Supporting Mobile Computing in Object-Oriented Middleware Architectures

Mads Haahr

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy

October 2003

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Mads Haahr

Dated: May 4, 2004

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Mads Haahr

Dated: May 4, 2004

Abstract

The distributed object paradigm is now widely accepted as a suitable abstraction for building distributed applications. Numerous system architectures based on the paradigm have been proposed, and software frameworks implementing such architectures have been widely adopted for the development and operation of distributed applications. These frameworks constitute middleware: services and protocols that reside 'in the middle,' in a layer above the operating system and networking software and below domain-specific applications. Although object-oriented middleware may incur a considerable runtime overhead, it can significantly simplify the development of applications for distributed, heterogenous environments.

In parallel, advances in technology are now making it possible to embed computational power and wireless communication capabilities in an increasing range of devices, including mobile devices. Such devices range from mobile telephones, personal digital assistants and laptop computers to the computers currently being embedded in cars, stereos, refrigerators and other everyday appliances. The mobile environment formed by such devices communicating with each other and with fixed computational infrastructure is both highly distributed and very heterogenous, and therefore a natural candidate for application support via object-oriented middleware.

Previous research into middleware providing mobility support has taken one of two approaches: the development of new middleware architectures or the adaptation of existing middleware architectures for use in mobile environments. A significant body of work exists in the former category, and the challenges faced by applications operating in mobile environments are therefore generally well understood. Different types of mobile middleware have been proposed and a number of existing mechanisms commonly used with distributed object architectures been adapted for use in mobile environments. Examples are events, tuple spaces and remote procedure call (RPC). The other approach has been to adapt existing distributed object architectures for use in mobile environments. Research in this category has generally focused on supporting a single distributed object architecture rather than finding generic solutions that are applicable across architectures. While such efforts have addressed problems

of heterogeneity and limited resources in terms of network bandwidth and processing power and to some extent also address migration, other problems such as short-term loss of connectivity, long-term disconnection have typically been left either to the transport or application layer. Although this may be suitable for some applications, transport-layer solutions tend to limit application flexibility (e.g., by hiding state related to mobility and connectivity) while application-layer solutions increase application complexity. Consequently, mobility support in current distributed object architectures is partial and architecture-specific.

This thesis also addresses the adaptation of object-oriented middleware for use in mobile environments. It is based on the assumption that current middleware architectures for distributed object computing are basically sound but only solve a portion of the problems caused by the mobile environment, and that the solutions tend to be architecture-specific.

The main contribution of the thesis is the definition of an architecture that allows mobility support to be added to any object-oriented middleware framework that supports a set of minimal requirements. The architecture itself is captured in a set of modular, reusable components that can be used to instantiate the architecture for different object-oriented middleware frameworks. Such modified frameworks have the attractive feature that mobility support remains completely transparent for those portions of distributed applications that do not reside on mobile hardware, while portions that do reside on mobile hardware can be aware or unaware of mobility as required for the application in question. Also, modified frameworks retain interoperability with unmodified frameworks implementing the same architecture.

The thesis describes the principles on which the architecture is based and presents in detail an instantiation of the architecture for one middleware framework. An instantiation of the architecture for one other middleware framework is outlined to show that the architecture is indeed generally applicable and does provide the desired level of transparency. Because the architecture allows support for mobility to be added to existing object-oriented middleware frameworks in a general and transparent manner, the architecture serves as a demonstration of what can be done in terms of mobility support within the confinements of existing models for distributed object computing.

Acknowledgements

Writing a doctorate dissertation is one of the more challenging projects I have undertaken. There were days when I would have sworn the letters PhD were short for panic, horror and desparation, but in retrospect I think perhaps patience, hard work and dedication are more accurate. Pizza, hamburgers and döner kebabs would be descriptive of my diet on many thesis days.

While a doctorate is by nature an individual task, there are many people to whom I am grateful for guidance, expertise, thoughts, advice, time and encouragement. My supervisor, Vinny Cahill, has done a fantastic job in all these respects and asked more difficult questions about the work presented here than I imagined was possible. IONA Technologies deserves a special mention for their generous support for the project, as do our former and current Heads of Department, John Byrne and David Abrahamson, for being understanding and flexible in relation to teaching duties during difficult times.

I am very grateful to Ray Cunningham and Greg Biegel for their excellent work on various parts of the greater body of work presented in this thesis. Their contributions are acknowledged in the respective places throughout the thesis. I am also grateful to Pete Barron, Andronikos Nedos and Stefan Weber for helping stress test my code and for technical assistance with networks and iPAQs, and to Eoin Curran for helping me find many tricky bugs.

I would also like to thank my family in Denmark who have been very supportive despite being physically elsewhere. Many of my friends also deserve a mention, but I am going to pick out Elizabeth Drew, Helena King and Christine Appel as particularly supportive. (Christine invented tandem supervision; our term for mutual encouragement involving food.) Lots of thanks go to my friends and colleagues in the Distributed Systems Group; 'tis a moyda place to be.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

'Kitty, can you play chess? Now, don't smile, my dear, I'm asking it seriously.' [36]

This thesis addresses the general area of middleware for mobility support. It is based on the assumption that the distributed object paradigm has proven merit for the development of distributed applications, but that the major architectures based on the paradigm lack features required by distributed applications operating in mobile environments. We describe a general approach through which mobility support can be added to existing object-oriented middleware frameworks in a general and transparent manner and thereby show what can be done in terms of mobility support within the confinements of existing models for distributed object computing.

This introductory chapter motivates the body of work described in the thesis, presents some background information related to object-oriented middleware architectures that is necessary to understand the remainder of the thesis. The chapter also defines the specific goals of the thesis and finally presents a document roadmap.

1.1 Motivation

The distributed object paradigm is now widely accepted as a suitable abstraction for building distributed applications. Numerous system architectures based on the paradigm have been proposed, popular examples of which are the Common Object Request Broker Architecture (CORBA) [83], Java Remote Method Invocation (RMI) [118] and the Simple Object Access Protocol (SOAP) [184, 185]. Software frameworks implementing such architectures have been widely adopted for the development and operation of distributed applications. These frameworks constitute middleware: services that 'sit

"in the middle," in a layer above the OS and networking software and below industry-specific applications' [22, p.88]. Although object-oriented middleware may incur a considerable runtime overhead [70, 69], it can significantly simplify the development of applications for distributed, heterogenous environments [165, 163].

In parallel, advances in technology are now making it possible to embed computational power and wireless communication capabilities in an increasing range of devices, including mobile devices. Such devices range from mobile telephones, personal digital assistants and laptop computers to the computers currently being embedded in cars, stereos, refrigerators and other everyday appliances. The mobile environment formed by such devices communicating with each other and with fixed computational infrastructure is both highly distributed and very heterogenous, and therefore a natural candidate for application support via object-oriented middleware.

Previous research into middleware providing mobility support has taken one of two approaches: the development of new middleware architectures or the adaptation of existing middleware architectures for use in mobile environments. The two approaches can be seen as the ends of a spectrum where work in the first category starts from the mobile environment and attempts to develop middleware to suit, while the latter starts from existing middleware and attempts to integrate support for mobile environments into the architectures. The latter approach is concerned with preserving existing semantics, whereas the former attempts to provide semantics that are better suited for the mobile environment.

A significant body of work exists in the former category, and the challenges faced by applications operating in mobile environments are therefore generally well understood [63, 159, 40, 57, 14]. For example, Forman et al. divide the problems into three groups related to characteristics of the mobile environment: wireless communication (e.g., connectivity and bandwidth variability), mobility (e.g., address migration) and issues related to portability of the devices (e.g., battery and processing power) [63]. Different types of mobile middleware have been proposed for use in mobile environments, and a number of existing abstractions commonly used with distributed object architectures have been adapted, often through the modification of semantics. Examples are events [174], tuple spaces [126, 49] and remote procedure call (RPC) [164, 48, 98, 17].

The other approach has been to add support for mobile environments to existing distributed object architectures. Research in this category has generally focused on supporting a single distributed object architecture rather than finding generic solutions that are applicable across architectures. Examples are the Java 2 Micro Edition platform [106] and the Minimum CORBA specification [81], both of which define subsets of functionality from their respective architectures in order to improve

support for devices with limited resources in terms of memory, battery and processing power. Address migration is another problem that has been addressed in an architecture-specific manner, for Java RMI via the MobileRMI toolkit [11] (designed for object mobility rather than device mobility, but the problem is similar) and for CORBA via the $\prod^2$ proxy platform [156] and the initial version of the architecture described in this thesis [84]. However, solutions to address migration problems have yet to be integrated into the middleware architectures themselves, perhaps because address migration can also be addressed in a more transparent manner through transport-layer support, such as Mobile IP [138, 140] or IPv6 [144].

Efforts to add mobility support to existing middleware have addressed in an architecture-specific manner the problems related to heterogeneity and limited resources in terms of network bandwidth and processing power, and to some extent also the problems related to address migration. Other problems related to the mobile environment, such as short-term loss of connectivity and long-term disconnection, and to some extent also address migration, have been left either to the transport or application layer. Although this may be suitable for some applications, transport-layer solutions tend to limit application flexibility (e.g., implementations of Mobile IP commonly hide state related to mobility and connectivity from higher layers) while application-layer solutions increase application complexity by requiring extra application code. Hence, current object-oriented middleware solves some (but not all) of the problems related to the mobile environment, and the solutions presented are generally specific to the middleware architecture for which they were developed.

This thesis also addresses the adaptation of object-oriented middleware for use in mobile environments. It is based on the assumption that current middleware architectures for distributed object computing are basically sound but that they only solve a portion of the problems caused by the mobile environment, and that the solutions tend to be architecture-specific. While it could be argued that mobile environments make current object-oriented middleware architectures obsolete, this thesis assumes otherwise. By presenting an architecture that allows mobility support to be added to object-oriented middleware frameworks in a general and transparent manner, this thesis shows what can be done in terms of mobility support within the confinements of existing models for distributed object computing.

1.2 Background

This section defines key concepts used throughout the thesis and provides background information on middleware and distributed object computing required for understanding the remainder of the thesis.

1.2.1 Architecture

The term 'software architecture' is difficult to define. Soni et al. write, '[t]he research area of software architecture is an emerging one with little agreement over the definition of architecture' [173]. A number of definitions of the term have been proposed, in particular by researchers working in software engineering, many of which are concerned with the development of formalisms to express and verify architectural properties. In comparison, researchers from the middleware community generally use the term less strictly and often completely without prior definition [165, 181]. The aim of this section is not to propose a general definition but to clarify what is meant within the scope of the thesis.

Hayes-Roth describes a software architecture as,

> [a]n abstract system specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections. [85]

This definition captures the notion that an architecture is abstract and also lists those of its components' properties that should be included in the architectural description. However, the definition does not mention the processes of instantiation and composition of components. Another source, Jackson, says that a software architecture also includes 'rules which govern how a system (or subsystem) may be composed from instances of the generic components' [93]. This definition captures the notion that architectural components are abstract and that there is a process of instantiation through which the components are made concrete.

For the purposes of this thesis, the term 'architecture' will be used in accordance with these two definitions. The architecture presented here allows certain categories of mobility support to be added to object-oriented middleware frameworks. The architecture itself is described in terms of a set of abstract functional components, including their behaviour, interfaces and interconnections with other components. Using the architecture involves selecting and combining a number of these components, depending on the desired functionality and instantiating them depending on the middleware framework in question. The architectural description includes the rules for composition and instantiation.

Although this use of the term 'architecture' is correct from a middleware perspective, a researcher in software engineering might feel the term is being used incorrectly. Shaw, a researcher in software engineering, divides software architecture into four different views: structural models, framework models, dynamic models and process models. About the former two, Shaw says,

> Structural models all hold that software architecture is composed of components, connections among those components, plus (usually) some other aspect or aspects ... Framework models are similar to the structural view, but their primary emphasis is on the (usually

singular) coherent structure of the whole system, as opposed to concentrating on its composition. Framework models often target specific domains or problem classes. Work that exemplifies the framework view includes domain-specific software architectures, CORBA or CORBA-based architecture models, and domain-specific component repositories (e.g., PRISM).[1]

This quotation makes the observation that the term 'software architecture' can be used to emphasise the coherent structure of a system rather than its composition. In this respect, Shaw's 'framework model' corresponds to what researchers in middleware typically call 'architecture.'

1.2.2 Framework

Another term that mandates definition is 'framework.' Bernstein says,

> A *framework* is a software environment that is designed to simplify application development and system management for a specialized application domain ... A framework is defined by an API, a user interface, and a set of tools. [22, p.92, original emphasis]

This definition defines the purpose of frameworks and also captures the notion of specificity to an application domain. In addition, it specifically defines the constituents of a framework, although Bernstein later in the same article, citing the Open Software Foundation's Distributed Computing Environment (OSF's DCE) as an example, points out that not all frameworks have user interfaces.

Another definition is given by Gamma et al.,

> A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software ... You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework. [66, p.26, original emphasis]

This definition is specific to object-orientation and in this respect less general than Bernstein's. However, it does make the important observation that the process of using a framework for development of a given application involves customising portions of the framework to fit that application's requirements. The definition does not mention tools explicitly, but it could be argued that any such functionality could be embodied as non-abstract framework classes.

[1] This unpublished quotation is attributed to Mary Shaw at the First International Workshop on Architectures for Software Systems. Source: Software Engineering Institute, CMU.

The two definitions agree that frameworks are specific to application domains and that interfaces are a significant portion of a frameworks. Object-oriented middleware, such as implementations of CORBA and Java RMI, would qualify as frameworks under both of the listed definitions.

While the architecture described in this thesis is abstract, it is captured in a set of software components that can be used to instantiate the architecture for different distributed object frameworks. These software components can themselves be called a framework according to the two definitions given above. In this case, the application domain is support for mobile computers in object-oriented middleware frameworks, and the framework's role is to simplify the process of application development (i.e., adding support for mobile computers to an object-oriented middleware framework). This involves using the supplied software components, some in customised form and others unchanged.

1.2.3 Middleware

'Middleware' is another term that is difficult to define. Bernstein observes that 'middleware is hard to define in a technically precise way' [22, p.90], and Geihs avoids giving a specific definition on the grounds that '[d]epending on the application environment, opinions differ as to which components comprise middleware' [68, p.24]. This section draws upon the literature to define middleware in terms of purpose, form and function.

Most sources agree that the purpose of middleware is to address the problems of *heterogeneity* and *distribution* [68, 102, 22, 25]. Blair et al. put it very succintly that '[t]he role of middleware is to present a unified programming model to application writers and to mask out problems of heterogenetity and distribution' [25, p.192]. This view may be in the process of becoming obsolete, as discussed below.

With regards to form, the general consensus in the literature [68, 102, 22, 25, 163] is consistent with the Bernstein quotation given in section 1.1 which defines 'middleware services' as programming interfaces and protocols that 'sit "in the middle," in a layer above the operating system and networking software and below industry-specific applications' [22, p.88]. Bernstein's definition not only defines the *form* of middleware services as programming interfaces and protocols but also places the services at a specific point in the protocol stack. Whereas there is general agreement as to the placement of middleware, most other sources are less specific about its form. For example, Schantz et al. simply write that '[m]iddleware is *systems software* that resides between the applications and the underlying operating systems, network protocol stacks, and hardware' [163, my emphasis].

Regarding functionality, Bernstein says, '[i]n general, middleware is replacing the nondistributed functions of OSs with distributed functions that use the network' [22, p.88] and lists distributed databases, remote file access and RPC as examples of middleware services. Another source, King,

agrees with Bernstein that 'middleware is to the network what an operating system is to the local computer resources' [102, p.59] and also lists message delivery systems as a type of middleware. Later sources categorise as middleware also group communication services [18, 68], workflow systems [18, 68], event services [174], tuple space services [126, 49] and distributed object architectures [163, 13].

Hence, the spectrum of middleware services is wide. Common to the listed services is that all offer developers a set of 'higher-level interfaces, which mask the complexity of networks and protocols' [22, p.88]. Furthermore, there seems to be a continuum or a trend towards services of greater complexity and higher abstraction levels over time. Bernstein's observations about middleware replacing operating system functions with comparable distributed functions hold for early types of middleware such as RPC and remote file access. However, later types of middleware, such as workflow systems and tuple space services, are based on higher levels of abstraction for which no comparable non-distributed functions exist in most operating systems. Such middleware services do more than mask out the problems of heterogeneity and distribution; they provide new functionality based on higher levels of abstraction. This trend has also been observed in the literature, for example by Schantz et al. who suggest that middleware for distributed object computing can be decomposed into four layers: host infrastructure (e.g., virtual machines), distribution (e.g., remote invocation), common services (e.g., events and transactions) and domain-specific services (e.g., e-commerce type services) [163]. The same trend can be observed in the development of the CORBA standards where initial efforts focused on basic services such as remote method invocation [181, 74] but later shifted towards higher-level services, for example the domain-specific standards used in finance [75] and healthcare [77].

1.2.4 Distributed Object Computing

A special class of middleware is based on the distributed object model. Also called distributed object computing (DOC) middleware, such frameworks offer distributed services based on the object-oriented paradigm, such as remote object instantiation, method invocation, etc. Schantz et al. give the following description,

> DOC is an advanced, mature, and field-tested middleware paradigm that supports flexible and adaptive behavior. DOC middleware architectures are composed of relatively autonomous software objects that can be distributed or collocated throughout a wide range of networks and interconnects. Clients invoke operations on target objects to perform interactions and invoke functionality needed to achieve application goals. [163]

Although there in this quotation is some confusion about what constitutes an 'architecture' (the description seems to match better what in this thesis is called a 'framework' or perhaps even an 'application'), the authors do describe the basic structure of an application based on the distributed object paradigm and identify support for flexibility and adaptivity as key features of this type of middleware. As discussed in section 1.2.3, distribution is a defining characteristic of middleware, and apart from being somewhat awkward, the term 'distributed object computing middleware' is therefore also redundant. Instead, this thesis uses the term 'object-oriented middleware' to denote frameworks implementing system architectures based on the distributed object model. It should also be noted that different object-oriented middleware architectures use different object models, and that the term 'the distributed object model' in this thesis is used in the abstract sense to refer to the idea of dividing and distributing application functionality via object-orientation. When individual object models are discussed, they will be referred to specifically, e.g., 'the Java object model' [190].

Many sources [163, 13, 165, 55, 26, 148, 135] agree that the distributed object model has proven a suitable abstraction for modelling distributed applications and that object-oriented middleware can significantly simplify the development of applications for distributed, heterogenous environments. About object-orientation, Bacon writes that '[o]bject concepts have stood the test of time in programming languages, operating systems and middleware platforms' [13, p.646]. On the merits of object-oriented middleware, Schmidt et al. write that it 'simplifies application development by providing a uniform view of heterogenous network and OS layers' [165]. Schantz et al. concur with this view and, speaking about one of the four layers of middleware mentioned in section 1.2.3, say,

> distribution middleware, such as CORBA, Java RMI, or SOAP, makes it easy and straightforward to connect separate pieces of software together, largely independent of their location, connectivity mechanism, and technology used to develop them [163]

Experiential records can be found in the literature that confirm object-oriented middleware to be suitable for the design and implementation of a variety of applications, e.g., multidatabase systems [55], multimedia platforms [26], support frameworks for mobile applications [9], soft real-time control systems [148] and systems for remote building monitoring and operation [135].

On the negative side, studies have shown object-oriented middleware to incur a significant cost in terms of runtime overhead for many applications. Gokhale et al. have measured data transmission rates in implementations of CORBA and found that 'conventional implementations of CORBA incur considerable overhead when used for performance-sensitive applications over high-speed networks' [70]. Another study by Gluzberg et al. concludes that 'Java ... RMI performance lags far below native ... RPC performance' [69]. Despite the runtime overhead observed by studies such as these, software

frameworks based on distributed object models are being widely adopted for the development and deployment of distributed applications. For example, Bacon observes that 'the CORBA approach has gained wide industrial and commercial acceptance' [13, p.626] although this acceptance has been 'steady, if relatively slow' [13, p.646].

1.3 Influential Systems

As is usually the case with research, the work presented in this thesis was inspired by previous work in the field of mobile computing and object-oriented systems. While a thorough state of the art review can be found in chapter 3, this section briefly discusses projects and systems that were particularly influential.

1.3.1 OnTheMove

The work described in this thesis was funded by IONA Technologies through their membership in the consortium of a European research project called OnTheMove. OnTheMove operated from August 1995 to August 1998 through funding from the Advanced Communications Technologies & Services (ACTS) Programme, a part of the European Fourth Framework Programme (FP4). The aim of the project was to develop an architecture for mobile multimedia and to propose a standardised mobile API to facilitate the use of multimedia applications in third generation Universal Mobile Telecommuncations System (UMTS) networks.

IONA's involvement in the OnTheMove project focused on bringing support for CORBA applications into mobile environments. This led to the design and implementation of the initial version of the architecture described in this thesis. While it did serve as a valuable prototype, this version of the architecture also had shortcomings, most notably its specificity to CORBA. The second version of the architecture (described in this thesis) addresses these shortcomings. The first version of the architecture is documented in [84].

1.3.2 Emerald

Emerald [99, 132] is an object-based language for distributed programming. Emerald has explicit support for fine-grained mobility of objects, meaning that objects can be small data objects as well as process objects. Implementations of Emerald need mechanisms to locate objects that have moved and therefore face problems similar to those encountered by systems supporting server objects residing on physically mobile hardware. Some of the ideas presented in this thesis—most notably the redirection

mechanism—were inspired by the author's previous work on an Emerald location service based on Fowler's work on decentralised object finding using forwarding addresses [64].

1.4 The Thesis

Section 1.1 argued that current object-oriented middleware provides only partial support for distributed applications operating in mobile environments, and that the adopted solutions tend to be architecture-specific. The work described in this thesis addresses these problems by contributing to the state of the art an architecture that allows mobility support to be added to any object-oriented middleware framework that supports a set of minimal requirements.

The architecture itself is captured in a set of modular, reusable components that can be used to instantiate the architecture for different object-oriented middleware frameworks. Such modified frameworks have the attractive feature that mobility support remains completely transparent for those portions of distributed applications that do not reside on mobile hardware, while portions that do reside on mobile hardware can be aware or unaware of mobility as required for the application in question. Also, modified frameworks retain interoperability with unmodified frameworks implementing the same architecture.

The thesis describes the principles on which the architecture is based and presents in detail an instantiation of the architecture for one middleware framework. An instantiation of the architecture for one other middleware framework is outlined to show that the architecture is indeed generally applicable and does provide the desired level of transparency. Because the architecture allows support for mobility to be added to existing object-oriented middleware frameworks in a general and transparent manner, the architecture serves as a demonstration of what can be done in terms of mobility support within the confinements of existing models for distributed object computing.

1.5 Objectives

As mentioned above, this thesis is concerned with the adaptation of existing object-oriented middleware frameworks for use in mobile environments. The work presented here is based on the idea that the distributed object model which underlies current object-oriented middleware architectures is basically sound, and that the middleware architectures themselves have proven merit. A convincing case can be made for reusing existing middleware rather than inventing new abstractions or models. First, current object-oriented middleware architectures are the product of several decades of research and constitute

a significant investment in terms of time, money and expertise. Many well-tested implementations of the architectures exist, and a significant body of experience in their use has been accumulated. Furthermore, object-oriented middleware architectures were designed explicitly to address problems of distribution and heterogeneity. As will be discussed in chapter 2, mobile environments have exactly these characteristics (perhaps to an even higher degree than many wired networks), and object-oriented middleware architectures are therefore natural candidates for supporting distributed applications in mobile environments.

