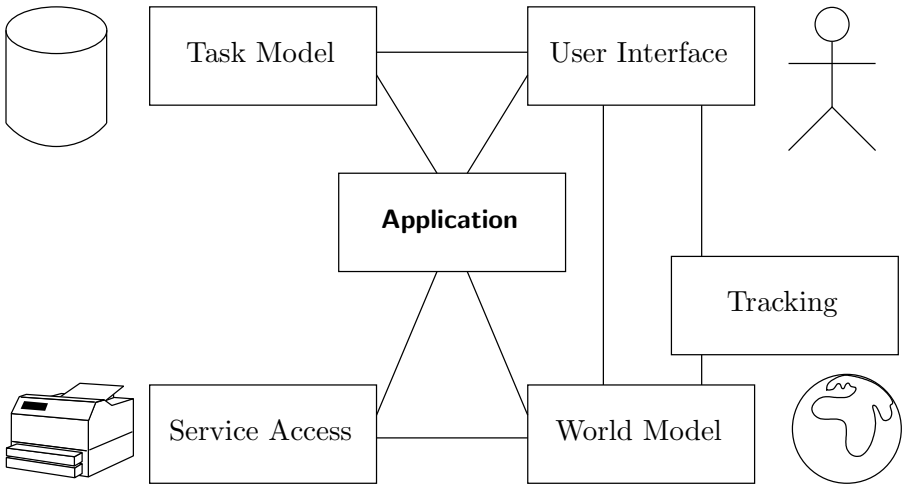


Figure 2.6: Necessary functionality in an Augmented Reality framework.
A framework for AR systems in intelligent environments must have a model of the world, be able to track the position of things in it, model tasks for the user to perform, access external services, and present this all to the user. The built-in connections between tracking, the world model and the user interface can correctly display virtual objects in three dimensions in the real world.

Tracking The framework contains support for selected tracking hardware devices as well as the possibility of accessing future devices using the same interfaces. The output of the trackers is accessible to the application, including quality-of-service parameters such as accuracy and lag.

The framework also includes methods for postprocessing and combining the output of the trackers with filtering and prediction algorithms.

Modeling the World The framework provides a World Model which stores all information the system has about its environment. All relevant objects of the real world the system is operating in are represented as objects in the World Model. In addition, virtual objects such as *stickies* (location-fixed textual tags) that the system uses to augment the user's reality are stored in the World Model.

Modeling Tasks The framework provides a Task Model containing a description of the tasks the user has to do or wants to do. It receives events from the application when a step is performed and then generates the user interface for the next step. this allows complex Taskflows for maintenance or navigation to be stored in a structured and uniform format.

User Interface The user interfaces for applications built with DWARF are described at a high level in terms of functions, operations and messages. This description is then converted into actual interface elements for the currently running user interface devices. These devices allow *multi-modal* user interaction, i.e. flexibly combining different user input and output devices depending on the situation and the preferences of the user.

Accessing External Services The framework provides mechanisms for the application to select and access external Services based on the current context of the user and the system, such as geographical location and battery power.

Automatic Combining of Components The various functionalities provided by the framework are designed to interact automatically. Using a model of the real and virtual world, the tracking information and the taskflow descriptions, the user interface can display relevant information to the user, registered in three dimensions in real time.

Middleware The framework provides middleware that allows the application and the framework components to communicate using various different communication protocols. This includes an event Service with a publish-and-subscribe mechanism, interprocess and low-level communication methods.

Nonfunctional Requirements and Pseudo Requirements

In developing DWARF, we began completely from scratch. Nevertheless there are some constraints that had to be taken into account during the design phase.

Performance Despite the high degree of flexibility the framework allows, its primary goal is AR, which needs real-time performance for such functions as tracking and user output. The individual components and the framework as a whole must provide this kind of performance, even in a distributed system.

Resource Issues The entire system allows for user mobility. This means it must be able to run on small wearable devices and deal with changes in connectivity and bandwidth while users move from one area to another. Moreover, mobile users are subject to different security policies as they are connected to different administrative domains.

User Interface and Human Factors The user interface has to be multi-modal and, as far as possible, should not restrict the user during his normal work. The applications using DWARF should not require the user to know anything about the internal structure of the system.

Hardware Considerations The whole system is deployed on several hardware modules. Modules encapsulate functionality and are, to a certain extent, independent. The framework allows independent developing and testing of these modules. Furthermore, modules can be extended, modified or replaced by new versions.

Every Service can be mapped to a single special hardware component. Hardware components may be added or removed at run-time, without the user having to configure their interaction.

Quality Issues As the major goal of the project is the development of a framework, industrial quality does not have to be reached on the implementation level. On the design level, however, the overall structure, interfaces, and events should be frozen after the initial development stage to ensure the immediate usability of the design.

Security Issues For the first version of DWARF, security was not an issue. Nevertheless, in the future, user and system service authentication and authorization should be investigated, as well as encryption. For performance reasons the components should be able to decide upon the security level used.

Documentation The primary focus was on design rather than implementation issues. In consequence, the whole design process had to be documented, including the design rationale.

In order to provide a proof of concept, at least one working prototype using all parts of the framework's functionality had to be implemented.

2.4.2 Related Work

DWARF relies on many fields of ongoing research in computer science. It aims to combine wearable computing with plug-and-play distributed systems, easy-to-use human-computer interfaces and advanced tracking technologies. In this section, we will take a brief look at some research projects that are related to DWARF, in that they are investigating similar problems.

MIThril

MIThril [45] is a context-aware wearable computing platform that is currently being developed at the MIT Media Lab.

The primary focus of the MIThril project is on the development of a wearable infrastructure using small RISC[1] processors with low power consumption like the Intel StrongARM, a single-cable power/data connection between the distributed components and a high-bandwidth data connection to the infrastructure of the surrounding environment.

The design of the system can be split into four distinct classes of components. The overall goal for each component class was to provide small, lightweight devices with low power consumption.

The *MIThril computing cores* are the base of the system design. These cores use small RISC processors with extremely low power consumption to perform all necessary computing tasks. They communicate with one another using an Ethernet connection. All computing cores run a full operating system, currently Linux is used.

The *MIThril body networks* consist of a Body Network and a Body Bus. The Body Network connects the computing cores with one another using a standard twisted-pair Ethernet cable modified to include power lines. The Body Bus is used to connect external devices such as microphones or cameras to the computing cores. It consists of two data buses, USB and I^2C and a power supply, all combined into a single cable that can be branched easily.

The *MIThril peripherals* are connected to the system either using the Body Bus or special connectors on some computing cores. Many cheap off-the-shelf components such as microphones, input devices or webcams can be connected via the Body Bus and its USB component. Devices with more complicated interfaces that would be restricted by the 12MBps constraint of the USB bus, such as head-mounted displays, are connected directly to special computing cores.

Finally, the *MIThril software* currently consists of special versions of Linux running on all computing cores. Special drivers have been developed to support the body networks and

[1] Reduced Instruction Set Computer

special hardware such as head-mounted displays. On top of this basic operating system runs the application. Up to now, almost no serious applications have been implemented.

MIThril seems to be a perfect match for the DWARF project. It focuses on hardware and operating system issues and can therefore be an excellent basis for truly wearable DWARF systems focusing on software issues. The concept of the body network allowing single cable connections between the various components is very promising and should be explored in the future. As the developers of the MIThril project intend to publicize their circuitry and device driver code, it could be a good starting point for extremely small DWARF systems.

jAugment

The *jAugment* [29] project at the computer science department of the University of Rostock aims to develop a set of applications to be used with different kinds of wearable computers. The applications take advantage of the unique features of a wearable computer. Examples that are given include text editors, e-mail clients and MP3 players as well as a path-finder, street maps and schedules for trains and buses.

They classify the applications into three groups by their purpose: infrastructure applications provide interfaces, helper classes, user interfaces and central services. Everyday applications are all the small things that a computer in general is expected to do, but that get a bit complicated in the case of a wearable computer. The last group are special-purpose applications that only make sense on a wearable computer.

The project does feature dynamic adding and removing of input and output devices as well as dynamic access to services. But the applications are basically stand-alone applications that need to bring all their resources with them. There is no direct support for reuse of resources between different applications, when it is not specially implemented together with the application; the overall architecture of the system is centered around the user interfaces.

UbiCom

The *Ubiquitous Communications (UbiCom)* program is a multidisciplinary research program at Delft University of Technology. The program aims at carrying out research needed for specifying and developing wearable systems for mobile multimedia communications. [72, 37]

The basic system architecture combines (non-self-sufficient) mobile units with stationary computing servers. Thus, the research focuses on wireless networks and small mobile systems.

There are many hardware-oriented subprojects involved in this program, such as head-mounted display technologies, high-bandwidth wireless networking, and small wearable systems. One interesting result is the LART (Linux Advanced Radio Terminal) [38], a small yet powerful embedded computer capable of running Linux. Its performance is around 250 MIPS[2] while consuming less than 1 Watt of power. In a standard configuration it holds 32MB RAM and 4MB Flash ROM, which is sufficient for a Linux kernel and a sizeable ramdisk image. Just like the MIThril computing cores, the LART would be well-suited for deploying DWARF.

An important software issue is quality of service, which led to the design of a dedicated quality-of-service architecture [39]. This involves algorithms for predicting network use and protocols for reserving network capacity. As of yet, these are only implemented in a simulator, and there does not appear to be a software framework for actually managing the network

[2] Million Instructions Per Second

connections. Another subproject [74] aims to make the mobile systems adapt their behavior to changing availability of resources.

A third area of research is aimed at the design of multimodal human-computer interfaces, but this is still in the design stage.

Ubicom is a broad research project investigating many aspects of building mobile multimedia systems. The program does not aim to build a framework or single system, but rather to develop basic technologies.

2.4.3 System Design

This section briefly outlines the DWARF system design. Further details can be found in [5].

DWARF is designed as a collection of Services that can be combined flexibly to build an efficient and wearable AR system. These Services can run on separate hardware components that are then networked together to provide the desired functionality.

Using the framework to build an AR application involves selecting the required Services and hardware components. The DWARF middleware will then connect together the Services that depend on each other. For simple systems, the application will not have to configure the Services' interaction at all, since they find each other automatically. This is even possible dynamically, so that new Services can be integrated into a running system as they are found. For more complex systems, the application will need to configure the Service's interaction appropriately.

The Services are designed to be as independent of one another as possible, so an AR system with limited hardware resources can be built that uses only a few of the framework's Services. On the other hand, the framework is also designed to scale upwards with increases in available computing resources.

The framework provides Services that range from low-level image processing for optical tracking to high-level interpretation of World and Task Models and the user's current context. An application using the framework provides these World and Task Models and defines the logical interdependencies between events in these models and output to the user—the framework takes care of the rest. Of course, an application can also access the low-level Services directly, if this is required.

Terminology To avoid confusion, we should point out that the term *service* can mean different things.

DWARF Services are the components of the framework, such as a GPS tracking Service. These are often referred to simply as "Services".

External services are not part of DWARF, but can be accessed using DWARF Services, e.g. external printers.

System services are used internally by the components of the framework, e.g. network or middleware services.

Subsystem Decomposition

AR is all about bringing information and things together for the user. Accordingly, DWARF provides Services to model things, access information, bring these things together and present

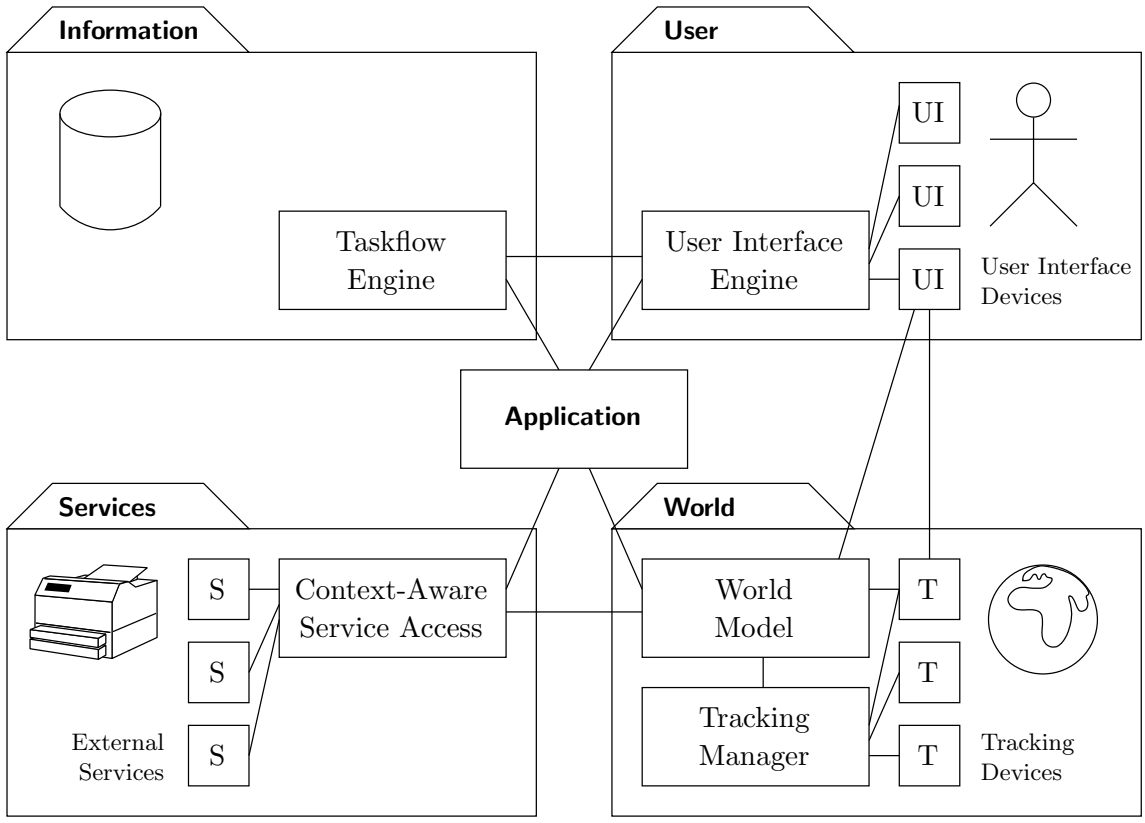


Figure 2.7: DWARF system design.
The DWARF services are conceptually decided into four different areas of functionality. Note that there are connections between the Services that bypass the application. For example, the combination of the World Model, a user interface device and a tracker can correctly register virtual objects in the real world.

them to the user. In addition, DWARF lets the user access external Services in the environment. See Figure 2.7 for an overview.

The application is generally "shielded" from the low-level devices, such as user interface devices, tracking devices, external services and so on. It can access these at a high level of abstraction using the various DWARF Services.

Modeling the World and Things in it For AR to work, the system must have an idea of both the position and appearance of things in the real world. For this, DWARF includes several different *trackers*, which can establish the position of things, and a *World Model*, which can store position information and other attributes of real and virtual things.

Accessing Information AR can display information about things to the user that he could otherwise not see easily. The information associated with things in the real world often involves a sequence of tasks, such as maintenance instructions or directions for navigation. DWARF models this sort of dynamic information in the *Taskflow Engine*.

Other models for information that can be presented to the user can be added to the framework in the future.

Interacting with the The User This is the central part of AR. DWARF provides several different kind of *user interface devices* and a *User Interface Engine* that lets the application access these devices in a high-level fashion. This allows multi-modal user interfaces to be created easily.

External Services Additionally, DWARF lets the user access external services in the environment which are not part of DWARF, such as printers or display devices. To allow the selection of such services based on user preferences and context such as geographical position, DWARF provides a *Context-Aware Packet Service*.

System Services The low-level system services allow DWARF to function as a distributed system. This includes the *network service*, the *communication services* and the *middleware*, which is used by all DWARF Services. These are *not* shown in Figure 2.7 on the preceding page—in a sense, they are omnipresent in DWARF.

Application The application in DWARF plays a rather unusual role for an application in a framework. From the point of view of DWARF, the application is divided into two parts:

- application logic modeled using the DWARF Services, such as the Taskflow Engine, the World Model or the User Interface Engine, and
- one or more specially developed DWARF Services with built-in application logic, such as for bootstrapping or configuring other Services.

Note that the only code needed for the application is written as a normal DWARF Service. There is no "special place" reserved for the application; rather, the application provides the "missing link" that the other Services need to cooperate usefully.

The application interfaces the with the DWARF middleware, and can access all of the DWARF Services and configure them, if it needs to. A simple AR application, however, can consist only of initializing the World Model and the the Taskflow Engine, and the information associated with the tasks will be displayed to the user in the correct three-dimensional position.

Hardware/Software Mapping

The DWARF Services can be distributed onto as many computers as is desired; the middleware will let them find each other, as long as they have a network connection. This allows computation-intensive Services such as an optical tracker to run on dedicated hardware which can be added to or removed from the system at run time.

This way, the user can configure his mobile AR system so that he never has to carry around more with him than is necessary.

DWARF takes advantage of existing software components where this is useful. For example, the user interface devices use existing VRML[3] rendering and voice recognition software, and the middleware makes use of CORBA[4] and third-party event Services.

The DWARF Services can run on many different platforms; the choice of hardware and operating system for a particular Service depends heavily on the availability of drivers for the specialized hardware (such as digital cameras or GPS[5] receivers). Performance-intensive and resource-constrained Services are generally written in C or C++, others are generally written in Java.

Persistent Data Management

Persistent data in DWARF is stored in various markup languages. This includes VRML for three-dimensional models, and various XML[6] dialects for Taskflows, the World Model, user interface descriptions, context-aware service requests, and descriptions and configuration of DWARF Services.

Global Software Control

Since DWARF consists of many cooperating Services, there are many simultaneously running processes, and many threads of control. The DWARF Services communicate with one another using the middleware's event service and remote procedure call mechanisms.

Boundary Conditions

The framework's Services can be started on demand by the middleware when other Services access them. Thus, startup of the whole DWARF system is automatic. The same mechanism allows new Services to be integrated into the system on the fly, and replace Services that have failed or cannot be accessed due to loss of network connectivity.

2.4.4 Component Walkthrough

This section provides a brief overview of each component of DWARF, indicating its functionality and how it is implemented. The components are grouped by the areas of functionality described above.

Modeling the World and the Things in it

The DWARF World Model contains geometric descriptions of objects in the real and virtual worlds. The tracking mechanisms provided by the framework allow the system to locate these objects in three dimensions in real time.

Tracking The tracking subsystem provides methods for determining the position of the user or other tracked objects. It is divided into two layers. The first layer consists of several more or less simple trackers that provide position information. This information is collected by

[3] Virtual Reality Modeling Language
[4] Common Object Request Broker Architecture
[5] Global Positioning System
[6] eXtensible Markup Language

the next layer called the Tracking Manager, which combines the possibly contradictory data and computes, according to the reliability of the data, the most probable position and the accuracy of this measurement.

We use two classes of tracking devices. First are simple devices that give less than the six-dimensional data (three translational and three rotational components) necessary for real three-dimensional registration, but have a large range of operation or require only limited computing power. Second are trackers that deliver six-dimensional data, but which are often constrained in rage or need a large amount of computing power.

Tracking Manager For an application using the tracking subsystem, the *Tracking Manager* is basically all it needs to get the most accurate position.

A Tracking Manager collects the information of all relevant trackers together and calculates a more accurate real position out of this data. This takes the general accuracy of the trackers into account as well as the time when the measurement was taken and the update frequency of the information.

One important feature of the Tracking Manager is the ability to dynamically add and remove trackers, which is invisible to the application.

In addition, facilities such as movement prediction may be added to the Tracking Manager. In short, its task is to make the whole of the trackers more than the sum of its parts. Further details on this, the simple trackers of DWARF and an extensive survey on various tracking mechanisms can be found in [3].

GPS Tracker The *GPS Tracker* uses the output of a standard GPS receiver to determine position and orientation. The position data consists of three-dimensional coordinates, although the altitudinal measurement is fairly imprecise, and an orientation angle from a magnetic compass. This GPS tracker only works outdoors, and only when it has an unobstructed view of the GPS satellites.

Radio Frequency ID Tracker The *RFID Tracker* uses passive tags and a reader for those tags to find the position of the user.

Radio Frequency ID (RFID) tags are small unpowered devices that are attached to known locations in the real world, like doors or significant points in hallways. A special RFID tag reader mounted on the user's wearable computer reads the tags' identification every time the user passes by. This way, it is possible to obtain precise location information, although only for a short moment.

Optical Tracker The *Optical Tracker* processes live video input from a camera mounted on the user's head and determines its orientation and position in real time. This six-dimensional data is crucial for performing "real" AR applications.

The basic principle of the Optical Tracker is simple. The video stream is analyzed for markers that are attached to known locations in the three-dimensional world. As a result, correspondences between two-dimensional image points and three-dimensional real world points are established. Sophisticated algorithms are now used to compute the camera's six-dimensional pose out of these correspondences.

Optical tracking is a computationally expensive task. To ensure stable high performance of this crucial Service for exact registration, the Optical Tracker should run on a dedicated

processor and deliver its result over a reliable network connection to the user interface.

Further details on the Optical Tracker are described in [77].

World Model The DWARF system needs to store data about real and virtual objects in a well-organized fashion. The first place to store all data describing the user's natural and virtual environment is the *World Model* Service. It can be seen as a large database that holds entries for every real or virtual object. Examples for real objects are buildings, floors, furniture in rooms etc. Virtual objects may consist of virtual stickies attached to real objects or highlighting information such as virtual arrows to indicate the directions the user has to take.

The crucial point for all these objects is their three-dimensional position and orientation towards one another. The World Model Service provides facilities that allow easy description and computation of these relations.

The World Model represents the world as a tree of *Things*, such as a campus, buildings, rooms, furniture items and so on. Each Thing has a geometric relation to its parent Thing, allowing the relative position of all Things to be computed. Each Thing can also have arbitrary attributes associated with it. Useful attributes include Services associated with Things, or VRML descriptions of a Thing's appearance. This tree-shaped structure is stored in an XML dialect.

As almost all DWARF components rely on such data, the World Model is a heavily used component. In consequence, efficiency was one of the major design goals.

The World Model Service is described in detail in [77].

Accessing Information

Dynamic information to be presented to the user can be modeled in the DWARF Taskflow Engine. Specific applications may want to model their own additional information, and in the future, new types of information models may be added to the framework.

Taskflow Engine The basic idea behind the development of the *Taskflow Engine* was to provide an easy-to-use possibility for the description of structured flows of tasks.

The advantages of this concept arise immediately if we think of maintenance applications using Augmented Reality technologies. Most maintenance tasks are characterized by a fixed flow of steps that have to be performed one after another. Using the Taskflow Engine, these steps can be described using an XML dialect. This feature is very useful for other application domains as well, such as navigation.

Internally, the Taskflow Engine may be seen as a state machine that switches to new states when it is triggered by certain incoming events.

Every state has information associated with it, which can be sent to the user interface to be displayed. This could be a textual or graphical description of a navigation task, e.g. "go up the stairs", or an animation of how to repair a certain machine part. By evaluating incoming events such as changes in the user's location or a spoken "done" command, the Taskflow Engine switches to new task descriptions.

Further details on the Taskflow Engine can be found in [57].

Interacting with the the User

All Services described up to now only processed data but did not provide any means of direct user interaction. The framework provides two levels of abstraction in interacting with the user: concrete user interface devices and a high-level User Interface Engine.

User Interface Engine The *User Interface Engine* provides a high-level encapsulation of interaction with the user.

Its main task is to display and process the user interface scenes provided by the application or the Taskflow Engine using multi-modal human-computer interfaces. The DWARF framework has been designed to support a large variety of application domains. In consequence, we cannot rely on a fixed class of user interface devices such as head-mounted displays or usual computer terminals. It may even be possible for the output device to change during the runtime of an application.

To handle these constraints, the User Interface Engine separates the description of the user interface from its actual instantiation. The input of the engine consists of an XML-based description of the user interface's functionality that does not contain much information about its final look and feel. This input is then transformed or *rendered* to a concrete user interface displayed on one or more available devices.

This approach allows high flexibility and reduced development time for highly platform-independent AR applications.

User Interface Devices Concrete user interface devices supported by the user interface engine include a VRML-based three-dimensional interface, a HTML-based interface for displaying textual and graphical information in two dimensions, and a voice recognition system that allows the user to give commands to the system.

Further details on these and on the User Interface Engine can be found in [59].

External Services

Systems built with DWARF can automatically integrate DWARF Services in the environment using the DWARF middleware. Other, non-DWARF external services can be accessed using the Context-Aware Packet (CAP) Service.

Context-Aware Service Selection and Execution Besides predefined flows of tasks, the user of a DWARF system should be able to spontaneously use external Services, such as printing in an unknown environment. This is handled by the *CAP Service*. Here, the user defines a Service he would like to use, such as "print out this document at a printer on the way to the meeting room", and the CAP Service takes care of it.

The major problem for this task is the user's current printer configuration. Even technically simple tasks such as printing can lead to problems in unknown computing environments, as a lot of contextual information such as the preferred paper size has to be considered for successful execution.

The basic idea of the CAP Service is now to encapsulate such information in packets that are further processed by software devices that route them in a suitable way. For the printing example, all of the user's configuration data, e.g. paper size, preferred color model etc., is stored in such as packet. The CAP Service gathers all information necessary for an optimal

fulfillment of the given task of printing from the other DWARF subsystems and executes the print job.

Further details on the CAP Service can be found in [43].

System Services

DWARF consists of many different location independent components such as a wearable user interface, external printers, tracking devices or databases. For these to cooperate, DWARF internally uses various system services, providing distribution and communication.

Middleware The DWARF middleware, described in chapters 3, 5 and 6 of this thesis, lets the components find each other and communicate with one another. It supports various modes of communication, such as event-based communication and remote procedure calls via CORBA.

The components of the DWARF system do not need to know each other specifically, but must only know what kind of functionalities of other Services they depend on. The middleware finds the appropriate Services and establishes the most efficient mode of communication that both Services support.

The middleware is designed to that it can be distributed onto separate network nodes, and does not rely on a central component. This means that even if one node of the system fails, the DWARF components on the remaining nodes can still communicate with one another.

Network Services For communication, the middleware and the DWARF components depend on the network subsystem. DWARF uses various networking technologies for communication: wireless networks for accessing remote Services, and local networks for connecting the separate hardware modules of a DWARF system together.

Since the availability and throughput of wireless networks varies, seamless handover between different types of networks should be possible. For this, in a future version of the framework, a DWARF Network Service is planned. This handles quality-of-service parameters and can, for example, trade off power consumption against bandwidth. The Network Service can take advantage of information in the World Model to recognize areas in buildings that are beyond reach of the wireless network, and it can provide contextual data to the other Services, such as a "network found" event.

Ethernet Communication For local communication between computers that are part of a mobile DWARF system, we currently use normal IP[7] communication over Ethernet.

WaveLan Communication In order to access external Services while roaming around a building or a campus, we use use IP over the *WaveLan* wireless network. Currently, we use IP version 4, but we could use IP version 6 with its Mobile IP features in the future.

Bluetooth Communication External Services that are not within the WaveLan network should also be accessed wirelessly. This allows the user to walk past devices providing location-dependent data and have the data downloaded with little or no user interaction.

The communication technology should have enough bandwidth so that useful data can be downloaded quickly (while the user is walking by), and should allow spontaneous connection.

[7] Internet Protocol

Of course, it should also be inexpensive in order to facilitate widespread use, and have low power consumption.

Bluetooth [6] is a new industry standard for low range wireless networks that fulfills these requirements. Thus, we developed the DWARF *Bluetooth Communication Service*, which provides a simple file transfer mechanism using Bluetooth technology. The bluetooth communication Service is described in [79].

2.4.5 Summary

The development of DWARF is far from complete. Indeed, the first version of the framework was developed in half a year. New components and missing functionality need to be added, and existing components will surely have to be redesigned.

Nevertheless, the framework has already proved its usefulness in a first demonstration system, described in Chapter 7. Using the framework, this (working!) system was built from scratch during three weeks' time.

By developing more applications using the framework, possibilities for improving and extending DWARF will become apparent, and we hope it will develop into a truly powerful framework.

3 Requirements Analysis for the DWARF Middleware

To build this framework, we need intelligent middleware that can describe and locate distributed services, manage resources, and let services communicate with each other.

DWARF relies on a collection of distributed Services being able to cooperate with one another. For this to work, we need an intelligent infrastructure—the middleware.

This chapter deals with the requirements I started out with when designing the DWARF middleware. Starting with the requirements coming from the design of DWARF as a whole, I use the methodology described in [7] to refine them step by step. This involves scenarios describing the interaction between the middleware and the rest of the DWARF system; use cases that refine these scenarios; functional and nonfunctional requirements; and models of the objects involved and their dynamic behavior.

The requirements analysis is quite thorough, in that it already identifies detailed interactions between objects. This makes the system design in Chapter 5 easier.

3.1 Problem Statement from the DWARF System Design

A *problem statement* briefly describes the scope of a system, including its high-level requirements, its target environment, and acceptance criteria. [7]

In this section—although it is not a complete formal problem statement—I briefly summarize the implications that the design of DWARF as a whole has for the design of the middleware.

Self-Assembling Distributed Services Systems built with DWARF consist of distributed Services that assemble themselves into a complete AR system. The functionality for the Services to find each other and to communicate with one another is handled by the middleware. This includes access to low-level and high-level communication methods such as remote method invocation and sending events. These functional requirements are on page 31.

Sufficient Performance for Augmented Reality For convincing AR, the communication between the tracking and display Services in DWARF has to be fast, or more specifically, have a low latency. This nonfunctional requirement is on page 33.

Connection-Oriented Communication Once the Services have assembled into a complete system, their communication needs are relatively static. For example, an optical tracker can send position updates to a display Service thirty times per second. This means that the communication that the middleware has to manage is *connection-oriented*. Services do not

spontaneously exchange information; rather, the middleware establishes connections between the Services, which these use for communication.

This is analogous to circuit-switched network communication (such as ATM[1]), as opposed to packet-switched networks (such as IP). A more "packet-oriented" approach in use of ad hoc services is offered by the CAP Service described on page 26.

Adaptability in Intelligent Environments As the user of an AR system built with DWARF roams through intelligent environments, the middleware must optimize the overall system functionality by connecting matching Services on the mobile system and in the environment together. It must also deal with the loss of connection between Services, such as when the user leaves a room or turns of a component of his wearable system.

Mediator Design Pattern Since DWARF is a framework, the individual components should be as independent of one another as possible, so they can be recombined in many different applications. To make this goal easier, we used the *mediator design pattern* [21], as shown in Figure 3.1. Rather than, for example, hard-coding the location of the optical tracker into the display, a "mediator", which knows both the display and the tracker, connects the two together. This way, the individual Services can be reused as-is in different systems.

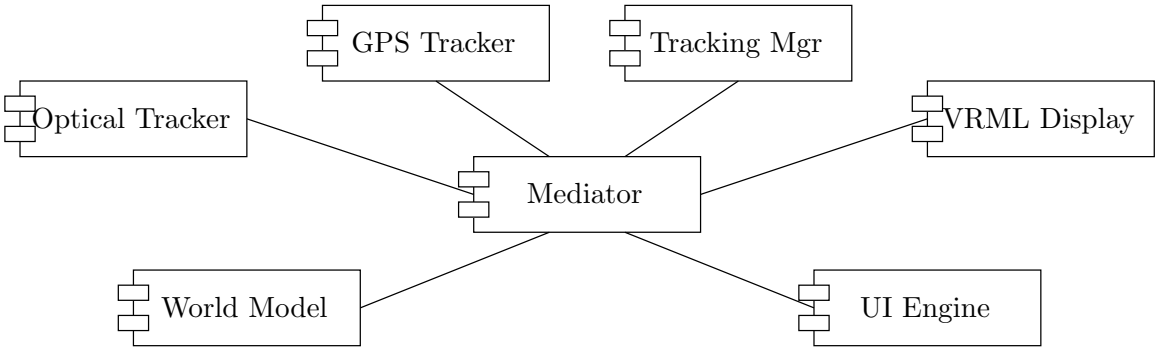


Figure 3.1: The mediator pattern in DWARF.
The DWARF Services communicate via a central mediator.

The low-level mediating functionality of coordinating between matching Services so that they can communicate with one another is in the DWARF middleware. If the middleware has descriptions of the individual Services and their communication needs, this task can be automated, allowing the Services to "self-assemble". For example, a display Service and a tracking Service can be connected together automatically.

Note that this is separate from high-level mediating functionality within the application. Complex application logic, such as coordinating between external services and virtual objects, must be written as a separate DWARF Service for each application.

Target Environment The DWARF Services run on many different hardware and operating system platforms, which the middleware must support. Furthermore, the middleware must be accessible from both Java and C++, the two languages the DWARF Services are written in.

[1] Asynchronous Transfer Mode

Acceptance Criteria The first implementation of the DWARF middleware had to successfully support the scenario of the demonstration system described in Section 7.1. The design of the middleware must be complete enough to support further development of the entire framework.

3.2 Functional Requirements

Functional requirements describe the interactions between the system and its environment independently of its implementation. [7]

In this section, I briefly describe the functional requirements of the DWARF middleware. Later, this is developed into more detailed scenarios (Section 3.6) and use cases (Section 3.7).

3.2.1 Locating Services

One main area of functionality for the middleware is locating Services. Note that the term *location* refers to logical locations such as network addresses, not to physical *positions*.

Advertise Services Many Services have abilities that they can offer to other Services. For example, trackers provide position data. The middleware must advertise these Services on the network so that other Services can use them.

Discover Services Conversely, the middleware must discover Services that have been advertised, so that the head-mounted display, for example, can receive position data from a tracker.