However, it should be noted that this approach is certainly not the only one possible, and it could even be argued that adapting frameworks based on existing architectures in the way suggested here is a somewhat conservative basis for contributing to the state of the art. A counter-view could be that the problems faced by distributed applications in mobile environments render the distributed object model obsolete, and it is better to rethink the model than to attempt to retain compatibility with existing architectures. This view is as valid as the one presented in this thesis, and for those who believe more radical changes to current models are required, this thesis will hopefully either weaken or strengthen that conviction by showing what can be done within the confinements of existing models.

Our objectives here are therefore not to invent novel types of mobility support or propose new middleware paradigms, but to show how mobility support can be integrated into existing object-oriented middleware frameworks in a manner that is both *general* and *transparent*.

1.5.1 Transparency

While mobile computers are becoming increasingly important as a platform for distributed applications, they are unlikely to replace traditional fixed hosts. Rather, it seems reasonable that mobile hardware will be used to complement fixed hardware, and that future distributed applications will consist of application logic distributed across a combination of mobile and fixed nodes. Compared to a traditional wired network environment, the mobile environment is difficult [63, 159] and causes a number of problems for systems and applications operating there. While some of these problems can be dealt with automatically (by system support), others will need to be addressed explicitly by the individual applications. A comparable example from mobile file systems is certain categories of conflict detection/resolution in Coda [161]. Coda will resolve most concurrent updates to a file system automatically, but some conflicts cannot be resolved without user input. For this reason, portions of applications will need to be *mobility-aware*.

For a distributed application operating across fixed as well as mobile nodes, it is desirable to restrict the requirement for mobility-awareness as much as possible, in particular to the portions of

the application that are actually mobile. By retaining transparency for the remaining (non-mobile) parts of the application, software complexity can be decreased, resulting in reduced development time and increased reliability. Also, for applications where (non-mobile) components already exist, a transparent approach will result in a greater degree of reusability. We call this type of transparency *application-level transparency.*

Another aspect of adding mobility support to existing object-oriented middleware frameworks is to retain interoperability with existing (unmodified) frameworks implementing the same architecture. For an unmodified framework interacting with one to which mobility support has been added, it is desirable that the modifications remain transparent to the unmodified framework. We call this type of transparency *framework-level transparency.* The architecture presented in this thesis supports a high level of application- as well as framework-level transparency.

1.5.2 Generality

Section 1.2.4 argued that object-oriented middleware has proven useful not only as an abstraction and a design tool but also as a platform for the actual implementation of distributed applications. This thesis is based on the assumption that the distributed object paradigm is not invalidated by mobility. At the time of writing, there are four dominant middleware architectures based on the distributed object paradigm: CORBA, SOAP, Java RMI and the Distributed Component Object Model (DCOM).[2] Despite differences in scope and complexity, these architectures share very similar notions of key concepts, roles and modes of interaction. In all the models, a distributed application consists of *objects* equipped with *methods.* Objects take on the roles of *clients* and *servers* (possibly both at the same time) and interact by the former obtaining *references* to the latter and *invoking* the latter's *methods.*

Despite the common characteristics of the different distributed object models, most efforts towards addressing mobility problems within existing object-oriented middleware have resulted in architecture-specific solutions. However, the problems related to the mobile environment are the same for all object-oriented middleware architectures. Combined with the similarity of the architectures, this makes it possible to develop solutions that can be applied more generally. General solutions are more attractive than those specific to individual architectures, because they make it easier to reuse existing ideas, expertise and software components.

The work described in this thesis has focused on developing solutions of sufficient generality to

[2] DCOM is in the process of being replaced by Microsoft's .NET framework, a SOAP-based object-oriented middleware architecture.

be applicable across several object-oriented middleware frameworks. The architecture presented here can be instantiated for a number of such frameworks. To demonstrate generality, an instantiation of the architecture for one object-oriented middleware framework is described in detail, while one more is presented as an outline.

1.6 Thesis Roadmap

The remainder of this thesis follows the following structure:

Chapter 2 defines the mobile environment and identifies the key problems it causes for the operation of distributed object applications. This chapter also defines the mobility model adopted for the purposes of the thesis.

Chapter 3 presents the state of the art in mobility support for distributed objects.

Chapter 4 presents the architecture including a set of modular, reusable components that can be used to instantiate it for specific distributed object frameworks.

Chapter 5 presents in detail an instantiation of the architecture for CORBA and outlines an instantiation for Java RMI.

Chapter 6 evaluates the work described in the thesis in the context of the objectives given in section 1.5.

Chapter 7 summarises and discusses future work.

Chapter 2

The Mobile Environment

> *Of course the first thing to do was to make a grand survey of the country she was going to travel through. 'It's something very like learning geography,' thought Alice, as she stood on tiptoe in hopes of being able to see a little further.* [36]

Mobile computing is a special case of distributed computing characterised primarily by the fact that many of the components are moving and that their degree of interconnectivity varies dramatically over time. Most of the hard problems in distributed computing (e.g., dealing with failed or unreachable nodes) are found in extreme forms in mobile computing. A significant body of work exists in the field of mobile computing, and the challenges of the mobile environment are therefore generally well understood [63, 159, 40, 57, 14].

This chapter characterises the mobile environment and presents the key problems that this environment causes for the operation of distributed object applications. The intention is to provide a frame of reference for the discussion of related work in chapter 3 and the architecture presented in chapter 4. First, a brief discussion of past and future mobile environments is presented. Second, the two main models of mobile computing—the ad hoc and the gateway models—are discussed. The problems related to mobile environments are presented under three headings: problems related to mobile devices, mobile networking (wired and wireless) and physical mobility. For each problem, we discuss how it affects the operation of distributed object applications and whether it is specific to distributed applications or applies also to standalone mobile applications. Based on this analysis, a number of themes are identified that emerge as generally useful for addressing the operation of such applications in mobile environments. Finally, the chapter is summarised.

2.1 Past and Future

The first mobile computer, the Osborne I, appeared in 1981. It weighed 10.9 kilos, cost $1,795 and featured a 5-inch display, 64 kilobytes (KB) of memory, a modem and two 5 1/4-inch floppy disk drives [127]. Such 'luggable' and later laptop computers were essentially compact versions of desktop computers whose screens, keyboards and disk drives had been integrated into the casing. These computers came with few options for interconnection—typically limited to a modem or serial port used for dial-up connectivity. Later, Local Area Network (LAN) connectivity options became available. At about the same time, the first handheld computers appeared. They were small, light devices with a minimum of processing power but sufficient memory to store addresses, telephone numbers, short memos and to run simple applications. For example, the Psion Organiser I, which appeared in 1984, weighed 225 grams, cost $199.95 (£99 in the UK) and featured a 16 character monochrome LCD display, a HD6301X processor clocked at 0.92 MHz, 10 KB of memory (2 KB on board and 8 KB as a plugin module) and an RS232 serial port [1, 113]. Like their luggable counterparts, devices such as these had few options for communication with their environment, typically limited to a serial line to connect to a desktop computer.

As such, the current state of the art in mobile computing is not radically different from what it was in the 1980s. Luggable computers have been replaced by laptops, and personal organisers are now called Personal Digital Assistants (PDAs). The capabilities of the devices have of course improved along with their technical specifications, and the applications have also grown in complexity. Despite these developments, the applications and the essential *form* of the devices—the portable and the handheld—remain the same. Perhaps the most significant advances have been in connectivity: current mobile computers (of both the laptop and handheld variety) boast a much wider range of options for connecting to other devices than did previous generations of these computers and even most current stationary computers. A number of these new modes of connectivity are based on wireless technologies, such as infrared (IR) and radio frequency (RF) communication. This development is explored in section 2.4.

In a 1991 article entitled 'The Computer for the Twenty-First Century' [187], Mark Weiser formulated his famous vision of 'ubiquitous computing' and suggested that computational power be embedded in our daily environment instead of in devices easily identifiable as 'computers.' Weiser's ideas struck a chord with the research community and have received much attention, not least among the distributed systems and mobile computing communities. The vision has been adopted in a variety of guises: 'ubiquitous computing' has also been called 'ambient' [121, 50], 'invisible' [115, 27] and 'pervasive' [130, 160] computing, the computationally augmented environments called 'smart' [115, 100]

or 'intelligent' [179] spaces and the technology 'calm technology' [188]. The ideas have also been the topic of major funding initiatives, such as the European Disappearing Computer initiative, and of academic publications such as *IEEE Pervasive Computing* magazine, the inaugural issue of which appeared in early 2002.

Weiser's idea of ubiquitous computing has been seen by many as the next paradigmatic shift in the ways computers will be used. Hence, if current research agendas can be trusted, the future computer is a vastly complex web of computational units of varying size and capability, interconnected via wireless networks. A substantial portion of these units can also be expected to be mobile, either actively (by moving on their own accord) or passively (by being carried). Together, the devices and their applications will form a layered distributed system of vastly bigger and more complex nature than our biggest and most popular system so far: the Internet with its favourite application the Web. This 21st-century 'computer' has many of the difficult characteristics of distributed systems: extreme scalability requirements, a high level of node mobility, dramatically varying network connectivity and an uncompromising need for decentralisation. In many ways, this environment is the ultimate distributed systems and mobile computing challenge.

The discussion of the mobile environment presented in this chapter employs current technology (e.g., laptop and handheld computers, current wireless communications) for purposes of illustration. However, an attempt is made to raise the issues in a general manner, such that the discussion will apply also to future environments (such as that described by Weiser) where mobility is expected to play a significant role.

2.2 Modeling the Mobile Environment

For the purposes of understanding a mobile environment, it is useful to model it at a high level of abstraction. Two such models are currently in use: the *ad hoc* and the *gateway* model. Each model defines the constituents of the mobile environment and the way in which mobile nodes interact with their surroundings. The basic difference between the models is related to the level of infrastructure assumed to be present in the physical environment. Whereas the gateway model is characterised by mobile nodes relying on a certain level of fixed infrastructure, the ad hoc model assumes no infrastructural support beyond that provided by the nodes themselves.

It should be noted that the two models are not mutually exclusive. A given environment (such as the ubiquitous computing environment proposed by Weiser [187]) could easily consist of micro-environments of both types, and some research has dealt explicitly with interfacing environments of

the two types [176]. Also, the models only describe those characteristics of mobile environments that are related to node mobility and to exchange/routing of data between nodes and their surroundings. Other characteristics, such as those related to security and to the general properties of mobile devices and networks, are not within scope of the models. Hence, two mobile environments belonging to the different categories will typically share a number of characteristics beyond those described by their respective models. For example, most mobile environments use wireless communication and therefore suffer from problems related to varying security, variation in connectivity, long-term disconnection, etc. Problems such as these are not described by the models.

2.2.1 The Ad Hoc Model

In the ad hoc model, the mobile devices are arranged dynamically to form networks in an improvised manner. Perkins et al. define an ad hoc network as 'the cooperative engagement of a collection of mobile nodes without the required intervention of any centralized access point or existing infrastructure' [141]. In such a configuration, nodes communicate directly and avoid relying on infrastructure present in the environment by acting as routers of data as well as sources and destinations. Because nodes may move, the network's topology can change rapidly. For this reason, the solutions typically adopted to addressing and routing problems in such networks are quite different from those that work well in fixed infrastructure networks.

In general, a network formed by a collection of mobile nodes in this manner is called a Mobile Ad hoc Network (MANET). It should be noted that while we use the dependence on fixed infrastructure to distinguish between the ad hoc and the gateway model, actual technologies sometimes support both modes. Also, according to some definitions, MANETs may optionally use fixed infrastructure. For example, Corson et al. write that a MANET 'may operate in isolation, or may have gateways to and interface with a fixed network' [45, p.3]. The same source identifies the four salient characteristics of MANETs as: having 'dynamic topologies,' subject to rapid reconfiguration as nodes move and wireless transmission conditions change; using 'bandwidth-constrained, variable capacity links,' that may be uni- or bidirectional and whose characteristics may vary unpredictably; requiring 'energy-constrained operation' because mobile devices typically rely on mobile power sources with limited lifetime; and having 'limited physical security' because of the inherently public nature of wireless media [45, pp.3–4]. It is worth noting that the last three characteristics identified by Corson et al. are descriptive of, but not specific to, MANETs. In particular, they cannot be used to distinguish between ad hoc and gateway type mobile environments.

Figure 2.1 shows a number of mobile hosts communicating in an ad hoc scenario. The hosts at

the top and the bottom are outside transmission range of each other but connected via a series of intermediate hosts that relay packets and thereby effectively act as routers. The two hosts at the top left are also outside each other's range but can communicate via a third host which is within range of them both.

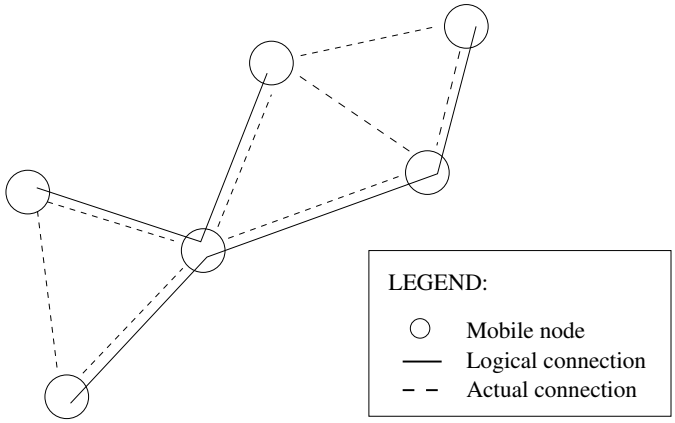


Figure 2.1: Mobile Hosts in an Ad Hoc Scenario

A significant body of research has been conducted in the area of ad hoc networks, in particular related to routing. Examples of ad hoc routing protocols include Dynamic Source Routing (DSR) [95], Temporally Ordered Routing Algorithm (TORA) [137] and Ad-hoc On Demand Distance Vector (AODV) routing [141]. It has been observed that the requirements on routing protocols for ad hoc networks are quite different from those on fixed networks and that ad hoc routing protocols 'do not scale to the Internet' [176].

Examples of technologies based on the ad hoc model are IEEE 802.11 [60] WLAN networks (when used in ad hoc mode) and Bluetooth [4].

2.2.2 The Gateway Model

In the second model of mobile environments, wireless networks are positioned in a peripheral role 'at the fringes of a universal infrastructure of high speed WANs and LANs, providing the "last link" to a mobile user' [57]. In this model, mobile hosts obtain connectivity via gateways placed at the border between the wired and wireless networks. These gateways, also called base stations, access points, mobile support stations or mobile support routers, are equipped with interfaces that allow them to communicate with mobile as well as fixed hosts. To participate in the network, a mobile host must be within range of a gateway that relays communication between the mobile host and other parties. Effectively, gateways lend functionality to mobile hosts by acting as proxies for a certain duration. In

most technologies based on the gateway model, mobile hosts can move between gateways, and in many systems a procedure called 'handoff' or 'handover' allows state pertaining to the mobile host and any open connections to be transferred from one gateway to another. A mobile host may be within range of several gateways at the same time or out of range of all gateways for an extended period of time. In the case of several gateways being within range, a single gateway is typically chosen to serve all the needs of the mobile host.

The gateway model forms the basis for a significant number of technologies and has also been described in the literature [14, 40]. Badrinath et al. describe the gateway model as a two-tier structure where the first tier consists of 'a fixed network with hosts ... having more resources in terms of storage, computing and communication' and the second tier of 'mobile hosts ... with limited capabilities connected by a weak connection to the fixed network' [14]. The definition adopted for the purposes of this thesis corresponds to that given by Badrinath et al., with the exception that our definition does not assume all fixed hosts to be gateways. It is worth noting that three of the four salient characteristics of mobile ad hoc networks identified by Corson et al. and discussed in section 2.2.1 also apply to most types of mobile environments based on the gateway model: bandwidth-constrained, variable capacity links (between gateways and mobile hosts); energy-constrained operation; and limited physical security due to the nature of wireless communications [45, p.3].

Figure 2.2 shows a number of hosts communicating in a gateway scenario. The mobile host on the left is communicating with a remote host on the fixed network through one gateway, while two other mobile hosts are communicating with each other through a second gateway. Even though the last two mobile hosts would be within each other's transmission range, they communicate via the gateway rather than directly. In a configuration based on the ad hoc model, these mobile hosts could have communicated directly. Handoff between gateways is not shown in the figure.

Examples of technologies based on the gateway model are IEEE 802.11 WLAN (when used in infrastructure mode) and GSM networks. Chlamtac et al. also mention dedicated packet data networks, such as Mobitex and Ardis, and hybrid systems, such as Cellular Digital Packet Data (CDPD) [40].

2.3 Mobile Devices

An important part of the mobile environment are the mobile devices themselves. Section 1.1 described the current spectrum of mobile devices as ranging from mobile telephones, PDAs and laptop computers to the computers currently being embedded in cars, stereos, refrigerators and other everyday appliances. This trend towards embedding of computational power in an increasing number of de-

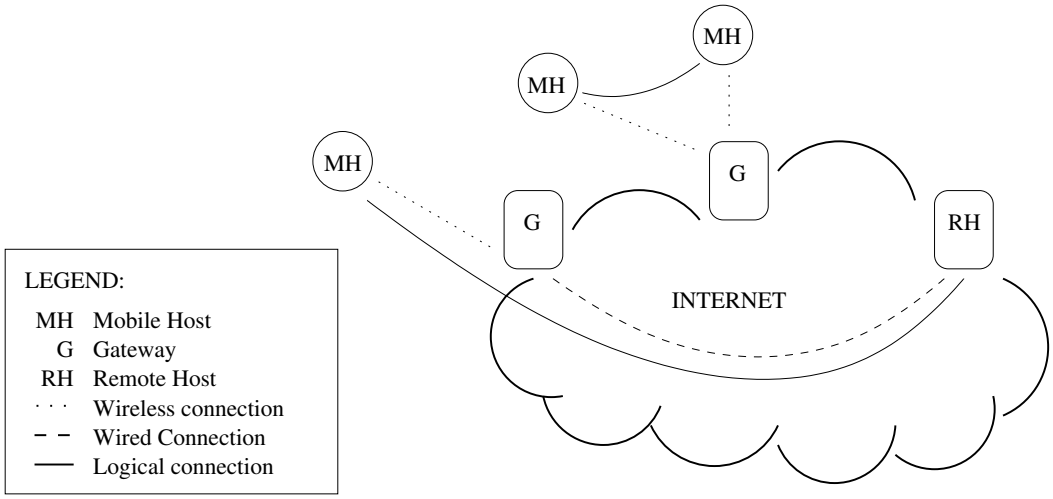


Figure 2.2: Mobile Hosts in a Gateway Scenario

vices can be seen as the first step towards the realisation of Weiser's vision of ubiquitous computing. This section discusses the capabilities of current mobile computers and their development over the last decade and identifies those problems of the mobile environment that are related to the devices themselves.

Mobile computers generally have significantly lower specifications than their stationary counterparts. In a 1994 paper, Forman et al. [63] discussed the challenges of the mobile environment and presented the technical specifications of a number of portable computers that were popular at the time. Table 2.1, taken from [63, p.44], summarises specifications for six PDAs, a tablet computer and a portable PC. Although Forman et al. do not discuss it in detail, it should be noted that some of the specifications in the table are difficult to compare. In particular, processor clock speeds cannot be compared directly across processor types because of differences in processor architecture. Also, it seems somewhat doubtful that the battery lifetimes given for the Tandy and Casio devices are for constant use, which presumably is the case for the Apple Newton Message Pad and the portable PC.

Table 2.2 lists specifications for a number of current mobile computers in the same format as that used by Forman et al. This table features low- and high-end PDAs as well as a wearable and a portable (laptop) PC. In addition, the table includes specifications for two sensor nodes, the Pushpin Computer [111] developed by Massachussetts Institute of Technology (MIT) and the Tiny Node [87] developed by University of California Berkeley (UCB). Because the two tables follow the same format, the pitfalls noted about comparing the specifications in table 2.1 also apply for table 2.2. The presence of the two sensor nodes means that the range of devices given in table 2.2 is slightly wider than that

Product	RAM	MHz	CPU	Batteries		Weight	Display	
				(No. hours)	(type)	(lbs.)	(pixels)	(sq.in.)
Amstrad Pen Pad PDA600	128 Kbytes	20	Z-80	40	3 AA's	0.9	240×320	10.4
Apple Newton MessagePad	640 Kbytes	20	ARM	6–8	4 AAA's	0.9	240×336	11.2
Apple Newton MessagePad 110	1 Mbyte	20	ARM	50	4 AA's	1.25	240×320	11.8
Casio Z-7000 PDA	1 Mbyte	7.4	8086	100	3 AA's	1.0	320×256	12.4
Sharp Expert Pad	640 Kbytes	20	ARM	20	4 AAA's	0.9	240×336	11.2
Tandy Z-550 Zoomer PDA	1 Mbyte	8	8086	100	3 AA's	1.0	320×256	12.4
AT&T EO 440 Personal Communicator	4–12 Mbytes	20	Hobbit	1-6	NiCad	2.2	640×480	25.7
Portable PC	4–16 Mbytes	33–66	486	1-6	NiCad	5–10	640×480 to 1024×768 (color)	40

Table 2.1: Technical Specs for 1994 Mobile Computers [63, p.44]

of table 2.1, and this requires an explanation. Sensor nodes are typically used to gather information about an environment, and a network of nodes can coordinate their efforts to collectively perform a larger sensing task [59]. Sensor networks have been seen as an important step towards the realisation of Weiser's vision of ubiquitous computing [116]. As such, the two sensor nodes listed in table 2.2 are not sufficiently powerful to host distributed object applications, and for this reason, this chapter focuses on the other types of mobile computers. The two sensor nodes are listed to show that the constraints related to current PDAs and portable PCs despite projected advances in hardware technology are likely to persist in the future, albeit for another category of devices.

Product	RAM	MHz	CPU	Batteries		Weight	Display	
				(No. hours)	(type)	(lbs.)	(pixels)	(sq.in.)
MIT Pushpin	32+2.25 Kbytes	22	Cygnal C8051F016	hours/ years	various	?	N/A	N/A
UCB Tiny Node	8+0.5 Kbytes	4	ATMEL 90LS8535	30–200 hours	Lithium	?	N/A	N/A
Handspring Visor Pro	16 Mbytes	33	Dragonball MC68VZ328	4 weeks	Lithium Ion	0.4	160×160 (b/w)	4.9
Compaq iPAQ H3870	64 Mbytes	206	StrongARM SA-1110	14	Lithium Ion	0.4	240×320 (colour)	6.8
CharmIT Transmeta 800	256 Mbytes	800	Transmeta Crusoe TM5800	12	Lithium Ion	3	640×480 to 1280×1024 (colour)	Head Mounted Display
Laptop PC	128–1024 Mbytes	1–2 GHz	Intel Celeron to P4	2–6	Lithium Ion	5–8	1024×768 to 1600×1200 (colour)	70–110

Table 2.2: Technical Specs for 2002 Mobile Computers

With reference to table 2.1, Forman et al. identifies the 'portability constraints' of the mobile computers to cause of a number of 'design pressures' [63, p. 43]. These pressures translate roughly to a number of 'constraints of mobility' defined by Satyanarayanan [159]. Synthesising the two sources yields the following list of respects in which mobile computers are constrained compared to their

stationary counterparts: battery power, data risks, user interface, storage capacity and processing power. The following discussion of the constraints draws upon the literature and the two tables and assesses each individual constraint's potential impediment to the operation of distributed applications.

As a preliminary note, the two tables show the weight of a PDA to have been roughly halved from 1994 to 2002, while the weight of a portable PC has remained roughly unchanged. This can be seen as a maturation of the PDA as an appliance—its form factor is progressing towards the ideal for its purpose. In comparison, the portable PC in 1994 seems to have had about the right size; its weight was largely unchanged from 1994 to 2002, and the most significant development during this period was the improved technical specifications as discussed below.