Abstract Description of Services In order to reduce coupling between the individual Services of DWARF, the Services should only access each other through well-defined interfaces. The middleware can maintain an abstract description of each Service, including the interfaces it supports and what kind of other Services it depends upon. Aside from formalizing the dependencies between Services, this also allows the middleware to start Services on demand, since it knows about Services even when they are not running.

Quality of Service In the three-dimensional display example, it is important that the position data be accurate and have a low temporal lag. This can be described in quality-of-service attributes. The middleware must be able to use such attributes, which can static (such as the resolution of a video camera) or dynamic (such as the current accuracy of a GPS device), to find the Services that are most appropriate for the task at hand.

Roaming and Handover In mobile applications, new Services can become available and the connection to old ones can be lost. The middleware must make this transparent to the user by changing the current Service configuration dynamically. For example, when the user walks out of the range of one video camera, another should take over.

3.2.2 Communication Between Services

Once Services have been located, they can communicate with one another using the middleware. Again, this functionality can be divided up:

Manage Communication Resources The middleware should provide a general mechanism for managing communication resources. This way, neither one of two communicating

Service have to deal with allocating event channels, network sockets, etc., making implementation of the DWARF Services easier. Additionally, this allows the middleware to configure the communication resources in order to optimize the communication in a group of Services—for example, by migrating an event service from one network node to another.

Event-Based Communication Many DWARF Services communicate with one another via events—for example, position events or user interface events. Event-based communication is flexible, since it frees the sender from having to know who will receive the event, and the receivers from having to know who sent it. The middleware must provide a mechanism for event-based communication. It also must manage the publish-and-subscribe mechanism for events, so that the individual Services do not have to implement this themselves.

Remote Method Calls Another useful kind of communication, especially when the order of steps in a complex interaction is important, is calling methods on remote objects. The middleware must provide mechanisms for this communication style, as well.

Extensibility to Other Communication Methods The middleware should be able to provide additional communication methods, such as shared memory blocks (when two Services are on the same host), streaming video, and so on. Since new communication methods will become necessary as the framework develops, it is important that these can easily be added later. Ideally, modules for new communication methods should be loadable at runtime.

Quality of Service As with the quality-of-service attributes of the Services themselves, the middleware should support quality-of-service attributes for the communication methods. For example, it might be able to guarantee in-order delivery of events, at the price of increasing event latency. The Services should be able to specify their communication needs.

3.2.3 Managing Services

The remaining functional requirements have to do with managing the Services as they are running.

Resource Management A crucial issue for mobile systems is power consumption. The middleware should provide a way of optimizing power consumption by carefully choosing which resources to use when. Other resources, such as a fee the user has to pay for using a Service, should be able to be modeled and optimized by the middleware as well.

Starting and Stopping Services A simple method of reducing power consumption is not to start Services until they are actually needed. The middleware should be able to offer Services' abilities on the network even when the Services are not running, and start the Services on demand when other Services need them. Analogously, the middleware should be able to shut Services down when they are no longer needed.

Installing Services This feature is currently not essential to the DWARF framework, but in the future, it would be nice to have a common mechanism of installing Services onto the networked computers that make up the DWARF system.

Service Status Information Again, this currently not very important. If the DWARF Services run on small wearable computers, some of which do not have user interfaces of their own, the middleware should provide a mechanism for gathering status information on the Services so that the user can see how his system is configured.

3.3 Nonfunctional Requirements

Nonfunctional requirements describe the user-visible aspects of the system that are not directly related to its functional behavior. [7]

The nonfunctional requirements of the DWARF middleware can be decided into two categories: on the one hand, the middleware must support convincing AR, and on the other hand, it should provide the flexibility for combining wearable computers and intelligent environments. The nonfunctional requirements identified here lead to design goals in Section 5.1.

3.3.1 Efficient Augmented Reality

Many current AR systems are built to run on a single computer, and do not have to deal with network communication at all. No AR system that I know of uses an elaborate and flexible middleware system such as the one proposed here. Thus, it is important that the extra flexibility that the middleware provides does not come at the cost of endangering the performance required to build an AR system. In a sense, these nonfunctional requirements represent constraints on the design and implementation of the middleware.

Availability The middleware should be at least as stable as all of the DWARF Services. If the middleware fails, the communication between the Services breaks down, and the system as a whole is no longer usable.

Reliability The middleware must perform its tasks correctly. If it tries to supply a printer with position data from a voice recognition system, neither side will be happy. Again, if the middleware fails, the DWARF system as a whole fails.

Low Overhead The middleware must not consume too much memory or processing power. This is especially important since it will be running on portable systems which are short on resources and already performing computation-intensive tasks such as image analysis, voice recognition or three-dimensional rendering.

Low Latency The event-based communication that the middleware supports must have a very low latency, so that position data can be transmitted from the trackers to the head-mounted display quickly enough that the user does not begin to feel sick. Not all events have to have this low latency, but a position event should not require more than 15 milliseconds (equivalent to half a video frame at 30 frames per second) from the time the tracker sends it until the time the display receives it.
Of course, the non-event-based communication should be fast as well.

Throughput The middleware must be able to handle fairly large volumes of data in communication. For example, position events are sent 30 times a second, and future DWARF systems might send streaming video data, as well.

Support Multiple Platforms For efficient AR, systems may use specialized hardware. Thus, the middleware may have to run on several different operating systems and processor architectures, and should be portable to others.

3.3.2 Flexibility for Mobile Systems in Intelligent Environments

The middleware must be able to arrange for communication between a wide variety of Services and under a wide variety of circumstances. This is the new idea of the DWARF middleware: not just to assemble one fixed system at runtime, but to reconfigure it dynamically. This kind of flexibility implies the functional requirements of locating Services described on page 31, but also the following nonfunctional requirements.

Fault Tolerance The middleware must be able to tolerate failures of network connections that are frequent in mobile environments. Indeed, it should shield the DWARF Services from these failures as much as possible and perform seamless handover.
This nonfunctional requirement leads to the important design decision of *distributing* the middleware, as described from page 69.

Robustness The middleware will connect Services together that it may never have known about before, such as when a mobile user walks into a new intelligent building. In the event that one such Service crashes, the middleware must survive, so that the other Services can continue running. Where possible, the middleware must shield the other Services from the crash.

Scalability At first, the middleware should mainly be able to support a small network of wearable computers. However, the design should be powerful enough so that the middleware can scale to run on large networks in intelligent buildings.

Response Time When a user walks quickly through a room with a short-range local wireless network, the middleware should nevertheless be able to (briefly) establish a connection between the wearable system and stationary Services. This allows the user to collect information from the environment while walking by.

Security Although security is currently not a major goal of systems built with DWARF, the middleware should be able to address security issues such as when multiple mobile users share Services in the same wireless network.

3.4 Pseudo Requirements

Pseudo requirements are requirements imposed by the client that restrict the implementation of the system. [7]

The only pseudo requirement for the DWARF middleware was our project team's agreement on using Java or C++ as implementation languages, not exactly a limiting decision.

3.5 Identification of Actors

Actors represent external entities that interact with the system. An actor can be human or an external system. [7]

For the DWARF middleware, there are human and nonhuman actors:

Service A Service in DWARF provides certain functionalities and requires others to work. It uses the middleware to find its communication partners, which are other Services, and to communicate with them. Services can also be notified by the middleware to start on demand or to stop when they are not needed.
Services can be internal to the wearable system, or they can be external Services in the environment. Examples for Services are a portable GPS tracking device, an external video-based tracker, an external printer, a head-mounted display driver, and a Taskflow Engine.

User The user does not usually interact directly with the middleware. Instead, she interacts with the framework's various Services and the application. However, by moving around in an intelligent environment, and by turning on and off various hardware devices, she can make new Services available or disable old ones.

Administrator Services can be installed on the various network nodes of a DWARF system by an Administrator, who does not necessarily have to be identical to the User of the system. This involves installing the Service executable and providing a description of the Service to the middleware.

3.6 Scenarios

A *Scenario* is a concrete, focused, informal description of a a single feature of the system from the viewpoint of a single actor. [7]

Here I will describe the scenarios I used when designing the DWARF middleware. Many of these are now actually supported and have been tested in our demonstration application, as described in Section 7.1. For simplicity, the scenarios only deal with tracking and display Services—of course, DWARF has many other Services as well.

The first scenario shows how the middleware can supply a Service with data from another Service.

Scenario: **Display needs position data**

Actor instances: `vrmlDisplay:Service, alice:User`

Flow of Events:

1. The `vrmlDisplay`, a Service for showing three-dimensional data on the user's head-mounted display, is started up by Alice. In order to register the virtual objects in three dimensions, the display needs position data about the user's head.
2. The display registers itself with the middleware, calling an appropriate method and identifying itself by its name, `vrmlDisplay`.
3. The middleware finds an appropriate tracker that can deliver the information that the display needs and sets up an event channel for sending position data over the network. It then gives the information about the event channel to the display.
4. The display connects itself to the event channel and, from then on, receives position data from it.

The next scenario is complementary to the above one: it shows how the tracker supplying the position data to the display is started up by the middleware.

Scenario: **Tracker provides position data**

Actor instances: `opticalTracker:Service`

Flow of Events:
1. The `opticalTracker`, a Service that can identify the position of an attached digital camera from the video input, is started by the middleware.
2. The tracker registers itself with the middleware as above.
3. The middleware notifies the tracker that it should start running. The tracker starts its digital camera and begins tracking.
4. The middleware creates an event channel for the optical tracker to send its position data to. It gives the information about the event channel to the tracker.
5. From now on, the tracker sends its position data to this event channel.

In order for the middleware to be able to start the optical tracker automatically when the display starts, the middleware needs to know that one supplies position data which the other can consume. This is described in XML Service descriptions, which the administrator gives to the middleware when he installs Services.

Scenario: **Install a tracking Service**

Actor instances: `joe:Administrator`

Flow of Events:
1. Joe, the administrator for a particular DWARF system, wishes to set up a new optical tracking Service on one of his wearable DWARF modules.
2. He downloads the executable for the tracking Service and installs it.
3. Joe edits an XML file describing this tracking Service, adjusting some parameters such as which camera device to use and the installation path of the executable, and stores this XML file in a directory known to the middleware.
4. The middleware now knows about the Service and can start it on demand.

A Service can also describe itself, as shown in the next scenario.

Scenario: **Service describes itself**

Actor instances: `opticalTracker:Service, alice:User`

Flow of Events:
1. The optical tracker is loaded manually by the user, Alice.
2. The tracker inquires the middleware to see if it already has a Service description named `opticalTracker`.
3. If not, the Service describes itself to the middleware using an appropriate interface.
4. The middleware now knows about the Service and can satisfy its communication needs.

The following scenario shows how the middleware dynamically reconfigures connections between Services while a mobile user moves around in an intelligent environment. It is rather complicated, but this shows how the middleware can establish whole hierarchies of dependencies between Services.

Scenario: **Dynamically reconnect Services**

Actor instances: `alice:User, inertialTracker:Service, opticalTracker:Service, trackingManager:Service, vrmlDisplay:Service`

Flow of Events:

1. Alice is wearing a mobile AR system built with DWARF. She has a gyroscope chip on her head-mounted display, which is used by an inertial tracking Service running on one of the wearable modules. Alice's wearable also has a `trackingManager` running on it, which can combine the output of various trackers. The `trackingManager` is inactive at the moment, since the `inertialTracker` is sending its data directly to the `vrmlDisplay`.
2. Alice walks into an intelligent room which has a camera mounted on the ceiling. This is connected to an optical tracking Service running on a server in the room. The mobile system comes within range of the room's Bluetooth wireless network.
3. The middleware recognizes that the position data of the `opticalTracker` could be sent to the `vrmlDisplay` or to the `trackingManager`.
4. Since the `trackingManager` now has access to data from two trackers, the middleware recognizes that the `trackingManager` can provide higher-quality output than either of the trackers themselves.
5. The middleware starts up the `trackingManager` and connects it with the outputs of the `inertialTracker` and the `opticalTracker`. The `trackingManager` computes accurate position data by combining both.
6. The middleware disconnects the `vrmlDisplay` from the `inertialTracker` and instead provides it with the more exact data from the `trackingManager`.

The next scenario shows what happens when a Service is no longer available. This example does not include a tracking manager to combine the output of two trackers.

Scenario: **Fall back to another Service**

Actor instances: `alice:User, gpsTracker:Service, rfidTracker:Service, vrmlDisplay:Service`

Flow of Events:

1. Alice is wearing a mobile DWARF system with a GPS tracker and a radio-frequency ID tag tracker. Her `vrmlDisplay` is showing a map, aligned in three dimensions using the position data from the `gpsTracker`.
2. As she enters a building, the GPS tracker loses sight of its satellites and can no longer provide position data.
3. The GPS tracker updates its quality-of-service parameters to indicate that it can no longer provide reliable data.
4. The middleware disconnects the display from the `gpsTracker` and reconnects it to the `rfidTracker`.

The last scenario illustrates different communication methods supported by the middleware. It also shows how data from one Service can be distributed to different consuming Services simultaneously.

Scenario: **Communication methods**

Actor instances: `localOpenGlDisplay:Service`, `remoteOpenGlDisplay:Service`, `opticalTracker:Service`

Flow of Events:

1. In order to demonstrate the capabilities of DWARF to a large audience, an OpenGL-based renderer is running on the same hardware module that the optical tracker is running on. This renderer can overlay the video data from the optical tracker's camera with three-dimensional objects.
2. The middleware allocates an area of shared memory for communication between the `opticalTracker` and the `localOpenGldisplay`. It tells the tracker and the OpenGL display about this.
3. The tracker now copies the video data it receives, as well as the position data it calculates, to the shared memory area, and the local OpenGL renderer accesses this data to generate images.
4. A second audience is watching the demonstration remotely, and is also using an OpenGL-based display on a remote workstation.
5. The middleware creates an event channel for transmitting position data and a streaming video connection for the video data and notifies the tracker and the remote display.
6. The `opticalTracker` sends its position data and a (smaller) copy of its video data across these network connections to the `remoteOpenGlDisplay`.

3.7 Use Cases

Every scenario is an instance of a *use case*, that is, a use case specifies all possible scenarios for a given piece of functionality. [7]

Here, I present the use cases for the middleware, as extracted from the above scenarios. They are quite detailed, as they involve a lot of dynamic interaction between multiple Services. Figure 3.2 on the following page shows the relationships between use cases.

The first two use cases are from the point of view of a Service such as the `vrmlDisplay`, which is started manually and needs data from other Services to operate.

Use Case: **StartManually**

Initiated by: `User`

Communicates with: `Service`

Flow of Events:

1. *(Entry condition)* The `User` starts the `Service` manually by starting the appropriate executable.
2. The Service checks whether the middleware has a Service description with its name, i.e. whether it has been described by an administrator using the *DescribeService* use case.

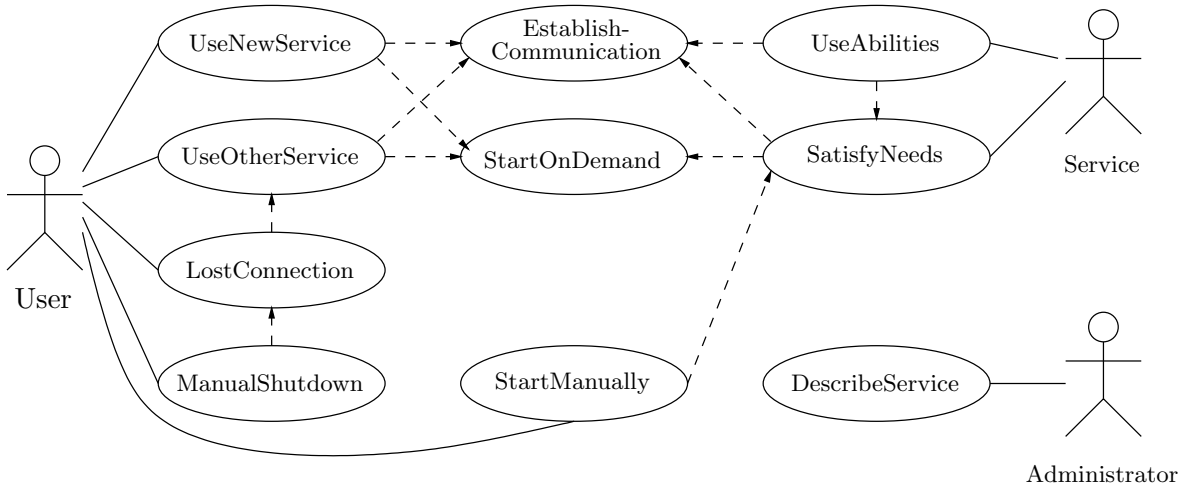


Figure 3.2: Use cases for the DWARF middleware.
The dashed arrows indicate "include" relationships between use cases; the solid lines indicate use cases being initiated by actors.

3. If there is no such description yet, the Service describes itself to the middleware using an appropriate interface.
4. The Service registers itself with the middleware by name.
5. The middleware invokes the *SatisfyNeeds* use case to satisfy the Service's Needs.
6. *(Exit condition)* If the Service's Needs can be satisfied, the middleware tells it to start running.

Once the Service has registered itself with the middleware, the middleware will try to fulfill the Service's needs.

Use Case: **SatisfyNeeds**
Initiated by: `consumer:Service`
Communicates with: `suppliers:Services`
Flow of Events:

1. *(Entry condition)* The `consumer` is a Service that needs data from another Service in order to operate. It has been loaded and has registered itself with the middleware, using the the *StartManually* or *StartOnDemand* use case.
2. The middleware attempts to find other Services that can deliver the data this Service needs. If necessary, it starts them on demand with the *StartOnDemand* use case.
3. For each Need of the `consumer` Service that the middleware satisfy, it establishes communication between the appropriate `supplier` and the `consumer`, using the *EstablishCommunication* use case.
4. *(Exit condition)* The Service is connected to enough other Services to satisfy all of its Needs, provided that enough other Services are available.

The next three use cases reflect what happens on the other end: a Service is started on demand, and one of its Abilities is used. For the Service to be started on demand, it must first be described by an administrator.

Use Case: **DescribeService**

Initiated by: `Administrator`

Communicates with:

Flow of Events:

1. *(Entry condition)* The `Administrator` places a valid XML Service description in a specified directory for the middleware.
2. *(Exit condition)* The middleware loads the Service description and can now start the described Service on demand.

Now that the Service has been described, it can be started when its Abilities are needed by other Services.

Use Case: **StartOnDemand**

Initiated by: `firstConsumer:Service`

Communicates with: `supplier:Service`

Flow of Events:

1. *(Entry condition)* The `supplier` is a Service that can deliver data to other Services. The Service has previously been described by an administrator, using the *DescribeService* use case. Enough other Services are available to satisfy any needs of the `supplier` Service. A `firstConsumer` Service needs data that the `supplier` can supply, and has also been started and registered itself with the middleware.
2. The middleware loads the `supplier`, using the executable file specified in the Service description.
3. *(Exit condition)* The Service registers itself with the middleware by name.

Once the desired Service has been started, its Needs are satisfied, and then it can offer its Abilities to other Services.

Use Case: **UseAbilities**

Initiated by: `firstConsumer:Service`

Communicates with: `supplier:Service`

Flow of Events:

1. *(Entry condition)* The `supplier` is a Service that can deliver data to other Services. It has been loaded and has registered itself with the middleware, using the the *StartManually* or *StartOnDemand* use case. Enough other Services are available to satisfy any needs of the `supplier` Service. A `firstConsumer` Service needs data that the `supplier` can supply, and has also been started and registered itself with the middleware.
2. The middleware satisfies all of the `supplier`'s needs, using the *SatisfyNeeds* use case.
3. The Service is notified that it should start operating.
4. The middleware establishes communication between the `supplier` and the `firstConsumer`, using the *EstablishCommunication* use case.

5. *(Exit condition)* The Service is running and can provide its requested Ability to the `consumer`.

When a Service's Abilities are no longer needed, the middleware can shut it down.

Use Case: **NoLongerNeeded**

Initiated by: `lastConsumer:Service`

Communicates with: `supplier:Service`

Flow of Events:

1. *(Entry condition)* At least one Ability of the `supplier` is being used, as per the *UseAbilities* use case. The last consumer using this Ability, `lastConsumer`, disconnects.
2. The middleware waits for a timeout specified in the Service description so that other consumers can reconnect to the Ability. If another consumer connects again during this time, this use case ends.
3. The middleware disconnects the Needs of the `supplier` from all other Services.
4. The middleware frees the communication resources used by the Service.
5. The middleware notifies the Service that it should stop running.
6. *(Exit condition)* The Service unregisters itself from the middleware and terminates.

Of course, a Service can be shut down by the user, as well.

Use Case: **ManualShutdown**

Initiated by: `User`

Communicates with: `Service`

Flow of Events:

1. *(Entry condition)* The user instructs the Service to shut down.
2. The Service unregisters itself from the middleware and terminates.
3. The middleware frees the communication resources used by the Service.
4. *(Exit condition)* For any Services that were using this Service's Abilities, the *LostConnection* use case is used.

The next two use cases deal with the middleware's dynamic behavior when the user roams around in an intelligent environment. First, a new Service can be used *in addition to* an old one.

Use Case: **UseNewService**

Initiated by: `User`

Communicates with: `newSupplier:Service,consumer:Service`

Flow of Events:

1. *(Entry condition)* All needs of the `consumer` Service have been satisfied, using the *SatisfyNeeds* use case. The `consumer`'s Service description allows the Need to be satisfied more than once (e.g. a Tracking Manager that can combine multiple position data inputs). The user enters a new wireless network, where the middleware finds a `newSupplier` that can satisfy one of the `consumer`'s needs.
2. The middleware establishes communication between the `newSupplier`

and the `consumer`, using the *EstablishCommunication* use case.

3. *(Exit condition)* The `consumer` can use the Abilities of the `newSupplier`.

Second, a new Service can be used *instead of* an old one.

Use Case: **UseOtherService**

Initiated by: `User`

Communicates with: `newSupplier:Service,oldSupplier:Service,consumer:Service`

Flow of Events:

1. *(Entry condition)* The `consumer` is currently using the `oldSupplier` to satisfy one of its Needs. The user either enters a new wireless network, where the middleware finds a `newSupplier` that can satisfy one of the `consumer`'s needs better than the `oldSupplier`, or the `oldSupplier` is disconnected.
2. The middleware disconnects the `oldSupplier` from the `consumer`. This can lead to the *NoLongerNeeded* use case being used for the `oldSupplier`.
3. If necessary, the middleware starts the `newSupplier` with the *StartOnDemand* use case.
4. The middleware establishes communication between the `newSupplier` and the `consumer`, using the *EstablishCommunication* use case.
5. *(Exit condition)* The `consumer` can use the Abilities of the `newSupplier`.

If a connection is lost to a Service, the middleware notifies the Services depending on it.

Use Case: **LostConnection**

Initiated by: `User`

Communicates with: `oldSupplier:Service,consumer:Service`

Flow of Events:

1. *(Entry condition)* The `consumer` is currently using the `oldSupplier` to satisfy one of its Needs. The connection is lost when the user leaves the wireless network.
2. The middleware notifies the `consumer` of the lost connection.
3. If a new supplier is available that can satisfy the `consumer`'s need, the *UseOtherService* use case is used.
4. *(Exit condition)* If a new supplier is available, the `consumer` can use its Abilities.

The last use case is never initiated directly by actors; it is included by other use cases.

Use Case: **EstablishCommunication**

Initiated by: other use cases

Communicates with: `supplier:Service, consumer:Service`

Flow of Events:

1. *(Entry condition)* The `supplier` and `consumer` Services have both been loaded and have registered themselves with the middleware. The consumer has a Need that the supplier has a matching Ability for.

2. The middleware sets up the necessary communication resources for communication. This can be, for example, an event channel, a shared memory block or a remote interface reference. The choice of communication resources depends on the Service's preferences and their location in the network.
3. If this is the first consumer to be connected to the `supplier`, the middleware notifies the `supplier` that its Ability is desired, and supplies it with the properly configured communication resources. Otherwise, the `supplier` is not notified.
4. The middleware notifies the `consumer` that its Need can be satisfied, and supplies it with the properly configured communication resources.
5. *(Exit condition)* The `supplier` and `consumer` Services are both connected to the communication resources and can communicate with one another.

3.8 Object Models

The *object model* describes the structure of a system in terms of objects, attributes, associations, and operations. The analysis object model represents the application domain, that is, the concepts that are visible to the user. [7]

In this section, I will identify objects from the scenarios and use cases described above, forming the analysis object model for the DWARF middleware. The descriptions of the objects' attributes and operations are still fairly informal at this level.

The object model is divided into "real-world" *entity objects*; *boundary objects* representing system's interface; and *control objects* representing interactions between the system and the outside.

3.8.1 Entity Objects

Entity objects represent the persistent information tracked by the system. This includes real-world entities that the system needs to keep track of. [7]

For the middleware, the "real world" consists of the Services using it, their descriptions, and so on.

Services

The first object class, *Service*, is easy to identify. The entire DWARF system consists of cooperating distributed Services, and *Service* actors appear in every use case except *DescribeService*.

A Service can be started and stopped by a user or by the middleware, as in the use cases *StartManually* and *StartOnDemand*. It is registered with the middleware, so it can advertise its functionality to other Services, and find the external functionality it needs. Services can be stopped manually or when they are no longer needed, as in *ManualShutdown* and *NoLongerNeeded*.

Design Rationale Services are not subclassed into "supplier" and "consumer" Services, even though these roles show up in the use cases. This simple division would limit the modeling power of the middleware: we have to be able to model Services that can play both roles simultaneously. This is accomplished by the next two objects identified.

Needs and Abilities

Services provide certain functionalities to the user and to other Services. I have called these *Abilities*. Analogously, Services may require functionalities of other Services to operate. I have called these *Needs*. A Service can have zero or more Needs, and zero or more Abilities (see Figure 3.3).

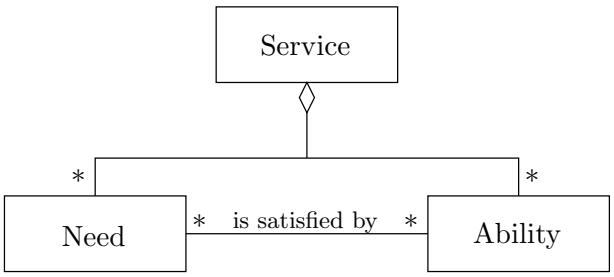


Figure 3.3: Services have Needs and Abilities.
Needs of one Service can be satisfied by Abilities of another Service.

For example, an optical tracker Service attached to a digital video camera might have the following Needs and Abilities:

- the Ability to provide position data,
- the Ability to provide a live video image,
- and a Need for a description of the world that it is supposed to recognize.

It is the middleware's responsibility to satisfy the Needs of all Services with the Abilities of others. Thus, at any given time, the middleware must maintain a dependency graph between Services, as shown in Figure 3.4 on the following page. This is where the middleware performs most of its work, as described in the use cases *SatisfyNeeds*, *UseAbilities*, *UseNewService*, *UseOtherService*, and *LostConnection*.

In addition, the middleware can start and stop Services on demand, since it knows when a Service is needed by others, and when a Service's Needs are satisfied.

Implementation of Services Modeling the Needs and Abilities as separate objects fits the implementation of many DWARF Services, which have separate objects to handle communication. In Figure 3.4 on the next page, for example, the tracker has a `PositionSender` object. In Chapter 5, these Needs and Abilities are refined into interfaces, which in fact were inspired by the DWARF trackers developed by Martin Bauer in [3].

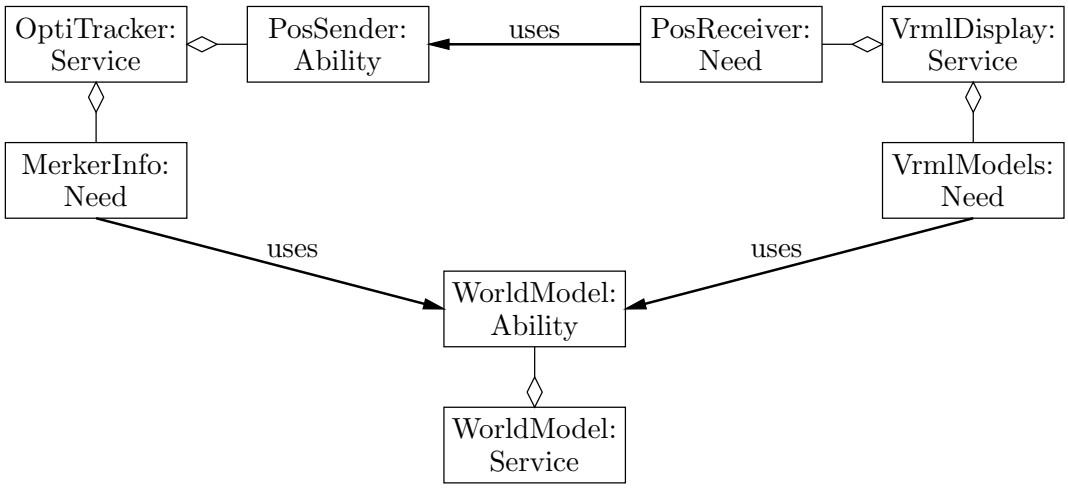


Figure 3.4: Example dependency graph between Services.
A VRML-based three-dimensional display Service needs models of real and virtual objects to display, which come from the World Model Service. It also needs position data from the optical tracker, which in turn depends on the World Model as well for descriptions of markers to recognize.
Compare this to the abstract view of Figure 2.5 on page 14.

Design Rationale This model of Needs and Abilities is not the only possible model for Services. Indeed, it would be quite possible to model both Needs and Abilities as one single entity, such as a "communication port". However, the idea of Needs and Abilities lets the middleware know what "communication ports" are actually needed before a Service can start, allowing better resource allocation.

Where is the User? As a rule of thumb showing when to model something as a Need and when to model it as an Ability, consider what side of the communication a human is on. Thus, it is natural to model a head-mounted display as *needing* position data, since this data is used to generate output for the user. On the other hand, a printing Service has the *Ability* to accept print jobs, since the user has documents that he wants to print out. This example shows that the direction of data flow does not necessarily coincide with the direction of a dependency between two Services.

Following the "where is the user" guideline, a Service's Need in one application may become an Ability in another. For example, when a user watches a television show, his television *needs* images to display in order to make the user happy. During commercials, this changes. The viewer would be happier without commercials, but the television has the *Ability* to show images, which another user, the advertiser, is paying for. This may seem like a very subtle difference, but I believe it is important to be able to model these two situations differently. An interactive television system may someday allow the viewer to adjust the advertising level he is willing to accept, going from pay-per-view (no advertising) to get-paid-for-viewing (lots of ads).

If the middleware is aware of this kind of distinction, it makes it easier to design applications around fulfilling the user's desires.

Service Descriptions

Even when a Service is not actually running, it can be described in some fashion. This description is partly hard-coded into the Service itself, and can partly be adjusted according to the application the Service is part of.

In other middleware systems, this description is often implicit, i.e. coded into the Service. If, however, this description is formalized as a *Service Description* object, it can be given to the middleware by an administrator (use case *DescribeService*), This conserves resources by letting Services be located before they are started, and also makes it possible to package Services as well-described stand-alone components. A Service Description potentially includes:

- a Service's Abilities, i.e. the functionalities it provides
- a Service's Needs, i.e. functionalities of other Services that this Service depends on
- quality-of-service information
- attributes of the Service, such as a printer's paper size
- configuration information
- authorization and billing information
- supported communication protocols
- how to start the Service, e.g. command line.

Some elements of this description are static (e.g. pages per minute) and can be queried even when the Service is not running, others can change dynamically while the Service is running (e.g. amount of toner remaining or number of pending print jobs). The middleware must provide a mechanism for these attributes to be accessed in a uniform fashion.

Communication Resources

There are many different ways that Services can communicate with one another. This includes different basic communication methods such as events, sockets and shared memory, and "simple" differences in protocol, such as CORBA or COM[2] events.

These infrastructure for these communication methods (such as an event service) is managed by the middleware, as described in the use case *EstablishCommunication*.

Entity Objects involved here are communication resources such as event channels, shared memory blocks, and sockets. Since the DWARF middleware takes advantage of existing operating system facilities and off-the-shelf components for communication, I have not modeled these communication resource objects in detail. Instead, the management of these resources is encapsulated as described below.

3.8.2 Boundary Objects

Boundary objects represent the interactions between the actors and the system. [7]

For the DWARF middleware, there is only one boundary object: the Service Manager.

[2] Component Object Model

Service Manager

Each Service must be able to access the middleware somehow. In order to give the DWARF middleware a coherent interface, I have defined an object called the *Service Manager*.

Upon startup, a Service registers itself with the Service Manager, from where it can gain access to all of the DWARF middleware's functions. The Service Manager must know the Service descriptions for the Services that register themselves with it. If the Service Manager has not received the Service description in the form of an XML file from an administrator, the Service must describe itself upon startup.

3.8.3 Control Objects

Control objects represent the the tasks that are performed by the user and supported by the system. [7]

There are two control objects, in the DWARF middleware: Connectors and Active Service Descriptions.

Connectors

In the use case *EstablishCommunication*, the middleware needs to be able to deal with the various communication protocols the Services use.

Since the DWARF middleware is supposed to be extensible to support many different protocols, I have encapsulated the functionality for establishing and managing communication into *Connectors*.

The middleware creates Connectors when Services want to communicate with one another. These allocate and configure the necessary communication resources on both ends of the connection. The Services can access the communication resources (event channels, shared memory blocks) through the Connectors, and use them to communicate with one another. This is shown in Figure 3.5 on the following page.

Design Rationale Having a basic Connector interface to encapsulate different protocols makes the middleware much more extensible, since protocol-specific connectors can be added later. Connectors also reflect the control flow in the use case *EstablishCommunication*. Services do not actively request connections to other Services. Instead, the middleware establishes a connection and delivers information about this connection to the Service. This information is encapsulated in a Connector.

Active Service Descriptions

When a Service becomes active, the middleware needs to keep track of it. A simple Service Description is not enough for this—we need an *Active Service Description*. This object knows the description of a Service and acts on its behalf. It coordinates the rest of the middleware functionality in satisfying the Needs of a Service and offering its Abilities to other Services. It encapsulates the functionality of the Service life cycle, described on page 50.

Design Rationale In identifying control objects, one suggestion [7] is to identify one control object per use case. Here, I only have a single control object for many use cases. Why is that?

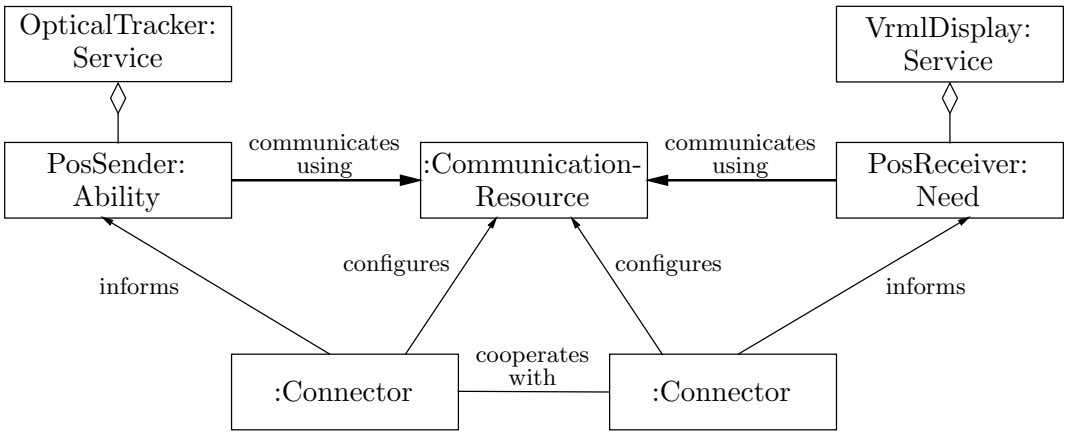


Figure 3.5: Communication between Services using Connectors.
A VRML display receives position data from an optical tracker. The data is sent using some communication resource (such as an event channel). This is configured by two cooperating connectors. The two Services receive information about the communication resources from their Connectors, so they do not have to know each other directly.

These use cases involve multiple Services, which can be on different hosts. If there were a control object for each such use case, it would obviously have to be on one host or the other. This would make the implementation asymmetrical, and could lead to a loss of stability if one of the hosts failed. By sharing the control between two Active Service Descriptions, each control object is associated with one single Service running on one single host.

3.9 Dynamic Models

Dynamic models document the behavior of the object model, in terms of state chart diagrams and interaction diagrams. Although this information is redundant with the use case model, dynamic models enable us to more precisely represent complex behaviors. [7]

For brevity, I have not modeled every possible use case, but the models here should suffice to explain the middleware's dynamic behavior.

3.9.1 Interaction Between Objects

Here, I will show how the objects interact in two important scenarios: *Display needs position data* and *Tracker provides position data*, described from page 35. These concrete examples give a better overview of how the objects interact than the more abstract use cases.

In this example, the display has not been described to the middleware yet, so it describes itself; the tracker is described to the middleware by an administrator so it can be started on demand.

Note that these two scenarios are complementary: when Alice starts the display, the middleware finds and starts the tracker automatically. This could be modeled in one huge interaction diagram, but to keep things simple, I have used two smaller ones.

Note also that the interactions shown here do not include the details of locating a tracking Service, which are internal to the middleware.

Display Needs Position Data Figure 3.6 shows how a head-mounted display starts up and receives position data from a tracker.

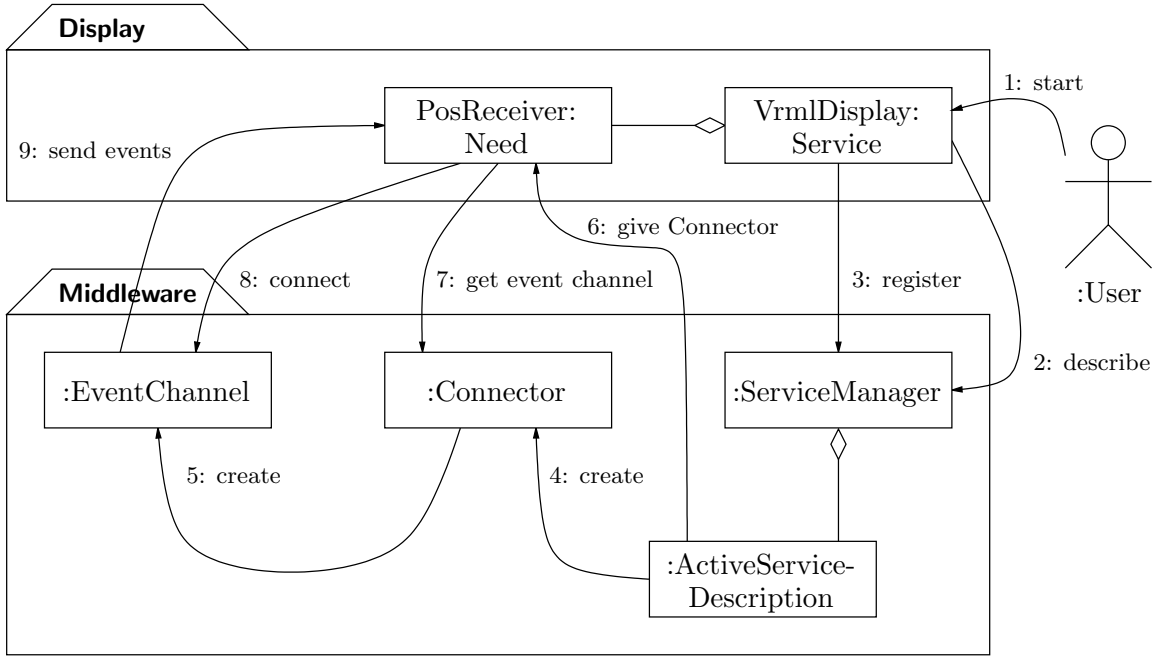


Figure 3.6: Example interaction between a display Service and the middleware

1. Alice starts the display Service.
2. The Service describes itself to the Service Manager, which creates an Active Service Description object for the Service.
3. The Service registers itself with the Service Manager.
4. After having found an appropriate tracking Service to provide position data for the middleware, the Active Service Description creates a Connector to handle the communication.
5. The Connector allocates an event channel and configures it to receive position data from the tracker's event channel.
6. The Active Service Description gives the display's Need a reference to the Connector.
7. The display retrieves a reference to the event channel from the Connector.
8. The display's `PosReceiver` connects itself to the event channel.
9. The event channel forwards incoming position events to the display.

Tracker Provides Position data Figure 3.7 on the following page shows how an optical head tracker is started on demand and supplies position data. For simplicity, this tracker is

modeled as having no Needs itself—in reality, however, most DWARF trackers need at least a World Model to know what they are tracking.

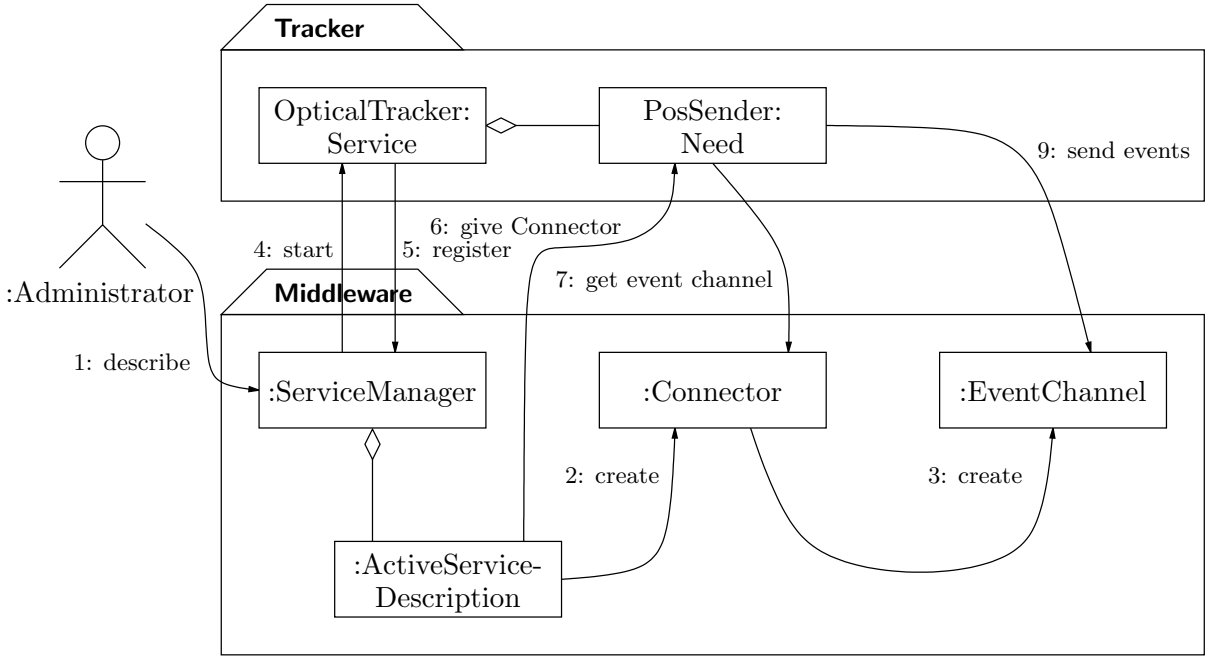


Figure 3.7: Example interaction between a tracker Service and the middleware. Note how this fits together with Figure 3.6 on the preceding page.

1. Joe, the administrator, describes the tracking Service to the middleware.
2. The optical tracker's Active Service Description creates a Connector to handle incoming communication requests.
3. The Connector creates an event channel and configures it to receive position data from the tracker.
4. The Active Service Description, having been notified that another Service needs position data, starts the tracker Service.
5. The Service registers itself with the Service Manager, which associates it with its Active Service Description.
6. The Active Service Description gives the tracker's Ability reference to the Connector.
7. The tracker retrieves a reference to the event channel from the Connector.
8. The tracker's `PositionSender` sends its position events to the event channel.

3.9.2 The Service Life Cycle

Services in DWARF can be in several states, the stages of the *service life cycle*. In each, they have different requirements of the middleware.

The DWARF middleware currently does not address all stages of the service life cycle, such as installation, upgrade, and so on. It only deals with starting and stopping Services and

arranging for communication between them. Managing these additional steps of the service life cycle would be a useful future extension of the middleware.

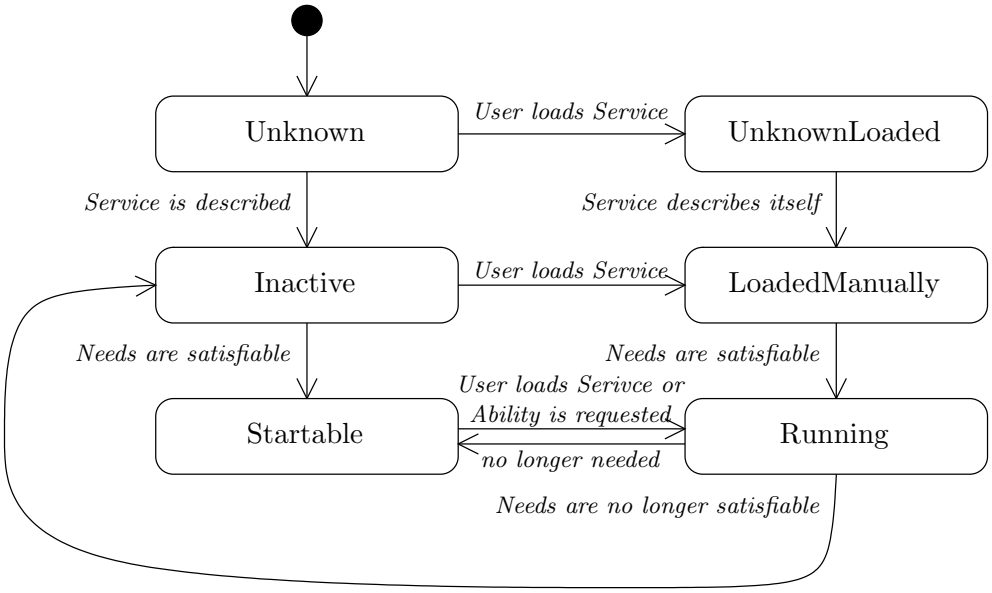


Figure 3.8: Stages in the service life cycle

The states of the life cycle that are currently supported by the middleware, shown in Figure 3.8, are as follows:

Unknown The middleware does not know anything about the Service.

UnknownLoaded The user has loaded the Service, but this Service has not been described yet. The Service describes itself to the middleware.

Inactive An administrator has described the Service to the middleware, but some of the Service's Needs are not yet satisfiable. The middleware tries to find Services to satisfy the Service's Needs.

LoadedManually The user has loaded the Service manually, and it has been described, but some of the Service's Needs are not yet satisfiable. As soon as the middleware has found other Service to satisfy the Service's Needs, it establishes the appropriate communication and starts the Service.

Startable The middleware has found other Services that can satisfy this (not yet loaded) Service's Needs. As soon as an Ability of this Service is requested, the middleware establishes the communication to satisfy the Service's Needs, loads the Service and starts it.

Running The Service is loaded, the middleware has satisfied its Needs, and told it to start running. When the Service is no longer needed, or when its Needs are no longer satisfiable, it stops.

3.9.3 Traversing Service Dependencies

In finding, starting and arranging communication between Services, the middleware essentially creates and traverses a dependency graph between Services. Since locating other Services happens before actually using them, the middleware must traverse this dependency graph twice, once for location and once for starting. In the first traversal, a possible configuration of Services using other Services is found, but Services that are not running are not actually started yet. In the second traversal, the Services are started one by one as they actually start using one another and communicating.

The double traversal of the Service dependency graph is shown in Figure 3.9 on the following page.

Design Rationale This two-phase traversal is more complicated than a single traversal, but it is also more powerful. It allows browsing for potential Services while they are not started yet, conserving resources. In the example, the GPS tracker is not started, since the display uses the optical tracker.

Avoiding Deadlocks In any distributed system, deadlocks are an issue. If there is a cyclic dependency between Services, the DWARF middleware will not deadlock—it will simply start the Services up in an unspecified order, and connect them together. The Services themselves may still deadlock, however—for example, if Services need to have data from other Services before they can send data themselves, and this dependency forms a cycle. Therefore, when designing DWARF Services, such dependencies should be considered carefully.

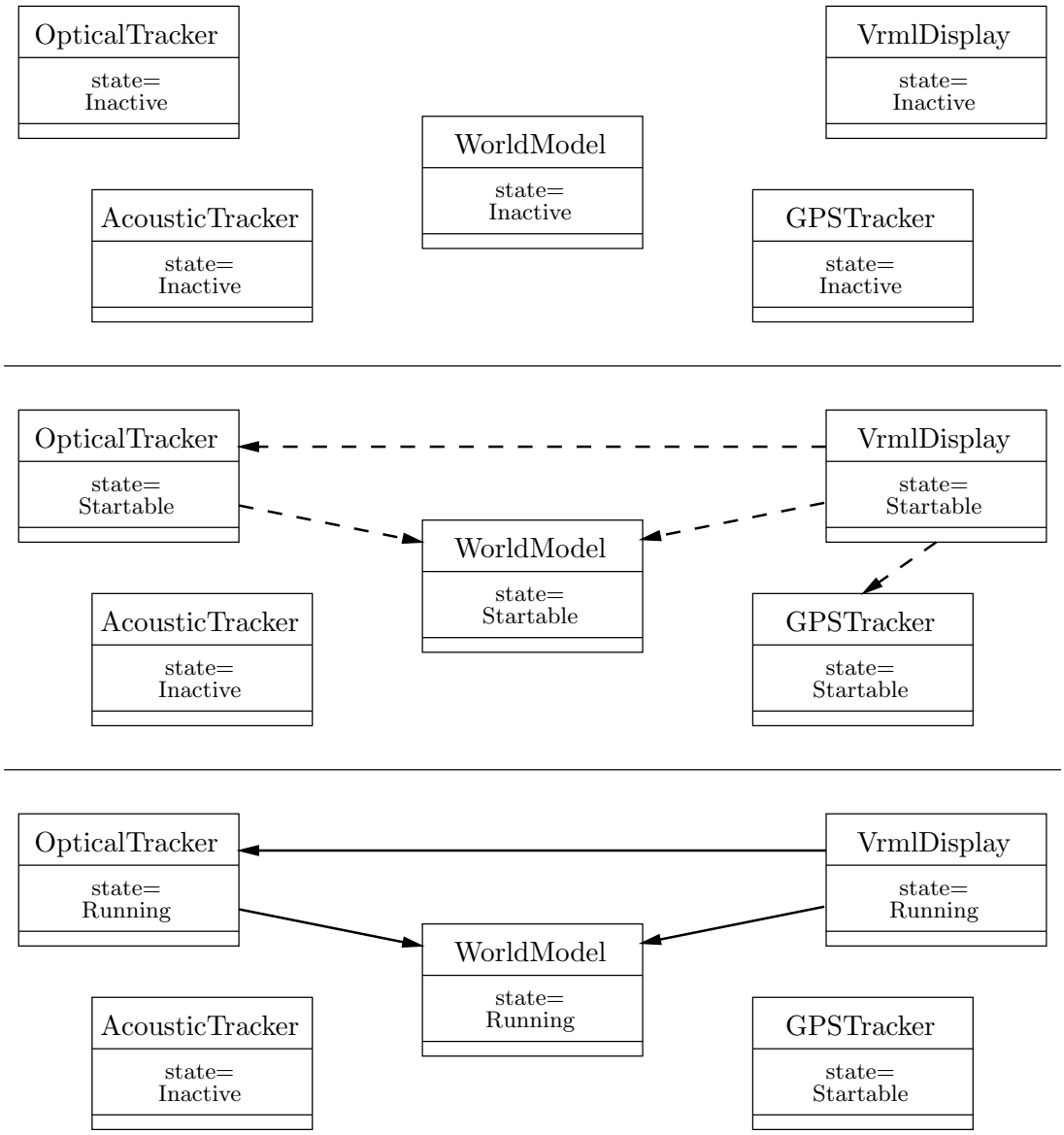


Figure 3.9: Traversals of the Service dependency graph.
The VRML Display needs position data and a World Model; the Optical Tracker needs a World Model; the GPS Tracker does not need anything, and the Acoustic Tracker needs a (nonexistent) Audio Data Service.
Before the first traversal (top), in which the Services are located, all Services are unsatisfiable.
After the first traversal (center), some Services become startable, since their Needs can be satisfied by other Service's Abilities (dashed arrows).
When the display is started, the second traversal occurs, after which (bottom) the optical tracker and the world model are started on demand, and the Services use each other's Abilities (solid arrows). The GPS tracker is not started, as it is less accurate.

4 Survey of Middleware Technology

Currently available technology can help in finding services and communicating with them.

With the requirements for the DWARF middleware in mind, this chapter examines various existing middleware technologies. Some of them can be used as-is as part of the DWARF middleware, and others can at least serve as an inspiration for the system design.

The subsections of this chapter vary in length and in detail. Some technologies are only examined briefly; others, the ones that I used in DWARF, are explained in more detail, including test results of various implementations.

The middleware technologies I examine fall roughly into two categories: communication infrastructure (Section 4.1) and service location mechanisms (Section 4.2).

4.1 Communication Middleware in Distributed Systems

One major issue for the middleware in distributed systems is to let the components communicate with one another. Various middleware technologies are available for this task.

4.1.1 CORBA

The Common Object Request Broker Architecture (CORBA) standard defines a mechanism allowing two applications which are written in entirely different languages and running on an entirely different computers and operating systems to communicate by calling the methods of each other's objects. The CORBA standard is developed and maintained by the Object Management Group (OMG) [47], the world's largest software consortium, which was founded in 1989 and is supported by nearly the entire software industry. CORBA 2.3 is the version of the standard for most implementations—the CORBA 2.4.2 specification [11] is available, but there are hardly any implementations of it.

Overview

I will not attempt to explain CORBA in detail; for this, see, for example, [24] or [52]. This section serves only as a brief overview.

Interfaces and IDL Objects in CORBA are accessed by way of *interfaces*. An interface consists of a group of well-defined, related operations. For example, a Service in DWARF has a `Service` interface, with the `startService` and `stopService` operations.

Interfaces in CORBA are described in Interface Description Language (IDL), a language very much like Java or C++. IDL allows the definition of complex custom types, sequences, etc. For an example of IDL, see Section A.1.

Clients and Stubs Objects that call methods of a CORBA interface are called clients.

From an interface definition in IDL, an *IDL compiler* generates stub code in the language that the client application is written in. CORBA specifies standard mappings from IDL to many target languages. For example, compiling the `Service` interface to Java stubs results in Java classes implementing the (Java, not IDL) `Service`, `ServiceOperations` and `ServiceHelper` interfaces.

When the client application calls a method of a stub object, the stub serializes the parameters, sends them over the network to a remote server, unpacks the results and returns again. The client application has the impression that it is calling a method on a local object. Of course, the object may also be local, in which case the stub simply performs a local method call itself.

Servants and Skeletons Objects that implement a CORBA interface are called *servants*, and the processes they run in are called servers.

On the server side of the network connection, the IDL compiler generates skeleton classes in the server application's language. The server application derives its own servant classes from these skeleton classes to implement the interface.

When a remote method call arrives over the network, the skeleton object receives and unpacks the parameters, calls the servant object locally, and sends the return value back over the network. The servant object has the impression of having been called by a local object.

ORBs The stub and skeleton classes are not self-contained; they are implemented using an Object Request Broker (ORB). This is a library which is linked to the client or server application, and provides the basic functionality of translating local calls into network requests and back.

Every application that wishes to use CORBA must use an ORB. There are ORB implementations for nearly every programming language and operating system combination. The CORBA standard defines a standard interface for ORBs, so that an application written using one ORB can (more or less) easily be ported to another.

Communication Protocol Remote method calls are translated to network messages between ORBs, and for this, CORBA defines the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP), which is based on TCP[1]/IP. Almost all ORB implementations "speak" IIOP, so it is possible for a client application to call a method on a server object written in an entirely different language and running on an entirely different operating system.

Resource Issues Communicating using CORBA has a certain overhead over using, raw network sockets. IIOP itself is very fast and compact, but the extra ORB libraries increase the size of the application.

However, ORBs of different sizes are available, and there is even a *MinimumCORBA* [11] specification for resource-constrained ORBs. One such implementation is e*ORB [20],

[1] Transmission Control Protocol

a C++ ORB for PalmOS. The features in MinimumCORBA are enough for the purposes of DWARF—we do not use CORBA transactions, security, etc.—so DWARF should be able to use CORBA even on very small wearables. See Section 5.4.2 for extra measures to keep the middleware on such systems to a minimum.

Tested ORBs

For DWARF, we used various different ORBs, and during the project, I tested even more. Both commercial and free ORBs are available, and we started with the free ones.

MICO [44] stands for "Mico Is CORBA", and is an open source C++ ORB for Unix and Windows platforms. It is compliant with CORBA 2.3, quite lightweight, easy to use, well-documented and free. Unfortunately, it is not multithreaded—which severely limits its use in many DWARF Services and especially in the middleware. However, for very small DWARF services, it might be appropriate, since it does not consume many resources.

TAO [71] is an open source C++ ORB developed at the University of Washington which is based on ACE, the Adaptive Computing Environment, which is a highly optimized, yet platform-independent library for communication and synchronization in distributed systems. TAO runs on nearly every platform that supports real multitasking. It uses design patterns implemented with C++ templates quite heavily, and thus requires a very long time to compile. TAO is fully multithreaded and quite fast, so it was a close candidate for use in DWARF. Unfortunately, it currently does not run well on Linux for PowerPC processors.

OmniORB [49] is an open source C++ ORB developed by the AT&T research labs. It is available for Windows and Unix platforms, fully CORBA 2.3 compliant, fully multithreaded, and very fast. Indeed, it has tested as one of the fastest ORBs available [12]. It is much more lightweight than TAO, runs stably on our development and deployment platforms, and supports all the features we needed. This is why we used it for all C++ code in the current implementation of DWARF.

ORBacus [50] is a commercial ORB for C++ and Java by Object Oriented Concepts, with a free license for noncommercial use. Like OmniORB, it is fairly lightweight and fully multithreaded, but it is not quite as fast as OmniORB, and requires run-time royalties for commercial use.

VisiBroker [76] is Borland's commercial ORB for C++ and Java, with a free evaluation license. This also requires run-time royalties, and was only evaluated briefly.

JavaORB [31] is a free ORB for Java by the Distributed Objects Group. It is fully multithreaded and supports the CORBA 2.3 standard. It is reasonably fast and small, and will also run inside applets within a browser's sandbox. We used this as an ORB for the DWARF Services written in Java, although it would be nice to find an ORB in the future that does not suffer from JavaORB's threading and interoperability problems.

JacORB [28] is another free ORB for Java, which is, however, not quite as fully developed as JavaORB. It had problems compiling user-defined IDL types.

Java 2 ORB [30] is part of Sun's Java 2 platform by default, but was not CORBA 2.3 compliant until very recently, and will only run in the Sun virtual machine. Thus, we opted not to use it.

Interoperability

We discovered a few "interesting" interoperability problems between the ORBs we tested. Here is one example.

Typedefs Between OmniORB and JavaORB In DWARF, we defined a structure `PositionEvent` that describes an object's position at a given time. When passing this as a return value from a C++ program using OmniORB to a Java program using JavaORB, the following definition did not work:

```
typedef unsigned long ThingID;

struct PositionEvent {
    ThingID thing;
    ThingID relativeTo;
    ...
};
```

The Java application would cause an exception, although the data was passed correctly over the network in IIOP format. To fix this, we had to change the IDL to:

```
typedef unsigned long ThingID;
typedef unsigned long RelativeToThingID;

struct PositionEvent {
    ThingID thing;
    RelativeToThingID relativeTo;
    ...
};
```

4.1.2 CORBA Notification Service

The *CORBA Notification Service* [14] is an OMG standard defining interfaces for accessing an event service. It is the successor to the CORBA Event Service [13] standard. Several implementations are available.

Overview

The CORBA Notification Service supports two methods of event communication: *push*-style events, in which the event supplier sends data to the event consumer, and *pull*-style events, in which the event supplier requests data from the event consumer. The push-style events are what one normally thinks of as events, and we did not use pull-style events in DWARF.

Structured Events There are three kinds of events: untyped events, typed events and structured events. Untyped events represent any kind of data; typed events represent calling specialized interfaces. A structured event, the most interesting event type, consists of:

- an event type string, e.g. `PositionEvent`
- an event name string, e.g. `where I was last night`
- a list of name-value attributes of standard headers, such as `priority=10`
- a list of name-value attributes called *filterable data*, e.g. `person="me",date="monday"`
- a *remainder of body*, a CORBA *Any*, which can hold any data, e.g. a DWARF `PositionEvent` structure.

The standard defines two relevant interfaces: a `StructuredPushSupplier` pushes structured events to a `StructuredPushConsumer` via a `push_structured_event` method. For details, see [14].

Event Channels Events are transmitted via event channels, which act like pipes. They consume events that come from the `StructuredPushSupplier`, and in turn supply them to the `StructuredPushConsumer`. For this, an event channel implements the `StructuredProxyPushConsumer` interface, which the supplier sends its events to, and the `StructuredProxyPushSupplier` interface, which sends the events to the consumer.

These interfaces are designed so that event channels can be composed, i.e. data can be sent through several event channels connected in series. Event channels transmit data from every supplier to all connected consumers. The standard also allows consumers to specify filters so that they only receive events they are interested in.

Event Service An event service implements the `EventChannelFactory` interface, which creates event channels. These event channels must then be connected to the suppliers and consumers so that they can communicate.

Note that suppliers and consumers can be connected to event channels by an external entity; they do not have to connect themselves. This is crucial for composing event channels.

Tested Implementations

Several Notification Service implementations are available today, and in the DWARF project, I tested nearly all of them.

OmniNotify [48] is a fairly new open source Notification Service implementation, written, like OmniORB, by AT&T research. OmniNotify is written in C++, and currently only available for Unix platforms. It is highly optimized, since it was adapted from an earlier (non-CORBA-compliant) event service [22]. We used OmniNotify for development on Linux PowerPC. OmniNotify is by far the fastest of these implementations. The current version (1.02) still has minor instabilities—it tends to crash when several hundred event channels are used simultaneously. Also, it is not available for Windows platforms, and manually porting the source to Windows did not produce the desired results.

ORBacus Notify [51] is an open source, but not royalty-free Notification Service implementation that uses ORBacus. It is written in C++, and compiles on many platforms, including Windows and Unix. It is reasonably fast, but had one of the strangest interoperability problems I found in the project, see below. Also, it does not allow event channels from the same notification service to be composed together, which is unfortunate—this ability was cited as a major design goal in [13].

dCon [18] is a commercial Notification Service implementation that uses VisiBroker, has a time-limited evaluation license, and is available for many platforms. It uses Java, and therefore has very high memory requirements, but is still reasonably fast. We used dCon on Windows for our demonstration system.

Interoperability

We discovered one very strange interoperability issue within the last days before the DWARF demonstration system was supposed to be deployed.

We sent user-defined structures, such as the DWARF `PositionEvent`, in the "remainder of body" portion of structured events, according to the specification. This worked flawlessly with OmniORB and dCon. When using ORBacus, however, data sent from a program using OmniORB to one using JavaORB, or between two programs using JavaORB, was corrupted. Strangely enough, programs using OmniORB could communicate fine using ORBacus, and programs using JavaORB could communicate fine using OmniNotify or dCon.

We did not discover the cause of this problem—it seems that it might involve slightly different implementations of the CORBA Dynamic Any specification between ORBs.

4.1.3 CORBA Audio/Video Streaming Service

Like the CORBA Notification Service, the CORBA Audio/Video Streaming Service [16] is a fairly new specification. Its basic idea is to allow for higher-bandwidth communication than CORBA method invocations over IIOP. The streaming data can be sent by raw TCP or other protocols, and there is a standardized CORBA interface for controlling the communication, for operations such as `start`, `stop` and `rewind`. Currently, an implementation using TAO [70] is being developed.

We do not use this specification in DWARF, but it should be considered as a method for sending audio or video data between DWARF Services in the future.

4.1.4 COM

The Component Object Model (COM) [9] consists of various technologies for object-oriented interprocess communication developed by Microsoft. COM+ is an extension to COM and DCOM (Distributed COM) that includes transactions, security and other features that had been previously missing.

COM follows a different architectural approach than CORBA: it has a more detailed binary specification including specific optimizations for programming using C++ on Windows platforms. Thus, for pure Windows desktop applications, it is faster and more robust than CORBA; but is is not a good choice for multi-platform systems. See [10] for a concise comparison of COM and CORBA.

In DWARF, cross-platform compatibility was a major design goal. For the middleware, the development platform was Linux, although the deployment platform was Windows; thus, we chose CORBA over COM.

4.1.5 Java Remote Method Invocation (RMI)

For communication between programs using Java, Java Remote Method Invocation (RMI) is an alternative to CORBA. RMI is built in to Java, and allows the invocation of methods on remote objects.

In contrast to CORBA, RMI passes objects by value, serializing them in the process. This makes it easier to transfer complex data structures across the network, but requires more bandwidth.

Of course, Java RMI only works with Java. Some DWARF services, such as the optical tracker, are written in C++, which was the main reason for not using Java RMI in DWARF.

With the development of the Java 2 ORB, Sun itself is planning on migrating Java RMI to CORBA; they call this "RMI over IIOP". [30]

4.1.6 Low-Level Protocols

When speed or latency is an issue, DWARF Services might want to use low-level communication protocols to exchange data. Of course, this is always platform-dependent, and complex data has to be converted into a communication format by hand.

Shared Memory can be used easily on Windows and Unix platforms; for example, to exchange video data between an optical tracker and an OpenGL-based renderer. Unfortunately, Windows and Unix have different API[2]s for managing an accessing shared memory, so the code would be hard to port.

Serial Connections are the method of choice for accessing small devices like GPS receivers. They could also be used to transmit position data between a tracking and a rendering computer, as this data requires very little bandwidth, and a serial connection provides a guaranteed bandwidth and latency. This could be used without any problems in DWARF.

Raw UDP[3] could be used for sending position data which does not have to arrive in order, but needs a low latency. Using multicasts, this data could be distributed to different displays. This is not used in DWARF, but could be investigated as a method of sending events.

Raw TCP can be used when medium or large amounts of data need to be transferred in streaming format, such as for XML descriptions. For this type of application, however, a protocol such as HTTP[4] would be preferable.

[2] Application Programming Interface
[3] User Datagram Protocol
[4] Hypertext Transport Protocol

4.2 Service Location

Service location is an active area of middleware development. Several new protocols have been developed in the last few years to solve the problem of finding services in a dynamically changing environment.

This section describes a few of these technologies. For a good comparison (in German), see [33].

4.2.1 Service Location Protocol (SLP)

SLP [63], the Service Location Protocol, is a simple protocol for finding services. Its functionality is limited—in contrast to several other protocols described here, it cannot do anything besides finding services.

I will describe SLP in more detail than the other protocols in this chapter, since I chose to use SLP in designing the DWARF middleware, and many other protocols are similar in nature. An extended version of this section appears in [41].