2.3.1 Battery Power

Forman et al. observe that batteries are the largest single source of weight in mobile computers, and there is an important trade-off between portability (related to weight) and usability (related to battery life) [63]. Badrinath et al. observe that 'power consumption is a serious practical consideration at a mobile host, unlike a fixed host' [14]. Satyanarayanan predicts that despite projected advances in battery technology, power consumption will remain an issue, the concern of which 'must span many levels of hardware and software to be effective' [159].

Forman et al. discusses power consumption of portable computer components and, examining a Sharp PC 6785 PDA, finds that the most demanding subsystems (in order) to be wireless communication (5.40W for an active mobile phone, 0.3W for a passive), base board (2.80–3.65W depending on processor clock speed), hard drive (0.7–3.0W) and screen backlight (1.425W). For this device, wireless communication was by far the most power-consuming feature, but it is worth noting that nearly twenty times more power was consumed when the interface was active than when it was passive.

Most of the devices listed in table 2.1 favoured disposable over rechargable batteries, presumably because of the superior lifetime of the former. This was the case despite the increased financial and environmental costs related to the use of disposable batteries. Comparing the two tables shows that disposable and NiCad batteries in the listed 2002 devices had been replaced with Lithium Ion batteries which are rechargable and have a higher capacity than NiCad batteries. The tables also show that the number of hours of operation did not change significantly from 1994 to 2002, neither for PDAs or portable PCs.

The battery power available to mobile hosts is a considerable restriction that applies to distributed as well as standalone applications. Wireless communication in general requires significant power, and any reduction in wireless communication will therefore result in increased operating time of the mobile

computer. For distributed object applications, some reduction in communication can be achieved by efficient on-the-wire data formats and through compression of the transmitted data. Another approach to reducing communication is through voluntary disconnection [14]. If distributed object applications can operate in voluntary disconnected mode, power consumption for wireless communication may be reduced and the operating time of the mobile computers increased.

2.3.2 Data Risks

Forman et al. and Satyanarayanan agree that data stored on mobile computers is vulnerable to breaches of privacy or total loss [63, 159]. Forman et al. observe that these issues can be addressed by minimising the essential data kept on the device, by the use of data encryption and through backups stored on secure, remote media [63, pp.44–45].

None of the PDAs listed in table 2.1 had wireless communications interfaces which made remote storage of data relatively cumbersome. In comparison, the devices listed in table 2.2 have several wireless modes of connectivity which can be used to reduce data risks. While not shown in the table, connectivity constitutes perhaps the most interesting respect in which mobile computers have developed from 1994 to 2002. Section 2.4 discusses this in detail and presents new tables that include characteristics of networking technologies commonly used with the mobile computers listed in table 2.2.

If a mobile device is stolen or damaged, application data may be lost, regardless of whether the application is standalone or distributed. Hence, with regards to data loss, distributed object applications are exactly as vulnerable as other applications running on mobile hardware. However, for threats of data loss, distributed object applications may also help solve the problem. Replication of data across physically dispersed machines is a key technique used with distributed systems and 'can provide enhanced performance, high availability and fault tolerance' [46, p.312]. In addition, replication can also be used to reduce threats of data loss, because remote replicas can act as backup copies of the data in question. Given support for object replication, distributed object applications operating in mobile environments can help prevent data loss by facilitating the remote replication of data. It should be noted, however, that several sources criticise current middleware architectures for being somewhat lacking in their support for object replication [107, 15] and have proposed improvements which have yet to be integrated into the architectures.

With regards to privacy threats, the problem can be divided in two: privacy of the communication between the device and other parties and privacy of data stored on the device. The former is discussed in section 2.4.5. All applications, distributed or otherwise, are of course vulnerable to privacy threats in case of theft of the mobile device.

2.3.3 User Interface

Forman et al. observe that limited display sizes make desktop windowing technology inadequate for small, portable devices and suggest wearable head-mounted displays (HMDs) as a possible alternative. In terms of input, space constraints cause buttons to be sacrificed in favour of analog input devices, such as handwriting, gesture and voice recognition. While gesture recognition has yet to become a common feature of mobile computers, all PDAs and many laptop computers do feature pens rather than mice as pointing devices. Compared to mice, pens leave more space available on the display (because they require no on-screen cursor) but they can also mislead the user when pointing, due to the parallax between the pen tip and the screen image [63].

On average, the PDAs listed in table 2.1 had displays that were just over a quarter the size of a portable PC display, and all PDA displays were monochrome. Table 2.2 shows that some 2002 PDAs are equipped with colour rather than monochrome displays, but that monochrome displays still exist. While display resolution for PDAs has remained largely unchanged from 1994 to 2002, the physical size of the displays has been reduced by approximately half. This corresponds to the general reduction in size indicated by the decrease in weight previously attributed to the progression of the PDA towards an ideal form factor. It seems safe to conclude that the reduction in size of the devices constitutes a sufficient improvement in terms of usability to mandate the associated reduction in display size. While a larger display would generally allow a better user interface, the fact that technology advancements have allowed resolution to remain unchanged, and also made colour displays possible, are mitigating circumstances. For portable PCs, display size and resolution have more than doubled. As a result, modern PDAs have displays that are less than a tenth of the size of a portable PC. The wearable computer listed in table 2.2 uses a relatively non-intrusive head-mounted display as predicted by Forman et al.

While restrictions in form factor and power consumption have driven the development of new types of user interfaces for mobile computers, the use of such interfaces apply to all applications running on mobile devices, distributed or otherwise. While potentially relevant to all applications running on mobile devices, the problems related to the user interfaces of such devices are not specific to distributed object applications.

2.3.4 Storage Capacity

Forman et al. and Satyanarayanan agree that storage space is limited by physical size and power requirements [63, 159]. In terms of memory, Badrinath et al. observe that 'mobile hosts have significantly lower memory capacity ... compared to fixed hosts' [14]. Forman et al. observe that disk

drives traditionally used for nonvolatile data storage, have significant power requirements and are fragile and therefore not very well suited for mobile devices [63]. Forman et al. also observe that data compression can reduce storage requirements and that the use of interpreted scripting languages can not only reduce the size of executables but also enhance software portability [63].

The PDAs listed in table 2.1 have approximately an order of magnitude less memory than the portable PC, and none can support disk drives. Comparing Random Access Memory (RAM) for the devices in the two tables shows that the amount has increased between one and two orders of magnitude from 1994 to 2002. This is the case for portable PCs as well as PDAs. For PDAs, this development corresponds to a significant improvement in terms of storage capacity. For portable PCs, which use disk drives for nonvolatile storage, this development corresponds to support for more complex applications and better support for multitasking. The observation by Forman et al. that disk drives are not well suited for mobile environments still applies. None of the PDAs listed in table 2.2 come with disk drives, although miniature disk drives are starting to become available as options for high-end PDAs, such as the Compaq iPAQ. Common for the two tables is that PDA memory sizes remain about an order of magnitude lower than that of portable PCs.

Limitations in terms of storage capacity are important for any application and no less so for distributed object applications. In general, all applications designed for mobile environments must attempt to minimise storage requirements. This applies to the basic footprint of the application as well as to runtime storage of data.

The persistent gap despite absolute increases in available storage between the two categories of mobile devices is noteworthy and represents a general trend. One source, Satyanarayanan, identified this trend with a 1996 prediction that '[w]hile mobile elements will improve in absolute ability, they will always be resource-poor relative to static elements' [159]. Similar cases to that of storage can be made for other characteristics of mobile devices, such as processing power and available network bandwidth, discussed in sections 2.3.5 and 2.4.3 respectively.

2.3.5 Processing Power

Satyanarayanan's observation that mobile computers are resource-poor compared to stationary computers [159] also applies to processing power. Badrinath et al. agree that 'mobile hosts have significantly lower ... computing power compared to fixed hosts' [14]. Although Forman et al. do not discuss processing power explicitly, table 2.1 supports this notion.

Half of the PDAs listed in table 2.1 are based on 8-bit CPUs (Z-80s and 8086s), whereas the other half use signficantly more powerful 32-bit CPUs (ARMs). The portable PC uses a 32-bit

CPU at a higher clock speed than any of the PDAs. An interesting observation is that the devices with the low-performance CPUs on average retain a significantly higher battery life than those with high-performance CPUs. This supports an observation by Chlamtac et al. that 'power required by the chip increases as a function of the clock frequency' [40]. In comparison with the 8-bit CPUs popular with 1994 PDAs, all the CPUs listed in table 2.2 are 32-bit. While the clock speeds for the PDAs in this table have increased significantly (up to an order of magnitude for the ARM-based PDAs) compared to those listed in table 2.1, clock speed for portable PCs has increased even more dramatically by more than two orders of magnitude. Despite advancements in PDA processing power, the gap between processing power of laptop PCs and PDAs has remained from 1994 to 2002. This supports Satyanarayanan's prediction discussed in section 2.3.4 that mobile devices will always be resource-poor relative to static ones [159].

Much like the storage limitations discussed in section 2.3.4, limitations in processing power require applications designed for operation in mobile environments to be as lightweight in terms of processing requirements as possible. This constraint is not specific to distributed object applications but applies to all applications on mobile devices.

2.4 Mobile Networking

Section 2.3 discussed the characteristics of mobile computers and identified a number of respects in which such computers are—and will continue to be—restricted compared to their stationary counterparts. The comparison between the 1994 and 2002 mobile computers also showed a considerable increase in capabilities and reduction in size. An area of functionality of particular interest is the devices' networking options. While none of the 1994 PDAs came with builtin wireless connectivity, the 2002 PDAs generally have a considerable number of native networking options. While this allows new types of applications to run on mobile devices, it also makes demands on the software operating on these machines.

Under the heading 'wireless communication,' Forman et al. define five areas in which network issues related to the mobile environment pose challenges to software design: disconnection, low bandwidth, high bandwidth variability, network heterogeneity and security risks [63]. In general, mobile computers are not restricted to wireless networks but also often avail of wired communication when possible. This section discuss issues related to mobile networking—wired and wireless—by following the outline suggested by Forman et al. and bringing in observations from other sources.

2.4.1 Network Heterogeneity

Despite the limited capabilities discussed in section 2.3, mobile devices avail of surprisingly diverse ways of connecting to their surroundings. Forman et al. observe that mobile devices may need to change transmission speed and protocols as they move between network transceivers and may have access to several network connections simultaneously [63, p.41]. Also, mobile computers may need to switch interfaces, for example when moving between indoor and outdoor environments [63, p.41].

Table 2.3 lists popular modes of connectivity for three categories of computers: desktop computers, laptop computers and PDAs. A bullet (•) indicates native support for that mode of connectivity and a circle (○) that the mode is available as an option. A blank means the mode is not available. The table only lists modes of connectivity generally used for communication between computers, mobile or otherwise. Interfaces used to communicate mainly with peripherals such as printers, mice and keyboards are not listed.

	Mode of Connectivity									
	Wired					Wireless				
Product	Modem	RS232	USB	Fast Ethernet	Gigabit Ethernet	GSM Modem	Bluetooth	InfraRed	GPRS	WLAN
Dell Dimension 4300	○	•	•	○	○	○	○	○	○	○
Gateway E-3600	○	•	•	•	○	○	○	○	○	○
Fujitsu Lifebook B142	•	•	•	○		○	○	•	○	○
Dell Inspiron 3800	•	•	•	○		○	○	•	○	○
Handspring Visor Pro	○	•	•	○		○	○	•	○	○
Compaq iPAQ H3870	○	•	•	○		○	•	•	○	○

Table 2.3: Networking Options for 2002 Desktops, Laptops and PDAs

The table shows that nearly as many modes of connectivity are available to PDAs and laptops as to stationary computers. The only exception is GigaBit ethernet for which interfaces are not currently available in PCMCIA card format and therefore cannot be used with PDAs and laptop computers. The table also shows that PDAs, despite the hardware limitations discussed in section 2.3, typically have about as many native modes of connectivity as laptop computers and more than average desktop computers. Laptop computers and PDAs tend to avail more of their diverse range of networking options than do desktop computers because the different types of networks have different characteristics that are suitable for operation in different circumstances. Section 2.4.4 discusses this in more detail.

Sometimes, a mobile device does not connect directly to a LAN but to another host that has a permanent LAN connection. A common example is a PDA connected to a desktop computer via a

docking cradle. At other times, a mobile device may be directly connected to the LAN via its own network interface. A mobile host may be connected to different desktop computers and different LANs at different points in time. In short, the networking options for a mobile host are more complex than those of a fixed host. Instead of a single high bandwidth interface connected to one network, a mobile host may have several interfaces of varying bandwidth and quality, each of which can be connected to many different networks at various points in time.

Standalone applications do not require use of the network and are therefore not affected by network heterogeneity. However, distributed object applications rely on connectivity for communication between the different parts of the application, and the heterogeneous network conditions therefore represent a significant challenge. If no system support for dealing with network heterogeneity is available, applications must deal with changes in transport explicitly. Such changes may also include variations in terms of bandwidth, error rates and cost, which it may not be possible (or desirable) to hide. Sometimes changes in transport can be anticipated by the user, and application performance may be improved if such 'hints' can be taken into account. For distributed object applications, there is a trade-off involved between awareness and unawareness of changes in transport. Awareness may allow some applications to offer improved performance to users, for example by adapting their behaviour to match the characteristics of the transport in question, at the cost of increased application complexity. Other applications may not be able to benefit from such awareness, and in such cases it may be better to mask out network heterogeneity to the extent that this is possible.

2.4.2 Disconnection

Forman et al. observe that '[stationary] computer systems often depend heavily on a network and may cease to function during network failures' [63, p.39]. Disconnection may also occur if the mobile host is out of range of any other devices. Compared to wired communication, wireless communication is significantly more susceptible to disconnection, and there is a design trade-off between spending the limited resources of mobile computers trying to prevent or to cope gracefully with disconnections.

However, not all disconnections are involuntary. Badrinath et al. give a reason why a mobile host might elect to enter a disconnected state out of power preservation concerns:

> mobile hosts have severe *resource constraints* in terms of limited battery life and often operate in a "doze mode" or entirely disconnect from the network [14]

Forman et al. distinguish between short and long disconnections and observe that the former can be hidden by techniques that decouple the act of communication between the parties, such as

asynchronous operation, prefetching and delayed write-back. Long disonnections can be addressed via optimistic approaches to conflict detection and resolution [161, 63]. However, not all network disconnections can be masked, and in these cases 'good user interfaces can help by providing feedback about which operations are unavailable' [63, p.40].

Like network heterogeneity, standalone applications are unaffected by disconnection, but to distributed object applications, disconnection poses a challenge because such applications rely on the network for communication between the different portions of the application. Requiring applications to deal with all disconnections explicitly can increase application complexity, and in most cases it may be desirable to shield applications, for example, from short-term involuntary disruptions in connectivity. Whether anticipated or not, long-term disconnections are more likely to require changes in application behaviour and should not necessarily be masked. Not unlike the case for network heterogeneity discussed in section 2.4.1, there is a trade-off between awareness and unawareness of the varying state of connectivity. Given knowledge about changes in connectivity, some applications may be able to adapt their performance and increase their usability at the cost of increased application complexity. For example, some web browsers designed for use on mobile computers have a disconnected mode of operation where content is served from a cache on the mobile device and no attempts to fetch new content are made, even when the cached content is deemed to have expired. Other applications may not benefit from awareness of variations in connectivity and it may be impractical to require them to deal with it. For example, there may be no reason for a mobile web server to change its behaviour when the computer on which it runs is disconnected, and it may be preferable if periods of disconnection appear simply as ordinary periods without client requests.

2.4.3 Low Bandwidth

In 1994, Forman et al. observed that '[w]ireless networks deliver lower bandwidth than wired products' [63, p.40] and compared 10 Mbps ethernet and 155 Mbps ATM networks with 9–14 Kbps cellular telephony and 2 Mbps radio communication. The difference in data rate between the fastest wired and wireless consumer technologies was approximately two orders of magnitude. Since then, the state of the art in networking technologies has advanced, but the relative difference remains. Table 2.4 shows data rates for a number of current network technologies. While the fastest wired network technology on the consumer market has a data rate of 1 Gbps, wireless LAN technologies have also improved and the difference remains approximately two orders of magnitude. Like the developments in storage capacity and processing power discussed in sections 2.3.4 and 2.3.5, this supports Satyanarayanan's notion that mobile devices despite improvements will remain resource-poor compared to static elements [159].

Technology	Bandwidth
Modem	56 Kbps
RS232	115 Kbps
USB	1.5 Mbps
Fast Ethernet	100 Mbps
GigaBit Ethernet	1,000 Mbps
GSM Modem	14.4 Kbps
Bluetooth	115 Kbps
InfraRed	500 Kbps
GPRS	115 Kbps
WLAN	11 Mbps

Table 2.4: Data Rates for Wired and Wireless Network Technologies

To remain functional in the face of low bandwidth connections, distributed object applications must use as little bandwidth as possible. As discussed in section 2.3.1, communication can be reduced through the use of compact on-the-wire data formats, data compression and support for disconnected operation. Standalone applications are not affected by low bandwidth.

2.4.4 Bandwidth Variability

Many wireless technologies are subject to fluctuation in bandwidth. Users of mobile devices may travel beyond the coverage of network transceivers or enter areas with high interference, caused for example by the surrounding environment blocking signal paths or introducing noise or echoes [63, p.39]. Such problems may also lead to the total loss of bandwidth in the form of dropped connections discussed in section 2.4.2. Also, different technologies may rely on the same frequency bands and therefore not easily coexist, as has been the case with IEEE 802.11 [60] and Bluetooth [4], both of which rely on the 2.4 GHz spectrum.

For some wireless technologies (e.g., Bluetooth), the available bandwidth is related to the density of transmitters and receivers because network bandwidth is divided among the users sharing a nework cell [63, p.40]. Hence, even if a mobile device is stationary, other devices moving in and out of the cell may cause the bandwidth available to the stationary device to fluctuate. In addition to variability for individual wireless technologies, bandwidth can vary across interfaces. The heterogeneity in terms of network interfaces discussed in section 2.4.1 means that the data rates for mobile devices can vary dramatically. For example, the iPAQ PDA listed in table 2.3 can communicate via a 9.6 Kbps GSM modem at one time and through a 100 Mbps Fast Ethernet card at another. This is a difference in data rate of four orders of magnitude.

While standalone applications are not affected by bandwidth variability, it is clear that distributed object applications can make few assumptions about the available bandwidth over extended periods

of time. The conservative approach is to assume low bandwidth connectivity, but this could lead to suboptimal performance and usability of some applications when high bandwidth connectivity is available. For example, multimedia applications tend to benefit from high bandwidth links when available but may be able to operate with reduced levels of service if only low bandwidth connectivity is possible [88, 44]. Hence, for each application there is a trade-off between awareness and unawareness of variations in bandwidth. The former may allow some applications to improve performance, but requires them to deal with the variations explicitly, resulting in increased complexity. The latter decreases complexity of applications (and of application development) but for some applications may result in suboptimal performance.

2.4.5 Security Risks

Forman et al. observe that 'the security of wireless communication can be compromised much more easily that that of wired communication' and that this 'increases pressure on mobile computing software designers to include security measures' [63]. In connection with ad hoc networks, Zhou et al. describe the security challenges of wireless networks as follows:

> [The] use of wireless links renders an ad hoc network susceptible to link attacks ranging from passive eavesdropping to active impersonation, message replay, and message distortion. Eavesdropping might give an adversary access to secret information, violating confidentiality. Active attacks might allow the adversary to delete messages, to inject erroneous messages, to modify messages, and to impersonate a node, thus violating availability, integrity, authentication, and non-repudiation [192].

In general, wireless links can be made private via encryption, and parties can be authenticated via public/private key encryption schemes combined with suitable public key infrastructure (PKI). However, the mobile environment may impede the use of PKI because public key verification (e.g., for X.509 certificates [89]) in many cases requires the availability of a certificate authority. If a mobile computer is disconnected or operating in ad hoc mode, it may be difficult for its applications to verify the authenticity of public keys offered by correspondent parties.[1] The active attacks discussed by Zhou et al. pose even more difficult problems, and measures to counter such threats are likely to increase the requirements on software support on the mobile devices. The remainder of this section focuses on privacy as an example.

[1] This observation, as yet unpublished, was made by Christian Jensen in 2001.

While standalone applications are generally unaffected by security threats related to mobile networking, distributed object applications require communication with remote application components and are therefore vulnerable. Not all distributed object applications may have privacy requirements, but others may benefit from the use of encryption to assure privacy of the communication. Encryption can be implemented at various points in the protocol stack, for example on top of (or as part of) individual transports. As an example, the Secure Sockets Layer (SSL) [65] and Transport Layer Security (TLS) [54] protocols reside above TCP in the Internet protocol stack and offer stream abstractions which, aside from being private, are comparable to those of TCP. All tasks related to encryption, such as key management and negotiation of cipher suites, are handled by the encryption layer. APIs for implementations of SSL and TLS typically differ a little from standard interfaces to TCP (e.g., Berkeley [175] or Java [2] sockets), but many SSL and TLS toolkits (e.g., Java Secure Sockets Extension (JSSE) [2]) aim to provide interfaces that resemble the underlying transports in an attempt to make it easier to use existing applications with the encryption toolkits. Applications that require influence over the encryption parameters (e.g., have special preferences regarding cipher suites or key lengths) are accommodated. For these reasons, transport-level encryption does not necessarily require a high level of application involvement. As was the case for bandwidth variability, discussed in section 2.4.4, there is a trade-off involved between awareness and unawareness. While the former may allow applications to offer improved service to users (e.g., by insisting on minimum key lengths), it also increases complexity of the applications.

2.4.6 Usage Cost

The financial cost of connectivity for mobile computers is an issue that has received relatively little attention in the literature, perhaps because it is not related to the network technologies themselves but to the policies and agreements under which they are used. One source, Kojo et al. [105], discusses the cost of GSM connectivity for the purposes of nomadic computing. For the user of a mobile computer, connectivity cost constitutes a significant constraint, and although this constraint is imposed by the user rather than the technology, it can be addressed naturally alongside other constraints related to mobile connectivity.

The heterogeneity of the network options discussed in section 2.4.1 is also reflected in their usage cost which can vary considerably. While typical uses of RS232 and USB interfaces are free, the use of conventional and GSM modems tend to incur a considerable cost on the user. Connectivity can be subject to a number of different charging schemes: flat subscription fees, billing per data unit transmitted or per unit of time spent online, or the combination of several different schemes. For

example, GSM and WAP connectivity typically require a monthly subscription fee and an additional charge per minute spent online [6]. In comparison, DoCoMo i-mode is based on a subscription fee plus an extra charge per data unit transmitted [3]. WLAN connectivity installed in some cafés is available for a flat monthly subscription fee with no extra charges [134].

For transports where a charging scheme is employed that depends on the amount of data transmitted or the time spent online, it may be worthwhile to reduce application requirements to communication. The methods previously described for reducing the amount of data transmitted by distributed object applications, for example, in section 2.3.1, also apply here. Some applications may be able to improve performance by being sensitive towards the cost of individual transports and modifying their behaviour accordingly, for example in conjunction with user hints or preferences. Like disconnection and bandwidth variability discussed in sections 2.4.2 and 2.4.4, the cost of connectivity is a parameter that can be made available to applications or not. The trade-off between awareness and unawareness has the same implications of possibly improved performance at the cost of increased complexity as discussed in previous sections. The issue of communications cost does not apply to standalone applications.

2.5 Physical Mobility

The previous sections discussed issues related to the mobile devices themselves and to the network technologies commonly used with such devices. Another group of problems is related to the mobility of the devices, i.e., to the fact that they are moving. These problems are independent of the devices themselves and the network technologies typically used with them. Forman et al. lists three issues related to physical mobility of computers: address migration, location-dependent information and migrating locality [63, pp.42–43].