Overview

Scope SLP can locate services in a network by service type and by attributes. It returns a URL[5] indicating the service's location to the client. SLP does not provide any additional mechanisms for configuring or communicating with a service—once a client has the service URL, it must contact the service itself, using the appropriate protocol. SLP is primarily designed to fit enterprise-sized networks with shared services. It includes mechanisms for scaling to larger networks, but has not primarily been designed for wide-area service discovery throughout the internet.

History Version 1 of the Service Location Protocol was first specified by the IETF[6] in 1997 [75]. This was updated to version 2 in 1999 [23], when an API was specified as well [34]. The IETF task force for developing SLP has completed its charter as of January 2001.

Supporters Supporters of SLP include Novell, which uses SLP in Netware 5 [67], Axis, which uses SLP in its this server products [66], and Apple, which uses SLP as an integral part of Mac OS X [64, 40]. Although the initial design of SLP was carried out at Sun Microsystems [53], Sun is not actively campaigning for SLP, presumably because they want to avoid competition for Jini.

Design Rationale Like many Protocols supported by the IETF, SLP is designed to be just large enough to fit the task at hand—in this case, service location. Thus, the IETF expressly excludes mechanisms for service access or configuration. SLP is designed to be simple to implement, even with limited hardware, yet provide a flexible and scalable mechanism for service location.

[5] Uniform Resource Locator
[6] Internet Engineering Task Force

Availability A reference implementation of SLP version 2 is available from Sun Microsystems [68]. This includes several testing tools and a C language API for Linux, Solaris and Windows. I have briefly tested this implementation, and it seems to work. An implementation for Windows operating systems is available from [65]. An open source implementation, OpenSLP, is also under development [62]. Other implementations, mostly still work in progress, are available from academic institutions, such as [56].

Protocol

Deployment SLP defines the interaction between three participants. Basically, a User Agent wishing to use a particular service tries to find a service offered by a Service Agent. Additional Directory Agents can improve scalability.

Quoting from [23]:

User Agent (UA) A process working on the user's behalf to acquire service attributes and configuration. The User Agent retrieves service information from the Service Agents or Directory Agents.

Service Agent (SA) A process working on the behalf of one or more services to advertise service attributes and configuration.

Directory Agent (DA) A process which collects information from Service Agents to provide a single repository of service information in order to centralize it for efficient access by User Agents. There can only be one DA present per given host.

SLP defines two modes of operation: with and without Directory Agents. In the simpler version without Directory Agents, User Agents communicate with Service Agents using a multicast network protocol (see Figure 4.1). If a Directory Agent is present, it acts like a caching proxy. User Agents and Service Agents communicate directly with the Directory Agent, eliminating the Need for network multicasts (Figure 4.2 on the following page).

User Agents and Service Agents are separate from the client application and the service itself. Once the client application has retrieved a service's location from the User Agent, it must contact the service directly, using some common network protocol.

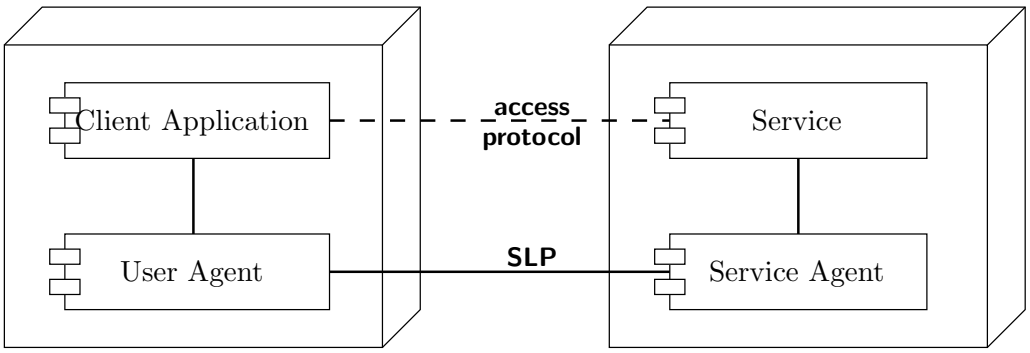


Figure 4.1: Deployment of SLP subsystems without Directory Agents

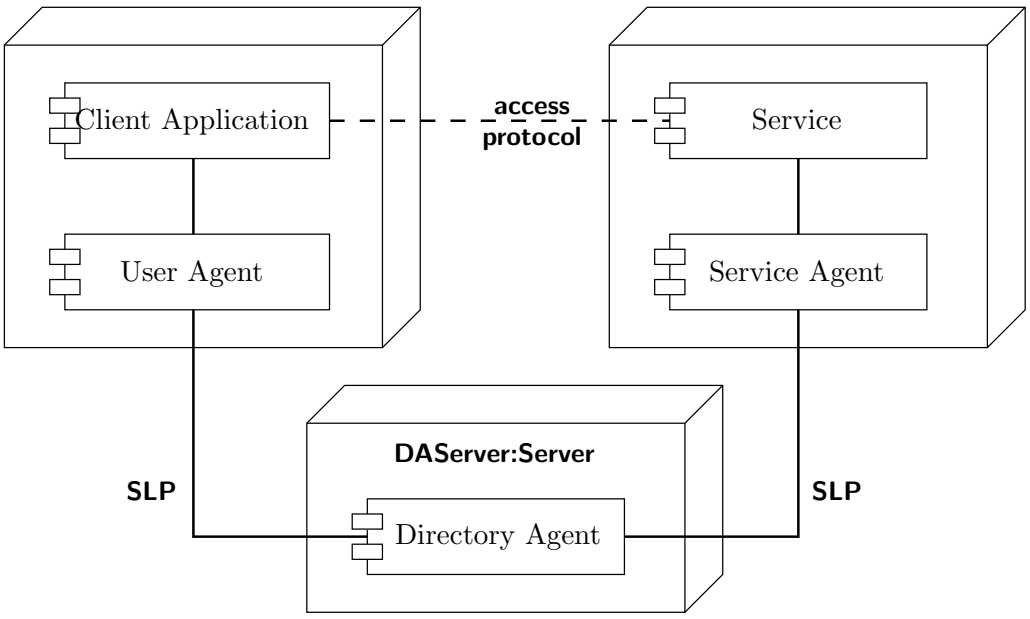


Figure 4.2: Deployment of SLP subsystems with Directory Agents

Services Services in SLP have a *service type* and a list of *attributes*. The type consists of two parts: an abstract type (e.g. `printer`), and an optional concrete type (e.g. `lpr`), specifying the communication protocol that should be used to access the service. A User Agent can search for services either by abstract type (any printers at all) or by full type (printers supporting the `lpr` protocol).

The service type is represented as the string

`service:`*`abstracttype`*`:`*`concretetype`*, e.g.

`service:printer:lpr`.

Attributes in SLP have a name (of type string) and a value (of type boolean, integer, string or raw binary). These attributes can be queried explictly by User Agents, and they also serve as a means of filtering services.

Every service also has a location, described below.

Finding Services User Agents and Service Agents communicate by sending messages back and forth. The most basic interaction, when no Directory Agent is present, is shown in Figure 4.3 on the next page.

A User Agent finds services by sending a Service Request (`SrvRqst`) to a well-defined multicast address (in local networks, this is like a broadcast). This contains a type specification and a predicate describing the attributes the service is supposed to have. This predicate, in LDAP[7] format, is essentially a boolean expression.

For example, a query for a fast and accurate position tracking device might specify the type `PositionData` and have a predicate of `(&(lag<=3)(accuracy>=10))`.

When a Service Agents receives a `SrvRqst` with a type and predicate that it has a matching service for, it answers with a Service Reply (`SrvRply`), containing the location of the service.

[7] Lightweight Directory Access Protocol

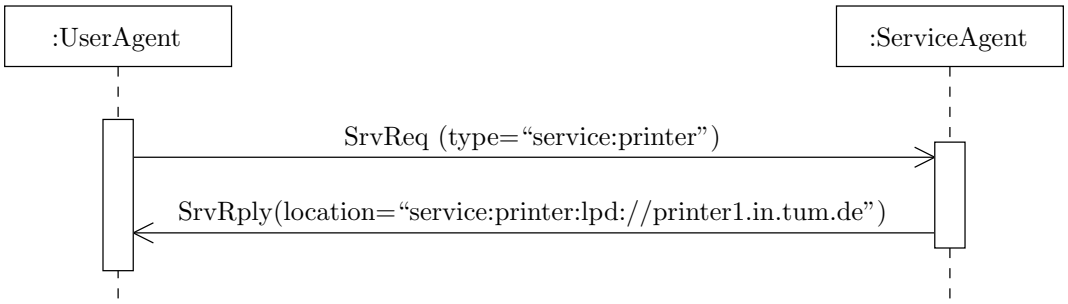


Figure 4.3: Locating a service with SLP

This is sent directly (not via multicast) to the User Agent that requested the service.

Service Locations The location of a service returned by such a request is specified in the form of a URL. This can be a normal HTTP URL, for example. However, [23] suggests the use of "Service URLs", for standardization. This is an URL of the form

`service:`*`abstracttype`*`:`*`concretetype`*`://`*`address`*`.`

An example for the return value of the above request could be

`service:PositionData:http://fasttracker.in.tum.de:1234/`.

Note that the protocol specification `http` would not necessarily be the protocol of choice for transmitting position data.

Network protocol Messages in SLP are generally sent via UDP. This has a low overhead, and allows multicasting to be used. [23] defines the format in which SLP messages are encoded into UDP packets.

If a message is too large to fit into a single UDP packet (i.e. greater than 1400 bytes), the User Agent will then open a TCP connection to the Service Agent and retransmit its request via TCP, which can deal with arbitrary message sizes.

If a User Agent does not receive an answer to its service request message (or not enough answers), it retransmits the request, up to a certain timeout. To prevent Service Agents from responding to these requests again, each `SrvRqst` message contains a "previous responder list", which consists of the IP addresses of the Service Agents that have answered so far.

Scaling with Directory Agents In larger networks, multicast and broadcast messages can cause an undue amount of traffic. To alleviate this, SLP allows the use of Directory Agents, which reduce an n-to-m relationship to two n-to-1 and m-to-1 relationships.

An additional advantage of this architecture is that multiple Directory Agents can be connected into a large federation via LDAP, allowing far-away services to be located without having to flood the network with broadcasts.

Upon startup, User Agents and Service Agents first try to discover a Directory Agent. This is done by sending a `SrvRqst` message for the type `service:directory-agent`. It is also possible for a system administrator to configure the address of a default Directory Agent, or for the address to be discovered via DHCP[8].

[8] Dynamic Host Configuration Protocol

When a Service Agent has discovered a Directory Agent, it registers its services with it, using service registration messages (`SrvReg`). It sends these by unicast to the Directory Agent, which responds with a `SrvAck` message. The `SrvReg` message contains the service type, attributes and location, as described on page 63.

When a User Agent has discovered a Directory Agent, it sends its `SrvRqst` messages by unicast directly to the Directory Agent, which responds with `SrvRply` messages in place of the Service Agents.

Service Browsing SLP defines several "optional" messages, which support browsing of services in a network, or inquiry of specific attributes.

User Agents can multicast (or send to the Directory Agent) `SrvTypeRqst` messages, Service Type Requests. This is a request for all service types in the network. The Directory or Service Agents respond with `SrvTypeRply` messages, which contain lists of service types.

Another message, the Attribute Request (`AttrRqst`), lets User Agents query all or specific attributes of a service. The Service Agent or Directory Agent responds with `AttrRply` messages.

Administrative Scopes For administrative purposes, SLP defines a "service scope" (e.g. `TUM-Informatik`) that can be used to divide clients and services into different administrative responsibilities, such as departments or faculties.

Internationalization SLP provides an additional "language tag" field in all messages, so that users can browse services and select services in their preferred language. Service Agents will attempt to answer `SrvRqst` messages with `SrvRply` messages in the given language tag.

Security SLP provides a simple and extensible security mechanism in the form of authentication blocks. Each SLP message can optionally contain a number of authentication blocks, which serve to identify the sending Agent. This can, for example, be a digital signature.

This lets the User Agents verify the digital signature of `SrvRply` blocks, so that client applications will only access trustworthy services.

Discussion

SLP is fast, powerful and has the advantage of being an open standard. It is a simple protocol that does exactly one thing—service location—and does it well. In areas where that is all one needs, it fits perfectly. If further services are needed, however, a more complex protocol suite such as UPnP, described in the next section, may be appropriate.

4.2.2 Universal Plug and Play (UPnP)

Universal Plug and Play (UPnP) [73] was developed mainly by Microsoft and consists of a whole suite of protocols for connecting and configuring small devices. UPnP is mainly designed for home and office networks, e.g. connecting printers and scanners to computers. UPnP is currently still being developed—the specification is not quite stable, and no development kits are available.

Service Location For locating services, UPnP uses the Simple Service Discovery Protocol (SSDP). This is quite similar in design to SLP, and can also use the equivalent of Directory Agents, but it does not allow searching for a service based on its attributes. SSDP returns the URL of a service description.

Service Description Services in UPnP are described in an XML dialect, and the corresponding XML file is placed on a *description server*. The XML description contains, for example, the device's manufacturer, model number, or a textual description for the user.

Service Control UPnP defines an additional layer, the Service Control Protocol (SCP). This is essentially a remote procedure call mechanism using XML over HTTP. Services define the interface they support in an *SCP Declaration*, and the client can invoke methods by sending short requests encoded in XML.

Event Architecture UPnP defines a publish-and subscribe mechanism for events generated when a service's status changes, such as a printer running out of paper. These events are also encoded in XML.

Discussion UPnP is intended for small home or office networks, and cannot scale as well as SLP. However, for these small networks, it offers much more than SLP does, since it defines methods of accessing services, and not only finding them. UPnP will probably be successful in its target market, but so far, no implementations are available.

4.2.3 Jini

Jini [32, 78] was developed by Sun Microsystems for service discovery and use in local networks. It is designed for the Java 2 platform, and provides a complete API for both clients and services.

Service Location Unlike SLP and SSDP, Jini will not operate without a central *Lookup Service*, which is analogous to the Directory Agent in SLP. Services register themselves with the Lookup Service, Clients find the Lookup Service by broadcast and query it for service information.

Service Access The interesting feature of Jini is how clients access services. Services upload a *service proxy* to the Lookup Service, which is a serialized Java object and the associated class file. When a client finds a service it wishes to use, it downloads the proxy, unserializes it, and loads the class into its own virtual machine.

In order to access the service, the client calls methods of the service proxy. This then handles the network communication with the service itself. The client does not have to deal with network communication at all, and the proxy and the service can communicate using whichever custom protocol the service desires.

Discussion The idea behind Jini's service proxies is intriguing, since it makes developing clients very easy and allows services to be very flexible. Of course, Jini only will work if object code can be migrated from the service to the client, as the Java platform allows. Since

Jini requires many resource-intensive features of the Java 2 virtual machine, it will be difficult to make it run on very small devices.

4.2.4 CORBA Trader Service

The *CORBA Trader Service* [15] finds and matches services based on CORBA interfaces, but does not discover services in ad hoc networks.

The Trader service specification defines services as collections of related CORBA interfaces, with named attributes. Services register a themselves with a Trader Service, and clients query the Trader Service in order to find the services. Clients can specify, in a *Trader Constraint Language*, attributes that the services should have. Once services find each other, they communicate using CORBA interfaces.

Trader Services can be connected together into a *Trader Federation*, which allows queries to propagate through a large-scale network.

Unfortunately, the Trader Service specification does not address ad hoc networks; it assumes that the Traders know each other and that the clients and services know their respective Traders. Thus, an implementation of the CORBA Trader Service would have to make use of a service discovery protocol such as SLP.

4.2.5 DEAPspace

DEAPspace [25] is a research project at IBM Labs aiming at finding services in small wireless networks quickly.

The project includes a detailed analysis of the time it takes for a service to be discovered. This should be as small as possible, so that a wearable computer can wirelessly access stationary devices as the user walks past them. For this, a special service discovery algorithm was developed, in which every node of the small wireless network regularly broadcasts information about itself and any other services it knows. This way, the information about which services are where spreads through the network very quickly.

DEAPspace is an interesting research project, and has resulted in a small implementation for testing the discovery algorithms. So far, however, it does not look like it will develop into a commercial product.

5 System Design for the DWARF Middleware

I have designed a middleware system for DWARF, using both new and off-the-shelf components.

This chapter contains the design of the DWARF middleware, the core of my thesis. After describing the important design goals, I present an overview of the system design. Here, I introduce the concept of *distributed mediating agents*, which implement the middleware's functionality without relying on a central component. Next, I divide the DWARF middleware into subsystems for communication between Services, locating Services and and managing Services. Then, I investigate such issues as security, global software control, initialization and shutdown. Finally, I describe the functionality provided by each middleware subsystem in detail, and show how these interact with the rest of the DWARF system.

5.1 Design Goals

In the design of the DWARF middleware, I had to pursue several sometimes conflicting goals. Most of these stem from the nonfunctional requirements listed in Section 3.3, but some come from the development process of the DWARF system.

My main design goals, divided into performance, dependability, maintenance and cost criteria, were:

Performance Criteria

Low Overhead The middleware should consume few memory and processing resources, so it can run on small, even wearable, systems.

Low Latency The latency of event-based communication should be as low as possible.

Response Time The middleware should find and connect new Services together quickly.

Dependability Criteria

Availability The middleware should run stably.

Reliability The middleware should perform its tasks as specified.

Robustness The middleware should tolerate being used incorrectly by faulty Services.

Fault Tolerance The middleware should handle network error conditions gracefully.

Maintenance Criteria

Portability The middleware should be easily portable to different platforms.

Extensibility It should be easy to add new functionality to the middleware.

Adaptability It should be adaptable to other application domains than AR.

Scalability It should scale to many networked hosts and high volumes of data.

Cost Criteria

Development Cost The first version of the middleware had to be implemented, by one developer, in time for the other framework components to be tested and our demonstration system to be built.

Trade-Offs Some trade-offs were necessary between the different possible design goals.

Functionality vs. Development Cost and Reliability Some of the functionality, such as starting Services on demand, SLP and XML support, are only specified in the design, but have not been implemented. None of these features is inherently difficult, but it was more important that other basic functionality, such as communication, be implemented reliably.

Security vs. Development Cost and Low Overhead Security issues were not investigated in detail in the design of the middleware, since this would have unduly increased design and development time, and would have added extra implementation overhead.

Portability vs. Low Overhead The middleware uses CORBA, which increases platform independence, but adds some implementation overhead.
On the other hand, it uses standard C++ rather than Java, which slightly reduces portability (the middleware currently runs on Windows and Linux, and should easily port to Mac OS X and other Unix systems), but greatly reduces run-time memory and processor requirements.

5.2 Overview

Here I will introduce the overall design of the DWARF middleware components and explain how it evolved from different concepts. From an AR system's point of view, the basic functionality is that of a *mediator* from the *mediator design pattern*, but additional steps were necessary to make this functionality available in a distributed system.

Distributed Mediating Agents The mediator pattern of the middleware in the DWARF system has one obvious disadvantage for distributed systems: since the mediator has to know all other subsystems that it connects, it could become a central component that cannot easily be distributed onto different network nodes. This would limit the user's ability to turn off arbitrary hardware components—if the mediator is turned off, the whole system falls apart. It would also be at odds with the *dependability* design goals.

The solution that I chose for this problem is based on the idea of *Distributed Mediating Agents*.

The term *agent* is somewhat overused in software technology. Here, I use it to mean software components running on the various computers in a network that proactively communicate with one another on behalf of other components.

It turns out that the functionality of mediating between the DWARF Services can be achieved just as well by distributed mediating agents as by a central mediator. These local

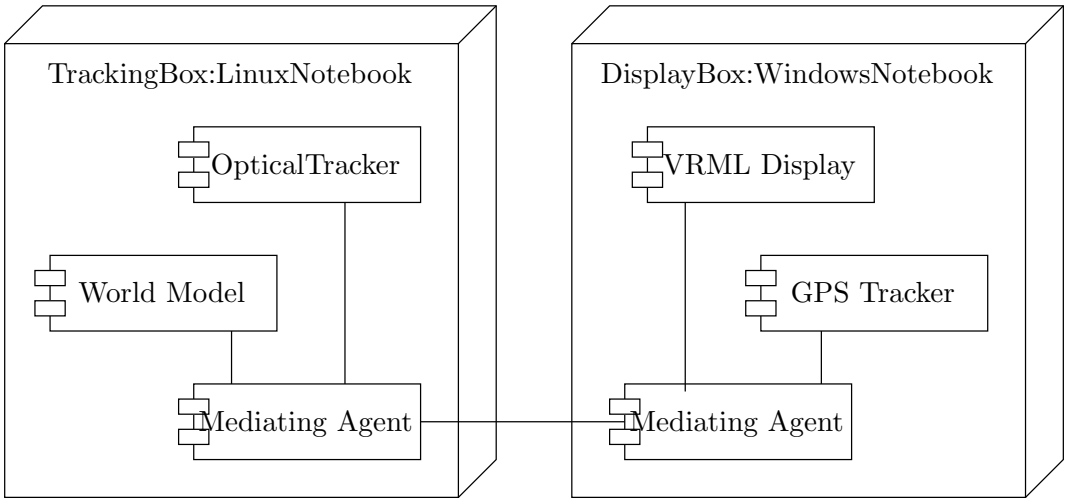


Figure 5.1: Deployment of Distributed Mediating Agents.
Each network node has its own Mediating Agent. The DWARF Services communicate locally with the Mediating Agents, which handle network communication. Compare this to Figure 3.1 on page 30.

mediating agents obviously have to communicate with one another, on a peer-to-peer basis, in order to set up communication between Services running on different nodes of the network. As far as the Services that use the mediating agents are concerned, these agents collectively act as a single mediator. The deployment of Distributed Mediating Agents is shown in Figure 5.1.

The middleware that I have designed is somewhat more "active" than many other middleware systems, in that the mediating agents proactively search for other Services, create connections, start and stop Services, and so on. Thus, these agents must be implemented (see section 6) as separate processes, not simply as a library whose functions are called by the DWARF Services.

5.3 Subsystem Decomposition

The Mediating Agents of the DWARF middleware can be divided into three distinct subsystems, as shown in Figure 5.2 on the next page.

Communication Subsystem Encapsulates the functionality of communication between known Services. This includes network communication, but also interprocess communication on single machines.

Location Subsystem Advertises and finds Services that are distributed in a network, so that they can communicate with one another. Note that the term *location* refers to logical locations such as network addresses, not to physical *positions*.

Service Manager Serves as an initial access point for the rest of the middleware, and acts on behalf of the DWARF Services. Maintains descriptions of active and inactive Services. Starts and stops Services on demand. Coordinates between communication and location subsystems and the DWARF Services.

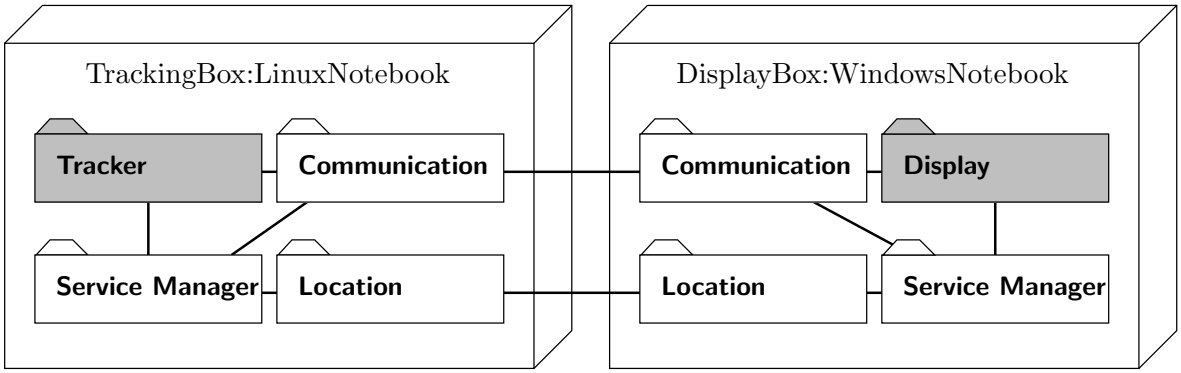


Figure 5.2: Subsystems of the DWARF middleware.
In this example, two DWARF Services are shown in grey. The display needs position data, which the tracker can deliver. The communication and location subsystems communicate across network boundaries, the Service Manager and the DWARF Services do not.

Design Rationale There are several reasons for this division into three subsystems.

The two areas of functionality of the middleware, finding Services and letting them communicate, have widely different nonfunctional requirements: communication has to be *fast* (especially for AR), whereas location has to be *flexible*. By using two different subsystems for communication and location, these distinct functionalities are separated into distinct subsystems. This makes it possible to optimize separately for speed and for flexibility, and to use different existing components for Service location and communication between Services, e.g. SLP and the CORBA Notification Service.

Separating these two subsystems from the rest of the middleware also separates the subsystems that use network communication from those that do not. Thus, only the communication and location subsystems have to deal with low-level network communication and problems such as loss of connectivity. The Service Manager and the other DWARF Services only communicate locally, making them both simpler and more reliable.

The rest of the middleware is contained in the Service Manager. It needs to coordinate between the DWARF Services, the communication subsystem and the location subsystem. This includes managing Service descriptions, starting Services, and so on. In the future, it might be desirable to further subdivide the Service Manager into smaller subsystems (such as storage of Service Descriptions), but this added complexity is not necessary at this stage.

Subsystems and Objects Figure 5.3 on the following page shows how the objects identified during requirements analysis (Section 3.8) are distributed into the three subsystems. Note that the location subsystem does not contain any problem-domain objects, since the decision to create a separate location subsystem was made after requirements analysis.

I will now describe the responsibilities of these subsystems in more detail, and explain the relationship between subsystems and objects shown in Figure 5.3 on the next page.

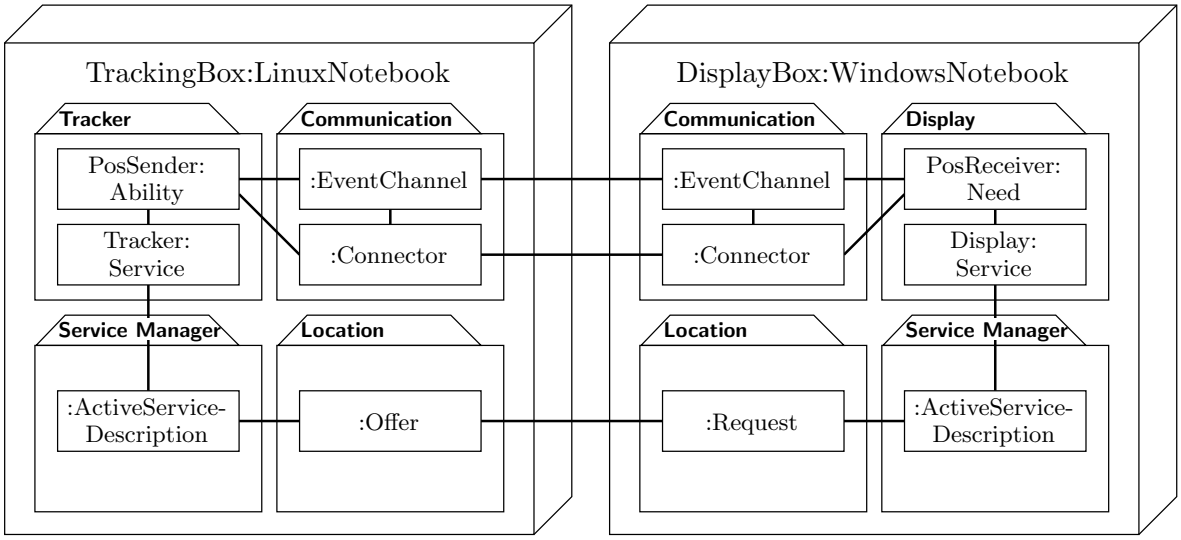


Figure 5.3: Subsystems and and the objects from requirements analysis.
As in Figure 5.2 on the page before, a tracker has the Ability to send position data, satisfying the display's Need. This also shows two newly identified objects in the location subsystem. Compare also with Figure 3.5 on page 48.

5.3.1 Communication Subsystem

The communication subsystem encapsulates the functionality of communication between Services. It manages communication resources such as event channels, shared memory blocks and TCP sockets.

Connectors The communication subsystem creates *Connectors* (page 47) when instructed to do so by the Service Manager. A Connector manages Communication Resources at one end of a connection between one Service's Need and another Service's Ability. The Service Manager can actively tell the Connector to connect to another Service, or it can tell it to passively wait for incoming connections. Once a connection has been established, the Service Manager passes the Service's Need or Ability a reference to the Connector.

The Service's Need or Ability then determines the protocol that the Connector supports, and extracts a reference to the protocol-specific communication resources from the Connector. It then accesses these communication resources directly, bypassing the Connector, to communicate with the other Service. The Service thus must be able to "speak" the protocol itself, but it does not have to deal with managing the infrastructure, e.g. allocating event channels or shared memory blocks. DWARF Services can access non-DWARF systems and vice versa, as long as a standard communication protocol, e.g. HTTP, is used.

A Service only receives one Connector per protocol for an Ability, even if more than one other Service connects to it to use that Ability. In contrast, a Service receives multiple Connectors for a Need if the Need is satisfied by connecting to multiple other Services.

Design Rationale As shown in Figure 5.3, the data flow between two Services goes directly through the communication resources (event channels), once it has been set up by the Con-

nector. The DWARF middleware thus causes no extra overhead once the connection has been established. This makes communication flexible and fast at the same time.

Since Connectors can be created before the Service is running, the middleware can accept connections on behalf of a Service and only then start the Service, saving resources.

5.3.2 Location Subsystem

The location subsystem provides the basic functionality of locating Services. It provides a fairly low level of abstraction, so that existing Service location technology (e.g. SLP) can be used to implement it.

Thus, the location subsystem does not know about communication protocols, connectors, Services, Needs, or Abilities. It simply deals with *Offers* and *Requests*. This gives us two new solution-domain objects:

Offers consist of a location and a set of attributes, and

Requests are predicates over these attributes.

The location subsystem periodically tries to find Offers to match its Requests, either from among its own Offers or by using network service discovery mechanisms to find Offers that other location subsystems have advertised.

The Service Manager creates Offers and Requests from each Service's description and registers them with the Service Locator, as shown in Figure 5.3 on the page before. It maps a Service's Abilities onto Offers, telling the location subsystem to advertise these on the network. Analogously, it maps a Service's Needs onto Requests, which the location subsystem tries to answer.

The communication protocol used between the location subsystems, SLP, is one main method of communication between the mediating agents on different network nodes. The two Service Managers do not communicate directly with one another.

Design Rationale The simple model of Offers and Requests that the location subsystem uses makes it easier to implement, especially because existing service location technologies use this model. Also, the location subsystem's functionality is just enough so that the Service Manager itself does not have to match up Needs and Abilities, simplifying the design of the Service Manager. Direct communication between the Service Managers of different mediating agents would not provide large enough benefits to make it worth the additional complexity.

5.3.3 Service Manager

The Service Manager provides the initial interface to the DWARF middleware. In this sense, it acts like a *facade pattern*. When a DWARF Service starts up, it has to be able to find the Service Manager, but once it finds that, it can access all other middleware functionality from there.

Active Service Descriptions Internally, the Service Manager consists of Active Service Descriptions, as identified on page 47. An Active Service Description exists throughout the life cycle of a Service and represents the Service within the Service Manager. It has a state reflecting the state of the Service (as shown on page 50).

Coordination The Service Manager coordinates the other subsystems. It creates Requests and Offers in the location subsystem to satisfy a Service's Needs and to make its Abilities available to other Services. It instructs the communication subsystem to connect the Services together that the location subsystem locates.

Description and Registration An administrator can describe a Service before the Service is started. Services are described by creating Service Descriptions, which describe a Service's attributes, its Needs and Abilities, and the communication protocols it supports.

When a Service starts, it registers itself with the Service Manager, which then associates the running Service with its stored description.

Starting and Stopping The Service Manager can create connectors for a Service's Abilities before the Service is even started. When such a Connector is connected to, the Service Manager can start the Service, potentially even loading it across the network (or even purchasing it first). When all other Services have disconnected themselves from the Service's Abilities, the Service Manager can shut down the Service. This start-on-demand and stop-on-no-use mechanism conserves resources.

5.4 Hardware/Software Mapping

This section describes the off-the-shelf software components that are part of the DWARF middleware, and the hardware that the middleware is designed to run on.

5.4.1 Third-Party Software Components

The DWARF middleware is designed to use available third-party middleware technology, as is described is in Chapter 4.

This includes CORBA ORBs as an interprocess communication mechanism, SLP for the location subsystem, and the CORBA Notification Service as an event-based communication protocol.

CORBA for Interprocess Communication

The various DWARF Services and the middleware are written in different languages, depending on speed and portability requirements, and they each have their own control flows. This means that the middleware and the Services must run in different processes, even when they are running on the same network node. For the Services to communicate with the middleware, some form of interprocess communication is needed. Since it is even possible to deploy the Services onto different network nodes than the middleware (as described below), this interprocess communication must be network-transparent, as well.

For this, we use CORBA, described in Section 4.1.1. It is an industry standard for object-oriented location-transparent interprocess communication, and ORBs are available for all of our target languages, hardware and operating system platforms (even on very small systems). CORBA interprocess communication is fast, scalable, and well-established.

The current middleware implementation uses OmniORB, and the DWARF Services use this and JavaORB, two publicly available CORBA implementations. For details, see Chapter 4.