2.5.1 Address Migration

The address migration problem arises from the fact that wired networks generally employ hierarchic routing protocols, such as IP. In hierarchic routing, host addresses belong to particular (physical) networks, and to move to a different network, a host must change its physical point of connectivity and hence obtain a new network address. Chlamtac et al. describe this as follows:

> In traditional desktop computing, the topology of the network is static. The physical location of the machine is encoded into its network address in protocols such as TCP/IP.

> In the mobile network, the topology can be constantly changing as the devices move from cell to cell. [40]

Badrinath et al. also identify this problem as related to dynamic network architecture:

> Mobility of a host implies that its location relative to the rest of the network changes with time; the connectivity of the entire network is thus modified as hosts move. Consequently, a logical link between two mobile hosts can no longer be mapped to a fixed sequence of phyiscal links in the underlying network. [14]

The need to change network addresses results in a number of problems. For example, Forman et al. observe that '[t]oday's networking is not designed for dynamically changing addresses. Active network connections usually cannot be moved to a new address' [63, p.42]. Hence, moving to a different point of connectivity will generally cause open connections to break, because connection endpoints are identified by network addresses and with current hierarchical routing protocols (most notably IPv4) cannot easily be reconfigured.[2] A level of indirection exists in the form of the Domain Name System (DNS) that maps hostnames to network addresses, but DNS is not suitable for the frequent updates required for mobile devices that move frequently. On this note, Forman et al. observe that '[o]nce an address for a host name is known to a system, it is typically cached with a long expiration time and with no way to invalidate out-of-date entries' [63, p.42].

Forman et al. discuss four basic mechanisms to determine a mobile computer's current address: selective broadcast, where a query is sent to a small set of cells estimated to be in the vicinity of the mobile host's current location; central services, where a (possibly replicated) location service maintains up-to-date information about current addresses of mobile hosts; home bases, where location information is maintained in a distributed, non-replicated fashion; and forwarding pointers, where mobile hosts leave traces of pointers behind them pointing towards more recent locations. Each of these methods has its advantages and drawbacks which are discussed in [63].

While not applicable to standalone applications, address migration causes two sets of problems for distributed object applications. First, the breaking of transport connections discussed above may cause invocations to fail. Extra functionality (e.g., support for transactions) can be added at the application or middleware layers to prevent inconsistencies despite frequent failures, but this approach increases application complexity and may cause code to be replicated at the application layer. As for short-term disconnections, a better approach may be to shield distributed object applications from transport connections broken as a result of changes in point of connectivity. Second, server references

[2] More recent initiatives—Mobile IP and IPv6—have features which help remedy this but suffer from other problems, discussed in section 3.1.

used by clients to invoke servers typically contain the network address (or host name which the client resolves to a network address using DNS) of the host on which the server is running. If the mobile host contains a server object, a change in the host's network address may cause server references held by clients to become obsolete because the server cannot retain the old address at the new point of attachment. Unless extra functionality is added, for example via one of the four methods suggested by Forman et al., a client holding an old server reference may not be able to locate the server at its new point of attachment.

2.5.2 Location-Dependendent Information

Forman et al. observe that '[t]he current location of a mobile computer affects configuration parameters as well as answers to user queries' [63, p.42]. Configuration parameters are low-level and application-independent, such as name servers, gateways, printers and the time zone used in the particular location; and Forman et al. propose that mobile hosts need to discover such basic services in a dynamic manner [63]. The authors also observe that the same requirement applies to user queries, i.e., to application-level functionality. Generally, applications for stationary computers rarely change behaviour depending on their location, and their degree of location-awareness is typically limited to rudimentary knowledge of the city and time zone in which the computer is located. In comparison, some categories of mobile applications can offer better service to users by being aware of the location and environment of the device hosting them. For example, given knowledge of their geographic location, guide and map type applications may be able to assist users in navigating an area or locating nearby services, such as libraries, petrol stations, restaurants, etc. Lancaster University's GUIDE project [39] is a good example of such an application. For these reasons, mobile computers generally need access to more location-sensitive information than stationary computers, also at the application-level. Forman et al. further distinguish between static and dynamic application-level location information. While the former (e.g., locations of nearby restaurants) remains unchanged over long periods of time, the latter (e.g., locations of nearby taxis) can be expected to change frequently. Static and dynamic location information may need to be managed differently [63, p.43].

Whether the information in question is low-level system configuration parameters or high-level application-specific information, the issue of location-dependent information applies at several levels in the protocol stack. As such, the problem also applies to all applications on mobile hosts, distributed as well as standalone. However, distributed applications are likely to require some types of location-dependent information (e.g., addresses of gateways and name servers) that standalone applications may not need. For this reason, the problem of location-dependent information is particularly relevant

but not specific to distributed applications, including distributed object applications.

A variety of general solutions have been proposed that allow applications to discover services in their vicinity dynamically. An example is Jini [186, 10], a Java-based middleware technology, that defines a set of protocols for discovery, join and lookup of services and a leasing and transaction mechanism for interacting with them. Jini is designed to be generally applicable to dynamic networked environments and can be used with applications running on mobile computers as well as mobile applications running on stationary computers. Another initiative that addresses this problem is Bluetooth [4] whose service discovery feature allows a device to describe itself and be discovered by other Bluetooth devices [120, p.214]. A network-layer solution can be found in IPv6 [53] which contains a feature called 'neighbor discovery' [129] that allows IPv6 nodes to solicit their (e.g., routing) services and be discovered by other IPv6 nodes in need of such services. The features of Bluetooth and IPv6 can be used to solve basic location-dependent information problems, such as the discovery and initial communication with nearby devices. Problems that rely on higher-level abstractions (e.g., geographical locations and concepts such as restaurants and taxis) can be addressed via middleware toolkits such as Jini.

2.5.3 Migrating Locality

Section 2.5.2 discussed information bound to a given locality. Even if a mobile host is able to discover and bind to services that may be nearby at a given point in time, these services may not remain nearby as the mobile host moves. Forman et al. describe this change in what is 'nearby' for a mobile host as 'migrating locality' and points out that the problem can be addressed by dynamically transferring service connections to servers that are closer, as the mobile host moves [63, p.43]. It is worth pointing out that many efforts in providing mobility support (e.g., Mobile IP [142] and IPv6 [53]) try to transparently maintain existing bindings, and this may not necessarily be the best approach. As an example, consider a web proxy to which a mobile application could conceivably discover and bind using a middleware toolkit such as Jini [186, 10]. When moving to a new location, it may be preferable to repeat the discovery process and create a new binding, rather than maintain the binding to the existing web proxy.

Like the problem of location-dependent information, migrating locality is applicable at several levels in the protocol stack. For example, basic configuration parameters (e.g., printers, gateways and time zones) may need to change repeatedly as mobile hosts move. For some applications—standalone as well as distributed—this can be problematic if they assume the information to be static and cache it for later use. As an example, the Unix functions `ctime()` and `localtime()` when invoked

set an external variable `timezone` indicating the current time zone. A natural way to address the migrating locality problem for time zone information would be to extend the system libraries that provide these functions to continually update the value of the `timezone` variable. However, modifying existing function semantics may cause problems if programmers expecting the conventional semantics mistakenly assume that there is no need to re-read the `timezone` variable. For non-mobile applications, this is not usually a problem since time zone changes are rare.

In line with the problem of location-dependent information discussed in section 2.5.2, distributed object applications are likely to require rebinding of some types of information or services that standalone applications do not, such as name servers and gateways. For this reason, the problem of migrating locality is particularly relevant but not specific to distributed applications, including distributed object applications.

2.6 Emerging Themes

The previous three sections have defined the problems related to the mobile environment with special attention to their impact on distributed object applications. For systems designed with relatively static network conditions in mind (such as object-oriented middleware frameworks), this environment constitutes a considerable challenge. For distributed object applications, extra functionality is required to deal with the mobile environment, either in the form of mobility-enhanced applications or special mobility support on the mobile hosts, or a combination. However, there is a trade-off involved in such solutions between addressing the three groups of problems discussed. Problems related to mobile networking (section 2.4) and physical mobility (section 2.5) can be addressed by increasing mobility support on mobile computers, but this is likely to increase the requirements on processing, storage and battery life on the mobile computers (section 2.3). Hence, addressing one group of problems makes it harder to address another.

Despite the variety of the problems discussed in the previous three sections, a number of general themes emerged in connection with their possible solutions. This section identifies and discusses five such themes: resource economy, communications economy, transparency vs. awareness, location management and disconnected operation. The five themes constitute a high-level synthesis of the possible approaches to addressing the challenges posed by the mobile environment. The themes are revisited in chapter 4 where they form the basis for key design decisions in connection with our architecture for adding mobility support to object-oriented middleware frameworks.

2.6.1 Resource Economy

This chapter has focused on two families of devices: the portable and the handheld computer. While the appearance of the handheld constituted a reduction in size from the portable, Weiser's vision of ubiquitous computing promises at least one more such reduction in the form of an even smaller family of devices embedded in everyday appliances. At the moment, such embedded devices are leading relatively isolated existences, not unlike the 1994 PDAs discussed in section 2.3 which came without wireless interfaces. If Weiser's vision can be trusted, the next years will see embedded computational functionality developing on a scale comparable to that of PDAs from 1994 to 2002. The two sensor nodes listed in table 2.2 serve as examples of the type of device many believe will be found in conventional appliances in the future.

For this reason, current limitations are likely to persist, albeit for a generation of smaller devices. While advances in chip design, wireless communications and power technology are certain to address many of these limitations, the constraints are unlikely to change within the foreseeable future. For hardware researchers and manufacturers, the challenge is to find the right trade-off between capabilities and size, as it seems to have happened for PDAs from 1994 to 2002 as they developed towards a better form factor. For software researchers and developers, the challenge is to maximise what can be achieved with the available hardware. In general, this requires the software for mobile hosts to use as few resources as possible in terms of data storage (section 2.3.4) and processing power (section 2.3.5).

2.6.2 Communications Economy

A significant number of problems discussed in this chapter had as their solutions a reduction in communications, either via compact protocol formats or through data compression. Reducing communications can result in power savings (section 2.3.1), improved bandwidth utilisation (section 2.4.3) and reduced financial cost (section 2.4.6). These are strong incentives for distributed object applications operating in mobile environments to use data formats optimised towards minimising the amount of data transmitted. For distributed object applications, talk may be silver, but for mobile devices, silence is gold.

Also regarding data formats for wireless transmission, improved privacy could be assured through the use of encryption (secion 2.4.5). If data streams transmitted to and from a mobile device are being compressed, it seems natural to add also a layer of encryption. Some encryption toolkits, such as the Secure Sockets Layer (SSL) [65], also include data compression algorithms, which can facilitate this.

2.6.3 Transparency vs. Awareness

For five of the six subproblems related to mobile networking, a trade-off between transparency and awareness was involved: network heterogeneity (section 2.4.1), disconnection (section 2.4.2), bandwidth variability (section 2.4.4), security risks (section 2.4.5) and usage cost (section 2.4.6). Generally, transparent solutions would hide these problems from applications while solutions based on awareness would require mobile applications to address the problems themselves. For applications, transparency has the advantage that no extra application code is required to deal with the changing environment. Also, for application developers, the freedom from having to deal with changing conditions constitutes a simpler programming model and is therefore likely to result in lower development time. However, some classes of mobile networking problems, e.g., long-term disruptions in connectivity, are difficult to mask. For such problems, attempts at retaining transparency may lead to suboptimal application performance, because applications are unable to adapt to the changing conditions. In comparison, awareness of changing conditions allows applications to adapt dynamically, and this can improve their performance. The cost associated with awareness is higher application complexity, because extra code is required to deal with the changing environment. Also, the more complex programming model will likely result in increased development time.

This theme corresponds to a taxonomy of adaptation strategies proposed by Satyanarayanan [159] and shown in figure 2.3. At one end of the scale, the *laissez-faire* approach leaves all mobility support to applications themselves. At the other end, fully transparent mobility support is provided. Between the two extremes is a spectrum of possibilities for various degrees of 'application-aware adaptation,' i.e., mobility support with varying degrees of application-awareness.

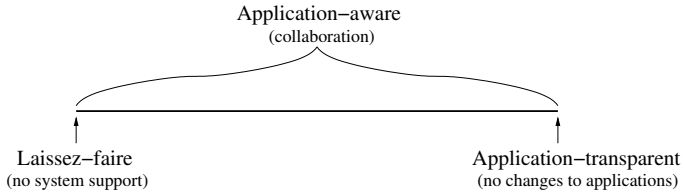


Figure 2.3: Range of Adaptation Strategies [159]

2.6.4 Location Management

For distributed object applications, the address migration problem (section 2.5.1) connected with physical host mobility is a significant challenge. A client holding an old server reference may not be able to locate the server at its new point of attachment. The problem of managing server references is

essentially a location management problem, the solution to which must make it possible for clients to obtain up-to-date information about a mobile computer's current location. Solutions to this problem can be distributed or centralised, and the resolutions performed by clients can occur at different levels in the protocol stack. For existing object-oriented middleware frameworks, this issue is particularly challenging, because such frameworks have not been designed to include mechanisms to frequently re-resolve server references. Chapter 3 presents and discusses in detail a number of approaches to dealing with this problem.

2.6.5 Disconnected Operation

Several of the problems identified in this chapter had as solutions support for disconnected operation. The ability to deal with involuntary disconnections can increase the usability of the device in the face of complete loss of network coverage or if battery power is too low to use wireless communications (section 2.4.2). Voluntary disconnection can also reduce power consumption (section 2.3.1) as well as communication expenses (section 2.4.6), and hence increase the usability of the device. Voluntary disconnection essentially corresponds to a complete reduction in communications (section 2.6.2).

Hence, for applications operating in mobile environments, there are considerable advantages to be gained from the ability to operate despite the lack of network connectivity. For distributed object applications, this challenge is particularly relevant because the different portions of such applications depend on the network to communicate. An approach to increasing support for disconnected operation can be to cache application functionality on either side of the failing link. Chapter 3 examines a number of approaches to dealing with this problem.

2.7 Summary

Section 1.1 argued that mobile computing environments form signficantly more complex operating environments than traditional computing environments. This chapter has defined the two main models used to describe mobile environments, identified three areas where such environments challenge the operation of distributed object applications and discussed the challenges related to each area. The discussion was subsequently used to identify five general themes that emerged in connection with possible solutions to the problems. The chapter also placed the thesis in context by outlining the development of the field of mobile computing from the appearance of the first mobile computers in the 1980s towards 'ubiquitous computing' which is generally perceived as the next paradigm after distributed computing. It was argued that despite projected advances in hardware technology, the

constraints and challenges currently associated with mobile environments are not going to go away but will continue to apply in the future, albeit to a different type of mobile environment.

Table 2.5 summarises these challenges and themes covered in this chapter. In the table, the columns titled 'Relevance' summarise the applicability of each of the fourteen challenges to distributed and standalone mobile applications. A bullet (•) means that the challenge applies to the type of application in question and a blank that it does not. Not surprisingly, challenges related to the devices apply to both types of applications, whereas challenges related to mobile networking are specific to distributed applications. In terms of physical mobility, only address migration was deemed specific to distributed applications. The columns of table 2.5 titled 'Emerging Theme' summarise which of the five themes identified in section 2.6 relate to the fourteen challenges. For these colums, a bullet indicates that the theme is relevant to the challenge and a dot indicates that it is not. It is worth noting that one challenge, data risks, is not covered by any of the themes identified, but that section 2.3.2 argued distributed object applications could be used to reduce data risks.

		Relevance		Emerging Theme				
General Category	Specific Challenge	Standalone Application	Distributed Application	Resource Economy	Communications Economy	Transparency vs Awareness	Location Management	Disconnected Operation
Mobile Devices (section 2.3)	Battery Power	•	•	•	•			•
	Data Risks	•	•					
	User Interface	•	•	•				
	Storage Capacity	•	•	•				
	Processing Power	•	•	•				
Mobile Networking (section 2.4)	Network Heterogeneity		•			•		
	Disconnection		•			•		•
	Low Bandwidth		•		•	•		•
	Bandwidth Variability		•		•	•		
	Security Risks		•			•		
	Usage Cost		•		•	•		•
Physical Mobility (section 2.5)	Address Migration		•			•	•	
	Location-Dependent Information	•	•					
	Migrating Locality	•	•					

Table 2.5: Summary of Mobile Environment Challenges

Chapter 3

State of the Art

> *They were indeed a queer-looking party that assembled on the bank—the birds with draggled feathers, the animals with their fur clinging close to them, and all dripping wet, cross, and uncomfortable.*
>
> *The first question of course was, how to get dry again: they had a consultation about this, and after a few minutes it seemed quite natural to Alice to find herself talking familiarly with them, as if she had known them all her life.* [35]

Chapter 1 divided previous research on middleware for mobile computing into two main groups: the development of new middleware architectures and the adaptation of existing middleware architectures for use in mobile environments. While the former typically involves modifying the semantics of existing middleware concepts to suit mobile enviroments (e.g., by introducing *queueing* of RPCs [97] or by *fragmenting* tuple spaces [126]), the latter has focused on retaining existing abstractions with few or no modifications to their semantics. For both types of approaches, the problems related to the mobile environment can be addressed at several levels in the protocol stack: from the network and transport layers up to the middleware and application layers. While some solutions reside mainly in one layer, hybrid schemes—where different problems are addressed at different layers—have also been proposed.

This chapter presents the state of the art in mobility support for distributed object applications. Although there exists a considerable body of work based on both of the mobility models discussed in section 2.2, only work based on the gateway model is considered. We first discuss lower layer solutions and then gradually proceed up through the protocol stack towards solutions whose main components reside at the middleware and application layers. Section 3.1 discusses network layer solutions in the form of IP mobility support while section 3.2 discusses transport-layer solutions (e.g.,

mobility-enabled sockets). Section 3.3 presents a number of middleware solutions (a file system and a distributed object system) tailored for mobile environments through the modification of the semantics conventionally provided by such middleware. Finally, sections 3.4 and 3.5 discuss mobility support initiatives specific to two main distributed object architectures: CORBA and Java RMI. The efforts covered in these two sections generally attempt to preserve existing semantics.

3.1 Network-Layer Mobility Support

In the Open System Interconnection (OSI) model, the *network layer* is responsible for functions related to routing of data as well as addressing, internetworking, error handling, congestion control and packet sequencing. In the Internet protocol stack, these functions are performed by the Transmission Control Protocol (TCP) [152] and the Internet Protocol (IP) [151] layers. IP is responsible for routing, addressing and error handling, while TCP performs congestion control and packet sequencing. IP addresses serve two functions: to identify nodes and to facilitate the routing of data [143]. This dual purpose becomes problematic when mobile nodes change their physical point of connection to the network and therefore (as discussed in section 2.5.1) need to change their network address, but at the same time retain their identity.

A number of network-layer solutions have been proposed which address this problem [177, 92, 94, 138, 167, 128]. By far the most influential of these is Mobile IP [138] which specifies a set of extensions to IP that improves support for host mobility by 'allowing the mobile computer to effectively utilize two IP addresses, one for identification, the other for routing' [143, p.68]. Mobile IP has been specified for IPv4 [151] as well as IPv6 [53]. This section presents and discusses the two versions of Mobile IP as well as an alternative approach to network-layer mobility support in the Internet protocol stack called Mobility Support using Multicasting in IP (MSM-IP) [128]. Section 3.1.4 summarises a number of other approaches not discussed in detail and concludes that current network-layer approaches to mobility support are not in themselves sufficient for comprehensive support of applications operating in mobile environments.

3.1.1 Mobile IPv4

Since proposed in 1994, Mobile IP has been implemented a considerable number of times, and Mobile IPv4 in particular has 'deployments numbering into the millions' [143, p.66]. However, the success of Mobile IP must be measured against the growth of the Internet as a whole. In a 2002 update [143] to his 1997 landmark paper [142], Perkins observes that '[t]he millions of existing Mobile IP

deployments represent only a very small fraction of the tens and hundreds of millions of network nodes currently attached to the global Internet' and on this basis concludes that 'Mobile IP has not lived up to its promise' [143, p.66]. However, Mobile IP remains the most influential proposal for network-layer mobility support in the Internet protocol stack and constitutes the only solution to reach any significant degree of adoption by the Internet community.

Description

Mobile IP operates with three primary entities: *mobile hosts*, *home agents* and *foreign agents*. Mobile hosts move between *networks* and interact with home and foreign agents (which are essentially routers) that reside in the networks and facilitate routing of data to and from the mobile host. One designated (possibly virtual) network acts as the mobile host's *home network*, while all other networks are *foreign networks*. The mobile host's home agent resides in the home network, while foreign networks typically hold at least one foreign agent with which the mobile host can interact. It is possible for the mobile host to act as its own foreign agent, in which case the presence of a foreign agent in the foreign network is not required. However, this places additional requirements on the mobile host in relation to routing and tunneling.

In Mobile IP, a mobile host has two addresses: a *home address*, which identifies the mobile host, and a *care-of address*, which facilitates the routing of data. Whenever a mobile host moves to a new network, it obtains a new care-of address, typically through interaction with a foreign agent. The home address remains unchanged. At any point in time, the binding between the two addresses is maintained by the home agent. For the purposes of routing, all IP datagrams sent to the mobile host are sent to the home address where they are received by the home agent. The home agent forwards the datagrams to the mobile host's current care-of address where they are received by the current foreign agent, which then forwards them to the mobile host. If the mobile host acts as its own foreign agent, the second level of forwarding is not required. Datagrams sent from the mobile host appear as if they originated from the home address rather than the care-of address. The functions of Mobile IP can be divided into three related groups: *agent discovery*, *registration* and *routing/tunneling*.

Agent Discovery

Mobile IP builds on the Router Discovery [52] feature of the Internet Control Message Protocol (ICMP) [150]. Router discovery allows routers to advertise their services through periodic broadcasts on their directly attached subnetworks. Mobile IP defines an extension to the ICMP router advertisement format which allows home and foreign agents to advertise their services. For foreign agents, one of

the functions of the extension field is to indicate a number of available care-of addresses that mobile hosts can use. A mobile host that wishes to use an address publicised in this manner must register it with the foreign agent from which the advertisement originated. If a home agent does not also act as a foreign agent, it does not need to include care-of addresses in its advertisements, but it still needs to advertise, because the mobile hosts for which it serves as a home agent rely on its advertisements to discover when they have returned to their home network. As is the case for ordinary Router Discovery [52], a mobile node can also solicit an advertisement by sending an ICMP Router Solicitation message. If a mobile host wishes to act as its own foreign agent, it must still obtain a care-of address via an independent service, such as the Dynamic Host Configuration Protocol (DHCP) [56], or through static allocation. This allows the mobile host to operate in the foreign network in the absence of foreign agents.

Registration

When a mobile host has obtained a care-of address, it registers it with the home agent. This typically involves the mobile host sending a registration message to the foreign agent which then forwards it on to the home agent. If the mobile host acts as its own foreign agent, it sends the registration message directly to the home agent. The registration message contains amongst other things an identifier for the mobile host, the new care-of address and a requested lifetime for the registration. If involved, the foreign agent can deny the registration (e.g., due to insufficient resources or failure to contact the home agent) or forward the message to the home agent. The home agent may also refuse registration, for example due to insufficient resources, but this is expected to be rare. If registration is successful, the home agent updates its binding to reflect the new care-of address and returns a registration reply containing a lifetime that indicates how long the home agent will honour the registration. If the registration was performed through the foreign agent, the foreign agent returns the reply to the mobile host; if not, the mobile host receives it directly. Mobile IP registration requests and replies are sent using the User Datagram Protocol (UDP) [149] to a well-known port (434).

Routing and Tunneling

When registration has been completed, the mobile host can start sending and receiving IP datagrams. Datagrams sent from the mobile host have their source field set to the mobile host's home address. To corresponding nodes, the mobile host effectively resides at the home address. Any traffic sent to the mobile host, is routed first to the mobile host's home address. If the mobile host is present on the home network, the datagrams are delivered to it directly. If the mobile host is away, the datagrams

are delivered to the home agent which encapsulates them and tunnels them to the foreign agent. The foreign agent decapsulates the datagrams and delivers them to the mobile host. Tunneling can be done in a variety of ways, but implementations of Mobile IP must always support at least simple IP-within-IP encapsulation [139], where an extra IP header is added to the datagram and the original header left intact.

For a mobile host communicating with a remote party, the flow of datagrams is asymmetric. All communication destined for the mobile host goes from the source to the home agent to the mobile host. Communication originating at the mobile host goes directly to the destination. This is shown in figure 3.1. Due to the three legs involved, this pattern of communication is called 'triangle routing.'

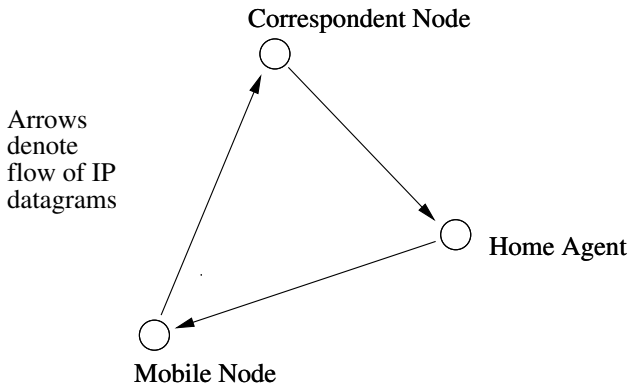


Figure 3.1: Triangle Routing in Mobile IP

Discussion

Mobile IP makes it possible for standard IPv4 nodes (which are unaware of mobility) to communicate with mobile hosts thanks to the level of indirection offered by the home agent. All functionality required for mobility support is kept on the home and foreign agents and on the mobile host. However, the consequent triangle routing is 'far from optimal' [143, p.75] because datagrams destined for the mobile node must travel through the home agent. Route optmisation efforts have attempted to improve performance by making the correspondent node aware of changing care-of addresses. However, for IPv4 this requires changes to the protocol stack on the correspondent node and is therefore a less attractive solution.

As discussed above, IP datagrams originating from mobile nodes have the source field set to mobile nodes' home addresses. The datagrams are transmitted directly to correspondent nodes, which as a result will send return traffic to the home address. While this approach facilitates triangle routing,

the somewhat creative use of the IP source field can also be problematic. One potentially difficult scenario is when a mobile host visiting a foreign network sends a datagram to a node on its home network. To the home network's firewall, such a datagram originates outside the home network but has a source address that belongs to the internal network. Firewalls are generally configured to reject such datagrams, because they constitute a common way to perform denial of service attacks using a technique known as IP source address spoofing. For a mobile node, such firewall configurations can make it difficult to communicate with the home network. Possible solutions are to use specialised firewall configurations (with some reduction in security) or reverse tunneling, where datagrams destined for the home network are tunneled through the home agent [124].

The modified source field can also cause a different but related problem. When the datagram goes out on the foreign network, the source address does not belong to the foreign network. In many enterprises, border routers between the enterprise network and the Internet are configured to discard packets coming from the enterprise network, if the packets do not contain a source IP address belonging to that enterprise network. This technique, called *ingress filtering* [61], can reduce denial of service attacks that employ IP source address spoofing, because packets with forged source addresses will never leave their originating network. For Mobile IP however, ingress filtering will prevent any packets originating at the mobile node from leaving the foreign network. Montenegro has proposed to address this problem by establishing a reverse tunnel from the care-of address to the home agent [123]. While this solution does offer topologically symmetric routing of packets, it effectively requires all traffic to and from the mobile node to be tunneled via the home agent and is therefore likely to result in a further reduction in performance than standard triangle routing.

Mobile IP constitutes a solution to the problem of hosts moving between domains. This type of inter-domain mobility has also been called *macro-mobility*. In Mobile IP, the handover triggered as a result of macro-mobility comes with a considerable cost, which may be disruptive for some types of applications. For example, Perkins observes that '[t]he base specifications for Mobile IPv4 and Mobile IPv6 do not really perform as well as one might like for realtime handovers' [143, p.66]. Some efforts have attempted to improve the performance of existing Mobile IP handover in different circumstances. For example, Choi et al. propose to parallellise Mobile IP and cellular handovers when Mobile IP is used in connection with data packet services [41].

While Mobile IP solves the inter-domain mobility problem, it is often possible for mobile hosts also to move *within* a given domain, e.g., from one wireless transceiver to another. This type of intra-domain mobility is called *micro-mobility*. Basic Mobile IP generally assumes that a mobile host's care-of address remains the same within a given mobility domain and assures routing of traffic to

and from this care-of-address. However, in order to improve routing efficiency within the mobility domain, it is often practical to retain more fine-grained information about a mobile host's current intra-domain location, for example in the form of a wireless access point to which the mobile host is connected. Hosts may move between such access points so frequently that it is impractical to track the them using (costly) Mobile IP handovers. On this note, Campbell et al. observe that

> IP micromobility protocols are designed for environments where mobile hosts change their point of attachment to the network so frequently that the base Mobile IP mechanism introduces significant network overhead in terms of increased delay, packet loss, and signaling [33, p.72]

A number of initiatives [7, 72, 155, 51, 37, 67] have been proposed to improve the operation of Mobile IP in environments with a high frequency of micro-mobility. Various approaches exist, but a common aim is to reduce the number of Mobile IP handovers by adding functionality for intra-domain location tracking and routing, for example through local handover schemes between base stations in the foreign network. Campbell et al. give a comparative analysis of a number of influential approaches [33].

Mobile IPv4 constitutes a pure and transparent network-layer solution to the address migration problem discussed in section 2.5.1. The approach does not address the other problems related to physical mobility, location-dependent information and migrating locality (sections 2.5.2 and 2.5.3 respectively), or the problems related to the nature of mobile devices or to mobile networking (sections 2.3 and 2.4). Mobile IPv4 expects such issues to be adressed elsewhere in the protocol stack. In this respect, the scope and boundaries of Mobile IPv4 are extremely well defined; the protocol solves the address migration problem with no 'functionality bleed' into adjacent layers, and applications using Mobile IPv4 remain completely unaware of mobility. On Satyanarayanan's scale of adaptation strategies [159] discussed in section 2.6.3, the approach therefore belongs at the application-transparent end of the strategy spectrum. Speaking generally about application-transparent solutions, Satyanarayanan observes that while they are backward compatible with existing applications, they are not always suitable and 'there may be situations where the adaptation performed by the system is inadequate or even counterproductive' [159]. Awareness is also championed by Noble et al. who support application-aware adaptation as the 'most general and effective approach to mobile information access' [131]. Another source, Angin et al., specifically criticises Mobile IP for 'lack[ing] the intrinsic architectural flexibility to deal with the complexity of supporting adaptive mobile applications in wireless and mobile environments' [9] but offers no further details. In effect, the trade-off is exactly that of transparency vs awareness raised in section 2.6.3. Whether the fully transparent approach adopted by Mobile IPv4 is suitable depends on the application in question. The popularity of Mobile IPv4 suggests that the

approach, if not optimal for all applications, is at least suitable for a considerable range of (possibly legacy) applications.

Perspective

Despite the problems outlined above, Mobile IPv4 has been extremely influential. It is the only network-layer approach to mobility support in the Internet protocol stack that has been adopted at any significant scale, and it has been the subject of a considerable number of suggested improvements and modifications [38, 72, 155, 51, 37, 7, 191, 33]. While it addresses only one of the fourteen problems discussed in chapter 2 (address migration), the layer boundaries are preserved in a manner consistent with traditional protocol design and the remaining problems are left cleanly to be solved by other layers. In terms of future adoption, Mobile IPv4 can be expected to be replaced by Mobile IPv6 as IPv6 gains gradual acceptance. In this respect, possibly the most important contribution of mobile IPv4 has been to influence the design of IPv6. In particular, Mobile IPv4 has highlighted the address migration problem and the consequent need to dynamically rebind IP addresses.

3.1.2 Mobile IPv6

IPv6 [53] is the latest revision of the Internet Protocol and is currently in the process of replacing IPv4 [151]. IPv6 has been under development since the early 1990s. At the time of writing, the protocol is still progressing through the Internet standards process controlled by the Internet Engineering Task Force (IETF). IPv6 has been a 'Draft Standard' since 1998 and still remains to reach the level of full 'Internet Standard.' IPv6 is backed by major industry players, such as Microsoft, Compaq and Cisco, and has been seen by the European Union as 'a prerequisite for the success of 3G [third generation wireless systems]' [117]. In terms of technical features, IPv6 represents a major advancement over IPv4, and the protocol contains many improvements, some of which facilitate host mobility.

Mobile IPv6 extends IPv6 with support for nomadic computing in a similar fashion to the way Mobile IPv4 extends IPv4. The Mobile IPv6 specification [96] became a 'Draft Standard' in June 2002 and should therefore still be considered work in progress. Mobile IPv6 builds closely upon the ideas of Mobile IPv4 but due to the availability of new IPv6 features is superior in a number of respects. Because of the similarity between the two versions of Mobile IP, the description given here focuses on the differences between the two and asumes a level of familiarity with Mobile IPv4 corresponding to the treatment given in section 3.1.1. A comprehensive treatment of the differences between the two versions of Mobile IP can be found in [101, 96].

Agent Discovery

One of the most significant differences between Mobile IPv4 and Mobile IPv6 is the removal of the foreign agent. Perkins writes that 'Mobile IPv6 uses the same basic network entities as Mobile IPv4, except that there is no need for the foreign agent' [143, p.66]. This is possible because of new features included in IPv6. In Mobile IPv4, one of the foreign agent's roles is to advertise available care-of addresses through extended ICMP router advertisements. In Mobile IPv6, mobile hosts can use the 'stateless address autoconfiguration' feature of IPv6 [178] to autonomously create (and test for uniqueness) a care-of address belonging to the foreign network. This allows mobile nodes 'to operate in any location without any special support required from the local router' [96, p.2]. Like Mobile IPv4, Mobile IPv6 also allows stateful autoconfiguration of care-of addresses, meaning that the foreign agent advertises and allocates (and hence retains state about) care-of addresses to mobile hosts. With Mobile IPv6, this is possible for example through the use of DHCPv6 [28].

The process of home agent discovery is largely unchanged between the two versions of Mobile IP. Like the Mobile IPv4 home agent, the Mobile IPv6 home agent uses an extended router advertisement message to advertise its services.

Registration

After having obtained a care-of address, the mobile host sends a registration to its home agent. This creates a binding between the mobile host's home address and care-of address. After binding has taken place, the home agent uses another feature of IPv6, 'proxy neighbor advertisement' [129], to intercept any IPv6 packets addressed to the mobile node's home address. This feature allows a router to 'indicate that it is willing to accept packets not explicitly addressed to itself' [129, p.65].

Routing and Tunneling

Another significant improvement of Mobile IPv6 over Mobile IPv4 is related to triangle routing. Triangle routing resulted from the fact that there is in IPv4 no way for a correspondent node to update its binding to reflect new care-of addresses, once that binding was established. This has been remedied in IPv6 where '[m]obile nodes can inform the correspondent nodes of the current location of the mobile node' [96, p.8]. In effect, a Mobile IPv6 correspondent node maintains two addresses for each endpoint: the home address, which is always known and uniquely identifies the mobile host, and the care-of address, which may be unknown. Traffic to the mobile host is sent to the care-of address, if known. Otherwise, traffic to the mobile host is sent to the home address where it is intercepted by the home agent and tunneled to the mobile host using IPv6 encapsulation of datagrams. The path

taken by such datagrams is equivalent to that of Mobile IPv4 triangle routing shown in figure 3.1. To a mobile host, the arrival of such tunneled datagrams indicate that the correspondent node lacks a binding to the current care-of address. The mobile host can choose to establish such a binding to prevent triangle routing of future datagrams. The designers of Mobile IPv6 expect that 'correspondent nodes usually will route packets directly to the mobile node's care-of address, so that the home agent is rarely involved with packet transmission to the mobile node' [96, p.8].

Firewall Configuration and Ingress Filtering

In Mobile IPv4, using the home address in the source field of datagrams originating from a mobile node visiting a foreign network caused problems when such nodes needed to communicate with correspondent nodes on their home network and when the foreign network's router was configured to perform ingress filtering. Mobile IPv6 addresses these problems by allowing datagrams to contain both the current care-of address (in the datagram's source field) and the home address (in an IPv6 header option). This allows datagrams destined for the home network to pass through the home network's firewall (still subject to filtering policies, but they appear less suspicous) and also allows datagrams to pass normally through routers that perform ingress filtering while still allowing direct delivery of datagrams from the correspondent to the mobile node.

Discussion

Mobile IPv6 addresses several of the problems raised in connection with Mobile IPv4. First, Mobile IPv6 completely solves the firewall configuration and ingress filtering problems to which Mobile IPv4 was subject. Second, the route optimisation features which were an optional add-on feature to Mobile IPv4 have become an integral part of Mobile IPv6 and require no special functionality from the correspondent node beyond what can be expected from any IPv6 node. Johnson et al. observe that route optimisation is 'important for scalability and reliability, and for minimizing overall network load;' that it 'eliminates congestion at the mobile node's home agent and home link;' and also that it increases fault tolerance because mobile and correspondent nodes are less dependent on the availability of the home agent and network [96, p.8].

In terms of macro- vs micro-mobility, Mobile IPv6 adopts the same position as Mobile IPv4: it facilitates inter-domain mobility and expects intra-domain mobility to be handled by the underlying link layer. Although less work has been done in terms of micro-mobility schemes for Mobile IPv6 than for Mobile IPv4, the issue is receiving attention from the Mobile IP working group within the Internet Engineering Task Force (IETF). For example, a recently produced Internet Draft [172] describes a

hierarchical mobility management scheme that aims to improve performance by reducing the amount of signalling between the parties.

Mobile IPv6 is designed in the same spirit as Mobile IPv4. It solves the address migration problem discussed in section 2.5.1 in a completely transparent manner. Other problems related to mobile environments are deliberately not addressed. Mobile IPv6 is therefore subject to the same critiques as Mobile IPv4 in relation to the trade-off between transparency and awareness.

Perspective

In conjunction with IPv6, Mobile IPv6 constitutes a very mature approach to network-layer mobility support in the Internet protocol stack. It builds upon the experience with Mobile IPv4 and solves the problems raised by Mobile IPv4 deployment in an elegant and consistent manner. It seems safe to assume the experience with Mobile IPv4 has also influenced the design of IPv6. Like its predecessor, Mobile IPv6 solves a well-defined problem (address migration) and suffers no 'functionality bleed' into adjacent layers. The most important open question is to what extent the fully transparent approach proves suitable for building mobile applications. As discussed in section 3.1.1, a number of sources argue that this approach has potential drawbacks, most notably in the form of inadequate adaptation capabilities and insufficient architectural flexibility to properly support adaptive mobile applications [159, 131, 9, 34].

3.1.3 MSM-IP

Mobile IP has remained the dominant approach to network-layer mobility support in the Internet protocol stack, certainly in terms of adoption, but also as a topic for investigation by the research community. While considerable research efforts have focused on improving or extending Mobile IP [7, 72, 155, 51, 37, 67, 168, 38], a number of alternative approaches to network-layer mobility have also been put forward. Some of these share characteristics with Mobile IP, while others differ significantly. This section presents a brief overview of one such approach proposed by Mysore et al. from the University of Illinois at Urbana-Champaign (UIUC).

Description

The proposed system is based on the observation that 'the fundamental issues faced by any mobile host are a sub-set of the issues faced by any network level multicasting approach' [128, p.162]. Their scheme, called Mobility Support using Multicasting in IP (MSM-IP), uses multicasting 'as the sole mechanism to provide addressing and routing services for mobile hosts' [128, p.161]. In MSM-IP,

a unique multicast address is assigned to each mobile host. All packets from a correspondent host to a mobile host are treated as multicast packets. All packets in the other direction are treated as standard unicast packets. As a consequence, the multicast routers serve many of the functions that in Mobile IPv4 are performed by the foreign agent, e.g., registration and connectivity in foreign networks. There is no concept of home network or home agent, and all data is routed to and from the mobile host via the multicast address. The overlay network formed by the multicast routers provide the level of indirection that Mobile IP provides via the home agent. The authors observe that the choice of multicast routing protocol is significant. The multicast groups used by MSM-IP will be small, consisting (except when using advance registration as explained below) of just one node. For this reason, sparse mode algorithms are expected to work better than dense mode algorithms.

The designers of MSM-IP claim handoffs to be efficient. When the mobile host can anticipate the handoff, it can perform advance registration with the new multicast router and thereby reduce packet loss during handoff. Even when this is not possible, the old and the new multicast routers are expected to be neighbours for the majority of handoffs, and the join and prune operations (for the new and old routers respectively) of the multicast overlay network can therefore be expected to be relatively inexpensive.

Discussion

In terms of implementation, a significant problem with MSM-IP is that a number of protocols used in conjunction with IP (e.g., ARP, RARP, ICMP, TCP, etc.) are not designed to accept multicast addresses as equivalent to IP addresses. For some of these protocols (e.g., TCP), kernel modifications on the mobile host are required. For others (e.g., ARP and ICMP), the mobile host must be configured to allocate an IP unicast address on arrival to a foreign network (e.g., via DHCP). This temporary unicast address is then used to perform network management functions.

Perspective

Like Mobile IP, MSM-IP solves the address migration problem discussed in section 2.5.1. Because advance registration can be performed before anticipated handoff, the system can also be said to address those short term disconnections (section 2.4.2) that are related to handoff. An advantage of MSM-IP is that the use of multicast allows mobility support with 'minimal or no changes to the backbone networks' [128, p.161–162]. Although MSM-IP constitutes an interesting alternative to Mobile IP, the scheme suffers from a number of problems that would impede its actual adoption. In addition to the problems concerning IP-related protocols mentioned above, MSM-IP requires all networks involved

in communication with mobile hosts to contain a multicast-capable router. Also, multicast addresses have been called 'an even more limited resource than unicast addresses' [5], and MSM-IP requires one such address for every mobile host. The authors acknowledge such impediments with the observation that the 'current [1997] IP multicasting infrastructure' is not 'sufficient to adequately support mobility in the Internet' [128, p.162].

3.1.4 Summary

This section has reviewed the state of the art in network-layer mobility support by discussing in detail the two versions of Mobile IP and presenting a brief overview of MSM-IP. In addition, a number of other approaches deserve a mention but will not be discussed in detail. A scheme proposed by Columbia University [92] supports mobility within a wireless campus environment through the use of mobility support routers (essentially gateways as discussed in section 2.2.2) that act as proxies and cooperate to keep track of mobile hosts. A scheme proposed by Sony [177] uses a dual-address and home agent approach similar to Mobile IP, but also extends the IP datagram format to include both destination addresses and introduces new functionality to cache address mappings in routers. A scheme proposed by Carnegie Mellon University [94] uses a home/foreign agent approach in conjunction with an existing IPv4 feature called Loose Source Routing (LSR) to route packets to the mobile host explicitly via the foreign agent. The Daedalus system proposed by the University of California Berkeley [167] uses an approach similar to Mobile IP but where the home agent multicasts packets to the mobile host to a cluster of foreign agents in the mobile host's vicinity in order to reduce packet loss during handoff.

Table 3.1 summarises the applicability of the two Mobile IP variations and the MSM-IP scheme discussed in this section to the fourteen challenges discussed in chapter 2. A bullet (•) indicates good support for dealing with the challenge in question, a circle (○) indicates partial support and a blank indicates no support. The table shows that the network-layer solutions reviewed aim mainly to solve the address migration problem discussed in section 2.5.1 and address few of the other problems related to mobile environments. On this basis, it seems safe to conclude that while comprehensive support for mobile applications could certainly include network-layer functionality, current approaches are not in themselves sufficient to deal with the variety of challenges faced by applications operating in mobile environments. Network-layer solutions could serve as a starting point but would have to be supplemented with extra functionality to address the remaining problems.

General Category	Specific Challenge	Mobile IPv4	Mobile IPv6	MSM-IP
Mobile Devices (section 2.3)	Battery Power Data Risks User Interface Storage Capacity Processing Power			
Mobile Networking (section 2.4)	Network Heterogeneity Disconnection Low Bandwidth Bandwidth Variability Security Risks Usage Cost			○
Physical Mobility (section 2.5)	Address Migration Location-Dependent Information Migrating Locality	•	•	•

Table 3.1: Summary of Network-Layer Approaches to Mobility Support

3.2 Transport-Layer Mobility Support

Section 3.1 presented a number of approaches to mobility support at the network-layer level, i.e., by modifying the IP layer in the Internet protocol stack. Common for the systems discussed was that they focused on solving the address migration problem and did not directly address any of the other challenges of the mobile environment. Another body of work [171, 103, 114] attempts to solve the problems related to the mobile environment at the transport layer, i.e., by modifying transport protocols such as TCP and UDP. This section reviews three such approaches and evaluates them in the context of the challenges described in chapter 2. We conclude that transport-layer solutions generally address a different set of mobility challenges than the network-layer solutions described in section 3.1. Where network-layer solutions focus on solving the address migration problem, transport-layer solutions address a number of the problems related to mobile networking.

3.2.1 Snoeren

Snoeren et al. [171] from the Massachussetts Institute of Technology (MIT) propose a system that uses the Domain Name System (DNS) to track host location and a TCP extension to allow open connections to be migrated to new endpoints as mobile hosts obtain new IP addresses. The approach requires no changes to the IP substrate but does require modification of transport protocols and applications at both of the corresponding hosts.

Description and Operation

Where Mobile IP uses the home address as an invariant host identifier and transient care-of addresses for routing, the Snoeren scheme uses the Internet hostname for identification and the IP address for routing. This use of the two data entities is perhaps more natural than that of Mobile IP. When a mobile host moves to a new connection point, it obtains a new IP address (e.g., via DHCP) and performs a secure DNS update, binding its hostname to the new IP address. After the update has completed, future resolutions of the mobile host's name will return the new address. DNS entries for mobile hosts have a Time-To-Live (TTL) value of zero, making the entries uncacheable. The authors observe that zero-value TTLs have less impact on scalability than one could be led to believe, because correspondent nodes performing lookups will cause the name server record for the mobile host's name to be cached by the correspondent node's own name server. When the correspondent node initially resolves the mobile host's name, the DNS tree will be traversed from the root until the name server holding the authoritative A-record for the mobile host has been found. After the initial resolution, the correspondent node's name server will retain a cached reference to the mobile host's name server, even though the A-record for the mobile host is uncacheable. Future resolutions of the mobile host's name will therefore start directly at mobile host's name server, avoiding traversal of the full DNS tree.

The extensions to TCP take the form of a series of options included in the TCP SYN segments. During connection setup, the mobile host uses a 'migrate-permitted' option to negotiate a token with the correspondent node. This token identifies the mobile host's open connections independently of its IP address. When the mobile host later acquires a new IP address, it sends a 'migrate' SYN packet containing the token to the correspondent node. The correspondent node's (modified) TCP implementation can then update its records of the open connections, such that the endpoints refer to the mobile host's new IP address instead of the old. The authors present an implementation of their scheme in the form of a modification to the IPv4 TCP implementation in the Linux kernel. The implementation also supports advance migration of connections, if the mobile host's new IP address is known in advance.