SLP in Location Subsystem

The location subsystem of the DWARF middleware is designed to use SLP as a service location mechanism, as described in Section 4.2.1. SLP is an open Internet standard, is used in products such as Novell NetWare 5 and Apple Mac OS X, and is simple enough so that it can run on small systems as well.

Although SLP support is currently not implemented in the middleware (see Chapter 6), I have evaluated two SLP implementations, OpenSLP and Sun's reference implementation, and either would be suitable for this task. They can be integrated into the location subsystem using a standard API [34].

SLP is already used by design, in that the locations of Services' Abilities are represented as `service:` URLs, according to the SLP specification. It would be possible to use a different service location mechanism, however—the location subsystem's interface is quite general.

CORBA Notification Service in Communication Subsystem

For event-based communication, we chose the CORBA Notification Service standard, described in Section 4.1.2.

Again, this is an open standard, and several commercial and non-commercial implementations are available. In performance tests, I established that the crucial latency time of events, even when sent across network boundaries, is low enough to meet the nonfunctional requirements. Also, the Notification Service standard allows user-defined event types, which can be dynamically filtered by type or value, using a powerful filtering language.

For development, we used the open source OmniNotify implementation for Linux, and for the demonstration system, we used dCon.

Operating System Services for Low-Level Communication

For low-level communication methods such as raw TCP sockets or shared memory areas, the communication subsystem can use operating system services. Since the DWARF middleware can load communication protocol modules on demand, the platform-dependent code can be written as separate processes or loadable libraries for each target platform.

Currently, no such low-level communication mechanisms are used by DWARF, but the middleware's design could accommodate them easily.

XML Parser

For parsing the Service descriptions, which are written in an XML dialect, the Service Manager can use a third-party XML parser, such as XALAN, which is already used in other DWARF Services [57, 59, 77].

The current implementation does not use XML yet, but again, adding XML support is a fairly straightforward exercise, as described in Section 8.3.

5.4.2 Hardware Deployment Methods

The DWARF middleware can be deployed on various different hardware platforms. This includes different processor architectures (such as Intel x86, Motorola PowerPC, or StrongARM) with widely different memory sizes and processing power. Since DWARF supports AR-ready

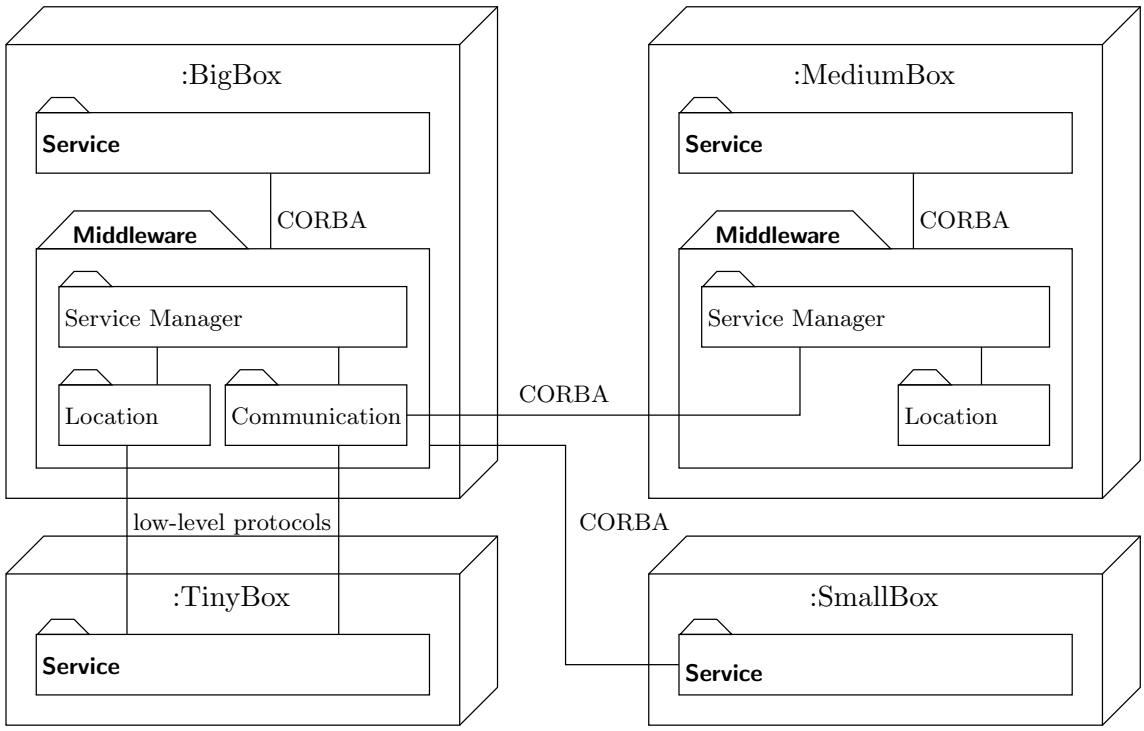


Figure 5.4: Deploying the subsystems of the DWARF middleware.
Each machine shown has a different hardware configuration and different middleware components. Note that the "SmallBox" communicates with the middleware as a whole using CORBA, whereas the "TinyBox" only uses low-level protocols.

intelligent environments, the middleware must run on systems ranging from PDA[1]-sized wearables to large servers within a building.

In designing the middleware subsystems, I have tried to keep the hardware requirements as low as possible. Additionally, different methods of deploying the middleware subsystems onto the hardware are possible—not every middleware subsystem has to be running on the local machine.

Here, I will show different possible configurations of DWARF middleware components, ranging from high to low hardware requirements. These are shown in Figure 5.4, clockwise from the top left.

Full Local Middleware Ideally, one Service Manager, one location subsystem, and one full communication subsystem runs on each node of a networked system. This way, all communication between Services and the middleware is local, and the Services do not explicitly have to deal with losing network connectivity. This, however, has the highest memory requirements.

Remote Communication Services To save memory and processing resources, it is possible to move parts of the communication subsystem (such as an event service) onto other, larger,

[1] Personal Digital Assistant

hosts. Local DWARF Services can still access the event service using CORBA—indeed, they will not even notice that the event service is not local, unless the network connection fails.

Note that if the DWARF Services running on a particular wearable system do not use an event service at all, the event service can be removed from the communication subsystem entirely. The communication subsystem is modular by design, as shown in Section 5.9.

No Local Mediating Agent The next smaller deployment method has hardly any local middleware at all. Each DWARF Service only has an ORB for using CORBA. The Services then must be configured to use the mediating agent of another host, meaning that they must know that host's network name. This means that the machine must have some form of reliable network connection to the machine whose middleware it is using, otherwise the local Services will not even be able to communicate with one another.

Use of Low-Level Protocols A system that cannot even accommodate an ORB implementation cannot run DWARF Services in the normal sense. It can, however, run Services that cooperate with the rest of the system, just like the DWARF system can use external Services such as printers. To do this, a Service must natively support the low-level protocols that the DWARF middleware uses. In the minimum requirement, this is SLP and a communication protocol such as HTTP. Since both SLP and HTTP require hardly any more resources than a TCP/IP stack, this allows very limited systems to cooperate with other DWARF Services. Of course, finding appropriate other Services, establishing connections, and so on all have to be implemented by the Service itself.

5.5 Persistent Data Management

In this section, I describe the persistent data stored by the middleware and the infrastructure required to store it.

The middleware hardly needs to store anything persistently, as the distributed mediating agents are designed to be running constantly on each network node. Status information for Services, communication ports etc. are all transient, as the middleware connects Services together dynamically.

XML Service Descriptions The only persistent data the middleware has consists of Service Descriptions. This enables the Service manager to advertise the Abilities of Services that have not been started yet. I have designed the data contained in Service Description so that these can easily be stored in XML files.

Each Service Description is stored in its own XML file. These XML files can reside in a file system directory that the Service Manager has access to. The Service Manager reads the files in this directory upon startup and regularly checks it for changes, loading any new Service descriptions. This way, installing a new Service is as easy as installing the executable file and copying the XML Service description into the Service Manager's directory.

An XML Service Description contains a description of a Service, its attributes, Needs and Abilities, with the communication protocols they support. The format of this file follows the `RegisterServices` interface described on page 84.

An example Service description for an optical tracking Service might be:

```
<service name="OpticalTracker" startOnDemand="true" stopOnNoUse="true"
         startCommand="/opt/dwarf/optical-tracker/bin/optical-tracker">
    <need name="world" type="WorldModel" predicate="wonderful=true"
          minInstances="1" maxInstances="1">
        <connector protocol="CorbaObjExport"
                   type="CorbaObjImporter"/>
    </need>
    <ability name="position" type="PositionData"
             attributes="accuracy=10,lag=30,trackedThing=/User/Head">
        <connector protocol="NotifyStructuredPushSupply"
                   type="NotifyStructuredPushSupplier"/>
        <connector protocol="SharedMemory" type="SharedMemoryWriter"/>
    </ability>
    <ability name="image" type="VideoData"
             attributes="resX=320,resY=240,fps=30">
        <connector protocol="SharedMemory" type="SharedMemoryWriter"/>
        <connector protocol="udp" type="udpSender"/>
    </ability>
</service>
```

Note that the functionality of accessing XML files is so far only present by design, and is currently not implemented. See Section 8.3 for details.

Design Rationale Service Descriptions are small, as shown above, and thus fit comfortably into text-based XML files. Using binary files would save only a marginal amount of space, and would necessitate extra editing tools. Storing the data in some larger database system would require far more resources and serve no useful purpose.

5.6 Access Control and Security

As explained on page 69, access control and security were not a major design goal for this version of the DWARF middleware. In fact, the current design contains no security mechanisms at all.

Design Rationale It has been pointed out that it is a bad idea to try to "retrofit" security onto an existing system. This is, of course, true for the DWARF middleware, as well. Thus, the decision not to address security issues had to be considered carefully.

However, for the application domain of mobile users in intelligent environments, security issues for the entire DWARF framework should be investigated in more detail. When a system assembles itself from distributed components, there must be some kind of trust mechanism between Services; there should probably be a billing mechanism for external Services, and a sophisticated user model is necessary to accommodate multiple users sharing Services.

Judging from this, security issues should be considered from a higher perspective, and, as mentioned in Section 8.3, this is probably a quite difficult task.

5.7 Global Software Control

This section describes how the control flow in the middleware is implemented, how requests are initiated and how subsystems synchronize.

Multithreading The DWARF middleware makes extensive use of threads in order to address the fact that it is active on the behalf of many different Services simultaneously, each with their own control flow. Additionally, the middleware must handle asynchronous events from both the Service location and the communication infrastructure.

Worker Threads for Each Service The middleware contains an Active Service Description for each Service using it. Each of these has an own thread, which acts on the Service's behalf and communicates with it. This follows the *robustness* design goal, and ensures that if one Service using the middleware fails, the other Services will not be affected.

Separate Threads for Location Subsystem The location subsystem can also use its own threads in order to communicate with the third-party Service location libraries and to handle asynchronous Service location events.

Monitoring Threads for Communication In order to detect failures of network communication resources and of the Services at the other end of a connection, the communication subsystem can use threads that monitor whether a connection is still open and functioning correctly.

Callbacks Between Subsystems To communicate with one another, the middleware subsystems and the DWARF Services use asynchronous callbacks. This means that they call methods to notify the other subsystems of status changes (e.g. the `connected` operation to indicate that a network connection has been established) and to request that actions be taken (e.g. the `startService` operation that DWARF Services must implement).

Avoiding Deadlocks For stability, these callbacks should not block the caller. This means that they should only update status information in the callee, and that any internal work that the subsystems or the DWARF Services have to perform should be done by separate worker threads. This pattern of synchronization is used extensively in the DWARF middleware to avoid deadlocks, as shown in Chapter 6.

5.8 Boundary Conditions

A *boundary condition* is a special condition the system must handle, including startup, shutdown and exceptions. [7]

Startup The startup of the DWARF middleware is fairly simple. First, the third-party components (such as event Services) that require their own processes must be started; then, the executable of the Service Manager is started and the middleware starts operating. This should happen before any DWARF Services are started, for example, on startup of the computer the DWARF system is running on.

Shutdown Shutdown is the same process as startup, only in reverse. Since the middleware is designed to run constantly, it can be shut down when the host using it is powered down.

Error Conditions Error conditions in network communication are handled by the communication subsystem, which notifies the rest of the middleware. Error conditions in communicating with Services using the middleware are detected by the corresponding Active Service Description. Currently, to ensure that the other Services can safely continue using the middleware, the middleware will simply stop communicating with a Service that has caused an error. As described in Section 8.3, more graceful error handling would be possible in the future.

5.9 Subsystem Functionalities

This section describes the functionalities offered by each subsystem of the DWARF middleware. I also describe the operations that a DWARF Service must provide in order to interact with it. For each subsystem, I describe the operations (grouped into interfaces), and then show how it interacts with the other subsystems. The IDL interface definitions are in Section A.1 in the Appendix.

This section can be seen as a high-level reference manual to using the DWARF middleware. In a system design document according to [7], it would be called "subsystem services". I have changed the name to avoid confusion with the DWARF Services.

DWARF Services do not access all of the subsystem functionalities described below; indeed, they usually only access the communication subsystem and the Service Manager. The interfaces that DWARF Services must support are explained below, and their interaction with the middleware is on the following page. Throughout this section, I have identified which functionalities are relevant to the rest of the DWARF system and which are "internal" to the middleware.

Unless you are interested in extending the DWARF middleware, I do not recommend reading the step-by-step details of the subsystem's interactions, but to continue directly with Chapter 6 or Chapter 7.

5.9.1 DWARF Services

Although DWARF Services are not a subsystem of the middleware, they have several middleware-relevant interfaces. Every DWARF Service that wishes to use the middleware must implement the `Service` interface. This is a callback interface that the Service Manager calls in order to tell the Service when to start and stop. When it is loaded, the Service registers its `Service` interface with the Service Manager using the `RegisterServices` interface (page 86).

Also, the Service must implement the `Need` interface for any Needs it has and the `Ability` interface for any Abilities. These are called by the Service Manager to provide the Service with the Connectors from the communication subsystem.

Service Interface

The `Service` interface supports two operations:

`startService` tells the Service that it can start running, i.e. all its Needs are satisfiable and one of its Abilities has been requested. If the Service has not been marked for starting on demand, this operation will be called immediately after the Service has registered itself with the Service Manager.

`stopService` tells the Service that it can stop running, i.e. none of its Abilities are required anymore. If the Service has not been marked for stopping when it is not in use, this will never be called.

Need and Ability Interfaces

A Service can implement the `Service`, `Need` and `Ability` interfaces in a single object (which is easiest for simple Services), or have separate objects for each Need and each Ability. These interface references are also registered with the Service Manager when the Service is loaded. The `Need` interface supports two operations:

`connectNeed` is called by the Service Manager when it has found and connected to another Service to satisfy one of this Service's Needs. It passes a `Connector` interface reference to the Service, which uses this to establish communication with the remote Service.

`disconnectNeed` is called by the Service Manager when a previously connected Need has become disconnected for some reason. This can be because the Service has requested the disconnection, because another (and better) Service has been found to satisfy the Need, or because network communication has failed.

The `Ability` interface supports two operations which are analogous to those of the `Need` interface:

`connectAbility` tells the Service that one of its Abilities has been connected to and provides the Service with a Connector it should use for communication.

`disconnectAbility` tells the Service that one of its Abilities is no longer being used, and it can stop communicating.

Interaction with the Middleware

Here, I will describe a DWARF Service's interaction with the middleware, from the Service's point of view. This consists of four phases: startup, Service description (which is optional), Service registration, and communication.

Figure 5.5 on the next page gives an example interaction for a tracking Service which is started on demand and starts sending data.

Startup Upon startup, the Service must find the middleware's interfaces.

1. The executable for a DWARF Service is loaded by the Service Manager, a startup script or the user.
2. The Service retrieves the Service Manager's `DescribeServices` interface, if it wishes to describe itself, from a well-known location.
3. The Service retrieves the Service Manager's `RegisterServices` interface, from a well-known location.

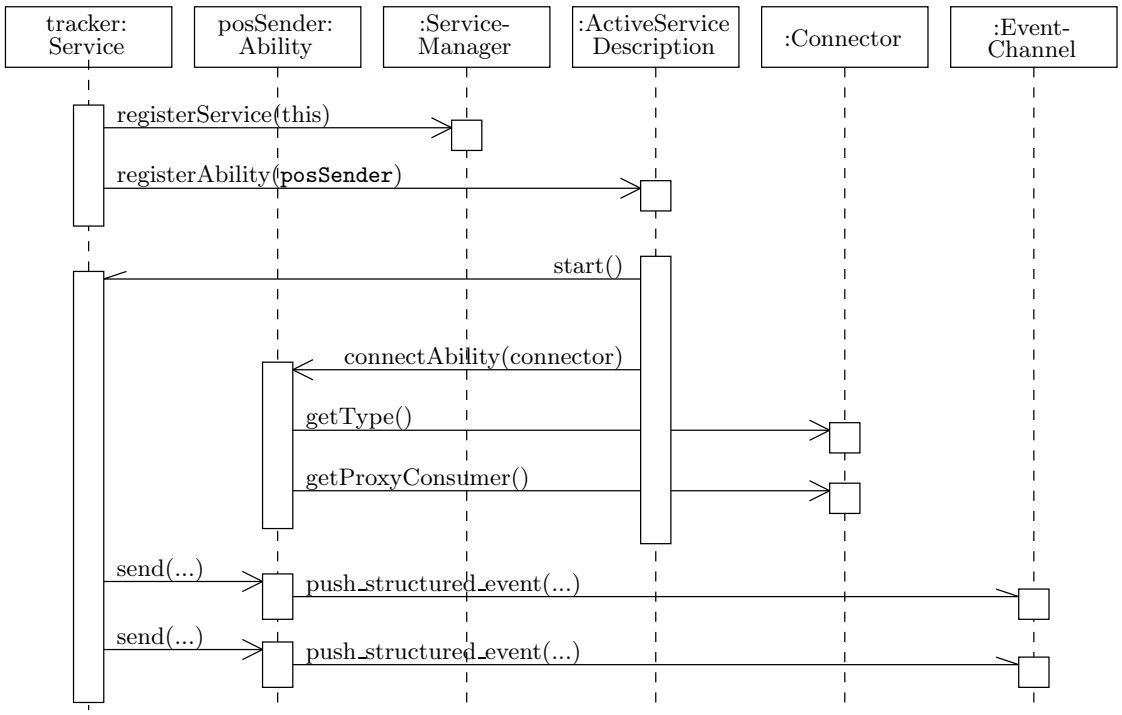


Figure 5.5: Interaction between a DWARF Service and the middleware.
A tracker Service registers itself with the Service Manager, is told to start up, receives a Connector, and starts sending events. Compare to Figure 3.7 on page 50.

Describe a Service This interaction is not needed if the Service Manager already has a description of the Service, either from an XML file or from a previous run of the Service. If the Service Manager has loaded the Service on demand, it obviously must have known the Service beforehand.

1. The Service invokes the `newServiceDescription` operation on the Service Manager's `DescribeServices` interface to create a new Service description with the specified name.
2. For each Need and each Ability, the Service invokes a `newNeed` or `newAbility` operation with the name of the Need or Ability.
3. For each communication protocol a Need or Ability supports, the Service invokes a `newConnector` operation with the type of the desired Connector.
4. The Service sets the attributes of the Service, its Needs, Abilities and Connectors, as described on page 84.
5. The Service invokes the `activateServiceDescription` operation to indicate that it is finished describing itself.

Register a Service and be Started This is mandatory for each Service wanting to use the middleware. The name a Service registers itself with is the same as its name in the Service description.

1. The Service calls the `registerService` operation on the Service Manager's `Register-`

`Services` interface with its name and a reference to its `Service` interface. This returns an `ActiveServiceDescription` interface.

2. The Service calls `registerNeed` and `registerAbility` on this `ActiveServiceDescription` interface for each of its Needs and Abilities.
3. If the `startOnDemand` flag in the Service's description is set, the Service Manager waits until one of the Service's Abilities is requested by another Service or by the user, and all of the Service's Needs can be satisfied.
4. The Service Manager invokes the Service's `startService` operation.

Get Connector for an Ability to Send Events, and Send them This is an example of a Service that can send events, such as a tracker. The Service must implement the `StructuredPushSupplier` interface to send events using the CORBA Notification Service.

1. The Service Manager invokes the `connectAbility` operation on one of the Service's `Ability` interfaces, giving it a Connector.
2. The Service checks the Connector's type with the `getType` operation,
3. narrows the `Connector` interface reference to a `SvcConnNotifyStructuredPushSupplier` reference,
4. retrieves the end of the event channel it should send its events to using this interface's `getProxyConsumer` operation,
5. and connects itself to this event channel using the `connect_structured_push_supplier` method.
6. Now, the Service can send events to the event channel using the `push_structured_event` method.

Get Connector for a Need to Receive Events, and Receive them This is a similar example of a Service that wishes to receive events, such as a display wanting position data. For this, the Service must implement the `StructuredPushConsumer` interface. If the Service has several different Needs wishing to receive events, it can implement this using a "receiver" object for each Need that implements both the `Need` and the `StructuredPushConsumer` interface.

1. The Service Manager invokes the `connectNeed` operation on one of the Service's `Need` interfaces, giving it a Connector.
2. The Service checks the Connector's type with the `getType` operation,
3. narrows the `Connector` interface reference to a `SvcConnNotifyStructuredPushSupplier` reference,
4. retrieves the end of the event channel it will receive events from using this interface's `getProxySupplier` operation,
5. and connects itself to this event channel using the `connect_structured_push_consumer` method.
6. From now on, events will be dispatched to the the Service's `push_structured_event` method.

5.9.2 Service Manager

The Service Manager provides two interfaces: `DescribeServices` and `RegisterServices`.

Service Description Interface

The Service Manager provides a `DescribeServices` interface, which allows Service Descriptions to be created and modified. These Service descriptions are designed so that they can easily be written in an XML dialect.

Just as Services consist of Needs and Abilities (see Figure 3.3 on page 44), Service Descriptions consist of Need Descriptions and Ability Descriptions (see Figure 5.6). Need and Ability descriptions themselves contain Connector Descriptions, which describe the communication protocols that Needs and Abilities can use.

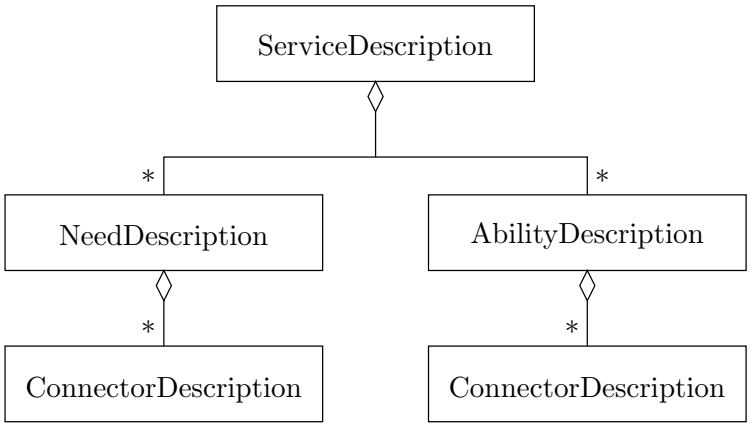


Figure 5.6: Service, Need, Ability and Connector Descriptions

Each of these descriptions has an interface, as described below.

DescribeServices Interface Services can either be described in an XML file, as described on page 77, or dynamically, using the Service Manager's `DescribeServices` interface. The main operations of this interface are:

`newServiceDescription` creates an empty new Service description with the given name. This name must be unique per Service Manager, i.e. no two Services described with the same Service Manager may have the same name. This operation returns a `ServiceDescription` interface reference.

`getServiceDescription` returns a reference to the Service description with the specified name.

`deleteServiceDescription` deletes a specified Service description.

`activateServiceDescription` indicates that the Service description with the specified name is now complete and can be activated.

ServiceDescription Interface This interface manages the attributes, Needs and Abilities of one Service description. The main operations of the `ServiceDescription` interface are:

`newNeed` creates an empty new Need description with the specified name. This name must be unique per Service, but otherwise can be arbitrary. Returns a `NeedDescription` interface reference.

The `getNeed` and `deleteNeed` operations allow these Needs to be accessed and deleted by name.

`newAbility` analogously creates an Ability description with the specified name. Returns an `AbilityDescription` interface reference. Again, there are also `getAbility` and `deleteAbility`.

Additionally, the interface contains `get` and `set` operations for the following attributes:

`startOnDemand` indicates whether the Service should be loaded on demand when one of its Abilities is requested and all its Needs can be satisfied.

`stopOnNoUse` indicates whether the Service should stopped when its Abilities are no longer required.

`startCommand` is the command line that launches the Service's executable. This is only relevant if `startOnDemand` is true.

AbilityDescription Interface An Ability description describes one Ability of a Service, i.e. a functionality that it provides. The main operations of the `AbilityDescription` interface are:

`newConnector` creates an empty Connector description with the specified communication protocol name. This identifies a protocol used for an Ability to communicate with other Service's Needs. It must come from a well-defined set of protocol names. For example, the protocol by which events are pushed by the CORBA notification Service from an Ability to a Need is called `NotifyStructuredPushSupply`.
This operation returns a `ConnectorDescription` interface reference.

`getConnector` and `deleteConnector` retrieve and delete Connector descriptions.

Additionally, there are get and set operations for the following attributes:

`type` describes the Ability, e.g. `PositionData`, and therefore should come from a well-defined set of Ability types.

`attributes` a list of named attributes describing the Ability, e.g. `accuracy=10,lag=30`.

NeedDescription Interface A Need description describes one Need that a Service has, i.e. a functionality of another Service that it depends on. The main operations of the `NeedDescription` interface are:

`newConnector` creates an empty Connector description with the specified communication protocol name, as above.

`getConnector` and `deleteConnector` retrieve and delete Connector descriptions.

Additionally, there are `get` and `set` operations for the following attributes:

`type` is the type of an Ability of another Service, e.g. `WorldModel`, and therefore should come from the same set of Ability types as above.

`predicate` A predicate on the attributes of the Ability needed, e.g. `accuracy>10`.

`minInstances` the minimum number of other Services' Abilities required for this Need to be satisfied. For example, a stereo tracking system needs two video cameras, both of the same type, to work.

`maxInstances` The maximum number of other Services' Abilities that this Need can use.

ConnectorDescription Interface The `ConnectorDescription` interface consists of get and set operations for the following attributes:

`ConnectorType` A communication protocol is often asymmetric, i.e. the two partners interact with the communication infrastructure differently: The supplier sends messages to the consumer, the caller calls the callee, etc. The Connector type identifies which end of the protocol specified above this Need or Ability is on. In the above example, the Ability would have the Connector type `NotifyStructuredPushSupplier` and the Need would have `NotifyStructuredPushConsumer`.

`manualURL` This can be used for Services that do not actually use connectors for communication. For example, if a Service's `TaskModelFile` Ability were provided by an external HTTP server, the Service could specify the appropriate URL here. In this case, the Connector type should be empty. This allows legacy systems to be easily integrated into DWARF systems.

`disconnectTimeout` specifies the amount of time after which a Connector for an Ability will time out if no Needs are using it. This allows the Service to be stopped when it is not needed anymore.

Service Registration Interface

Once a Service has been started, it must register itself with the Service Manager. It does that through the `RegisterServices` interface. This allows the Service Manager to give it the Connectors it needs to communicate with other Services.

RegisterServices Interface The `RegisterServices` interface contains the following operations:

`registerService` registers a Service's `Service` interface with the Service Manager and associates it with the Service description of the specified name. The Service Manager then knows that this Service is active and can supply it with the necessary connectors.
This operation returns an `ActiveServiceDescription` interface reference, which points to the active Service description of this Service.

`registerServiceAndNeedAndAbility` is a helper function allowing simple Services to register themselves with one call. For this, the Service must implement the `Service`, `Need` and `Ability` interfaces.

`unregisterService` unregisters the specified Service from the Service Manager. A Service should call this upon termination.

ActiveServiceDescription Interface The `ActiveServiceDescription` interface is derived from the `ServiceDescription` interface, so that a Service can access its description once it is registered. This allows the Service to access configuration information that has been stored in the XML file.

`registerNeed` associates the provided `Need` interface with the Need description of the specified name. This interface will receive connectors for the Need.

`registerAbility` analogously registers an `Ability` interface reference.

Interaction with other subsystems

The Service Manager must coordinate between the other subsystems and the DWARF Services. Here, I will show three examples of how it does this.

Description and Registration of a Manually Started Service

1. The user starts a Service that has not been described to the middleware yet.
2. The Service calls the Service Manager's `DescribeServices` interface to establish that it has not been described yet, an describes itself. The Service Manager creates an Active Service Description to represent the Service.
3. The Service invokes the `ActivateServiceDescription` operation. The Active Service Description tells the communication subsystem to allocate the necessary communication resources for the Service's Abilities, creating an appropriate Connectors from the appropriate Connector Factories, and advertises the Abilities by creating Offers with the location subsystem.
4. The Service calls the Service Manager's `RegisterServices` interface, to register itself as running. The Service Manager associates the Active Service Description with the running instance of the Service, and the Active Need and Ability Descriptions with the running instances of the Service's Needs and Abilities.

An Ability is Used by Another Service

1. One of the Connectors in the communication subsystem accepts an incoming Connector for a Service's Ability. It notifies the Active Service Description.
2. If the Service is not loaded yet, the Service Manager loads it and waits for it to register itself. It then calls the Service's `start` operation.
3. If this is the first time this Ability is being used, the Active Service Description notifies the Service by invoking its Ability's `connectAbility` operation.

Satisfy a Need

1. When a Service is supposed to start, either because an Ability is requested or because the `startOnDemand` flag is false, the Active Service Description tries to satisfy the Service's Needs.
2. For this, it selects other Service's Abilities found for the Requests it has registered with the the location subsystem.
3. The Active Service Description instructs the communication subsystem to connect to the other Services, creating an appropriate Connector from the appropriate Connector Factory.
4. For each connection that is established, the Active Service Description notifies the Service by invoking its Need's `connectNeed` operation.

5.9.3 Communication Subsystem

Abstract Factory Pattern The communication subsystem follows an *abstract factory* design pattern. Connectors for various communication protocols are created by *Connector Factories*,

which share a common interface. There is a Connector Factory for each communication method, e.g. one for the CORBA Notification Service, one for the simple exchange of CORBA interfaces, one for shared memory, etc. (This pattern is shown in Figure 6.2 on page 99.)

The Connectors themselves also have a simple common interface, but additionally support protocol-specific derived interfaces. For a Service to be able to make use of a particular Connector type, it must know the protocol-specific interface of the Connector, which includes operations like `getProxyPushSupplier` (for the CORBA Notification Service) or `getSharedMemAddr` (for communication via shared memory).

Connector Interface

For the rest of the DWARF system, the Connector interface is the main point of interaction with the middleware's communication subsystem. A DWARF Service receives a reference to a Connector from the Service Manager, as described below. At this point, communication resources have already been allocated by the Connector. If this Connector is for a Need, a connection to an Ability satisfying this Need has been established. If this Connector is for an Ability, there may be no connection yet—instead, the Connector is passively waiting for incoming connections.
The `Connector` interface is very simple, supporting only three operations:

`getType` returns the type of this Connector. This allows the Service to determine which protocol should be used to communicate. Example types are `NotifyStructuredPushSupplier` and `CorbaObjImporter`.

`disconnect` ends all communication initiated by this Connector. A Service generally only needs to call this function to indicate that it is dissatisfied with this Connector, e.g. the other Service that it is communicating with is sending invalid data.

Protocol-Specific Connector Interfaces

To encapsulate protocol-specific information in Connectors, there are two interfaces derived from the `Connector` interface for each protocol, representing the two ends of a communication protocol. When a Service receives a reference to a Connector, it must narrow this reference to a protocol-specific Connector interface. As a naming convention, if a Connector's type is *type*, it supports the protocol-specific interface `SvcConn`*`type`*.

CORBA Notification Service Connectors Communication using the CORBA Notification Service is based on structured events. There are two types of connectors, `NotifyStructuredPushSupplier` and `NotifyStructuredPushConsumer`, reflecting senders and receivers of events. The connectors encapsulate structured event channels, which implement the actual delivery of events.
The `NotifyStructuredPushSupplier` Connector supports the `SvcConnNotifyStructuredPushSupplier` interface, with only one operation:

`getProxyConsumer` returns a reference to a `StructuredProxyPushConsumer` interface of an event channel. The Service should connect itself to this proxy and send its events into it by calling the proxy's `push_structured_event` method. For this, it must implement the `StructuredPushSupplier` interface, as described in [14]. The event channel consumes the events and delivers them to subscribers.

The `NotifyStructuredPushConsumer` Connector supports the `SvcConnNotifyStructuredPushConsumer` interface, with only one operation:

`getProxySupplier` returns a reference to a `StructuredProxyPushSupplier` interface of an event channel. The Service should connect itself to this proxy in order to receive events. For this, it must implement the `StructuredPushConsumer` interface, as described in [14]. This interface's `push_structured_event` method will be called by the event channel when a new event arrives.

By convention, the protocol types for this kind of communication are `NotifyStructuredPushSupply` (the Ability sends data to the Need, e.g. tracker to display) and `NotifyStructuredPushConsume` (the Ability receives data from the Need, e.g. pizza advertising Service to printer).

CORBA Object Reference Connectors These connectors are very simple. There are two types, `CorbaObjExporter` and `CorbaObjImporter`. The exporter allows a Service to export a CORBA reference to a CORBA object, and the importer allows other Services to import this, so they can call methods on the exporting Service's object.
The `CorbaObjExporter` Connector supports the `SvcConnCorbaObjExporter` interface, with only one operation:

`setObject` passes the Connector a reference to the object which the Service wishes to export.