Discussion

The proposed scheme constitutes 'an end-to-end architecture for Internet host mobility' [171, p.155] and is therefore designed to address those challenges of the mobile environment that are related to physical mobility (section 2.5) rather than those related to mobile devices or mobile networking (sections 2.3 and 2.4 respectively). Like Mobile IP, the system focuses on the address migration problem (section 2.5.1) and does not address the problems of location-dependent information or migrating

locality (sections 2.5.2 and 2.5.3 respectively). Unlike Mobile IP, the system does not suffer from triangle routing or impediments due to firewall configuration or ingress filtering, but it does have other limitations. For example, it is not possible for two corresponding parties to move simultaneously and unexpectedly, because connection migration requires each of them to send a 'migrate' SYN packet to the other. If both change address simultaneously, both SYN packets will be sent to the old addresses. Mobile IP does not suffer from this problem, because the home agent serves as an anchor point whose address does not change. Another problem with the Snoeren approach is related to application-level caching of DNS resolutions; some applications keep copies of values returned by DNS resolutions (e.g., via the `gethostbyname()` function) instead of re-resolving them before each use. In addition, disconnections (section 2.4.2) are poorly supported; if the mobile host fails to migrate its open connections within the time frame allowed before TCP times out, these connections will be lost.

Perspective

Focusing exclusively on the address migration problem, the scheme can be seen as a transport-layer alternative to Mobile IP. The use of non-cacheable DNS entries to maintain the level of indirection required for location management is a simple and elegant solution that seems to fit naturally with the existing Internet protocols and services. The main drawback is the requirement that implementations of transport protocols be updated on all parties involved in communication with mobile computers. This requirement is a serious impediment to adoption beyond research environments and can explain why the approach has not become a serious competitor to Mobile IP in IETF efforts to deal with the address migration problem.

3.2.2 MOWGLI

Mobile Office Workstation using GSM Link (MOWGLI) [103, 104, 105] was a joint project between the University of Helsinki, Digital Equipment Corporation, Nokia and Telecom Finland. The project included the design and implementation of an architecture that allows mobile hosts (e.g., laptops) to connect to the Internet through a GSM network. The MOWGLI architecture is based on the gateway model discussed in section 2.2.2. The gateways are called Mobile-Connection Hosts (MCHs) and the mobile computers are called Mobile Nodes (MNs). The architecture is designed to be used in conjunction with Mobile IP [138, 140].

Description

The MOWGLI architecture 'replace[s] the client-server paradigm with a client-mediator-server paradigm' [105, p.1337]. TCP connections between mobile clients and remote (fixed) servers are *split* at the MCH, which acts as a mediator during communication between mobile clients and fixed servers.[1] The splitting of the connection allows the use of different protocols between the MN and MCH and between the MCH and the fixed host. This is an advantage, because standard TCP implementations have been shown to perform poorly over wireless links [30]. MOWGLI uses a special-purpose protocol stack on the MN-MCH link and conventional TCP/IP between the MCH and the fixed host. (The architecture presented in chapter 4 of this thesis follows a similar approach.) The MOWGLI protocol stack used on the wireless link performs a number of functions. First, it shields mobile applications from disruption in GSM connectivity by performing transparent reconnections. Second, it can be configured to shut down the GSM link in case connections are idle and reestablish the link when data is transmitted. This is done transparently to applications and can reduce the cost associated with GSM connectivity.

MOWGLI provides 'a socket interface, which is similar to and highly compatible with the Berkeley sockets ... as well as the Windows sockets' [103, pp.2–3]. MOWGLI sockets effectively emulate BSD/Windows sockets and transparently let applications on the MN use the TCP/IP stack of the MCH. The MOWGLI architecture also includes a layer of indirection above the extended socket functionality that allows sockets owned by mobile applications to be rebound at runtime. For example, a MN that connects to a LAN may wish to exchange its MOWGLI sockets (which are optimised for operation over the GSM link) for ordinary BSD/Windows sockets. Like standard sockets, MOWGLI sockets provide TCP as well as UDP interfaces. They are implemented as a kernel extension on Linux and a dynamic link library (DLL) on Windows.

While the MOWGLI architecture allows mobile applications to remain largely unaware of mobility, the architecture 'also provides a possibility to implement enhanced functionality needed by users who are aware of the impacts of mobility and wireless communication' [103, p.3]. Mobile clients can be divided in two portions where one resides on the MN and the other on the MCH. This *compound client* structure allows the MCH to perform actions on behalf of the MN. For a complex application, such as a distributed transaction processing client, the majority of the functionality can be placed on the MCH and controlled from the MN. This can reduce the amount of data transmission required over the GSM link. In some cases the GSM link can be taken down during the operation, which can reduce cost.

[1] The idea of TCP connection splitting was first introduced by Bakre et al. with the introduction of indirect TCP (I-TCP) [16].

The MOWGLI protocol stack consists of three main protocols. At the top is the MOWGLI Socket Protocol (MSP) which allows the remote management of sockets between the MN and MCH. In the middle is the MOWGLI Data Channel Protocol (MDCP) which allows the multiplexing of TCP connections and UDP datagrams over a single underlying connection. At the bottom is the MOWGLI Wireless Link Protocol (MWLP), a modified version of the Point to Point Protocol (PPP), suitable for data transfer over GSM links.

Operation

Whenever TCP/IP connectivity is required, a MN places a GSM data call to a MCH. The MN can choose one of several MCHs, but a closer MCH will typically incur a lower cost, especially if movement occurs across national boundaries. When the MN connects to a MCH, a proxy is created on the MCH to act as mediator for the MN during the period of the connection. The MCH also creates a new virtual network interface and assigns the MN's IP address to that interface. The MCH also acts as a Mobile IP foreign agent and registers the MN's new location with the corresponding home agent. This allows IP datagrams destined for the MN to arrive at the MCH via the home agent. Outgoing datagrams from the MN are delivered using ordinary IP routing.

When a MOWGLI socket is created on the MN, the proxy on the MCH creates a standard (TCP or UDP) socket. Data sent through the MOWGLI socket is transferred via the MOWGLI protocol stack over the wireless link to the MCH where it is relayed by the proxy over the conventional socket to the fixed host. In case the connection is lost between the MN and MCH, the MN transparently reconnects to the MCH and resumes the transfer of data. The MOWGLI architecture does not support handoff between MCHs or tunneling of open connections, so if any open TCP connections remain, the MN must reconnect to the same MCH. Otherwise, another MCH can be chosen. The MOWGLI literature [103, 104, 105] does not discuss how long connections between the MCH and fixed hosts are retained after a connection is lost between the MN and the MCH.

Discussion

The MOWGLI project specifically targeted the laptop computer as its hardware platform. As shown in section 2.3, laptop computers are relatively powerful compared to smaller devices such as PDAs. For this reason, none of the five categories of device limitations discussed in section 2.3 were considerations in the design of the MOWGLI architecture and are not directly addressed in the literature [103, 104, 105]. However, the MOWGLI architecture is implemented in the C programming language which generally incurs relatively little runtime overhead (e.g., significantly less than Java and somewhat less

than C++) in terms of processing requirements and memory required for execution, and it therefore seems reasonable to expect the architecture is relatively lightweight in these two respects. Data risks were also not of particular concern for the designers of MOWGLI, although it could be argued that the architecture (like any architecture facilitating distributed applications) can be used to reduce risks of data loss by making it easier to store data remotely.

In relation to mobile networking, the MOWGLI architecture addresses five of the six problems discussed in section 2.4 to various extents. Although the architecture is tailored to the use of GSM as a transport, the transparent rebinding of MOWGLI sockets to standard sockets allows applications to use a LAN and thereby allows some degree of network heterogeneity. Better support for heterogeneous network environments could be added in the form of support for other wireless technologies, such as Bluetooth and InfraRed (IR). The architecture successfully shields mobile applications from short-term disconnections occuring as a result of dropped GSM connections and also includes some support for operation during long-term disconnections through the use of compound clients. However, the literature contains no detailed descriptions of such applications, and it is therefore uncertain how well this approach will work in practice. The issue of low bandwidth is addressed by replacing TCP with a protocol specialised for low-bandwidth wireless links and through the optional use of compound clients which can further reduce the amount of data transmitted. The MOWGLI protocol stack does not compress data transmitted over the link but leaves this issue to the GSM or application layer. (Most GSM modems conform to the V.42bis standard which includes support for data compression.) Small variations in bandwidth are addressed through the architecture's ability to deal with pauses in data transmission as a result of change in transmission/reception conditions. Large variations in bandwidth are addressed through the socket rebinding mechanism. Security is not a high priority of the MOWGLI architecture, and the protocol stack does not encrypt data sent over the wireless link. However, such functionality could conceivably be implemented with relative ease at the application-level using a compound client. The architecture addresses the issue of connectivity cost by allowing connections to MCHs in the vicinity and through the support of transparent disconnection and reconnection of the GSM link.

Regarding problems related to physical mobility, the MOWGLI architecture addresses the three problems discussed in section 2.5 to various extents. The address migration problem is solved through the use of Mobile IP [138, 140] in conjunction with the dynamic assignment of virtual network interfaces on the MCH. This approach allows a fixed client in possession of a MN's (home agent) address and a port number to send datagrams to a particular process on the MN whenever the MN is connected to a MCH. This constitutes support for server mobility. However, MOWGLI does not implement

handoff between MCHs, meaning that state pertaining to open TCP connections cannot be transferred between MCHs and no inter-MCH tunneling is possible. As a consequence, a MN with open TCP connections will have to reconnect to the same MCH in case a GSM connection is lost or voluntarily closed. During movement with an open GSM connection, handoff between GSM base stations takes place at a lower level in the protocol stack, but this is transparent to upper layers, except for short delays in communication. Using GSM as the only transport, the lack of MCH-level handoff is not a problem because it is always possible to reconnect to the last MCH. However, given transports where no underlying handoff exists (e.g., Bluetooth, IR or WLAN), this approach will not work. Regarding location-dependent information, such as name servers, printers, etc., MOWGLI MNs could conceivably be configured to use 'local' resources via the MN's TCP/IP stack. Since access to such resources (e.g., name server queries) would in reality go through the stack on the MCH, they would effectively give access to services on the MCH, i.e., local services. However, this approach does not solve the migrating locality problem, because no MCH-level handoff exists.

Perspective

The MOWGLI architecture solves a number of problems related to nomadic computing using laptop computers. In particular, very convincing solutions are presented to the problems related to mobile networking. The address migration problem is solved fairly elegantly, although the use of Mobile IP raises other problems as discussed in section 3.1.1. A number of additional open issues pertain to the MOWGLI architecture. First, the only transport supported between MCHs and MNs is GSM. As demonstrated in section 2.4, current mobile devices have a considerable number of options for wireless communication of which GSM is only one. The lack of MCH-level handoff leaves open the question of the extent to which the MOWGLI architecture will support other wireless transports. Second, the idea of compound applications seems to require further investigation. In particular, it remains unclear how the partitioned client functionality can be distributed, i.e., whether client components can be installed automatically on the MCH at runtime or whether static preconfiguration is required.

3.2.3 MSOCKS

Maltz et al. [114] propose an architecture called MSOCKS also based on the gateway model discussed in section 2.2.2. In MSOCKS, mobile hosts interact with proxies hosted on gateways which perform TCP connection splitting in a similar manner to the MOWGLI architecture discussed in section 3.2.2. As opposed to MOWGLI, which assumes GSM as the only transport between the mobile host and the gateway, an important consideration in relation to the MSOCKS design was the observation that

'[c]urrent mobile nodes can choose between many types of wireless network interfaces, each with wildly different bandwidth, error-rate, cost and latency characteristics' and subsequently that '[a]pplications need to be able to specify the network interfaces over which each data stream should be sent and received' [114, p.1037]. The proposed solution provides functionality comparable to that of MOWGLI in that it allows mobile hosts to change their point of attachment to the Internet but also includes better support for multiple interfaces and the ability for applications to decide which interfaces to use for the different kinds of data leaving from and arriving at the mobile host.

Description and Operation

MSOCKS is designed for use with mobile computers (called mobile nodes) which have different modes of connectivity, e.g., RF-based, IR-based and wired network interfaces. Gateways (called proxies) are equipped with corresponding interfaces and also with a permanent Internet connection. When an application on a mobile node creates a connection to a remote host, a connection is first created to the proxy over one of the mobile node's wired or wireless interfaces. (A mobile node may prefer a wired connection, if one is available.) The proxy then creates a second connection to the remote host (completing the connection setup) and begins forwarding data in both directions. If the mobile-proxy part of the connection breaks, it can be restored without disrupting end-to-end connectivity between the two parties. In addition, the mobile application can choose to voluntarily close down a mobile-proxy connection and reestablish it over a different interface. This allows mobile applications to adapt to changes in connectivity, for example by switching from a wireless to a wired interface or vice versa without dropping open connections. Maltz et al. call the process of re-associating an existing proxy-remote connection with a new mobile-proxy connection 'resplicing' TCP connections.

The implementation of MSOCKS consists of three components: a user-level proxy process running on the gateway, a modification to the proxy node's kernel that implements TCP connection splicing, and a dynamic link library (DLL) on the mobile host which replaces the standard sockets library. No modifications are required to the remote party, and the mobile application can remain unmodified unless it wishes to avail of the mobility-awareness features. The MSOCKS implementation builds on the SOCKS [109] protocol for firewall traversal. MSOCKS adds functionality to identify end-to-end TCP connections despite changes in the mobile host's IP address and to resplice a new transport connection between the mobile host and the proxy.

To an application on the mobile node, the MSOCKS library provides replacements for the ordinary BSD sockets functions, e.g., `send()`, `recv()` and `setsockopt()`. The replacement functions constitute the 'Msockets' API and while their signatures are identical to those of BSD sockets, the

Msockets functions interact with the proxy instead of directly with the remote server. To a client application using Msockets instead of BSD sockets, the change in functionality is transparent; all the semantics of TCP are preserved. For mobile servers, the implementation of MSOCKS supports incoming connections to the proxy which can be relayed to the mobile host with the restriction that only one incoming connection per server port is allowed. On this note, the authors write '[s]ince most mobile nodes will be acting as clients, this is not a significant limitation' [114, p.1040]. An algorithm for extending the system to support full server socket semantics is described but not implemented.

MSOCKS manages data flows between the mobile host and the proxy over a series of communications interfaces through a combination of two features. First, individual IP addresses are allocated to the different interfaces on the mobile host. This allows the proxy to associate a data stream destined for the mobile host with a particular interface by routing it to that interface's IP address. Second, the proxy is also configured with a number of different IP addresses. This allows the mobile host to set up 'host routes' for the individual flows, specifying that data destined for a particular proxy IP address should always be sent over one particular (local) interface. When establishing a new connection via the proxy, the mobile host can choose the proxy IP address that corresponds to the interface it wants to use for that data stream.

Discussion

Like the MOWGLI architecture, MSOCKS does not specifically address issues related to the nature of mobile devices, and none of the five challenges discussed in section 2.3 can be said to be addressed by the architecture. The main focus of MSOCKS is to address the issues related to mobile networking, and the proposed architecture contains solutions for two of the six challenges discussed in section 2.4 and possible solutions to two more. First, the architecture directly addresses the network heterogeneity problem (section 2.4.1) by allowing mobile applications to adapt their behaviour depending on the available transports. Second, the system can automatically reconnect to the proxy in case the connection between the mobile host and the proxy is lost and therefore also addresses short-term disconnections (section 2.4.2). The uncertainty about the other two challenges is due to the fact that the description of the MSOCKS architecture found in the literature [114] is somewhat vague about how mobility-awareness can be used in actual applications. In particular, it is unclear exactly what information about the transport characteristics is available to mobility-aware applications that wish to adapt. While it seems safe to assume that applications can obtain information about currently active interfaces and their general characteristics (e.g., typical bandwidth) it is unclear whether the MSOCKS implementation maintains and lets applications monitor performance statistics for individ-

ual interfaces. While we will assume this feature is not included in the current version of MSOCKS, it would conceivably be relatively straightforward to introduce and would allow MSOCKS to address the problem of bandwidth variability (section 2.4.4). In a similar vein, the authors mention cost as one of the characteristics that can vary significantly across interfaces [114, p.1037] but leave it uncertain to what extent the architecture allows this parameter to be taken into account by adaptive mobile applications. If the cost parameter is available to mobility-aware applications, the architecture can be said also to address the usage cost problem (section 2.4.6).

In relation to the physical mobility challenges discussed in section 2.5, MSOCKS offers a partial solution to the address migration problem (section 2.5.1) and does not address the other challenges. MSOCKS deals with the address migration problem by fixing the proxy endpoint of the proxy-remote part of the connection at the network layer. In this way, host mobility is hidden from the remote party through the use of the proxy's static IP address. While this approach allows end-to-end connectivity to be maintained between the mobile and the remote party, it does not cater well for all situations. For example, the architecture features no inter-proxy handoff or tunneling of open connections. As a result, mobile clients with open connections must connect to the same proxy in order to resume these connections. This can be problematic if the mobile host is no longer within range of the proxy. Because there is no inter-proxy handoff or tunneling, being within range of another proxy does not remedy the situation. Another consequence of the proposed approach is poor support for server mobility. As discussed, mobile server support in MSOCKS was not a high priority and for that reason is somewhat rudimentary in the current version. However, even if remedied along the lines suggested by the authors, MSOCKS has no features that can help clients locate servers that have moved.

Perspective

The MSOCKS architecture solves a significant portion of the problems related to mobile networking. In particular, convincing solutions are presented to the problems of network heterogeneity and disconnection, and it is clear that the architecture could also address the problems of bandwidth variability and usage cost, even if it does not already do so. In this respect the MSOCKS approach is superior to that of MOWGLI. The opposite is true in relation to the two architectures' approaches to the address migration problem. In this respect, MOWGLI's use of Mobile IP provides better support for server mobility than MSOCKS. While the authors of MSOCKS assumed mobile hosts will contain mostly clients, this assmption will not necessarily hold for distributed object applications, as discussed in section 2.5.1.

A particularly interesting feature of MSOCKS is its support for adaptive application behaviour. In

this respect, MSOCKS makes a different trade-off between transparency and awareness (section 2.6.3) than the network-layer solutions discussed in section 3.1. Whereas the network-layer solutions aimed for complete transparency, MSOCKS falls somewhere between the extremes on Satyanarayanan's scale of adaptation strategies discussed in section 2.6.3. Maltz et al. describe the ability for applications to 'specify which interfaces are used for each type of traffic' as a 'qualitatively different kind of control over their sessions' [114, p.1045] and attribute the possibility of supporting this type of adaptation to the fact that their scheme is implemented at the transport layer.

3.2.4 Summary

This section has reviewed the state of the art in transport-layer mobility support by discussing in detail three different proposals and reviewing the support offered in the context of the fourteen mobility challenges identified in chapter 2. The discussion is summarised in table 3.2 which uses the same notation as table 3.1. A comparison of the two tables shows that while the network-layer solutions focus almost exclusively on the address migration problem, the transport-layer solutions discussed in this section generally also address the challenges related to mobile networking. Most of the transport-layer approaches reviewed here include less comprehensive solutions to the address migration problem than the network-layer approaches. In particular, MSOCKS only offers a partial solution and the MOWGLI architecture solves this problem through the use of Mobile IPv4, a network-layer solution.

General Category	Specific Challenge	Snoeren	MOWGLI	MSOCKS
Mobile Devices (section 2.3)	Battery Power			
	Data Risks			
	User Interface			
	Storage Capacity			
	Processing Power			
Mobile Networking (section 2.4)	Network Heterogeneity		○	●
	Disconnection		○	○
	Low Bandwidth		●	
	Bandwidth Variability		○	○
	Security Risks			
	Usage Cost		○	○
Physical Mobility (section 2.5)	Address Migration	●	●	○
	Location-Dependent Information		○	
	Migrating Locality			

Table 3.2: Summary of Transport-Layer Approaches to Mobility Support

3.3 Middleware for Mobile Environments

The previous two sections have reviewed a number of approaches to mobility support at the network and transport layers respectively. The OSI protocol stack precedes the term 'middleware,' and does therefore not specify a middleware layer, but based on the discussion in section 1.2.3, we assume middleware to reside 'in a layer above the operating system and networking software and below industry-specific applications' [22, p.88], i.e., above the transport layer.

Section 1.1 argued that previous research into middleware providing mobility support has taken one of two approaches: the development of new middleware architectures or the adaptation of existing middleware architectures for use in mobile environments. This section reviews a number of systems belonging to the former category. A number of systems belonging to the latter category are discussed in sections 3.4 and 3.5. New types of middleware for mobile environments are generally based on adaptation of existing *concepts* rather than adaptation of existing *architectures* and propose changes to the semantics of existing middleware concepts to suit mobile environments. Examples of such changes are the idea of *relaxing* consistency in file systems (e.g., Coda [161, 158]) and of *relocating* objects and *queuing* remote invocations (e.g., Rover [97, 98]).[2] Examples of other concepts whose semantics have been adapted for use in mobile environments would be tuple spaces [126, 49] and events [174].

Two systems are reviewed in this section: Coda and Rover. While both constitute middleware, only Rover is *object-oriented* middleware. We have chosen also to discuss Coda for two reasons. First, Coda has been highly influential through its pioneering of support for disconnected operation in mobile environments. Although Coda is a file system, the approach taken to address the disconnection problem (section 2.4.2) is highly relevant also for object-oriented middleware. Second, Coda and Rover both constitute approaches to middleware based on the modification of semantics of existing middleware concepts. The treatment of two such systems (rather than just one) allows us to compare better this approach with approaches that attempt to retain semantics of (and therefore, compatibility with) existing middleware, such as the systems discussed in sections 3.4 and 3.5.

3.3.1 Coda

Perhaps the most influential system in the area of support for mobile applications is the Coda file system developed by Satyanarayanan et al. [161] at Carnegie Mellon University (CMU). Coda is a distributed file system whose explicit goal is to 'offer clients continued access to data in the face of

[2] Recall that distributed file management and remote procedure call have both been classified as middleware [102, 22], as discussed in section 1.2.3.

server and network failures' [158, p.27]. Even though Coda was designed before the term 'middleware' became commonly used, it addresses the two key problems that middleware (cf. section 1.2.3) generally attempts to solve: *distribution* and *heterogeneity*. Coda, like other distributed file systems, such as Sun Microsystems' Network File System (NFS) [133] and Coda's predecessor, the Andrew File System (AFS) [90], can therefore be categorised as a type of middleware that offers file management services. Compared to NFS and AFS, Coda achieves significantly improved operation in mobile environments by slightly changing the conventional file system semantics. Hence, while Coda does constitute an adaptation of an existing middleware framework (AFS), it also constitutes an adaptation of an existing middleware *concept*, namely the distributed file system abstraction and its associated operations.

Description

Coda operates with mobile hosts (e.g., laptop computers), called 'clients,' that are connected to fixed networks at various points in time, either via wired or wireless links. The Coda file name space is divided into shared 'volumes' each of which is maintained by a file server. On each client runs a cache manager called 'Venus' that manages data cached from the different volumes and information about volume mappings. A key feature of Coda is that it allows data cached on the client to be modified during periods while the client is disconnected from the fixed network and therefore without access to the volumes where the authoritative copies of the data are stored. While this improves the range of file operations that clients can perform during disconnections, it also introduces the possibility for inconsistencies. Coda contains facilities for detecting and resolving such conflicts with a high degree of transparency.

Operation

Venus manages access to the entire Coda name space on the client, when the client is connected as well as disconnected. Venus operates in one of three states depending on the client's state of connectivity: *hoarding, emulating* and *reintegrating*. When the client is connected, Venus is in *hoarding* state. In this state, file operations on the Coda name space are relayed to the remote volume servers. Therefore, the entire name space is available and updates take place immediately. The hoarding state is so named because in this state Venus, in addition to forwarding requests, also *hoards* copies of critical files on the client such that these files will be cached (and hence available) in case the connection to the fixed network is lost. Coda uses command scripts called 'hoard profiles' to allow the user to customise the operation of Venus in hoarding state. By configuring a set of profiles, users give Venus hints about the files in which they are most interested.