The `CorbaObjImporter` Connector supports the `SvcConnCorbaObjImporter` interface, with only one operation:

`getObject` returns a reference to the object that the other Service has exported.

By convention, the protocol types for this kind of communication are `CorbaObjExport` (the Ability exports an object reference, e.g. the world model) and `CorbaObjImport` (the Ability imports an object reference, e.g. an object property logging Service).

Communication Resources

All communication resources supported by the DWARF middleware are provided by third-party components, e.g. CORBA Notification Service implementations, as explained in Section 5.4.1.

Connector Factories

The Connector Factories described above have a very simple interface, `ConnectorFactory`. This is only used by the Service Manager. This basically can tell which Connector types the factory supports and can create Connector of those types:

`supportsType` tells whether the factory supports a certain Connector type.

`createConnector` returns a new Connector of a certain type.

ProtocolConnector Interface

Internally, Connectors that are created by the Service Manager for a Service's Abilities also support the `ProtocolConnector` interface, with two additional operations:

`connect` tells the Connector to connect to a certain location.

`setConnectCallback` gives the Connector a reference to a `ConnectCallback` interface which is notified when a connection is established.

`destroy` destroys the Connector.

Additionally, the `ProtocolConnector` interface is derived from the `LocatorOffer` interface. This way, a Connector can act as an Offer for the Service Locator, by supplying a location that other Services can connect to.

ConnectCallback Interface

The `ConnectCallback` interface is for the Service Manager, and provides status information about the Connector. It has two operations:

`connected` the Connector is connected to a certain location.

`disconnected` the Connector is disconnected.

Interaction with other Subsystems

Here, I will show how the communication subsystem interacts with the other middleware subsystems and the DWARF Services.

Create a Connector for Sending Events In this example, the Service Manager creates a `NotifyStructuredPushSupplier` Connector for a Service's Ability, so that it can send events to other Services.

1. The Service Manager queries the `ConnectorFactory` interfaces registered with it, using the `supportsType` operation to find a Connector Factory supporting `NotifyStructuredPushSupplier` connectors.
2. It then invokes the `createConnector` operation on that Connector factory.
3. The Connector factory creates a Connector.
4. Upon initialization, the Connector creates an event channel from the CORBA Notification Service's `EventChannelFactory` interface.
5. The Connector factor returns a `ProtocolConnector` reference.
6. The Service Manager invokes this Connector's `getLocation` operation, which returns its stringified CORBA object reference as a location. The Service Manager can now advertise this location.

Accept a Connection From Another Service Here, the local Connector accepts a connection from a remote Service.

1. The remote Service's `NotifyStructuredPushConsumer` Connector uses the stringified CORBA object reference to find the local `NotifyStructuredPushSupplier` Connector.

2. It then connects its own event channel to the local Connector's event channel.
3. The local Connector calls the Service Manager's `connected` method.
4. The local Service, which gets a reference to the Connector from the Service Manager, calls the Connector's `getProxyConsumer` method. This returns a reference to one end of the local event channel.
5. The Service connects itself to this event channel and sends events.
6. The events are forwarded through the local and remote event channels to the remote Service.

Connect to Another Service Here, the Service Manager creates a `NotifyStructuredPushConsumer` Connector for a Service's Need, and connects to a remote Service's Ability.

1. The Service Manager creates a `NotifyStructuredPushConsumer` Connector from the appropriate Connector factory (as above).
2. The Service Manager then invokes the `connectTo` operation of the Connector, giving it the location of a remote `NotifyStructuredPushSupplier` Connector.
3. The local Connector connects to the remote Connector, as above.
4. The Connector calls the Service Manager's `connected` method.
5. The Service Manager passes a reference to the Connector to the Service, which calls the Connector's `getProxyConsumer` method. This returns a reference to one end of the local event channel.
6. The Service connects itself to the event channel to receive events.
7. The events coming from the remote Service are forwarded through the remote and local event channels to the local Service.

Connection Failure When a network connection fails. The Service Manager is notified about this, and can take appropriate action.

1. A local Connector's event channel is connected to receive events from a remote event channel.
2. The network connection fails.
3. The local Connector notices this as its periodic ping operation fails on the remote event channel.
4. The Connector calls the Service Manager's `disconnected` operation.
5. The Service Manager can then notify the local Service, and find another Service to connect to.

5.9.4 Location Subsystem

The location subsystem is generally not directly used by other DWARF Services, with the exception of the CAP Service. Instead, it is used by the Service Manager.

Offers and Requests

The Offer and Request objects identified on page 73 each have a corresponding callback interface, called `LocatorOffer` and `LocatorRequest`. Thus, the Service Manager must implement

these interfaces when it wishes to register Offers and Requests with the location subsystem.

Offer Interface The `LocatorOffer` interface is called by the location subsystem to retrieve dynamically changing information about an offer. It has the following operations:

`query` returns the offer's current attributes of the Offer.
`getLocation` returns the offer's location, as a URL.

Request Interface The `LocatorRequest` interface is called by the location subsystem. It has only one operation:

`found` indicates that an offer has been found that matches this request, and provides the location and attributes of this offer.

Locator Interface

The main interface of the location subsystem is the `ServiceLocator` interface, with the following operations:

`registerOffer` Registers a given offer with the location subsystem.
`registerRequest` Registers a given request with the location subsystem.
`doQuery` Asks the location subsystem to try to find offers for the registered requests.

Interaction with other Subsystems

Here, I will show how the location subsystem interacts with the other middleware subsystems.

Register Offer and Have it Queried This shows how the location subsystem deals with Offers from the Service Manager.

1. The Service Manager registers an object supporting the `LocatorOffer` interface with the location subsystem, using the `registerOffer` operation. It provides the type of the offer to be registered.
2. The location Subsystem calls the provided `LocatorOffer` reference to retrieve the location and attributes of this offer.
3. Then, the location subsystem registers the offer with with the SLP library.
4. The SLP library will then respond to matching queries with the location of this offer.
5. Periodically, the location subsystem checks the `LocatorOffer`'s attributes using the `query` operation, and registers any changes with the SLP library.

Register Request and Have an Offer Found This shows how the location subsystem deals with Requests from the Service Manager.

1. The Service Manager registers an object supporting the `LocatorRequest` interface with the location subsystem, using the `registerRequest` operation. It provides the type of offer that is supposed to be found, and a predicate describing attributes that that offer must have.

2. The location subsystem registers the request with with the SLP library.
3. Periodically, the location subsystem queries the SLP library for new matches for the requests.
4. When the SLP library finds a remote offer that matches the request, the location subsystem notifies the Service Manager, calling the `LocatorRequest`'s `found` method.

6 Implementation of the DWARF Middleware

The first implementation is stable, fast, and covers most of the basic functionality.

This chapter is quite technical, as it documents my first implementation of the DWARF middleware. For each subsystem, I describe the objects I used in the implementation. In addition, I describe how the DWARF Services access the middleware, including CORBA implementations and the bootstrapping process.

This chapter is mainly of interest to anyone wishing to develop the DWARF middleware further.

6.1 Interprocess Communication With CORBA

This section briefly describes how the middleware subsystems use CORBA to communicate.

6.1.1 OmniORB

The ORB I used for the middleware is OmniORB (see Section 10). This is available under the GNU[1] Public License and the GNU Library Public License, so that there are no royalties for using it, and it can be freely modified. I used version 3.0.2, for Linux on Intel x86 processors, Linux on PowerPC processors, and Windows NT, 98 and 2000.

OmniORB provides a lightweight platform-independent threading library called OmniThreads. I used this to create threads in the middleware subsystems and for mutexes and condition variables. On Linux, OmniThreads uses the pthreads library, and on Windows, it uses the Win32 API.

Aside from the threading code, of which ten lines are OmniThread-specific, the middleware's code is not specific to OmniORB, and should compile with any other multithreading ORB compliant to the CORBA 2.3 standard.

6.1.2 Servant Implementation

Almost all objects in the middleware have a CORBA interface and are therefore servants.

Portable Object Adapter To implement these servants, CORBA specifies the so-called Portable Object Adapter (POA), which is a skeleton class generated by the ORB's IDL compiler that a servant implementation can derive itself from. This POA dispatches incoming calls from other processes to the methods of the servant object.

[1] GNU is Not Unix

Naming Convention Classes implementing interfaces are not allowed to have the same name as these interfaces; for that reason, several classes have the suffix `_i` to indicate that they only implement a particular interface. For example, the `ActiveServiceDescription_i` class implements the `ActiveServiceDescription` interface.

Implementing Multiple Interfaces For various reasons, the POA does not support multiple inheritance, i.e. a servant implementation class may not be derived from two different POA skeleton classes, even though C++ supports multiple inheritance. CORBA interfaces, however, do support multiple inheritance. Thus, I defined an empty helper interface in IDL whenever a servant needed to implement more than one interface. For example, the `ServiceManager_i` class implements the `ServiceManager` interface, which is derived from the `DescribeServices` and `RegisterServices` interfaces and is otherwise empty.

6.2 Service Manager

Implementation Status The Service Manager is fully implemented, except for the following points:

- XML support is missing, i.e. Services must describe themselves using the `RegisterServices` interface.
- Because of this, Services are not yet started on demand—the `startCommand` is not evaluated. This would be trivial to add, however.

6.2.1 Object Design

During *object design*, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects. [7]

The relationships between the objects and interfaces of the Service Manager are shown in Figure 6.1 on the following page.

Tree-Structured Active Service Descriptions

The Active Service Description object identified in Chapter 3 has a quite complex functionality. It has to coordinate between DWARF Services and the other middleware subsystems, location and communication, which do the actual work of finding and connecting Services. To simplify implementation, I have identified several additional object classes, which together describe and manage a DWARF Service.

This division into objects was based on minimizing the amount of state information that needs to be stored in each object, so that these can be implemented as deterministic finite automata, as described on page 98.

`ActiveServiceDescription_i` implements the `ActiveServiceDescription` interface. There is one `ActiveServiceDescription_i` for each Service which has been described to the middleware. Since the `ActiveServiceDescription` interface is derived from the `ServiceDescription` interface, an `ActiveServiceDescription_i` is created by the Service Manager whenever a new Service is described.

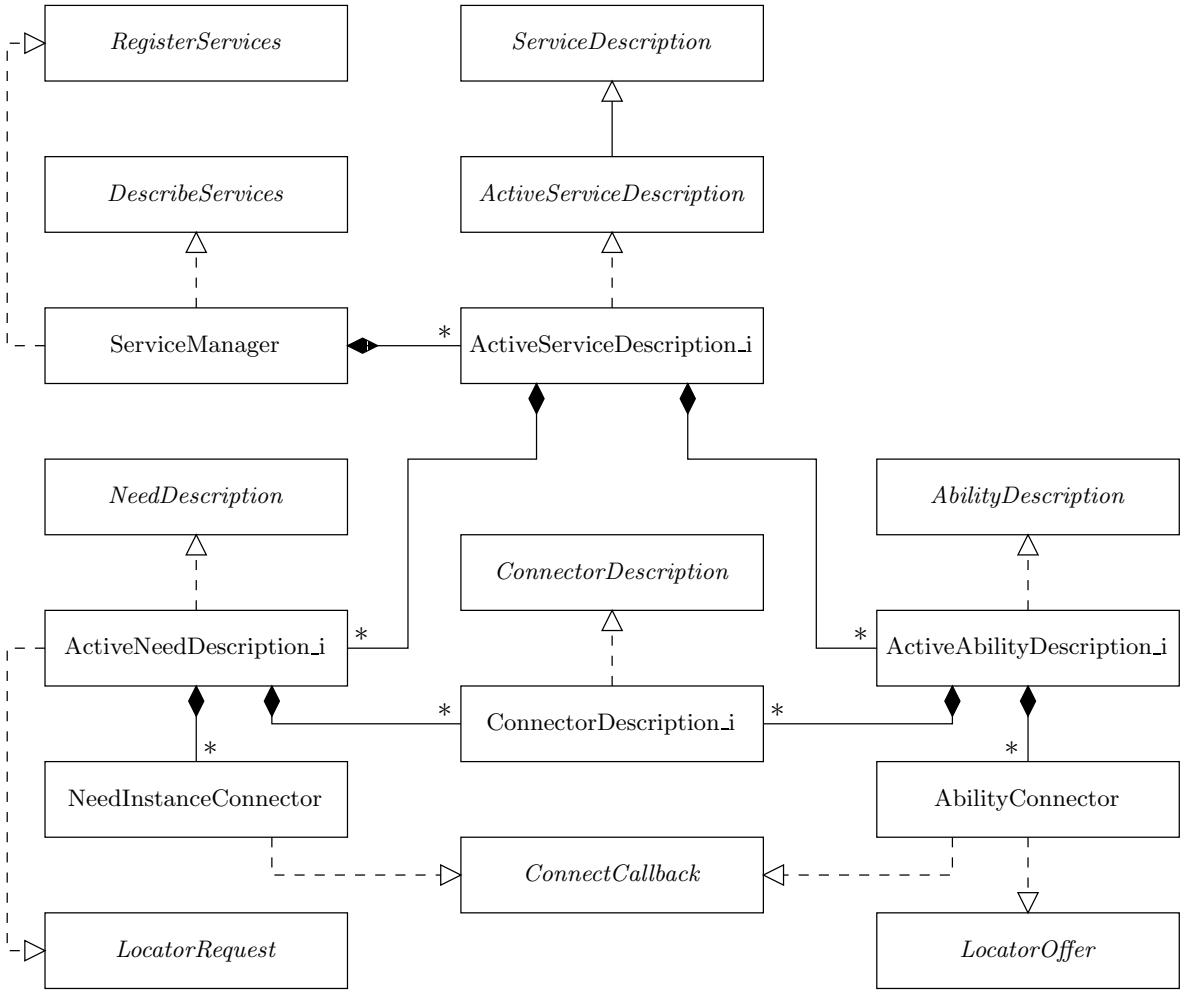


Figure 6.1: Classes within the the Service Manager and the interfaces they implement.
Note that a `ConnectorDescription_i` object can only be associated with *either* an `ActiveNeedDescription_i` *or* an `ActiveAbilityDescription_i`, as is denoted by the solid diamond indicating composition in UML.
The interfaces that are *above* classes implementing them in this diagram are used by the DWARF Services, whereas the interfaces *below* classes are used by other middleware subsystems.

The `ActiveServiceDescription_i` is responsible for calling all methods of the Service's `Service` interface, loading the Service, and processing the Service's `registerNeed` and `registerAbility` requests. An `ActiveServiceDescription_i` maintains a map of `ActiveNeedDescription_i`s and `ActiveAbilityDescription_i` objects, representing the Service's Needs and Abilities.

`ActiveNeedDescription_i` implements the `NeedDescription` interface. It represents one Need of a Service and is responsible for communicating with the Service's `Need` interface. It maintains a map of `ConnectorDescription_i`s, representing the communication protocols that the Service wishes its Need to be satisfied in. It maps the Need to Requests that are registered with the location subsystem, and processes the `found` callback of the location subsystem by creating `NeedInstanceConnector_i`s. It maintains a list of these `NeedInstanceConnector_i`s, representing Abilities of other Services that potentially could satisfy this Need, and selects `NeedInstanceConnector_i`s from this list to satisfy the Need.

`NeedInstanceConnector_i` represents one potential or active connection to another Service, to satisfy a particular Need. It contains a reference to a `ProtocolConnector` interface, which is the Connector that manages the network connection, and is responsible for telling this Connector where to connect to.

`ActiveAbilityDescription_i` implements the `AbilityDescription` interface. It represents one Ability of a Service and is responsible for communicating with the Service's `Ability` interface. It maintains a map of `ConnectorDescription_i`s, representing the communication protocols with which the Service wishes its Need to be satisfied, and a list of `AbilityConnector_i`s, representing connectors that are kept open for incoming connections to this Ability.

`AbilityConnector_i` represents the Connector for one communication protocol that this Ability supports. It contains a reference to a `ProtocolConnector` interface, which is the Connector that manages the network connection.

`ConnectorDescription_i` implements the `ConnectorDescription` interface. It represents one communication protocol that a Need or Ability supports.

Boundary Class: Service Manager

The `ServiceManager_i` class provides a uniform interface to the Service Manager subsystem. Its functionality is quite simple: it maintains a map of Service names to `ActiveServiceDescription_i` objects. It implements the `DescribeServices` and `RegisterServices` interfaces.

Service Descriptions in XML

The reading and parsing of Service descriptions written in XML will be done by the Service Manager in in the future, or perhaps by a separate class calling the Service Manager's `DescribeServices` interface.

6.2.2 Implementation

C++ and STL All objects of the Service Manager are implemented in standard C++. They use the C++ Standard Template Library (STL) to implement data structures such as maps

and lists.

These libraries are available on both compilers used for the middleware: the GNU C++ compiler *g++*, and Microsoft's *Visual C++* compiler.

Deterministic Finite Automata All "active" objects within the Service Manager are modeled as deterministic finite automata. The `ActiveServiceDescription_i`, `ActiveNeedDescription_i`, `NeedInstanceConnector_i`, `ActiveAbilityDescription_i` and `AbilityConnector_i` classes each have a `state` member indicating their current state.

Additional state is only kept in the lists of related objects (for example, the `ActiveServiceDescription_i`'s list of its `ActiveNeedDescription_i`s), and—as a special case—by the `AbilityConnector_i` class, which has a member `useCount` indicating how many incoming connections are currently open.

Transitions between the states can be triggered by changes in the state of related objects. For example, an `ActiveNeedDescription_i` switches from the state `Satisfying` to `Satisfied` when enough of its `NeedInstanceConnector_i`s are in the `Connected` state.

Transitions can also be triggered by external callbacks. For example, a `NeedInstanceConnector_i` switches from the state `Connecting` to `Connected` then its `connected` method is called. I have included the state transition diagrams in the appendix.

Worker Thread, Transitions and Callbacks Each `ActiveServiceDescription_i` has a worker thread, which is responsible for all actions associated with state transitions. I have strictly adhered to the rule that incoming calls from outside of the Service Manager only trigger state transitions that are not associated with actions. This ensures that the objects of the Service Manager will never block on an incoming call. Actions that must be taken as the result of an incoming call are then taken later, by the worker thread. This prevents deadlocks between the Service Manager, other middleware subsystems and the DWARF Services.

As an example, when a `NeedInstanceConnector_i`'s `connect` method is called, it switches to the `ShouldConnect` state, but does not connect yet. Later, within the worker thread, it starts connecting and switches to the `Connecting` state.

Multithreading Since the the ORB can dispatch incoming method calls to the Service Manager in multiple threads, there is a semaphore protecting access to the `state` variable of each `ActiveServiceDescription_i` and all the objects associated with it. The worker thread releases this mutex when making calls outside of the Service Manager, so that the Service Manager will not deadlock even if another subsystem or a DWARF Service calls the Service Manager back reentrantly while the Service Manager is calling it. This is similar to the *active object* design pattern in [61].

6.3 Communication Subsystem

Implementation Status The communication subsystem currently supports event-based communication using the CORBA Notification Service specification and communication by calling methods on exported CORBA object references. Certain optimizations that could be performed on event channels are missing, but otherwise the implementation is complete.

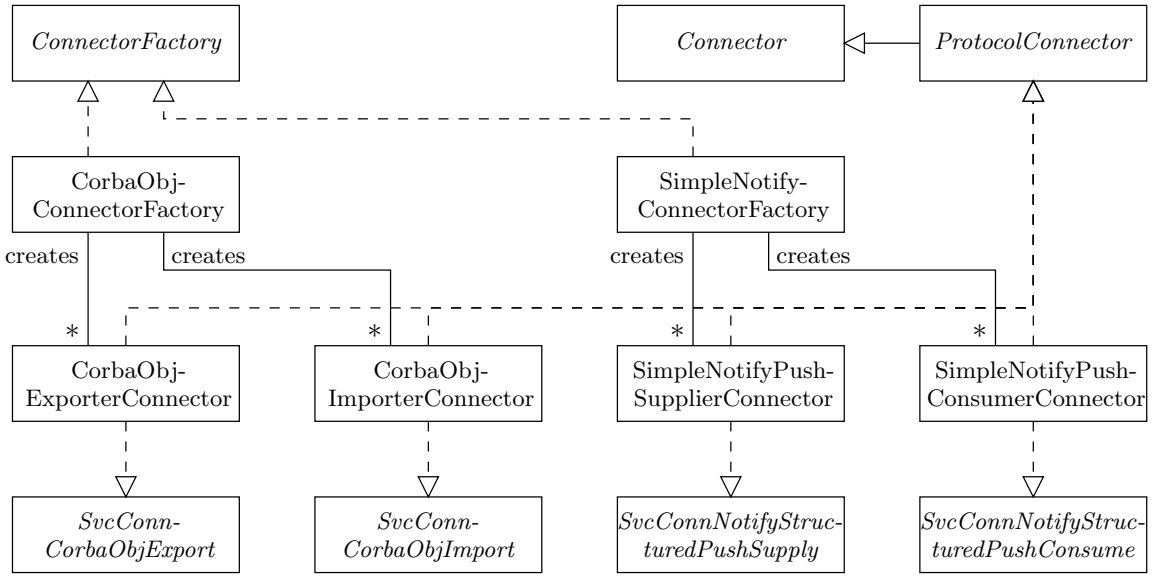


Figure 6.2: Classes and interfaces of the communication subsystem.
The communication subsystem follows an *abstract factory* design pattern. The interfaces `ProtocolConnector` and `ConnectorFactory` are used by the Service Manager, the other interfaces are used by DWARF Services to establish communication with one another.

6.3.1 Object Design

The communication subsystem is designed using the *abstract factory* pattern [21].

Both the `CorbaObjConnectorFactory` and the `SimpleNotifyConnectorFactory` implement the `ConnectorFactory` interface, and they can create Connectors for communication using CORBA interfaces and the CORBA Notification Service, respectively. The classes and interfaces of the communication subsystem are shown in Figure 6.2.

CORBA Object Connectors are very simple: the `CorbaObjExporterConnector` exports a CORBA object reference, and the `CorbaObjImporterConnector` imports it. Both Connector classes make their own stringified object references available to the location subsystem, so that it can advertise them on the network.

CORBA Notification Service Connectors are slightly more complex. The Connector Factory is initialized with a reference to a CORBA Notification Service's `EventChannelFactory` interface. When a Connector is created, it allocates an event channel from the `EventChannelFactory`. One end of the event channel is for the local Service to attach to, the other end is exported via the location subsystem. When told to connect to a remote Service, the Connector connects its own event channel to the remote Service's event channel, so the two Services communicate through two composed event channels. For optimization, it is also possible to use only one single event channel, on the side of the Ability, that the other Services' Needs then connect to directly.

The events sent across the event channels are CORBA Notification Service structured

events, which explains the rather long interface names. Some event types have been standardized for DWARF, such as the `PositionEvent` described in [3].

Future Connectors Future implementations could easily extend the communication subsystem by implementing factories for shared memory, raw socket, streaming video, or other communication methods. Since Connector Factories can be added to the Service Manager independently of one another.

Interaction with Service Manager Connector Factories are registered with the Service Manager using the `RegisterConnectorFactories` interface. In the current implementation, the two connector factories are compiled into the same executable as the Service Manager, and are registered with it on startup. It would be possible to implement Connector Factories as separate processes, however, even written in a different language, as long as they can access the Service Manager using CORBA.

6.3.2 Implementation

Like the Service Manager, the communication subsystem is implemented entirely in C++. Its functionality is so simple (since it can take advantage of third-party components) that it does not even need to use the Standard Template Library.

6.4 Location Subsystem

Implementation Status Since I have defined a generic `ServiceLocator` interface for the location subsystem, it is possible to use different Service location mechanisms without affecting the rest of the middleware. The design is intended to use SLP in the future; however, this is not implemented yet. I have implemented a simple "local" Service locator, which can only find Services that have been registered with it. In order to be able to use the middleware with the Distributed Mediating Agents as it is designed, SLP support must still be added.

6.4.1 Object Design

Simple Service Locator The `SimpleServiceLocator` simply implements the `ServiceLocator` interface. It maintains a list of `LocatorOffer`s and a list of `LocatorRequest`s, and when the method `doQuery` is called, it tries to match these requests. Requests are matched to offers simply by type; the boolean expression is not evaluated, since this should be done by the SLP library when the SLP Service Locator is implemented.

SLP Service Locator The `SLPServiceLocator`, when it is implemented, should similarly implement the `ServiceLocator` interface. It can use the SLP C language API [34] to register requests and offers with the SLP implementation. This should be a one-to-one mapping. For interaction with the SLP library, the `SLPServiceLocator` should presumably have its own thread of control.

6.4.2 Implementation

The simple local Service locator is, again, implemented entirely in C++, and uses the Standard Template Library for its internal data structures.

6.5 Accessing the Middleware from DWARF Services

Implementation Status The DWARF Services access the middleware using CORBA. This has been implemented and tested successfully on several platforms.

6.5.1 ORBs on Target Platforms

The following combinations of ORBs and platforms have been successfully used to access the DWARF middleware from DWARF Services. Both OmniORB and JavaORB fully support the CORBA 2.3 standard, including the POA, so that porting the DWARF Services to use other ORBs should only require recompiling the Services with a different library.

- OmniORB with GNU C++ on Linux for PowerPC and for Intel x86 processors
- OmniORB with Microsoft Visual C++ on Windows 98 and 2000
- JavaORB with the Java Development Kit 1.1.8 on Apple Mac OS 9
- JavaORB with the Microsoft Java Virtual Machine on Windows NT, natively and within the Internet Explorer's sandbox

Other freely available ORBs were tested as well, with varying results.

- TAO would work in principle for C++, but (currently) does not run well on Linux PowerPC, and requires approximately ten times as long to compile as OmniORB does.
- Sun's Java 2 ORB does not yet support the full CORBA 2.3 specification. In particular, it does not support the POA, meaning that code using it is harder to port to different ORBs.
- JacORB's interface compiler could not deal with the (legal) type definition of the DWARF `PositionEvent`.

6.5.2 Initial References

The DWARF Services must be able to access the Service Manager somehow at startup. For this, I have defined the CORBA location of the Service Manager, when it is running on the local host, to be `corbaloc://localhost:39273/ServiceManager`. (The number 39273 is "DWARF" as dialed on a telephone keypad.)

For this, the Service Manager uses OmniORB's `omniINSPOA`, a Portable Object Adapter that allows a servant to specify its object key itself.

6.5.3 Service Adapters for C++ and Java

Since accessing the middleware is similar for all DWARF Services, this raises the possibility of writing generic adapter classes to handle routine tasks such as ORB initialization.

For C++, the initialization is easily done by hand, and the DWARF Services written in C++ (the optical tracker, the World Model, and the Bluetooth communication Service) simply have similar initialization code. A simple wrapper class would be useful here in the future.

For accessing the middleware from Java, Martin Bauer kindly developed a simple adapter class that encapsulates ORB initialization, finding the Service Manager, and so on. This is described in [3], and is used by all DWARF Services written in Java.

7 Building Augmented Reality Systems with DWARF

With our new new framework, we can build Augmented Reality systems quickly. We have built one example system already, and have designed others.

In this chapter, I show how DWARF, the presented in Chapter 2, can be used to build mobile AR systems quickly and easily.

For this, I present two examples: first, the demonstration system we built and presented in December 2000; and second, the design for a maintenance system that could be built with DWARF as well.

7.1 Navigating with Use of Services: Our Demonstration System

In this section, I describe the first demonstration system built with DWARF. This is essentially a campus navigation system allowing the wireless use of Services such as printers. Parts of this section were written jointly and published simultaneously with Martin Bauer and Martin Wagner in [3] and [77].

7.1.1 Scenario

For our demonstration system, we chose a rather complex scenario. A visitor is headed for a meeting in a room on the campus of the Technische Universität München, and his mobile AR system navigates him to the meeting room and lets him print out his handouts on the way.

The choice of our scenario was driven by three goals:

- The scenario should involve not only classical AR, but also the dynamic use of services,
- it should take place in different types of environment to show the flexibility of our framework,
- and it should use and test all of the DWARF components developed up to now.

Here is the scenario, in the same notation used in Chapter 3:

Scenario: **Demonstration Scenario**

Actor instances: `Fred: User`

Flow of Events: 1. Fred is invited to a meeting with some software engineering students at the TU München. He is equipped with a backpack with two laptops, a head-mounted display with an attached a digital video camera, a headset and microphone for voice input, a GPS/compass combination and

a RFID tracking device. Fred has a PostScript handout on one of his laptops, and has already registered the handout to be printed as soon as he reaches the TU main building. The students Fred is supposed to meet with have told him to take the subway to the station Königsplatz.

2. Fred emerges from the Subway station and walks towards the exit. As he comes within reach of an information terminal on his way, an option appears on his display letting him download personalized navigation instructions to the meeting room. He says "yes" to accept this data transfer. He sees a a message that the download is in progress. After a while a message appears saying that the data transmission is complete.
3. On Fred's head-mounted display, a three-dimensional map of the area appears. It shows his own position with a red dot and rotates as he turns, showing his current orientation. A blue arrow indicates his destination. Fred uses this map to guide him to the entrance of the TU.
4. As Fred reaches the TU, an option appears to let him send off the print job for his handouts by wireless LAN[1]. Fred confirms this by saying "yes" again.
5. Inside the building, Fred is guided by a schematic two-dimensional map, indicating which room he is currently in (the position is read from RFID tags), to the hallway outside of the meeting room.
6. Here, he sees a red arrow appear in his head-mounted display, pointing to one of two printers, which has printed his handouts.
7. Fred picks up his handouts from this printer, says "ready", and a three-dimensional blue arrow appears, pointing him to the meeting room.
8. Fred enters the meeting room, takes off his head-mounted display and backpack and greets the students.

7.1.2 Requirements Analysis

Here, I will very briefly analyze some requirements for this demonstration system.

Participating actors for the mobile system are Fred and the information terminal. We assumed that the Information Terminal is a black box which can generate navigation instructions on demand, and transfer them via Bluetooth.

Application-Domain Objects Two interesting application-domain objects in this scenario are:

- The *Room* that Fred is in. The indoor navigation instructions guide him from one room to the next, and the behavior of the system changes between outdoors (essentially a large room), indoors, and the hallway outside the meeting room.
- The *Printer* that prints the handouts.

[1] Local Area Network

Functional Requirements The system has to support three main tasks for the user:

- selecting navigation instructions,
- outdoor, indoor and in-room navigation,
- and using dynamically found Services such as printers.

Nonfunctional Requirements are essentially that the navigational maps are accurate and up-to-date enough to be useful; that the mobile system is lightweight enough to carry around, and that its battery life is at least several hours.

7.1.3 System Design

The system was built with DWARF, so the basic system design follows the framework design shown in Figure 2.7 on page 21.
The mapping of DWARF components onto the demonstration system, shown in detail in in the next section, was:

- The navigation instructions were assumed to have been generated by the information terminal, and were downloaded via the DWARF Bluetooth Service.
- These navigation instructions were then executed by the the Taskflow Engine.
- The World Model received a model of the TUM campus and the relevant rooms, which was also downloaded from the information terminal.
- User interface scenes for initialization, navigation, printer highlighting and Service selection were modeled in the User Interface Engine.
- The selection of a printer for the handouts was managed by the Context-Aware Packet Routing Service.
- The various tracking service determined the user's position, both indoors and outdoors.

Some of the application's logic had to be programmed specifically for the demonstration system, however:

- bootstrapping functionality that lets the user select (via a scene in the User Interface Engine) navigation instructions to download, initiates the Bluetooth transfer, and passes the downloaded data to the DWARF Services,
- mediating functionality between the CAP Service and the User Interface, for the confirmation and status reports for the print job,
- generation of *ContextData* events from the tracking data, indicating which room the user is in, which causes the Taskflow Engine to move to the next navigation step,
- and generation of *ContextData* events based on availability of the WaveLan network, as the DWARF Network Service has not been built yet.

7.1.4 DWARF Components in the Demonstration System

Here, I show how the DWARF Services are used in the demonstration system, in the order in which they appear in the demonstration scenario.

Context-Aware Service Selection and Execution

In the fist step of the scenario, the user registers a print job to be printed when he enters the TU. He does this by creating a Context-Aware Packet (using an appropriate tool) and registering this with the DWARF CAP Service.

In order to be able to print in an unknown environment, all of the user's configuration data, e.g. paper size, preferred color model etc., is stored in the packet. The packet also specifies contains the contextual conditions under which the print job should be started. In this case, that means "when the user is within the TUM's wireless network."

The CAP router gathers *ContextData* events from the other DWARF Services, and starts the print job when the wireless network becomes available.

Bluetooth Communication

The demo scenario involves an information terminal that allows the user to download location dependent data. To make the interaction with the information terminal as painless as possible for the user, we chose to communicate wirelessly using Bluetooth.

For this, we used the DWARF Bluetooth Communication Service's file transfer mechanism. The mobile system tells the information terminal that its user wants to go to the meeting room (and print something along the way), and the information terminal sends a compressed file which contains:

- a geographic and geometric description of the area of the TUM campus, including locations of printers along the way, which is loaded into the World Model,
- a Taskflow which guides the user towards the meeting room step by step and room by room, which is loaded into the Taskflow Engine,
- and abstract user interface descriptions of the navigation scenes to be displayed during this navigation process, which are stored locally so the Taskflow Engine can send them to the User Interface Engine at the appropriate time.

In an ideal world, this information would be generated dynamically using intelligent routing algorithms—but for our demo application, it was hand-coded to support navigating to exactly one meeting room.

World Model

The geographical and geometric information from the information terminal is stored in the mobile system's DWARF World Model Service.

The World Model's tree-like representation includes the TUM campus and the rooms in it, as well as their three-dimensional representations. This is accomplished by using one XML file to describe the campus and the rooms, and embedding pointers to VRML files which contain the three-dimensional descriptions, e.g. a large-scale campus map.

Taskflow Engine

The DWARF Taskflow Engine stores and handles the remaining data transferred from the information terminal. A Taskflow represents the process of navigating from the subway station to the meeting room. States in this Taskflow are, for example, "outside", "in the hallway on the first floor" and "in the meeting room".

Every state has a textual or graphical description of a navigation task, e.g. "go up the stairs" or an image of the stairs that have to be taken. This is sent to the User Interface Engine.

The Taskflow Engine reacts to incoming *ContextData* Events with updates on the user's position, which are generated by the application from the tracking data. By evaluating them, it can switch to new states of the navigation task.

User Interface Engine

The DWARF User Interface Engine is responsible for interaction with the user. For our demo application, it displayed three different kinds of scenes to display:

- outdoor navigation, showing a three-dimensional map of the TUM campus, rotated to match the user's orientation,
- indoor navigation, showing two-dimensional maps of rooms, halls, stairs and doorways,
- and three-dimensional highlighting of a specified object, the printer used for the hand-outs.

Only the last scene, overlaying an arrow over the printer, fits the classical definition of AR [2]. However, we found that for the task of navigation, schematic maps can provide a better overview than virtual arrows floating in mid-air would. For and example image displayed to the user, see Figure 7.3 on page 110.

The information on which scene to display when comes from the taskflow engine.

Additionally, the User Interface Engine can display status information such as "print job started" and processes user input by voice recognition, in order to start the bluetooth download or to confirm the print job.

Tracking Devices

Simple Trackers For outdoor navigation, we used a simple GPS Receiver attached to the GPS Tracking Service. This provides accurate enough data to display the user's position on a map, but not to accurately overlay outlines of buildings in three dimensions. This is one reason why we chose a schematic map for outdoor navigation.

For indoor navigation, we originally intended to use RFID tags, which can identify each doorway as the user walks through it. Unfortunately it was not possible to find a obtain such tags in the narrow time frame of the project. As an alternative, we implemented a software simulation of RFID tags—a "manual tracking Service" in which an assistant entered the ID of the room the user was entering. Porting this software to use RFID tags should be trivial.

Optical Tracker For the AR highlighting of the printer that printed the user's handouts, we needed accurate and fast tracking. For this, we used the Optical Tracking Service.

The area around the printer was measured accurately, and the optical tracker could detect carefully-placed fiducial markers on and around the printer. It used this to calculate the user's position, and sent position data to the VRML display component of the user interface.

7.1.5 Deployment

Figure 7.1: A side view of our prototype wearable computer built with DWARF.
Note that two laptops are used, and that there are no trailing cables. The display on the lower laptop was kept open so that the user's view could be shown to an audience as well.

The wearable system we built is shown in Figure 7.1. We used two laptops, with various attached peripheral devices.

- A Sony Vaio PCG-C1XD PictureBook (above) with a Pentium II-400 processor and 128 MB of RAM, running Microsoft Windows 98, with
  - a Sony DFW-VL 500 FireWire Camera for optical tracking,
  - an Ericsson Bluetooth device for access to the information terminal,
  - and a PC Card Ethernet adapter for access to the other laptop.
- A Dell Inspiron 5000 (below) with a Pentium III-450 processor and 192 MB of RAM, running Microsoft Windows NT 4.0, with
  - a Sony PLM-S 700 Glasstron see-through head-mounted display,
  - a Garmin eTrex Summit GPS receiver,
  - a Lucent WaveLAN wireless network PC Card adapter (using Apple AirPort base stations),
  - and a PC Card Ethernet adapter for access to the other laptop.

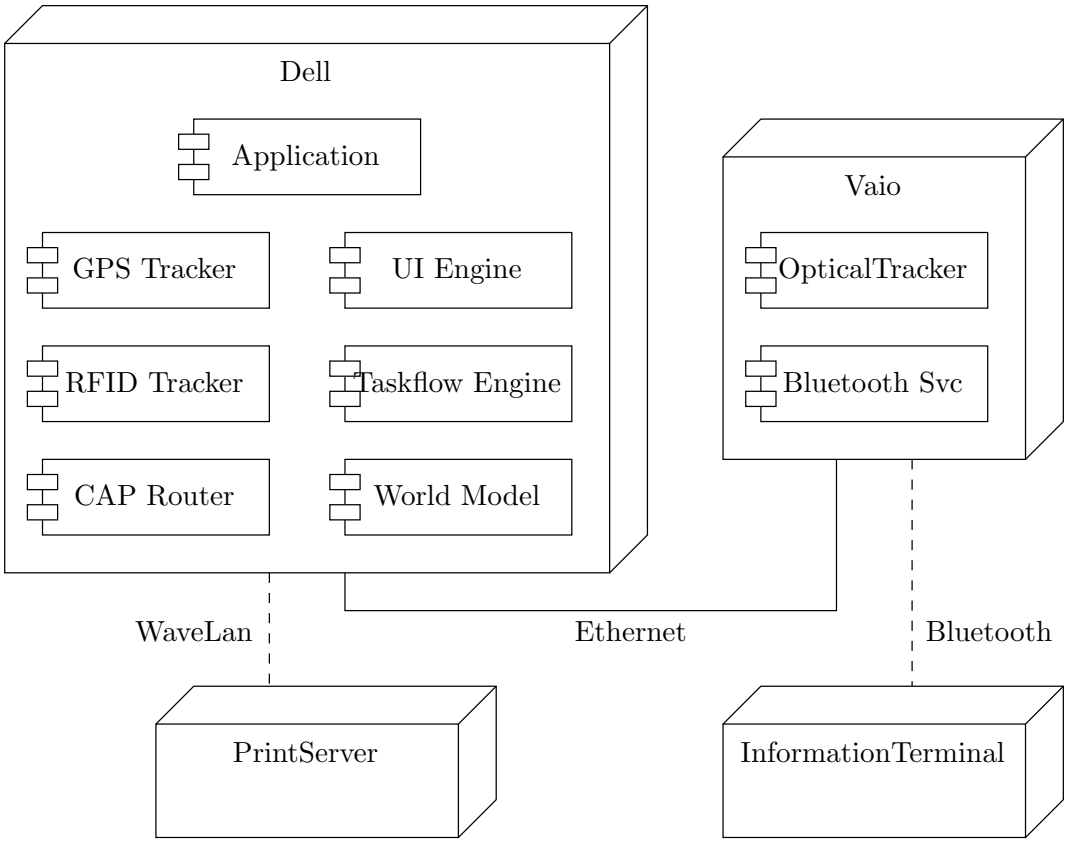


Figure 7.2: Deployment of the DWARF Services in the demonstration system

All devices are battery-powered, and the time of operation is more than two hours.

The deployment of the DWARF Services on the laptops is shown in Figure 7.2. For the middleware, we used the "no local mediating agent" deployment on page 77 for the Sony Vaio, as all the middleware was on the Dell Laptop and there was a reliable ethernet connection between the two.

7.1.6 Results

The demonstration system was implemented within a short period of time—for setting up the demo application using the existing components, no more than three weeks of work were necessary. To be fair, of course, the demonstration scenario had been carefully chosen so that the currently implemented DWARF components could be demonstrated successfully.

The scenario was demonstrated successfully, and the system performed as it should. Please see the DWARF web page at [19] for a movie.

Figure 7.3 on the following page shows a scene from the outdoor navigation part of the scenario, as the user would see it (without, however, the arrow indicating where to go).

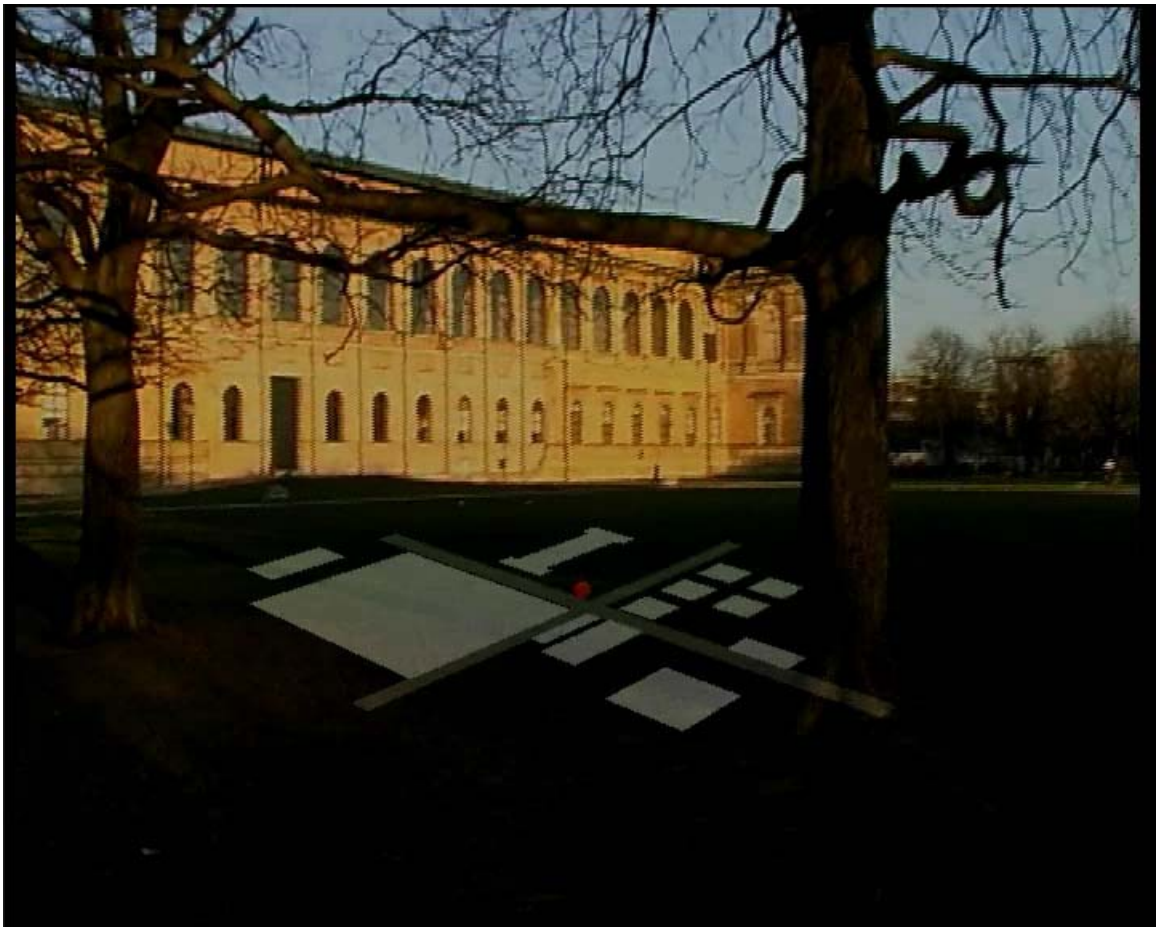


Figure 7.3: User's view of outdoor navigation with the DWARF demo system.
A three-dimensional map is superimposed on the user's field of view. Note that the building ahead is also directly ahead on the map, since the map is oriented as the user is.

7.2 Maintenance: The STARS Scenario

In this section, I describe how a system built with DWARF could address a maintenance scenario, which was inspired by the Sticky Technology for Augmented Reality Systems (STARS) [69] project.

7.2.1 Scenario

This scenario uses *stickies*, virtual text tags that stay attached to real objects. These are essentially virtual post-it notes.

Scenario: **Maintenance of an F-18 fighter using stickies**

Actor instances: `Sam, Frank, Jeff: User`

Flow of Events:

1. Sam is supervisor for maintaining an F-18. He is wearing a mobile AR system built with DWARF. He inspects the plane and finds several rusty spots. He points towards them, and says "rust".
2. The wearable recognizes his gesture and adds stickies to the rusty spots with the label: "rust, please repair!"
3. Later, the mechanic Frank arrives at the plane with his mobile AR system and the stickies are displayed in his head-mounted display at the corresponding places.
4. Frank removes the rust, points to the stickies and says "paint". This alters the text of the stickies to: "please paint!"
5. Jeff, a varnisher, is notified that his skills are needed. When he arrives at the plane with his wearable system, he sees the stickies in his head-mounted display.
6. Jeff paints the corresponding parts, says "done", and the stickies are removed.

7.2.2 Building the System with DWARF

I will not attempt a detailed system design here. I would simply like to show the extent to which the current framework could support such a scenario.

Current DWARF Components Most of the basic functionality can be accomplished with the current DWARF components.

- The positions and labels of the stickies are stored in the World Model.
- The Optical Tracker establishes the user's head position to keep the stickies in the correct location.
- The stickies are displayed by the VRML Display, which is controlled by the User Interface Engine and extracts the information of where to display stickies from the World Model.
- The voice recognition Service can recognize simple spoken commands.

- The User Interface Engine processes user interface scenes and generates appropriate events when the user says “paint”, “rust” or “done”.

New DWARF Components One new Service could be added to the framework for this system:

- A finger tracker. For example, this could use a video camera to locate the position of a brightly colored glove. This does not have to deliver the coordinates of the fingers in three dimensions, as the system can assume the hand is on the flat surface of the airplane hull.

Application Logic The following Services are application-specific and would have to be programmed specially for this system.

- A Sticky Management Service which receives the events from the User Interface Engine and combines them with the position from the finger tracker to make appropriate modifications to the stickies in the World Model.
- A Service that notifies Frank and Jeff that they have work to do when stickies are created for them.

8 Conclusion

We now have a useful first prototype of the framework and its middleware, although there is still more work to do.

This chapter concludes my thesis. First, I would like to take a step back from the technical details of the previous chapters and take a look at the results of the DWARF project in general and my design of the middleware in particular. Finally, I will show some possibilities for future work for both of these areas.

8.1 Results

8.1.1 Framework for Mobile AR in Intelligent Environments

The main result of the DWARF project is the DWARF system design.

New Way of Interacting with Computers Using AR in intelligent environments is a whole new form of human-computer interaction. We do not know yet how useful it will be or whether it will become accepted in the long term. We believe it has great potential, however, and DWARF is thus part of an exciting field.

Framework for Augmented Reality DWARF is the first framework that we know of to have been designed to target the entire field of Augmented Reality. Again, we do not know whether this will work, but the test results so far have been encouraging. In any case, a framework encourages AR research, since new components can easily be integrated into test systems.

8.1.2 Validation of the DWARF Architecture

But not only did we design DWARF, we also tested it in order to validate the architecture.

Demonstration System was Built Quickly The whole idea of a framework is to be able to build systems quickly—and our demonstration system was built with DWARF in three weeks.

Scenario was Successfully Demonstrated We used a complex navigation scenario that took place both indoors and outdoors for our demonstration, and it worked.

Performance Our demonstration system was performant enough for AR, despite the fact that it was a distributed system built from off-the-shelf, battery-powered components.

8.1.3 Middleware Design for Self-Assembling Systems

The middleware I designed for DWARF supports a new kind of system: one that assembles itself dynamically from its constituent parts.

The Way It Should Be Building Systems with DWARF is easy: you just plug the components together, and they work. This makes life easier for two groups of people: for users, who do not have to deal with configuring components at run time, and also for developers, who do not have to deal with configuring components at development time. Especially for users, this is "The Way It Should Be."

Are We Still Building Systems? The classical definition of a *System* has two points: first, a system has a boundary defining what is inside and what is outside, and second, a system consists of subsystems. "Systems" built with DWARF consist of subsystems (the DWARF Services), but their boundary constantly changes, as new Services in the environment are added and removed. This raises issues in many areas, e.g. for security, manageability, and how such "systems" should be modeled.

Implications for Manageability In fact, in developing DWARF in our lab, we we discovered a potential disadvantage of systems that try to assemble themselves. A display Service was receiving bad position data from somewhere, but we did not know where. After a thorough search, we discovered that someone had left a "fake" tracking service running on a computer and forgotten about it. This should not happen to the user once a system has been deployed—systems that are "too" intelligent become hard to control.

Meta-Framework A framework allows developers to build many different applications quickly. With DWARF, we may ultimately build systems that can configure themselves into different applications automatically. In that sense, we will be building frameworks, not systems. Have we designed a meta-framework?

New Role for the Middleware Finally, in this kind of self-assembling system, the middleware plays a new role. It is more active than the classical idea of middleware—it changes the shape of the system using it. In fact, the term "middleware" may not be appropriate anymore. I would warmly welcome any suggestions for a new and better name!

8.1.4 First Implementation of the DWARF Middleware

The middleware I designed is not vaporware; a first implementation has been written, tested and used.

Tested in the Demonstration Scenario The middleware was tested in our demonstration scenario, in which it had to coordinate nine different Services on two computers. This was a fair test for a first implementation. Furthermore, it satisfied the crucial nonfunctional requirement that the latency of events be low enough: it remained under ten milliseconds.

Allows More DWARF Services to be Developed With the middleware, the infrastructure is in place to develop new DWARF Services and assemble new systems, so research can continue in this area. The Services are easier to develop, too, since tedious details of network communication are taken care of by the middleware.

Successfully Deployed on Multiple Platforms The middleware was developed on Linux for PowerPC processors and deployed on Windows for Intel x86 Processors—and worked.

Easily Extensible Using the modular interfaces of the communication and location subsytems, new communication methods or service discovery protocols can be implemented and tested.

8.2 Lessons Learned

In this section, I would like to note a few lessons I learned in working on the DWARF middleware. Mainly, these involve portability and compatibility.

Interoperability of CORBA Implementations CORBA is great, when it works. As mentioned in Section 4.1.1, we discovered several interesting interoperability issues, which took a substantial amount of time to fix.

Availability of CORBA Notification Service Implementations CORBA standards are great, when they are implemented. It proved quite difficult to find an implementation of the CORBA Notification Service that actually met our needs.

Java Cross-Platform Compatibility Writing portable code in Java is great, when it works. In attempting to write a Java applet to control the VRML browser, Christian Sandor and I tested two browsers, three virtual machines, and four ORBs in all possible combinations. One worked.

C++ Cross-Platform Compatibility Writing portable code in C++ is great, and it works. Although Microsoft C++ tends to rename standard C++ functions to something slightly different (thanks to Martin Wagner for helping me with this), writing the middleware on Linux and deploying it on Windows caused fewer problems than I expected.

8.3 Future Work

This section describes several possible future projects for extending, expanding, consolidating, and using the DWARF middleware and framework.

8.3.1 Extensions to the Middleware's Implementation

The current middleware implementation is not complete, and could be extended in various ways.

Integration of an SLP Service Locator This is the highest-priority item on the list of future projects. Without a working location subsystem, the concept of Distributed Mediating Agents will not work.

Integration of an XML Service Description Parser The Service Manager currently does not parse XML files for Service Descriptions. This could be added easily by adding a SAX-style XML parser.

Less Paranoid Error Handling The main goal in the current implementation's error handling is that the middleware itself stay running. Thus, it refuses to communicate with Services that have behaved incorrectly. This could be made somewhat nicer by using detailed exceptions to indicate the severity of error conditions.

Extending Connectors The communication subsystem is easily extensible, and it would be useful to build Connectors with new features. For example, a new CORBA Object Connector could be written that tunnels IIOP through SSL or SSH, providing transparent encryption. Thanks to Thomas Reicher for this idea.

Implementation of Shared Memory Connectors This would be one of the most useful new Connectors to implement, as it would allow the optical tracker to communicate with a local OpenGL display via shared memory. The Connector could provide a uniform interface for accessing shared memory in a platform-independent fashion.

Make Middleware Self-Assembling Why shouldn't the Middleware for a self-assembling system be self-assembling itself? It would be easy to modify the Service Manager executable so that it contains only Connector Factories for CORBA Objects, and has a Need for other Connector Factories. Additional Connector Factories, such as for the Notification Service, could then be written as DWARF Services that are started on demand when the Service Manager needs them.

8.3.2 Extensions to the Middleware's Design

The middleware's design could be extended in many different areas.

Add Optimization Criteria to Service Search Need Descriptions could be extended to specify which attributes of other Services they want to optimize for, e.g. speed over accuracy. The Service Manager could then provide them with the best matching Ability.

Access Other Service's Descriptions By extending either the Connector interface or adding parameters to the Need and Ability interfaces, a Service could receive a reference to another Service's Description when it is connected to it. This would be useful for Services like the Tracking Manager that need to know the properties of the trackers it is receiving data from. Thanks to Martin Bauer for this idea.

Full Service Life Cycle Support The proposed DWARF system consists of many very small computers. It should be possible to administer these remotely. For this, the middleware should support installation and configuration of Services as well as accessing Service status information.

Security The great dilemma of ad hoc networking involves security: how do you know whether to trust a Service you just found? Results of ongoing research in this area could be investigated as to their usefulness for the DWARF middleware.

Deadlock Detection Avoiding deadlock caused by cyclic dependencies between Services could be accomplished by adding a dynamic attribute to each Service, the recursive list of the other Services it depends on. The middleware could then avoid creating cycles.

Visualization An attribute like this would be useful for visualization as well, as it would allow a tool to display the dependency graph between Services.

Making a Design Pattern The DWARF middleware combines aspects of the *Component Configurator* and *Acceptor-Connector* patterns in [61]. Perhaps a new pattern could be identified from this.

8.3.3 Extensions to the DWARF Architecture

Aside from implementing missing features, the DWARF architecture could be extended in many ways.

Network Services For effective roaming and handover between wireless networks, a DWARF Network Service which delivers connectivity events to the middleware and the other Services should be developed.

New Ways of Representing Information Information to be displayed can currently be represented in the Taskflow Engine and in the World Model. These are both appropriate for a particular kind of information. For representing new types of information, such as visualization of complex data, new DWARF Services would be necessary.

User Model We need a good model of the user with his personal preferences, history and security information in order to be able to deliver context-sensitive data.

Security As mentioned above, Security is a serious issue. How can we build a secure system when we do not even know what the system's boundary is? An idea would be to associate a small hardware device, such as a wrist watch running Linux, with the user, that could provide authentication, trust and configuration services. This would be a good place to store the user model as well.

Multiuser Capability Once we have a user model, we should think about modeling multiple users. This would involve extensions to many Services, such as tracking, the Taskflow Engine, and the User Interface Engine. Several Services already are multiuser capable by design.

More Input Devices Our only current input device is voice recognition—gesture recognition or eye tracking would be useful extensions.

Distribute the Application This is perhaps the most difficult research topic associated with DWARF. The mediating role of the middleware has been distributed across the network using the design in this thesis. The World Model can be distributed using replication mechanisms. We still, however, have a central *application* that configures how the DWARF Services interact.

To eliminate this central component, we could try to model simple AR applications by describing dependencies between events and conversion of data types between Services. This would essentially be a programming language for AR. If this language is designed so that it can be replicated and distributed though the system, we no longer need a central component. Essentially, this would allow the user to "project his will onto the system"—the application would diffuse through the DWARF Services and configure them to do what the user wants.

8.3.4 Extensions to the Demonstration System

The demonstration system we built is still rather clumsy, and it could be extended in two ways.

Build as Permanent Testing and Development Platform A backpack-based demonstration system is useful for development and testing—all components are accessible, and new ones can be added easily. It is also useful for demonstrations, as an audience can see what the user sees. Thus, the "almost-wearable" should be maintained and extended to a development and research platform.

Build Wearable Wearable Ultimately, of course, the wearable should be wearable. This means using very small embedded systems—such as [45] or [38]—to deploy DWARF on. The components can be attached to a belt and be added or removed at run time. The DWARF middleware is designed so that it can be recompiled as-is for these tiny systems.

A Appendix

Reference information for programming with the DWARF middleware.

This appendix contains information which is only of interest to users and developers of the DWARF middleware. It provides a reference to the middleware's interfaces, instructions on how to compile and install the middleware, and additional documentation for future developers.

A.1 Interface Definitions for the Middleware

This section contains the CORBA IDL interface definitions for the DWARF middleware. The functionality of the interfaces is described in Section 5.9.

A.1.1 DWARF Services

```
// these interfaces are implemented by Services that want to use the Service Manager.
interface Service {
        void startService();
        void stopService();
};

interface Need {
        void connectNeed(in string name, in short instance, in Connector conn);
        void disconnectNeed(in string name, in short instance, in Connector conn);
};

interface Ability {
        void connectAbility(in string name, in Connector conn);
        void disconnectAbility(in string name, in Connector conn);
};

// helper interface for a simple Service that is implemented as one object
interface ServiceAndNeedAndAbility:
        Service,
        Need,
        Ability { };
```

A.1.2 Service Manager

Describing Services

```
interface DescribeServices {
        ServiceDescription newServiceDescription(in string name);
        ServiceDescription getServiceDescription(in string name);
        void deleteServiceDescription(in string name);
        void activateServiceDescription(in string name);
```

```
};

interface ServiceDescription {
        string getName();
        void setStartCommand(in string cmd);
        string getStartCommand();
        void setStartOnDemand(in boolean yesno);
        boolean getStartOnDemand();
        void setStopOnNoUse(in boolean yesno);
        boolean getStopOnNoUse();
        NeedDescription newNeed(in string name);
        NeedDescription getNeed(in string name);
        void deleteNeed(in string name);
        AbilityDescription newAbility(in string name);
        AbilityDescription getAbility(in string name);
        void deleteAbility(in string name);
};

interface NeedDescription {
        string getName();
        void setType(in string type);
        string getType();
        void setPredicate(in string name);
        string getPredicate();
        void setMinInstances(in short num);
        short getMinInstances();
        void setMaxInstances(in short num);
        short getMaxInstances();
        ConnectorDescription newConnector(in string protocol);
        ConnectorDescription getConnector(in string protocol);
        void deleteConnector(in string protocol);
};

interface AbilityDescription {
        string getName();
        void setType(in string type);
        string getType();
        void setAttributes(in string attributes);
        void getAttributes(out string attributes);
        ConnectorDescription newConnector(in string protocol);
        ConnectorDescription getConnector(in string protocol);
        void deleteConnector(in string protocol);
};

interface ConnectorDescription {
        string getProtocol();
        void setConnectorType(in string type);
        string getConnectorType();
        void setManualURI(in string uri);
        string getManualURI();
        void setDisconnectTimeout(in double seconds);
        double getDisconnectTimeout();
};
```

Registering Services

```
interface RegisterServices {
        ActiveServiceDescription registerService (in string name, in Service svc);
```

```
        ActiveServiceDescription registerServiceAndNeedAndAbility(
                in string name, in ServiceAndNeedAndAbility svc);
        void unregisterService(in string name);
};

interface ActiveServiceDescription:
        ServiceDescription {
        void registerNeed(in string name, in Need n);
        void registerAbility (in string name, in Ability a);
};
```

Registering Connector Factories

```
interface RegisterConnectorFactories {
        void addConnectorFactory(in ConnectorFactory cfactory);
        void removeConnectorFactory(in ConnectorFactory cfactory);
};
```

A.1.3 Communication Subsystem

Connectors

```
interface Connector {
        string getType();
        string getProtocol();
        void disconnect();
};

interface ProtocolConnector:
        Connector,
        LocatorOffer {
        void setConnectCallback(in ConnectCallback callback);
        void connectTo(in string location);
        void destroy();
};

interface ConnectCallback {
        void connected(in string location, in Connector conn);
        void disconnected(in string location, in Connector conn);
};
```

Connector Factories

```
interface ConnectorFactory {
        ProtocolConnector createConnector(in string type);
        boolean supportsType(in string type);
};
```

CORBA Object Connectors

```
interface SvcConnCorbaObjExporter {
        //set the object that is to be exported
        void setObject(in Object obj);
};

interface SvcConnCorbaObjImporter
{
        //get the object that has been imported
        Object getObject();
};
```

Notification Service Connectors

```
interface SvcConnNotifyStructuredPushConsumer {
        //return proxy. Use connect_push_consumer to connect.
        CosNotifyChannelAdmin::StructuredProxyPushSupplier getProxySupplier();
};

interface SvcConnNotifyStructuredPushSupplier {
        //return proxy. Just push into it.
        CosNotifyChannelAdmin::StructuredProxyPushConsumer getProxyConsumer();
};
```

Helper Interfaces

```
//The following are convenient interfaces to implement if you have a Service,
//Need or Ability that wants to send or receive events.
// ability to supply events / need to receive events (e.g. tracker/display)
interface AbilityIsNotifySupplier:
        Ability,
        CosNotifyComm::StructuredPushSupplier { };

interface NeedIsNotifyConsumer:
        Need,
        CosNotifyComm::StructuredPushConsumer { };

// ability to consume events / need to supply events (e.g. printer/word processor)
interface AbilityIsNotifyConsumer:
        Ability,
        CosNotifyComm::StructuredPushConsumer { };

interface NeedIsNotifySupplier:
        Need,
        CosNotifyComm::StructuredPushSupplier { };

//all-in-one-object services that want to send and/or receive events
interface ServiceAndNeedAndAbilityIsNotifySupplier:
        ServiceAndNeedAndAbility,
        CosNotifyComm::StructuredPushSupplier { };

interface ServiceAndNeedAndAbilityIsNotifyConsumer:
        ServiceAndNeedAndAbility,
        CosNotifyComm::StructuredPushConsumer { };

//this wins the prize for the longest interface name
interface ServiceAndNeedAndAbilityIsNotifySupplierConsumer:
        ServiceAndNeedAndAbility,
        CosNotifyComm::StructuredPushSupplier,
        CosNotifyComm::StructuredPushConsumer { };
```

A.1.4 Location Subsystem

```
interface LocatorOffer {
        string getLocation();
        void query(out string attributes);
};

interface LocatorRequest {
        void found(in string location, in string attributes);
};
```

```
interface ServiceLocator {
        void registerOffer(in string serviceID, in string type, in LocatorOffer offer);
        void registerRequest(in string serviceID,
                in string type, in string predicate, in LocatorRequest request);
        void unregisterOffer(in LocatorOffer offer);
        void unregisterRequest(in LocatorRequest request);
        void unregisterAllOffers(in string seriviceID);
        void unregisterAllRequests(in string seriviceID);
        void doQuery();
};
```

A.2 Installation Instructions for the Middleware

This section briefly describes how to compile and install the DWARF middleware. This is not a trivial task, and if you run into difficulties, please contact me. I am currently assembling a CD-ROM to accompany this thesis containing the source code and libraries, which interested parties may have a copy of.

A.2.1 Required Libraries

The DWARF middleware uses the following compilers and libraries. Newer versions should probably work as well. Aside from the compilers, I will include these (including evaluation versions of the commercial products) on the CD-ROM, and they are available on our file server.

- g++ version 2.95.2 (for Linux)
- Microsoft Visual C++ version 6.0 (for Windows)
- OmniORB version 3.0
- OmniNotify version 1.01 (for Linux)
- dCon version 2.2 (for Windows)
- VisiBroker version 4.1 for Java (for Windows)
- JavaORB version 2.2.6 (if you want to build stubs to access the middleware from Java)

These should all be installed according to their respective installation instructions.

A.2.2 Compiling the Source Code

Compiling consists of two steps: compiling the IDL files and compiling the actual source. Use the source tree from our source code repository, or the copy of it on the CD-ROM.

Compiling the IDL files

The directory `src/idl` of the cvs tree contains the IDL definitions, and the directory `src/stub` contains makefiles for various platforms.

The first step is to build the OmniORB stubs in the directory `src/stubs/omni`. A Makefile is provided for Linux, and a batch file is provided for Windows.

Compiling the Sources

The source code for the Service Manager is in the `src/macwilli/servicecomm/servicemgr` directory. Here, there is a makefile for Linux and and developer studio project for Windows. You may need to adjust the makefiles if you have installed OmniORB in a different path.

The source code for the test program is in `src/macwilli/servicecomm/tests`. Again, there is a makefile for Linux and a developer studio project for Windows.

A.2.3 Installation and Use

Once the libraries have been installed, you can copy the Service Manager executable or the test program to any directory you wish.

Starting the Service Manager requires the following steps:

1. Start the Notification Service and save its IOR to a file.
   - For OmniNotify on Linux, run
     `notifd -DFactoryIORFileName=/tmp/notification.ior`
   - dCon on Windows can simply be started without parameters. It saves its IOR to `c:\tmp\notification.ior`.
2. Start the Service Manager, giving it a reference to the IOR of the Notification Service:
   - For Linux, run
     `servicemgr -ORBpoa_iiop_port 39273 file:///tmp/notification.ior`
   - For Windows, run
     `servicemgr -ORBpoa_iiop_port 39273 file://c:\tmp\notification.ior`

The Service Manager will provide detailed debugging and log messages.

A.2.4 Test Program

A simple test program called `test1` is available which tests the various features of the middleware. It contains three Services, a producer, a transformer, and a consumer. The test program can be started in two different ways:

- Without parameters: `test1`
  This first describes all three services and then starts them manually.
- With one (rather cryptic) parameter: `test1 [d][s][p][t][c]`
  The letters of this parameter, which can all be combined in any order, mean:

  d Describe the specified services.

  s Start the specified services.

  p Describe and/or start the producer.

  t Describe and/or start the transformer.

  c Describe and/or start the consumer.

As a simple test, start `test1 dsp` in one window, and `test1 dsc` in the other. The producer will send position events to the consumer. It is instructional to look at the source code to see how the middleware can be used.

A.2.5 Building Stubs

Before you compiled the Service Manager, you had to build the OmniORB stubs in the directory `src/stubs/omni`. These can be used for DWARF Services written in C++, as well.

If you wish to build stubs to access the middleware form Java, use the Makefile (for Linux) in `src/stubs/javaorb`.