When connection is lost, Venus enters *emulating* state. In this state, Venus emulates the volume servers' functions by serving requests from its cache. While disconnected, only the previously hoarded files are available and updates are made only to the cached copies. In case of a cache miss, a fatal error is returned to the application. When the client's connection to the fixed network is restored, Venus enters *reintegration* state. In this state, Venus reintegrates its cache with the volume servers. Any conflicting updates are detected and an attempt to resolve them made. Coda takes an 'optimistic' approach to replication management, meaning that possibly conflicting updates are allowed under the (optimistic) assumption that any conflicts can be detected and resolved during the reintegration stage. (The authors generally reflect positively upon this assumption, and after two years of using Coda in a mobile computing environment with three servers, 15 desktop computers and 25 laptop computers conclude that the decision to use optimistic replica control was 'the correct one for our type of environment' [162, p.20].) Because Coda is a Unix-style file system, it treats files as byte streams and has no knowledge of the internal structure of data stored in files. This prevents it from resolving intra-file conflicts. In such cases, Coda allows applications to provide their own Application-Specific Resolvers (ASRs) to transparently resolve conflicts. If an ASR fails to resolve a conflict, the inconsistency is presented to the user who can then repair it manually.

Optimisations

In addition to the basic operation of Coda described above, a number of extensions have been developed to deal with 'intermittent, low-bandwidth or expensive networks' [158, pp.29–30]. One optimisation reduces the number of validations that Venus has to perform when it enters reintegration mode. In the original Coda design, a version stamp was maintained for each file. Reintegration involved validating all cached file objects—a potentially slow process over low bandwidth links. By introducing per-volume version stamps, reintegration of unmodified volumes can be performed with less network traffic. Another optimisation for low-bandwidth connectivity, called 'trickle reintegration,' introduces an additional state to Venus called the 'write-disconnected' state. In this state, file operations are served from the cache, and updates transmitted to the volume servers in batches. This allows updates to be performed efficiently (on locally cached files) and updates to 'trickle' back to the relevant volume server. The usual conflict detection and resolution mechanism still applies. A third optimisation introduces the option of user involvement in the case of cache misses. A user can configure a profile that instructs Venus to fetch a file as a result of a cache miss instead of reporting a failure. The model is based on a 'patience threshold.' If a cache miss occurs, the estimated time to fetch the file is computed as a function of the available bandwidth and the estimated file size. If the value is less than

the configured threshold, the file is fetched. If not, a failure condition is returned to the application.

Discussion

As discussed above, Coda implements a file system abstraction very similar to a traditional Unix-style networked file system. However, two main changes have been made to file operation semantics in order to make Coda suitable for operation in mobile environments. First, files may be unavailable while the client is disconnected, and situations can arise where Venus is unable to hide a cache miss. To the naïve client, it will seem as if files appear and disappear at various times. Second, the explicit involvement of the user is required to resolve some types of conflicts. Although Coda aims to make both situations rare occurrencies (using such means as hoarding profiles and ASRs), both situations constitute changes in semantics compared to the traditional Unix-style networked file system.

Support for dealing with long-term disconnections (section 2.4.2) is very strong in Coda. As discussed in section 2.6.5, this has a number of advantages, perhaps most importantly the ability of the client to operate when no network is available. In addition, the ability to operate without network connectivity can be employed voluntarily to reduce wireless communication and thereby increase battery life and/or reduce expenses related to mobile communications. For this reason, Coda's support for disconnected operation also constitutes a partial solution to improving the battery life (section 2.3.1) of mobile devices and reducing the usage cost (section 2.4.6) of mobile networks. The three opimisations—per-volume version stamp, trickle reintegration and user-configurable cache miss handling—constitute good support for low bandwidth connectivity (section 2.4.3) and bandwidth variability (section 2.4.4).

Perspective

Coda was one of the first efforts to propose and demonstrate the concept of 'disconnected operation' [158, p.28] and the system has been highly influential in the area of software support for mobile computing. While Coda is not concerned with problems related to physical mobility (section 2.5) and only indirectly addresses one of the problems related to mobile devices (section 2.3), it does offer convincing solutions to a considerable portion of the problems related to mobile networking (section 2.4).

A particularly interesting feature of Coda is its high degree of configurability and adaptability. While Coda generally attempts to hide its modified semantics from applications, it also deliberately breaks transparency in several cases to allow (require) applications and users to become aware of (deal with) the problems. Hence, on Satyanarayanan's range of adaptation strategies discussed in

section 2.6.3, Coda falls somewhere between the two extremes. Coda's trade-off between transparency and awareness is therefore different from the network-layer solutions discussed in section 3.1, all of which attempted to retain full transparency, and more in line with the support for adaptive application behaviour later adopted in MSOCKS as discussed in section 3.2.3.

3.3.2 Rover

Rover [97, 98] was developed at the Massachussetts Institute of Technology (MIT) and is a toolkit designed to support mobile ('roving') applications by offering them 'a distributed object system based on a client/server architecture with client caching and optimistic concurrency control' [98, p.338]. Rover borrows a number of ideas from Coda, discussed in section 3.3.1, most notably the optimistic approach to replica maintenance and the transmission of updates via a logging facility. Where Coda supports the replication and distribution of files, Rover makes it possible to move object code and state between hosts, and to replicate and reconcile objects. Rover targets mobility-aware as well as mobility-unaware applications by offering 'a set of software tools that supports applications that operate obliviously to the underlying environment, while also enabling the construction of applications that use awareness of the mobile environment to adapt to its limitations' [98, p.337].

Description and Operation

Rover is based on the client/server model for distributed applications. The system is based on two central ideas: relocatable dynamic objects (RDOs) and queued remote procedure calls (QRPCs). Together, RDOs and QRPCs constitute the Rover object model. Rover RDOs consist of four components: 'mobile code, encapsulated data, a well-defined interface, and the ability to make outcalls to other RDOs' [98, p.341]. RDOs communicate by making QRPCs on other RDOs. When a client issues a QRPC, the QRPC is logged locally (along with any parameters) and control returned immediately to the client. If the client has registered a callback function, it will receive notification when the QRPC has completed. Alternatively, the client may block until the QRPC completes. RDOs can be copied, marshalled and moved between clients and servers by being passed as parameters to and return values from QRPCs. Each RDO has a 'home' server that maintains the primary (authoritative) copy. Clients use QRPCs to obtain secondary copies of primary RDOs. Updates performed on secondary RDOs are tentative until the operation is committed, i.e., received and accepted by the server holding the primary copy. Applications that are mobility-aware can retain knowledge of whether their updates are in tentative or committed state, whereas mobility-unaware applications will not distinguish between the two. Hence, if a conflict causes a server to reject a tentative update, a

mobility-aware client can prevent inconsistencies between the primary and secondary RDOs, e.g., by alterting the user and undoing the update on the secondary. In comparison, a mobility-unaware client may become inconsistent with the server and other clients.

Three main Rover components reside on the client side. An object cache holds secondary copies of RDOs and has the ability to interact with servers to import and export RDOs. An operation log is used to record mutating operations (i.e., QRPCs that modify the state of an object) made on secondary copies of RDOs. The log component can interact with servers to propagate changes to the primary copies. When a client (for example during disconnected operation) invokes methods on an RDO, log entries describing mutating invocations are added to the operation log and control is returned immediately to the application. This constitutes tentative updates. The operation log is later exported back to the server and used to commit the changes to the primary copy. If an application has registered a callback function, it will be invoked when the changes are committed. Client applications interact with the log and cache components through the Rover library. Both log and cache components interact with remote servers through a third component, the network scheduler, whose role is to 'interact[s] with the operating system to initiate or terminate connection-based transport links (e.g., dial-up) or to power-up/power-down wireless transmitters' [98, p.339]. Mobility-aware applications can specify priorities to updates, which are taken into account by the network scheduler. This allows applications to distinguish urgent from less urgent updates. If only an expensive link is available, the client may wish to reduce communication by only transmitting urgent updates and delaying less urgent updates until a later time. This means QRPCs may be delivered out of order (i.e., non-FIFO).

On the server side, the application's main responsibility is to manage primary copies of RDOs. A server can receive and process QRPCs from clients. This may include creating a secondary RDO (from a primary) and returning it to the client as a result of a QRPC. In addition, servers receive operation logs from clients and commit the entries by applying them to the primary copies. The server also contains functionality for detecting and resolving conflicts that may occur as a result of client operations. Because conflict detection and resolution is application-specific, Rover cannot perform these operations transparently, but instead offers assistance to applications in the form of version vectors for RDO replicas, and applications then provide their own algorithms for resolving conflicts. Applications also have other responsibilities, such as doing their own prefetching during periods of connectivity.

Object Mobility and Transport Issues

RDOs are executed in a 'controlled enviroment' [98, p.341], a virtual machine. In the current (1997) version of Rover, RDOs are implemented as Tcl/Tk programs. RDOs can be moved from server to clients as described, but also in the opposite direction. This can be useful in cases where the client wishes to perform a time-consuming or data-intensive operation on the server. Rover allows functionality to be moved close to the data and vice versa, depending on what is more suitable for the application in question.

In terms of communication, Rover supports a number of different transports, for example HTTP and SMTP. These are managed by the network scheduler. The network scheduler also implements a number of optimisations in order to achieve better utilisation of potentially slow links, such as batching of QRPCs and standard data compression. Rover allows the request and response to be transmitted over different channels. If a mobile host is disconnected between issuing a request and receiving a reply, the server has the option of relaying it over a different channel, such as a broadcast-only sattelite or a receive-only pager. If processing the request is time-consuming, the decoupling also allows the client to disconnect in order to save battery power or reduce communications cost.

Discussion

The Rover toolkit offers comprehensive support for mobility-aware as well as mobility-unaware applications. The approach is based on a new object model tailored for operation in mobile environments. Compared to traditional object models designed for used with fixed networks, e.g., the CORBA object model, Rover's RDOs have strong native support for object replication, reconciliation and mobility. Rover's object model also has other features, e.g., split-phase communication, which are not found in traditional object models but which can improve performance in mobile environments.

Object-oriented application design generally uses data encapsulation to hide object internals. While this offers an improved programming model, it also impedes system support for conflict detection and resolution. Rover addresses this problem by offering hooks that mobility-aware applications can use to provide their own conflict detection and resolution algorithms. Hence, data encapsulation is preserved in the Rover object model.

Of the fourteen challenges related to mobile environments discussed in chapter 2, the set addressed by Rover is rather similar to that addressed by Coda. As shown, Rover has excellent support for long- as well as short-term disconnections (section 2.4.2). The variety of transports supported by the network scheduler in combination with the option for split-phase communication between clients and servers constitutes very good support for network heterogeneity (section 2.4.1). Rover's comprehen-

sive support for disconnected operation combined with the possibility of associating priorities with individual QRPCs constitutes good support for the problem of usage cost (section 2.4.6) and bandwidth variability (section 2.4.4) of mobile networking technologies. The data compression techniques used constitute good support for low-bandwidth links (section 2.4.3).

Perspective

The authors of Rover conclude that its use of caching achieves 'high availability, concurrency, and reliability' [98, p.342]. In many respects, Rover is to object-oriented middleware what Coda is to file systems. A number of changes to semantics compared to standard object models allow significantly improved operation of applications in mobile environments. Some categories of applications can remain unaware of issues related to mobility, but many are expected to benefit from some level of mobility-awarenes. Rover allows various degrees of mobility-awareness and therefore covers a range (starting from the application-transparent end of the spectrum and not reaching the laissez-faire end) rather than a fixed point on Satyanarayanan's spectrum of adaptivity strategies discussed in section 2.6.3. The main disadvantage of Rover is related to compatibility with existing object models. To benefit from mobility support, application designers have to adopt the Rover object model and toolkit, including the Tcl programming language and the Tk toolkit. Nevertheless, Rover constitutes an excellent example of what is possible in terms of mobility support for object-oriented middleware, if compatibility with existing architectures is not a priority.

3.3.3 Summary

This section has reviewed two influential middleware systems that provide support for applications operating in mobile environments. Both of the systems were based on the modification of conventional semantics used in their respective domains to better suit mobile environments. The discussion is summarised in table 3.3. Comparing this table with table 3.2 shows that the middleware as well as the transport-layer solutions are mainly concerned with addressing the issues related to mobile networking, and that the support offered by the middleware systems is a little more comprehensive than that of the transport-layer solutions. Where the transport-layer solutions also addressed the address migration problem, the two middleware systems reviewed here leave this challenge to other layers.

The contributions of Coda and Rover as a result of the changes in semantics are twofold. First, the optimistic replication strategy makes it possible to build convincing solutions to the problem of long-term disconnection (section 2.4.2). In comparison, the transport-layer solutions discussed in

General Category	Specific Challenge	Coda	Rover
Mobile Devices (section 2.3)	Battery Power	○	○
	Data Risks		
	User Interface		
	Storage Capacity		
	Processing Power		
Mobile Networking (section 2.4)	Network Heterogeneity		●
	Disconnection	●	●
	Low Bandwidth	○	○
	Bandwidth Variability	○	○
	Security Risks		
	Usage Cost	○	●
Physical Mobility (section 2.5)	Address Migration		
	Location-Dependent Information		
	Migrating Locality		

Table 3.3: Summary of Middleware for Mobile Environments

section 3.2 only provided support for short-term disconnections. As discussed in section 2.6.5, the ability to deal with long-term disconnections has considerable advantages, such as reduction in power consumption and communication expenses and the ability to operate in the absence of connectivity. Second, Coda and Rover allowed a considerable degree of flexibility in relation to application awareness and involvement in the device's connectivity. This makes it possible for applications to tailor their behaviour to the circumstances. As discussed in section 2.6.3, this can result in increased usability of the application.

3.4 CORBA Mobility Support

As mentioned in section 1.1, a number of research initiatives have concerned the adaptation of existing object-oriented middleware architectures for use in mobile environments. One such architecture is the Common Object Request Broker Architecture (CORBA), specified by the Object Management Group (OMG), a consortium of more than 600 members. Some work within the OMG has directly addressed the use of CORBA in mobile environments and two official specifications, Minimum CORBA [81] and Wireless CORBA [79] have been produced. In addition to the work performed within the OMG, some research has also been undertaken by independent parties. This section presents the state of the art in CORBA mobility support by reviewing the two OMG initiatives and two independent efforts, DOLMEN and $\prod^2$, which aim to make CORBA suitable for mobile environments. In addition, two software mobility schemes, the Object Migration Service and the Jumping Beans product, which address only the address migration problem are given shorter reviews.

The discussion in this section requires some familiarity with key CORBA concepts, such as the General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP) used to perform remote invocations and the Common Data Representation (CDR) format used to encode data transmitted as part of invocations. A detailed introduction to CORBA is beyond the scope of this thesis and the reader is referred to the CORBA 3.0 specification [83]. However, several of the approaches discussed in this section propose modifications to CORBA object references, and these references are therefore worth a special mention. In standard CORBA, objects are identified via globally unique Interoperable Object References (IORs). An IOR contains one or more *profiles*, each of which specifies a server endpoint, typically in the form of an Internet *host name* and a *port number* identifying a server process and an *object key* (a string) identifying the object within the server process. A client invoking a method on an object connects to the server process using the host name and port specified in the IOR's first profile and supplies the object key as part of the invocation. If the IOR contains multiple profiles, the client should try them in turn until one succeeds; this may involve connecting to several server processes on different hosts. IORs can be published in a number of ways, e.g., through the CORBA Name Service, but can also be returned as results of method invocations or simply converted to strings and transmitted outside CORBA.

3.4.1 Minimum CORBA

The Minimum CORBA specification describes 'a subset of CORBA designed for systems with limited resources' [81, p.1-2]. While the specification as such does not focus on mobile environments, the device limitations addressed by Minimum CORBA do constitute a significant part of the challenges related to such environments (section 2.3). The key idea of Minimum CORBA is to move certain expensive functions (e.g., runtime support for dynamic typing) normally performed by the ORB into the application layer. The intention is that many Minimum CORBA applications will not need full ORB functionality and some of the functionality can therefore be omitted, resulting in reduced resource consumption on the host device. Even if the application does require omitted functionality, only a subset will be required (compared to that offered by a full ORB) and this subset can be provided by the application itself. The Minimum CORBA specification observes that '[t]he omission of a feature of CORBA represents a trade-off between usability and conserving resources' [81, p.1-2]. ORBs conforming to the Minimum CORBA standard remain fully interoperable with ORBs implementing the full CORBA standard and with other Minimum CORBA ORBs.

Description

The omitted functionality falls into one of three groups. First, Minimum CORBA has no support for dynamic type checking; the specification recommends the omission of support for type safety with respect to the CORBA 'any' type, to TypeCodes (used to maintain dynamic type information) and to narrowing of Object References. This removes the need for a Minimum ORB to include code to parse type information at runtime and hence decreases the ORB code footprint. Second, Minimum CORBA recommends the omission of support for user and system exceptions when the former are not used in the application. This results in a reduction in code required to detect and throw exceptions. Third, Minimum CORBA recommends that support for multiple inheritance (i.e., inheritance from multiple interfaces) be omitted if the application does not require this feature. This removes the need to maintain multiple inheritance tables at runtime and also means that code normally required to process the tables during method invocation can be omitted.

Discussion

Version 1.0 of Minimum CORBA appeared in August 2002, and details of savings in terms of storage and processing power of actual implementations have therefore yet to appear in the literature. Despite the reduction in functionality, the current specification has already been criticised for retaining too much and remaining too big for some classes of systems [20, p.15]. In particular automotive, sensor and board-level systems which typically use 8-bit or 16-bit processors with less than 512 kilobytes of memory have been cited as being to low-spec to comfortably run an implementation of Minimum CORBA, whereas a 32-bit processor with more than 512 kilobytes of memory is expected to be sufficient [20, p.16]. Work within the OMG is currently ongoing to revise Minimum CORBA for use on the smaller class of devices.

Of the five challenges related to devices with limited characteristics discussed in section 2.3, the Minimum CORBA standard can certainly be said to address issues related to the limited storage capacity (section 2.3.4) and processing power (section 2.3.5). Also, issues related to battery power (section 2.3.1) can be said to be indirectly addressed through the expected reduction in memory and processor requirements. Issues related to data risks (section 2.3.2) and user interface (section 2.3.3) are not addressed and seem to be beyond the scope of the specification.

Perspective

Despite the critique, it is clear that Minimum CORBA even in its current form constitutes a significant reduction in complexity compared to a fully fledged CORBA-compliant ORB. It seems reasonable to

expect a considerable reduction in code and data footprints for implementations of Minimum CORBA compared to full ORBs. Because of the newness of the specification, relatively few implementations exist and experience with the specification is therefore limited. Examples of ORBs that support the Minimum CORBA standard are TAO by Washington University in St. Louis [166] and e*ORB by Vertel [180].

3.4.2 Wireless CORBA

Whereas the Minimum CORBA specification discussed in section 3.4.1 describes how CORBA can be made to operate on devices with limited capabilities, another OMG specification called 'Wireless Access and Terminal Mobility in CORBA' (or 'Wireless CORBA' for short) addresses a number of other challenges related to the mobile environment. The Wireless CORBA specification has been under development since 1998 and is currently in the finalisation phase. Few, if any, changes are expected to the current (June 2001) version [79]. The specification is also documented in a white paper by Black et al. [24]. Wireless CORBA retains full compatibility with the CORBA 3.0 specification [83] such that ORBs that do not implement Wireless CORBA will be able to interoperate with ORBs that do. Somewhat surprisingly, the Minimum CORBA and Wireless CORBA specifications do not relate themselves to each other, but it appears that Wireless CORBA does not require functionality that was omitted from Minimum CORBA.

Description

Wireless CORBA is based on the gateway model described in section 2.2.2. Mobile hosts, called *terminals*, interact with the fixed network through *access bridges*, software components that reside in the fixed network. Each terminal also hosts a software component called a *terminal bridge* which is responsible for interacting with access bridges. A connection between a terminal and an access bridge forms a *GIOP tunnel* through which the General Inter-ORB Protocol (GIOP) messages required to perform CORBA invocations can be sent and received. Terminals can move between access bridges. Each terminal also has an optional *home location agent* that keeps track of which access bridge the terminal is currently connected to.

Mobile Object References

Wireless CORBA defines a *Mobile IOR,* an extension to the standard IOR format that includes an identifier for the terminal on which the object resides (as would a normal IOR) but also details about the access bridge to which the terminal was connected when the IOR was created and, optionally, the

home location agent for the terminal in question. Like an ordinary IOR, a Mobile IOR contains a series of profiles that specify endpoints where the server can be located. The information about the access bridge and the home location agent is contained in the form of extra profiles. The extended format allows clients holding Mobile IORs to find servers that have changed location through a combination of redirection (between access bridges) and the use of a centralised directory (the home location agent). The details about the access bridge are included twice in the Mobile IOR (in two different formats) because of compatibility considerations between GIOP versions 1.0/1.1 and 1.2.

To allow redirection between access bridges, the Mobile IOR contains at least one profile with the address of the access bridge to which the terminal was connected when the IOR was created. Like an ordinary profile, this profile contains an object key. Ordinarily, the object key specifies the object within the server for which the invocation is destined. However, in the case of the access bridge profile, the object key contains an identifier for the terminal to which the IOR belongs. Such an object key is called a Mobile Object Key (MOK). Like an ordinary object key, a MOK will be passed to the access bridge as part of a standard GIOP 1.0/1.1/1.2 request. This scheme allows a client unaware of Wireless CORBA to make invocations on the terminal through the access bridge. When a client connects to the server, it will connect to the bridge specified in the access bridge profile and pass along the MOK that identifies the terminal in question. If the terminal is still connected to that bridge, the bridge has a GIOP tunnel to the terminal and can relay the invocation to the terminal through tunnel. If the terminal has moved to a different access bridge, the old bridge can redirect the client using a GIOP LOCATION_FORWARD message. All interaction between the client and the access bridge is performed using only the standard (non-Wireless CORBA) features and will therefore work with ordinary CORBA clients using any version of GIOP. Hence, using the address of the access bridge in the IOR allows clients to find servers that have moved through repeated redirections.

While this scheme makes it possible to find servers that have moved, it requires access bridges to maintain forwarding pointers indefinitely or risk that IORs held by clients become invalid. For this reason, a Mobile IOR may also contain details about a home location agent, which maintains information about the current whereabouts of the terminal. If present, the home location agent's address is contained in the Mobile IOR in the form of a special 'mobile terminal profile.' Because the authors of Wireless CORBA consider the idea of embedding the terminal identifier in the MOK a somewhat unorthodox use of the profile's object key field, the mobile terminal profile also contains the address of the access bridge to which the terminal was connected when the IOR was created. (Hence, the bridge address is contained twice in the Mobile IOR.) The intention is that clients using GIOP 1.2 or newer will use the mobile terminal profile, while clients using GIOP 1.0 or 1.1 will use the MOK.

In the long term, the MOK approach will become obsolete.

The special mobile terminal profile is part of the Wireless CORBA specification and therefore not recognised (and consequently ignored) by clients that do not implement the Wireless CORBA specification. The special profile contains an identifier for the terminal and, if the terminal has a home location agent, a CDR-encapsulated reference for the home location agent. When a client holding a Mobile IOR connects to an access bridge using GIOP 1.2, the access bridge will request that the client supply the entire object reference used to make the invocation. This is done using the NEEDS_ADDRESSING_MODE feature of GIOP 1.2 and gives the access bridge access to the terminal identifier, which allows it to forward the request or redirect the client as befits the situation. Also, if the access bridge knows nothing about the whereabouts of the terminal (i.e., does not have a connection to it and no knowledge of its more recent location), the home location agent reference allows it to query the home location agent for the terminal's current location or simply redirect the client directly to the home location agent.