A.3 State Transition Diagrams for the Service Manager Classes

I have included the (somewhat simplified) state transition diagrams which I used as a reference during implementation of the `ActiveServiceDescription_i`, `ActiveNeedDescription_i`, `NeedInstanceConnector_i`, `ActiveAbilityDescription_i`and `AbilityConnector_i`classes.

This will mainly be of interest to anyone who would like to extend or enhance the Service Manager's functionality. When you read the source code, have these diagrams handy.

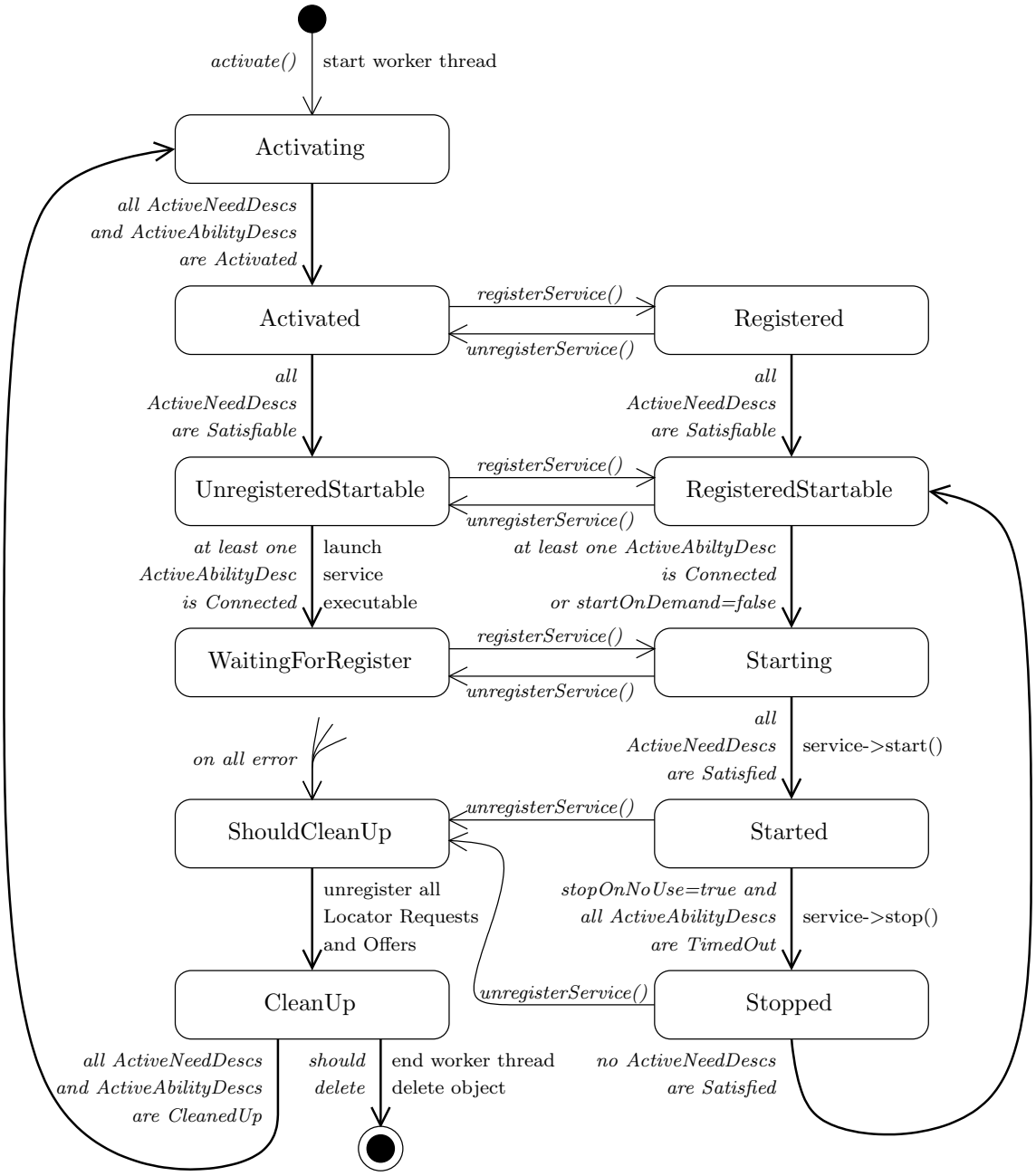


Figure A.1: State diagram for the `ActiveServiceDescription_i` class.
UML statechart diagram. For clarity, transitions that are taken by the worker thread are thicker than those that are done within operations called from the outside. Conditions for a transition are set in *italics*, actions to be taken during a transition are in regular text.
Note the two loops: a small loop on the right-hand side for starting and stopping a Service that stays loaded in memory, and a large one for starting and stopping separate Service processes.
Also note that the state ShouldCleanUp can be entered from all other states, on error conditions.

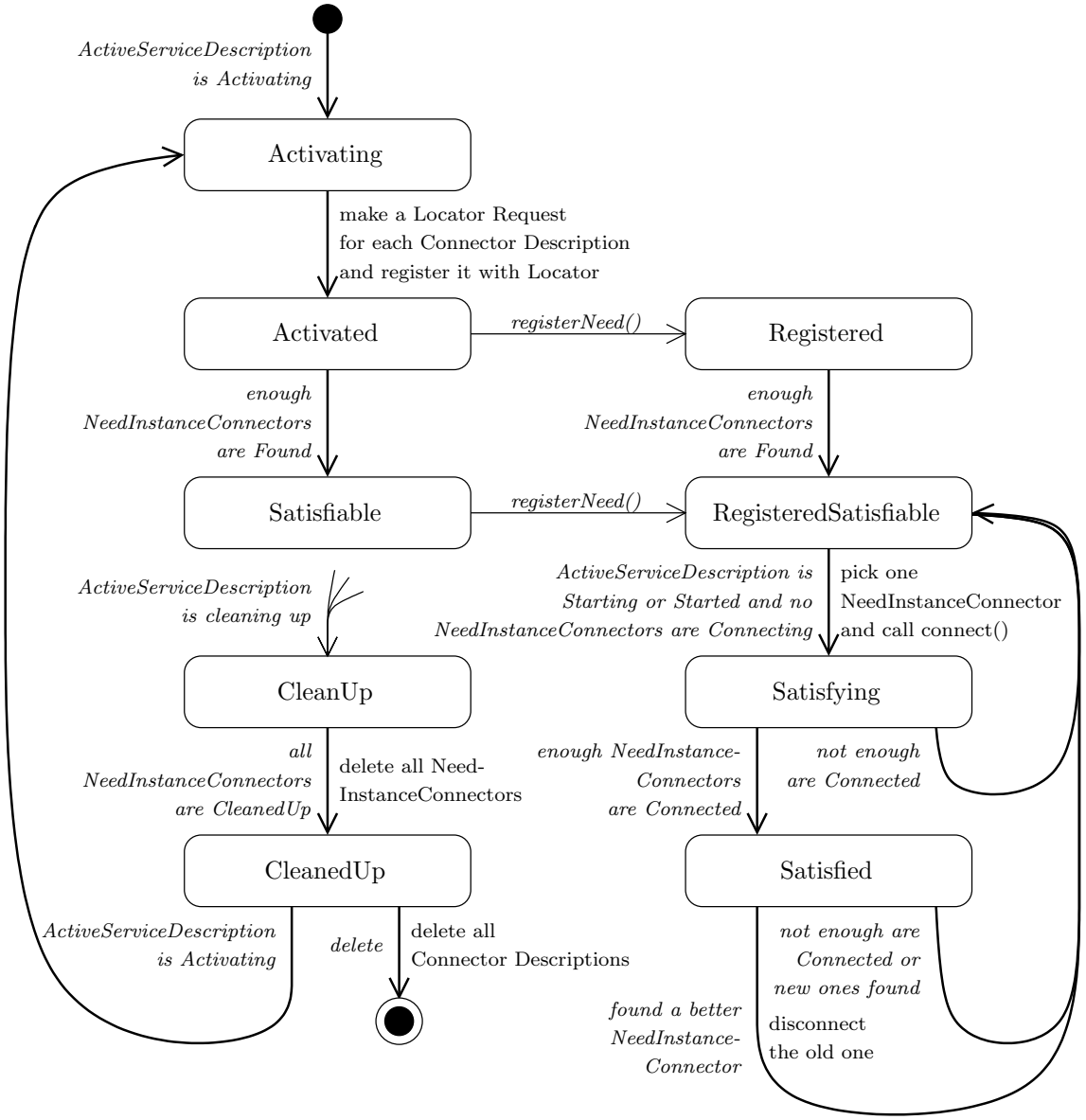


Figure A.2: State diagram for the `ActiveNeedDescription_i` class.
UML statechart diagram, with the same conventions as Figure A.1 on the preceding page.
Note the main loop on the right-hand side that manages the NeedInstanceConnectors, telling them to connect and disconnect appropriately.

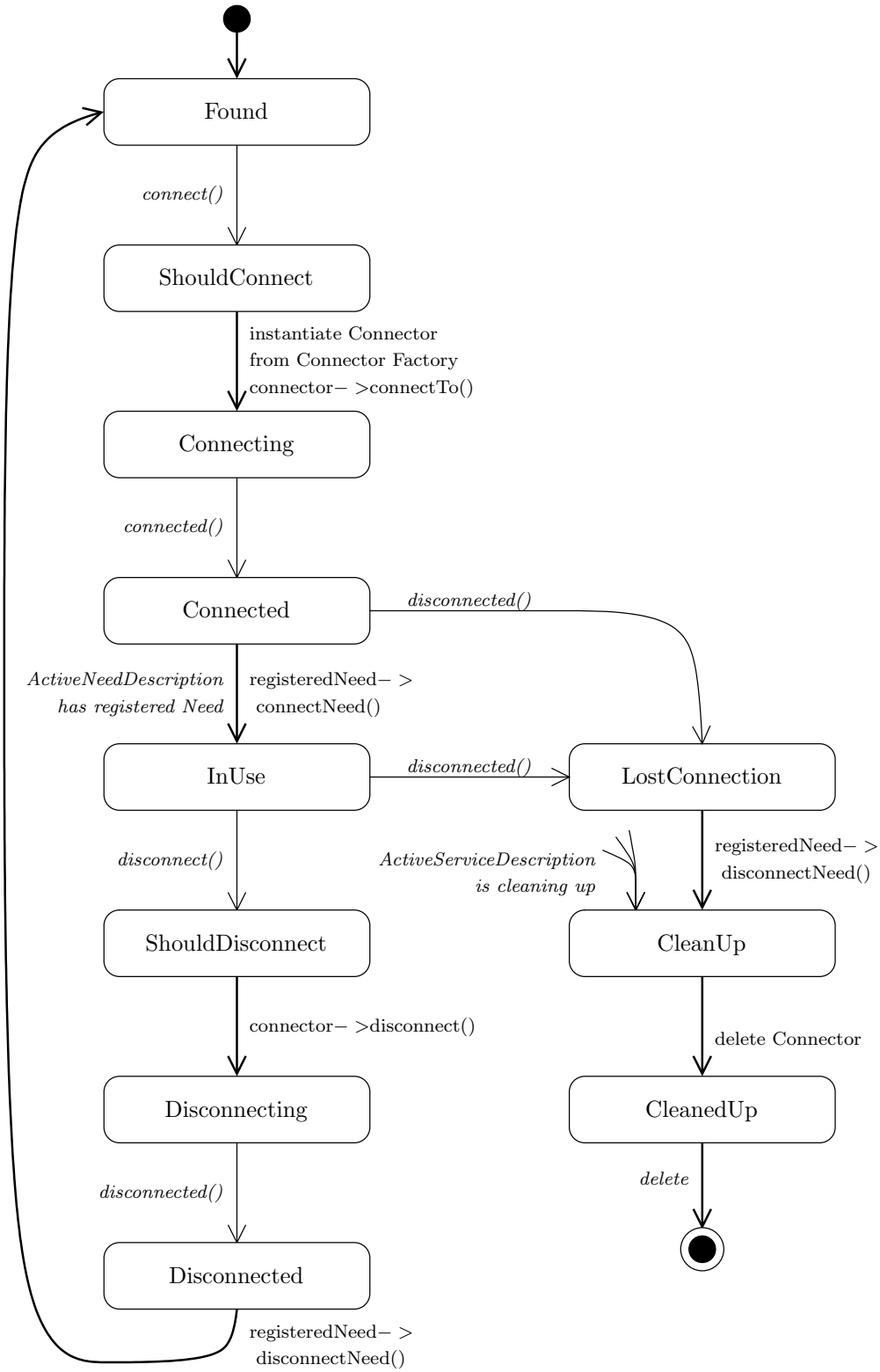


Figure A.3: State diagram for the `NeedInstanceConnector_i` class.
UML statechart diagram, with the same conventions as Figure A.1 on page 126. Note the different states for an intentional disconnection (Disconnected) and an unintentional one (LostConnction).

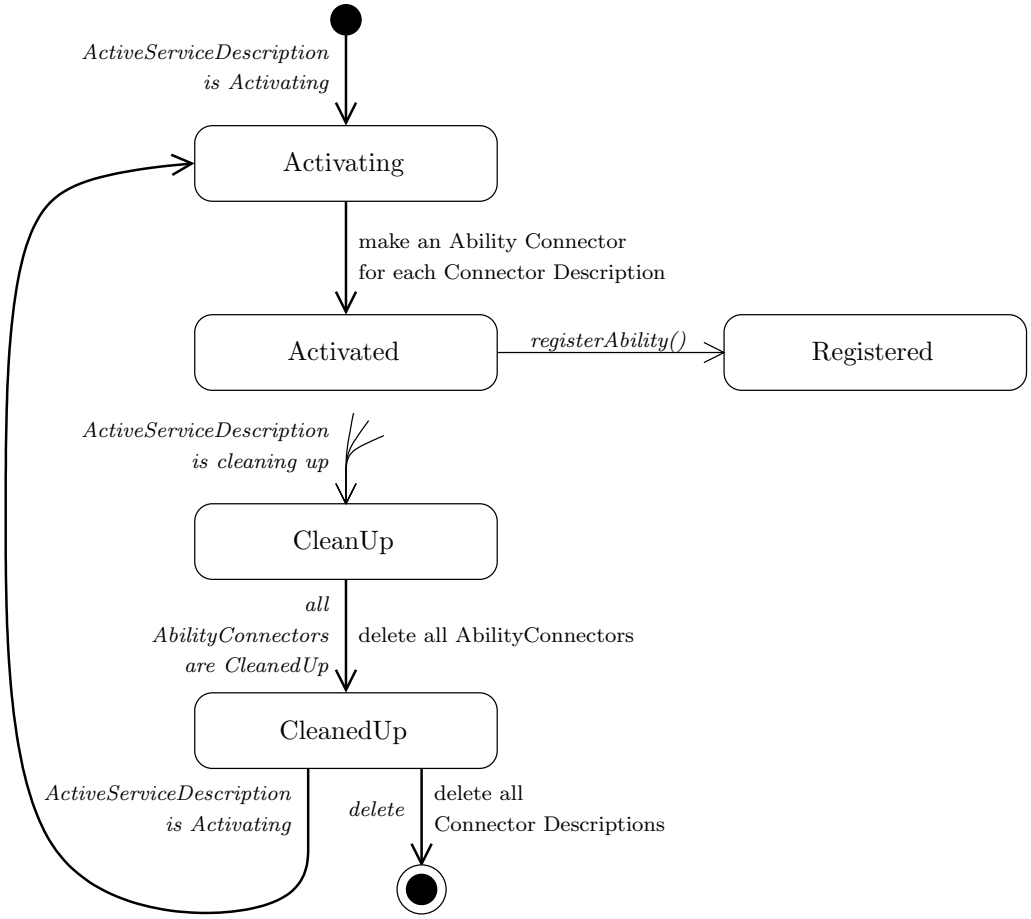


Figure A.4: State diagram for the `ActiveAbilityDescription_i` class.
UML statechart diagram, with the same conventions as Figure A.1 on page 126.

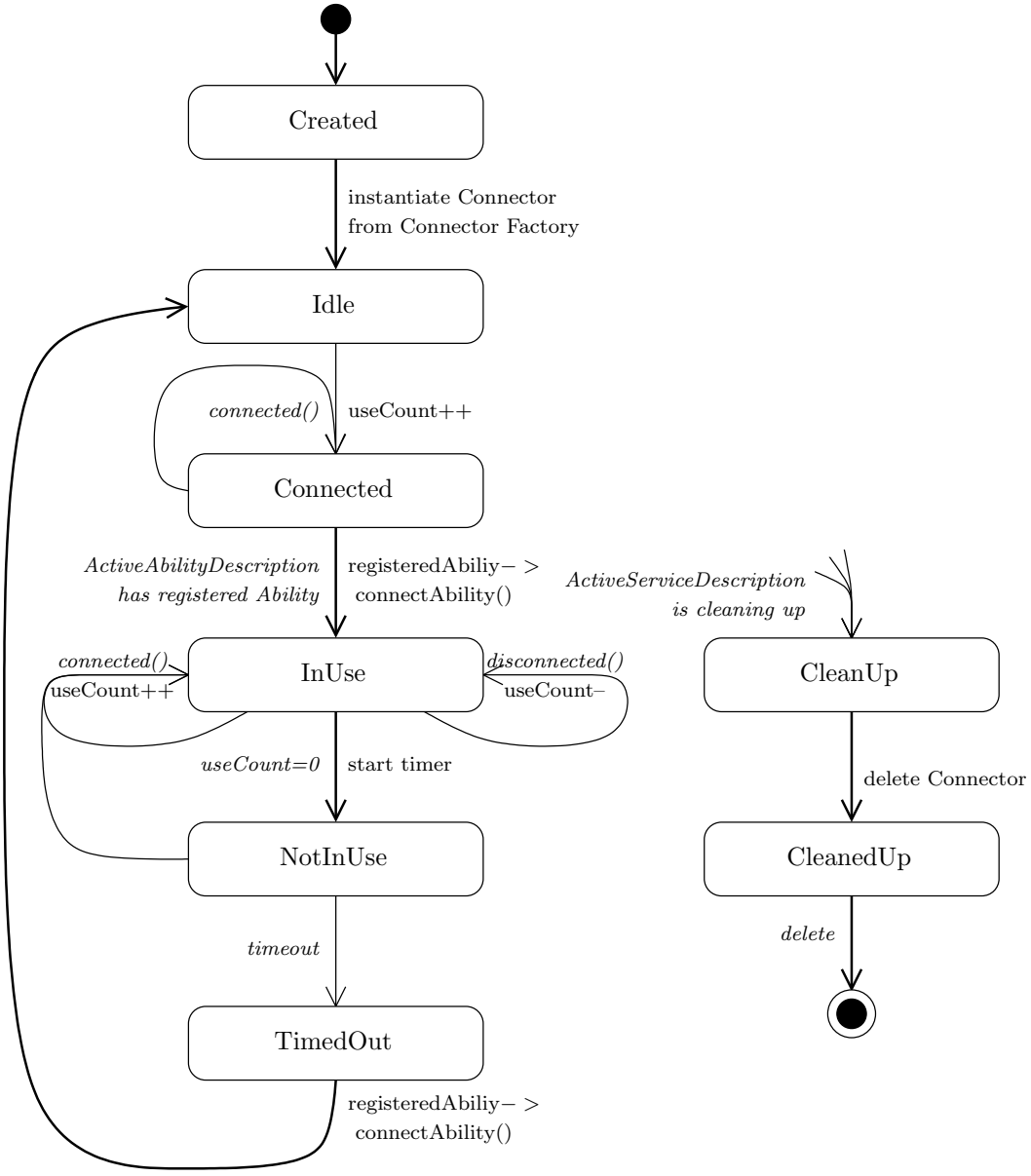


Figure A.5: State diagram for the `AbilityConnector_i` class.
UML statechart diagram, with the same conventions as Figure A.1 on page 126. Note how the variable useCount is incremented and decremented in the InUse state.

Bibliography

[1] R. Azuma, B. Hoff, H. Neely, and R. Sarfaty, *A Motion-Stabilized Outdoor Augmented Reality System*, in Virtual Reality '99, Houston, March 1999. 3

[2] R. T. Azuma, *A Survey of Augmented Reality*, Presence: Teleoperators and Virtual Environments, 4 (1997), pp. 355–385. 3, 107

[3] M. Bauer, *Design and Implementation of a Module for the Dynamic Combination of Different Position Trackers*, Master's thesis, Technische Universität München, February 2001. 15, 24, 44, 100, 102, 103

[4] M. Bauer, A. MacWilliams, F. Michahelles, C. Sandor, S. Riss, M. Wagner, B. Zaun, C. Vilsmeier, T. Reicher, B. Brügge, and G. Klinker, *DWARF: Requirements Analysis Document*. Unpublished, 2000. 15

[5] M. Bauer, A. MacWilliams, F. Michahelles, C. Sandor, S. Riss, M. Wagner, B. Zaun, C. Vilsmeier, T. Reicher, B. Brügge, and G. Klinker, *DWARF: System Design Document*. Unpublished, 2000. 20

[6] *Bluetooth Special Interest Group*. http://www.bluetooth.com, January 2001. 28

[7] B. Brügge and A. H. Dutoit, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000. 4, 11, 29, 29, 31, 33, 34, 34, 35, 38, 43, 43, 46, 47, 47, 48, 79, 80, 95

[8] B. Brügge, R. Pfleghar, and T. Reicher, *OWL: An object-oriented Framework for intelligent Home and Office Applications*, in Proceedings of CoBuild '99, Second International Workshop on Cooperative Buildings, N. Streitz, J. Siegel, V. Hartkopf, and S. Konomi, eds., Pittsburgh, Oct 1999. 3

[9] *Microsoft COM Technologies - Information and Resources for the Component Object Model-based technologies*. Microsoft Corporation, http://www.microsoft.com/com/default.asp, February 2001. 59

[10] *COM versus CORBA: A Decision Framework*. QUOIN Inc., http://www.quoininc.com/quoininc/COM_CORBA.html, February 2001. 59

[11] *CORBA 2.4.2 Specification*. Object Management Group, http://www.omg.org/cgi-bin/doc?formal/01-02-33, February 2001. 54, 55

[12] *CORBA Comparison Project*, tech. rep., Charles University, Prague, June 1998. http://www.kav.cas.cz/~buble/corba/comp/report.pdf. 56

[13] *CORBA Event Service Specification*. Object Management Group, http://www.omg.org/technology/documents/formal/event_service.htm, February 2001. 57, 59

[14] *CORBA Notification Service Specification*. Object Management Group, http://www.omg.org/technology/documents/formal/notification_service.htm, February 2001. 57, 58, 88, 89

[15] *CORBA Trader Service Specification*. Object Management Group, http://www.omg.org/technology/documents/formal/trading_object_service.htm, February 2001. 67

[16] *CORBA Audio/Video Streaming Service Specification*. Object Management Group, http://www.omg.org/technology/documents/formal/audio.htm, February 2001. 59

[17] D. Curtis, D. Mizell, P. Gruenbaum, and A. Janin, *Several Devils in the Details: Making an AR App Work in the Airplane Factory*, tech. rep., Boeing Applied Research & Technology, Linius Technologies, UC Berkley, 1990 - 1997. 3

[18] *dCon Notification Service*. DSTC, http://www.dstc.edu.au/Products/CORBA/Notification_Service/, February 2001. 59

[19] *DWARF Project*. Technische Universität München, http://www.augmentedreality.de, February 2001. 109

[20] *e*ORB*. Vertel Corporation, http://www.vertel.com/corba/default.asp, February 2001. 55

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995. 30, 99

[22] R. E. Gruber, B. Krishnamurthy, and E. Panagos, *The Architecture of the READY Event Notification Service*, tech. rep., AT&T Labs – Research, 1999. 58

[23] E. Guttman, C. Perkins, J. Veizades, and M. Day, *Service Location Protocol*. IETF, RFC 2608, June 1999. 61, 62, 64, 64

[24] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, Reading, MA, 1999. 54

[25] R. Hermann, D. Husemann, M. Moser, M. Nidd, C. Rohner, and A. Schade, *DEAPSpace—Transient Ad-hoc Networking of Pervasive Devices*, tech. rep., IBM Research, May 2000. 67

[26] *Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000*, Munich, October 2000. 3

[27] *Proceedings of the IEEE International Workshop on Augmented Reality – IWAR 1999*, San Francisco, October 1999. 3

[28] *JacORB*. http://www.inf.fu-berlin.de/~jacorb/, February 2001. 57

[29] *jAugment Project*. Universität Rostock, http://www.informatik.uni-rostock.de/~mawol/jaugment/, February 2001. 19

[30] *Java 2 ORB*. Sun Microsystems, http://java.sun.com/j2se/1.3/docs/relnotes/features.html, February 2001. 57, 60

[31] *JavaORB*. Distributed Objects Group, http://www.multimania.com/dogweb/details_javaorb.html, February 2001. 56

[32] *Jini*. http://www.jini.org, February 2001. 66

[33] R. Kehr, *Spontane Vernetzung. Infrastruktur-Konzepte für die Post-PC-Ära*, Informatik-Spektrum, 3, 23 (2000). 61

[34] J. Kempf and E. Guttman, *An API for Service Location*. IETF, RFC 2614, June 1999. 61, 75, 100

[35] G. Klinker, T. Reicher, and B. Brügge, *Distributed User Tracking Concepts for Augmented Reality Applications*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, October 2000. 3, 7

[36] G. Klinker, D. Stricker, and D. Reiners, *Augmented Reality: A Balancing Act Between High Quality and Real-Time Constraints*, in Mixed Reality - Merging Real and Virtual Worlds, Y. Ohta and H. Tamura, eds., Springer Verlag, March 1999. 9, 9, 9

[37] R. L. Lagendijk, *The TU-Delft Research Program "Ubiquitous Communications"*, in 21st Symposiom on Information Theory in the Benelux, May 2000. 19

[38] *LART Project*. TU Delft, http://www.lart.tudelft.nl/, February 2001. 19, 118

[39] A. Lele, S. Nandy, and D. Epema, *Hamony – An Architecture for Providing Quality of Service in Mobile Computing Environments*, Journal of Interconnection Networks, (2000). 19

[40] *Mac OS X Server*. Apple Corporation, http://www.apple.com/macosx/server/features.html, February 2001. 61

[41] A. MacWilliams, *SLP – Service Location Protocol*, in Developing Ad Hoc Component Systems for Mobile Computing, M. Fahrmeier, F. Marschall, and C. Salzmann, eds., Technische Universität München, 2001. (Unpublished). 61

[42] A. MacWilliams, *Systems and Design Considerations*, in Erweiterte Realität: Bildbasierte Modellierung und Tragbare Computer, B. Brügge, H. Niemann, and T. Reicher, eds., Technische Universität München, Universität Erlangen-Nürnberg, 2001. 9

[43] F. Michahelles, *Designing an Architecture for Context-Aware Service Selection and Execution*, Master's thesis, Ludwig–Maximilians–Universität München, February 2001. 27

[44] *MICO*. http://www.mico.org/, February 2001. 56

[45] *MIThril Project*. Massachussets Institue of Technology, http://www.media.mit.edu/wearables/mithril/, February 2001. 11, 18, 118

[46] D. Mizell, *Augmented Reality Applications in Aerospace*, in Proceedings of the IEEE and ACM International Syposium on Augmented Reality – ISAR 2000, Munich, October 2000. 3, 6

[47] *Object Management Group*. http://www.omg.org, February 2001. 54

[48] *OmniNotify*. AT&T Laboratories Cambridge, http://www.research.att.com/~ready/omniNotify/index.html, February 2001. 58

[49] *OmniORB*. AT&T Laboratories Cambridge, http://www.uk.research.att.com/omniORB/index.html, February 2001. 56

[50] *ORBacus*. Object Oriented Concepts, http://www.ooc.com/ob/, February 2001. 56

[51] *ORBacus Notification Service*. Object Oriented Concepts, http://www.ooc.com/notify/, February 2001. 59

[52] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*, Wiley, New York, NY, 1997. 54

[53] C. Perkins, *Service Location Protocol White Paper*. Sun Microsystems, http://playground.sun.com/srvloc/slp_white_paper.html, May 1997. 61

[54] W. Piekarski, B. Gunther, and B. Thomas, *Integrating Virtual and Aumented Realities in an Outdoor Application*, in Proceedings of the IEEE International Workshop on Augmented Reality – IWAR 1999, San Francisco, October 1999. 10

[55] D. Reiners, D. Stricker, G. Klinker, and S. Müller, *Augmented Reality for Construction Tasks: Doorlock Assembly*, tech. rep., Fraunhofer IGD, Department Visualization and Virtual Reality in cooperation with BMW's Virtual Environment Group, July, 10 1998. 9

[56] C. Renner, *SLP Demo Implementation*. Technische Universität München, http://www.lkn.e-technik.tu-muenchen.de/~chris/slp/, February 2001. 62

[57] S. Riss, *An XML-Based Task Flow Description Language for Augmented Reality Applications*, Master's thesis, Technische Universität München, November 2000. 6, 25, 75

[58] D. Roberts and R. Johnson, *Evolving Frameworks. A Pattern Language for Developing Object-Oriented Frameworks*. University of Illinois, http://st-www.cs.uiuc.edu/~droberts/evolve.html, February 2001. 9

[59] C. Sandor, *CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces*, Master's thesis, Technische Universität München, November 2000. 26, 75

[60] K. Sato, Y. Ban, and K. Chihara, *MR Aided Engineering: Inspection Support Systems Integrating Virtual Instruments and Process Control*, in Mixed Reality - Merging Real and Virtual Worlds, Y. Ohta and H. Tamura, eds., Springer Verlag, March 1999. 3

[61] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*, Wiley, New York, NY, 2000. 98, 117

[62] *OpenSLP Implementation*. OpenSLP Group, http://www.openslp.org, February 2001. 62

[63] *Service Location Protocol*. http://www.svrloc.org/, February 2001. 61

[64] *The Service Location Protocol and the Macintosh*. http://www2.opendoor.com/shareway/SLP.html, February 2001. 61

[65] *Service Location Protocol Implementation*. Kempf & Associates, http://www.svrloc.org/kaslp/, February 2001. 62

[66] *Service Location Protocol in Axis Products*. Axis Communications, http://www.axis.com/corporate/research/usability.htm, February 2001. 61

[67] *Service Location Protocol in NetWare 5*. Novell Corporation, http://developer.novell.com/research/appnotes/1998/septembe/03/07.htm, February 2001. 61

[68] *Service Location Protocol Reference Implementation*. Sun Microsystems, http://www.sun.com/research/slp, February 2001. 62

[69] *STARS Project*. Technische Universität München, http://wwwbruegge.informatik.tu-muenchen.de/projects/stars2001/. 111

[70] N. Surendran and S. Mungee, *Control and Management of Audio/Video Streams and Real-time ORB Protocols*. http://www.cs.wustl.edu/~schmidt/orb-protocols.html, February 2001. 59

[71] *TAO*. Washington University, http://www.cs.wustl.edu/~schmidt/TAO.html, February 2001. 56

[72] *UbiCom*. TU Delft, http://www.ubicom.tudelft.nl/, February 2001. 19

[73] *Universal Plug and Play*. http://www.upnp.org, February 2001. 65

[74] H. van Dijk, K. Langendoen, and H. Sips, *ARC: a Bottom-Up Approach to Negotiated QoS*, in 3rd IEEE Workshop on Mobile Computing Systems and Applications, October 2000. 20

[75] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan, *Service Location Protocol*. IETF, RFC 2165, June 1997. 61

[76] *VisiBroker*. Borland Inc., http://www.borland.com/visibroker/, February 2001. 56

[77] M. Wagner, *Design, Prototypical Implementation and Testing of a Real-Time Optical Feature Tracker*, Master's thesis, Technische Universität München, February 2001. 15, 25, 25, 75, 103

[78] J. Waldo, *Jini Architecture Overview*. Sun Microsystems, http://java.sun.com/products/jini/whitepapers/architectureoverview.pdf, February 2001. 66

[79] B. Zaun, *A Bluetooth Communication Service for DWARF*. Systementwicklungsprojekt, Technische Universität München, March 2001. 28

Acronyms

API Application Programming Interface

AR Augmented Reality (page 1)

ATM Asynchronous Transfer Mode

CAP Context-Aware Packet (page 26)

COM Component Object Model (page 59)

CORBA Common Object Request Broker Architecture (page 54)

DA Directory Agent (page 62)

DHCP Dynamic Host Configuration Protocol

DWARF Distributed Wearable Augmented Reality Framework (page 4)

GNU GNU is Not Unix

GPS Global Positioning System

HTTP Hypertext Transport Protocol

IDL Interface Description Language (page 55)

IETF Internet Engineering Task Force

IP Internet Protocol

LAN Local Area Network

LDAP Lightweight Directory Access Protocol

MIPS Million Instructions Per Second

ORB Object Request Broker (page 55)

PDA Personal Digital Assistant

POA Portable Object Adapter (page 94)

RFC Request for Comment

RFID Radio Frequency ID (page 24)

RISC Reduced Instruction Set Computer

SA Service Agent (page 62)

SCP Service Control Protocol (page 66)

SSDP Simple Service Discovery Protocol (page 66)

SLP Service Location Protocol (page 61)

SMB Service Message Block

STARS Sticky Technology for Augmented Reality Systems (page 111)

STL Standard Template Library (page 97)

TCP Transmission Control Protocol

UA User Agent (page 62)

UDP User Datagram Protocol

UML Universal Modeling Language

UPnP Universal Plug and Play (page 65)

URL Uniform Resource Locator

VRML Virtual Reality Modeling Language

VR Virtual Reality (page 2)

XML eXtensible Markup Language

Index