While this works well for GIOP 1.2, the NEEDS_ADDRESSING_MODE feature is not supported by GIOP 1.0 and 1.1, which therefore have to rely on the MOK approach. The authors of Wireless CORBA consider the use of the mobile terminal profile preferable and recommend that this profile be included by servers using GIOP 1.2 or newer, even if the MOK approach is also used for GIOP 1.0/1.1 compatibility reasons.

Terminal Mobility

The tunnel between the terminal and the access bridge can operate on top of three different transports: TCP, UDP and WAP Wireless Datagram Protocol (WDP). A specialised underlying tunneling protocol exists for each of the three transports in order to provide a reliable, ordered message delivery service to the tunneling protocol itself. When a terminal moves, it can establish a new tunnel to another access bridge. State related to the old tunnel is transferred to the new access bridge as part of the handoff procedure. Wireless CORBA operates with two types of handoff: forward and backward handoff. In the former case, the connection is lost and reestablished to the new access bridge. In this case, the new bridge initiates the handoff. In the latter case, the connection to the old bridge persists while the connection to the new bridge is being set up. This is expected to be the normal case. Backward handoff can be initiated by the terminal or the network as appropriate; forward handoff only by the terminal.

When a terminal has changed access bridge, the new access bridge updates the terminal's home location agent by performing a CORBA invocation, passing its own identifier as a parameter to the

invocation. When a terminal moves along a series of access bridges, it effectively leaves a series of forwarding pointers behind, pointing towards more recent locations. Wireless CORBA allows access bridges several hops back to register interest in any future handoffs related to a particular terminal. This is useful in case a terminal maintains a long-term GIOP connection, since it allows the access bridge involved in that connection to keep tracking the terminal for as long as the GIOP connection persists.

Discussion

As we have seen, mobility management in Wireless CORBA is based on the use of a decentralised forwarding pointer scheme in connection with a home agent approach as known from Mobile IP [138]. The MOK approach in conjunction with forwarding pointers correspond roughly to the scheme adopted for the first version of the architecture presented in this thesis [84]. The use of the home location agent as a fallback solution gives an additional level of fault tolerance. In combination with the handoff facilities, this hybrid scheme constitutes a convincing solution to the address migration problem (section 2.5.1). In relation to MOK, it could be argued that the use of the special profile is somewhat superfluous, since the MOK approach will work with all versions of GIOP. However, the MOK compatibility mode is inferior to the special profile mode in one respect: Using MOK, the access bridge can never obtain the home location agent's reference and can therefore never forward the client to it, in case the access bridge has no information about the terminal's current whereabouts. However, in all other cases, the two schemes are functionally equivalent.

The availability of three different transports, each with its own specific tunneling protocol constitutes a CORBA-specific solution to the problems of network heterogeneity (section 2.4.1) and short-term disconnections (section 2.4.2). The functionality provided by each of the Wireless CORBA transports is comparable to that of the MOWGLI transport-layer mobility support scheme discussed in section 3.2.2, but specific to the GIOP protocol. In this respect, the MOWGLI approach is superior because of its higher degree of generality.

There is some confusion about the relation between Wireless CORBA and network-layer mobility support schemes, such as Mobile IP. As discussed in section 3.1.1, Mobile IP addresses inter-domain mobility. In relation to Wireless CORBA, Black et al. write that '[t]he middleware level is the best one to take care of mobility between administrative or service provisioning domains' [24, p.1] and it therefore seems that Wireless CORBA and Mobile IP both target the macro-mobility problem. However, Black et al. also mention that '[t]he objective of the specification is not ... to replace Mobile IP' [24, p.16]. The Wireless CORBA specification [79] does not mention Mobile IP. From the perspective

of a CORBA application, Wireless CORBA is marginally superior to Mobile IP. Both approaches solve the address migration problem, but Wireless CORBA also addresses the problems related to network heterogeneity and short-term disconnections. For non-CORBA applications, Wireless CORBA offers no support and to such applications, the more general support provided by Mobile IP is preferable.

Perspective

It seems that Wireless CORBA is somewhat of a misnomer. While the specification does include a protocol specification for one wireless transport (WAP), the main achievement of the standard is to solve the address migration problem at the middleware level. Since address migration is related to physical mobility (section 2.5) rather than mobile or wireless networking (section 2.4), perhaps the name Mobile CORBA would have been more descriptive. That said, the hybrid solution to the address migration problem is quite elegant and promises good availability for mobile CORBA servers with full transparency to clients, even those that do not implement the Wireless CORBA standard. The main weakness of Wireless CORBA is that the problems of network heterogeneity, disconnection and address migration are solved in a GIOP-specific manner, and that the solutions therefore only apply to applications using CORBA. For some categories of devices (e.g., some embedded systems) that could conceivably rely exclusively on CORBA for communication, this is not a problem, but for others (e.g., general-purpose PDAs) where GIOP would merely be one of several protocols used, such middleware-specific solutions may lead to replication of functionality, increased resource requirements and reduced performance.

As discussed in section 2.6, increasing mobility support is likely to increase the requirements on processing power, storage capacity and battery life on mobile computers. For Wireless CORBA, the critical component is the terminal bridge which resides on the mobile computer. Because of the specification's newness, experience with actual implementations of Wireless CORBA is still limited and performance results remain to be published. However, the first implementations are starting to appear, for example the open source MICO project [154], Vertel's e*ORB [180] and Nokia's M2M platform [145]. As discussed in section 3.4.1, some of these systems also implement Minimum CORBA. When used in conjunction, the reduction in functionality offered by Minimum CORBA may prove sufficient to mandate the extra overhead imposed by Wireless CORBA.

3.4.3 DOLMEN

DOLMEN was a collaborative project funded by the European Union under the Advanced Communications Technologies & Services (ACTS) programme. It ran from 1995 to 1998 and had 12 partners.

While the main purpose of the project was to provide mobility support in the Telecommunication Information Networking Architecture (TINA), the effort also included the development of a CORBA-based mobility support scheme. This mobility support scheme later became the basis of the Wireless CORBA standard discussed in section 3.4.2. Because of the similarities between Wireless CORBA and the DOLMEN approach to CORBA, the latter will be given comparatively less attention in this state of the art review.

Description

Liljeberg et al. [112] identify and target two of the three general categories of mobility challenges: mobile networking (section 2.4) and physical mobility (section 2.5). Like Wireless CORBA, the DOLMEN approach is based on the gateway model discussed in section 2.2.2. Mobile hosts, called *terminals*, interface with the fixed network through *access points*. The CORBA concept of *bridges*, software components that act as gateways between CORBA and non-CORBA environments, is used in a similar way to that of Wireless CORBA. On each DOLMEN terminal runs a Mobile Distributed Processing Environment Bridge (MDBR), a software component that communicates with Fixed Distributed Processing Environment Bridges (FDBRs) residing on access points in the fixed network. DOLMEN generally assumes the link between the terminal and the fixed network to be wireless. The ORBs on the terminal and the access point interface with their respective bridges. During operation, the FDBR acts as a proxy for the terminal, relaying outgoing and accepting incoming invocations. A Location Register placed in the fixed network is used to track the terminal's current point of connectivity by associating terminal identifiers with FDBR addresses.

CORBA allows Environment-Specific Inter-ORB Protocols (ESIOPs) to be defined for specific domains where the General Inter-ORB Protocol (GIOP) is less suitable. DOLMEN uses an ESIOP called the Lightweight Inter-Orb Protocol (LW-IOP) for communication between the MDBR and FDBR. LW-IOP is functionally equivalent to GIOP but has additional features for improved performance in mobile environments. First, LW-IOP employs caching of unsent data combined with an acknowledgement scheme to deal with the unreliability of the underlying wireless medium. Second, its object references contain not the actual names and addresses of machines (as do ordinary IORs) but references which are translated at runtime via the Location Register. Third, LW-IOP aims to reduce the amount of data transmitted over the wireless link through the use of a more compact header format than GIOP, a more space-efficient data representation format than CDR, the ability to reference data units (such as strings) sent in previous messages, and through the use of general data compression algorithms.

Reconnection and Handover

When a terminal loses the connection to an access point, any outstanding data is cached by the old FDBR. If the terminal reconnects to the same FDBR, communication is simply resumed. If the terminal connects to a new FDBR, a handover takes place between the two FDBRs. If there are outstanding invocations (i.e., open GIOP connections between the old FDBR and any remote parties), this involves setting up a tunnel between the two FDBRs. The tunnel persists until all pending invocations have been completed. Handover also involves updating the Location Register with the new FDBR. The combination of transparent reconnection and handover allows mobile devices to move between bridges without losing open connections.

Object References

DOLMEN introduces a modification to the IOR format along the lines of the MOK scheme discussed in section 3.4.2. For server objects residing on mobile terminals, the object reference would usually contain a profile with the hostname of the terminal. In DOLMEN, this hostname is replaced with the hostname of the FDBR and a terminal identifier is encoded in the object key field of the profile. When a mobility-unaware client invokes the server object, it will connect to the FDBR given in the IOR and pass the object key to the FDBR as part of the invocation. The FDBR then extracts the terminal identifier from the object key and if the terminal is still hosted by this FDBR, the invocation is relayed directly to the MDBR. If the terminal has moved, the FDBR finds the new FDBR by looking up the terminal in the Location Register. It then constructs a new IOR where the profile hostname refers to the new FDBR and returns this reference to the client using the IIOP LOCATION_FORWARD feature. DOLMEN implements a centralised Location Register but recommends that a distributed implementation be adopted. In this way, the DOLMEN approach supports servers on mobile hosts but relies on the Location Register to maintain up-to-date information about the current location of a mobile host.

In DOLMEN, the FDBRs also perform the roles of translating IORs. Each FDBR monitors traffic going from the MDBR and into the fixed network. Any object references passed as parameters to outoing invocations or returned as results of incoming invocations are examined and, if they pertain to objects residing on the terminal, translated in the manner described above. This makes it possible for clients on the terminal to use ordinary (non-translated) references to local servers and hence make invocations without going through the FDBR. Any IORs passed to remote clients will have been translated by the FDBR, and invocations using such IORs will therefore be made on the FDBR.

Discussion

DOLMEN addresses a number of challenges related to mobile environments. Like Wireless CORBA, the scheme constitutes a middleware solution to the address migration problem (section 2.5.1). The handover scheme combined with the caching and retransmission features of LW-IOP constitutes good support for short-term disconnection (section 2.4.2). The compact data format of LW-IOP directly addresses the problem of low bandwidth links (section 2.4.3) and to some extent also the problem of usage cost (section 2.4.6). In terms of usage cost, a superior solution would probably allow the user more fine-grained control over the use and cost of the different interfaces.

While LW-IOP includes a number of features supporting optimisation of GIOP for improved bandwidth utilisation, Liljeberg et al. do not discuss the actual reduction in data transmitted during a typical CORBA invocation, and some uncertainty remains as to whether the optimisation efforts are worthwhile. For example, some of the types of wireless media that LW-IOP is designed for already use hardware data compression (e.g., typical GSM modems), and is not clear that the compression techniques employed by LW-IOP will actually improve performance in such cases. Also in this context, a 2001 paper by Adwankar examines standard GIOP performance in three types of wireless networks (802.11, CDPD and the iDEN packet data system) and concludes that the GIOP message formats 'are already sufficiently optimized and further optimization will only result in reduced interoperability' [8, p.52].

Like Wireless CORBA, the LW-IOP approach to mobility support is CORBA-specific. While the CORBA-specific solution allows IORs leaving the terminal to be transparently translated at the FDBR, it also means that any non-CORBA applications residing on the terminal can not benefit from the transparent reconnection or tunneling features available to CORBA applications. Like Wireless CORBA, the features offered by DOLMEN are comparable to those offered by Mobile IP, but only for CORBA applications.

Perspective

As this and section 3.4.2 have shown, DOLMEN and Wireless CORBA employ similar means to achieve comparable functionality. Both offer convincing, although CORBA-specific, solutions to the address migration problem, and both can hide short-term disconnections from applications. In relation to mobile networking, the focus of the two solutions is slightly different. Where Wireless CORBA addresses the problem of network heterogeneity, offering specialised tunneling protocols for a number of transports, DOLMEN is designed to work with a single low bandwidth link. As discussed in section 2.4.1, the diverse range of networking options is a significant characteristic of current mobile

computers, and this change in emphasis is therefore mandated. The two schemes also differ in respect to the location management scheme used to track mobile terminals. The (centralised) location register, arguably one of DOLMEN's weaker features, was in Wireless CORBA replaced with a decentralised forwarding pointer approach in combination with a home agent type scheme. DOLMEN precedes Wireless CORBA by several years, and it is clear that the former has had a significant impact on the latter. Perhaps the most significant contribution of DOLMEN has been to help explore functionality suitable for inclusion in Wireless CORBA.

3.4.4 The $\prod^2$ Proxy Platform

Ruggaber et al. describe an approach for mobile CORBA application support called the $\prod^2$ Proxy Platform [156, 157]. The $\prod^2$ platform allows CORBA requests from clients on mobile computers to be relayed to remote servers but does not in itself support server mobility. Pählke et al. [136] describe how $\prod^2$ can be used in conjunction with a Mobile IP variant called the Firewall-Aware Transparent Internet Mobility Architecture (FATIMA) [122] to achieve support for mobile servers.

Description

The $\prod^2$ platform is based on the gateway model described in section 2.2.2. Mobile Nodes (MNs) interact with gateways called Access Nodes (ANs) that act as proxies for MNs. The MN always must connect via an AN, even if it has its own address on the fixed network (e.g., using a wired network interface). In this case, any fixed node can act as an AN. The interaction between the MN and the AN is performed by two components, $\prod_m^2$ which resides on the MN and $\prod_f^2$ which resides on the (fixed) AN. $\prod_m^2$ is integrated into the MN's ORB and $\prod_f^2$ is a standalone application. The platform allows general-purpose message filters to be inserted in both components to allow outgoing requests and incoming replies to be modified according to predefined rules. The filters are applied on a per-message (request or reply) basis, but state can be maintained across messages through the use of a persistent data structure. Because filters can modify requests destined for and replies originating from any server object in the fixed network, the filters need access to type information (i.e., method signatures) for the invoked methods. In CORBA, type information is not included in request and reply messages but embedded at compile time in the stubs and skeletons residing in the client and server. For $\prod_m^2$ this is not a problem, because it is integrated into the client ORB which has access to the method signatures. For $\prod_f^2$ it is necessary to query the server's Interface Repository (IFR) to obtain the type information at runtime.

Like DOLMEN and Wireless CORBA, $\prod^2$ introduces a proxy between the client and the server

to allow broken connections between the MN and AN to be reestablished transparently. $\prod^2$ does this by using the platform's filtering capabilities to add a level of indirection to CORBA invocations originating at the MN. All outgoing requests generated by the MN's ORB are subject to filtering. The request message format includes the IOR which allows the target of the invocation to be modified by the filter. (The IOR is not included in the GIOP-format of the request.) For requests destined for remote servers, the hostname and port number of the MN is replaced with that of the current AN, and the original hostname and port number are encoded in the ServiceContext field of the profile. The ServiceContext field is ordinarily used to transfer implicit context information, such as security details, from the client to the server. The modification of the request means that the ORB on the MN will automatically relay outgoing invocations to the AN to which the MN is currently connected, rather than making the invocation directly on the server. On the AN, the request is received by $\prod^2_f$ where it is restored to its original state and the remote server invoked.

Reconnection and Handover

The presence of the proxy between the client and the server makes it possible to reestablish broken connections between the MN and AN transparently to both client and server. $\prod^2$ administrates a number of transports (e.g., WAP, GSM, WLAN) and can recreate broken connections over the same or a new transport, and to the same or a new AN. Like DOLMEN, $\prod^2$ is designed with characteristics of wireless links in mind, and the protocol used between the AN and MN features a number of bandwidth optimisations, for example in relation to repeated transmission of stringified object references [156, p.167]. Also, a header compression scheme is introduced that reduces the size of a GIOP header from 12 to 6 bytes. For each transport, a reconnection policy can be used to define how often reconnection should be attempted if a connection over that transport is lost. In addition, $\prod^2$ allows network interfaces, applications and user preferences to be described through profiles. During selection of which network interface to use for communication, the profiles are applied one at a time until the best interface (according to the policy expressed in the profiles) is found. The characteristics that can be described in profiles include cost and bandwidth for network interfaces, timeliness requirements for applications and the cost of communication users are willing to accept.

Handovers in $\prod^2$ are initiated by the MN. As opposed to DOLMEN and Wireless CORBA, $\prod^2$ does not use a tunneling scheme. If a client on the MN has moved to a new AN while a request was in progress, an outstanding reply may be cached at the old AN. In this case, the MN makes an explicit invocation to the old AN in order to fetch the reply. It is up to $\prod^2_m$ to keep track of such outstanding requests and initiate the retrieval of the replies.

Performance

Of the CORBA mobility schemes currently described in the literature, $\prod^2$ is the only one to include published performance figures. The results show that the platform causes considerable overhead (a threefold increase) in invocation round-trip times when used with 2 Mbps (802.11) and faster networks, but only about a 5% increase when used over low-bandwidth links such as 9.6 Kbps GSM.

Server Mobility

On its own, the $\prod^2$ platform does not offer support for server mobility. The address translation scheme implemented using the filtering mechanism pertains only to outgoing requests issued by mobile clients. While the handover mechanism allows clients to retrieve outstanding replies to requests that were made before disconnection from the old AN occurred, it does not offer any sort of proxy functionality for clients in the fixed network to access a mobile server. For this reason, and because no tunneling is involved, $\prod^2$ handovers are simpler but the platform in itself only constitutes partial CORBA support because server mobility is not possible.

Pählke et al. [136] describe how the platform can be used in conjunction with a Mobile IP variant called FATIMA [122] to also support mobile servers. Like Mobile IP, FATIMA solves the address migration problem at the network layer, and changes to the MN address therefore remain invisible to clients in the fixed network. FATIMA addresses the Mobile IP firewall configuration problems discussed in section 3.1.1 by requiring special software to be installed on the firewall hosts in the home and foreign networks. Like Mobile IP, FATIMA suffers from of performance penalties due to triangle routing, and the home agent constitues a possible bottleneck and single point of failure.

Discussion

The $\prod^2$ platform addresses a significant portion of the problems related to mobile networking. It does not attempt to solve problems related to the nature of mobile devices and only addresses problems related to physical mobility when used in conjunction with FATIMA. The variety of transports offered and the profile-based selection scheme constitute convincing solutions to the problems of network heterogeneity (section 2.4.1), bandwidth variability (section 2.4.4) and usage cost (section 2.4.6). The optimisation of the GIOP protocol between the MN and AN constitutes good support for low bandwidth links (section 2.4.3), although the reservations regarding such optimisations made in section 3.4.3 in connection with DOLMEN also apply here. The transparent reconnection features constitute good support for short-term disconnections (section 2.4.2). The general filter mechanism currently used to intercept and modify outgoing requests is quite elegant and, given support for object replication

and conflict detection/resolution, seems a natural candidate for also providing support for operation during long-term disconnections. If server functionality could be cached on the MN, filters could be added to $\prod_m^2$ that would intercept outgoing requests and redirect them to local server replicas instead of $\prod_f^2$.

The proposed solution for server mobility based on FATIMA is not unlike the MOWGLI architecture discussed in section 3.2.2. Compared to the decentralised location management schemes employed by Wireless CORBA, $\prod^2$ with FATIMA is weaker in terms of scalability and fault tolerance because of the centralised home agent and also potentially more problematic because of its requirements on firewall configuration, etc. Given server mobility based on FATIMA, one potentially difficult scenario with $\prod^2$ could be in relation to Minimum CORBA. As noted, $\prod_f^2$ relies on the Interface Repository (IFR) being available on the server side in order for the proxy to determine the method signatures. While this is not a problem when the server resides in the fixed network (we assume such servers implement the full CORBA standard, including the IFR), it could make it difficult for a mobile client to interact with a mobile server based on Minimum CORBA, because '[t]he majority of the Interface Repository ... is omitted from minimumCORBA' [81, p.1-5].

Perspective

Like DOLMEN and Wireless CORBA, $\prod^2$ solves the problems related to mobile networking in a CORBA-specific manner. While the solutions are sound, their specificity to CORBA is also the platform's main weakness. As discussed in section 3.4.2, middleware-specific solutions may be suitable for some, but not necessarily all, categories of mobile computers.

3.4.5 Object Migration Service

In a 1999 white paper, Choy et al. [42] from the German software developer IKV++ describe a scheme to achieve object mobility (code, state and computation) within CORBA. Although the paper's focus is on software rather than hardware mobility, the address migration problem (section 2.5.1) caused by server mobility is the same. The idea in the proposed Migration Service is to suspend object execution, marshal the object's state and code into a stream, transfer the stream to the receiving side and create a new copy of the object there. The new object restores its state from the stream whereupon the old copy of the object is destroyed. In case an object is part of an object graph (i.e., has references to other objects), the entire object graph must be transfered together. Choy et al. do not discuss how objects shared between several object graphs should be handled.

The Object Migration Service relies on a number of standard CORBA services. The CORBA

Externalization Service is used to marshal and unmarshal object state and the CORBA Life Cycle Service is used to create new and destroy old copies of objects. (The Object by Value standard is suggested as a possible alternative to the Externalization Service.) The CORBA Naming Service is used to handle the address migration problem by updating the Naming Service whenever IORs change as the result of object mobility. Presumably this requires clients to resolve server references more often than usual. Choy et al. leave relatively open the issue of how object code can be moved and does not discuss how mobility of computation will be handled. Until now, the proposed Migration Service has not been adopted as part of the CORBA standards.

3.4.6 Jumping Beans

Jumping Beans [58] is a commercial framework originally designed by Ad Astra Engineering in the late 1990s and now marketed by Aramira Corporation. The framework lets 'jumping applications' (essentially CORBA server objects) written in Java move between networked nodes. Although the framework is designed for software mobility, the fact that jumping applications are CORBA servers mean that they face the same address migration problem (section 2.5.1) as CORBA servers running on mobile hardware.

The Jumping Beans approach is based on a centralised server, called a Management and Security Console, and a Jumping Beans daemon that runs on each node in the network. Each daemon has the ability to send and receive applications via the Management and Security Console. The daemons do not interact directly. When a jumping application moves, it first moves to the console and then to its final destination. This means that server mobility is transparent to the clients.

3.4.7 Summary

This section has reviewed the state of the art in mobility support for CORBA by discussing the two relevant OMG initiatives and two independent efforts. In addition, two software mobility schemes were briefly reviewed. The Minimum CORBA specification [81] defines how the footprint of a CORBA application can be reduced to allow it to run on devices with limited capabilities. The Wireless CORBA specification [79] tackles the address migration problem and some of the problems related to mobile networking. The DOLMEN project [112], which can be seen as a predecessor to Wireless CORBA, provides slightly less mature solutions to nearly the same set of problems as its successor. The $\prod^2$ Proxy Platform [156, 157], when used in conjunction with FATIMA [122], provides perhaps the most comprehensive suite of mobility support functionality. However, its solution to the address migration problem is less attractive than that of Wireless CORBA, because it (like Mobile IP) suffers from

performance penalties due to triangle routing and also relies on the availability on a (possibly distant) home agent to track mobile hosts.

Table 3.4 summarises the applicability of the systems and specifications reviewed in this section to the fourteen challenges discussed in chapter 2. The legend is that used also in table 3.1. Of the systems reviewed, table 3.4 shows the most comprehensive mobility support would be achieved by combining FATIMA/$\prod^2$ and Minimum CORBA, a solution which would address nine of the fourteen challenges. However, it is unclear to what extent FATIMA/$\prod^2$ can be implemented on the basis of Minimum CORBA. A combination of Minimum CORBA and Wireless CORBA, the two official specifications, addresses only six of the fourteen challenges. On this basis, we conclude that mobility support in CORBA is still *partial*.

General Category	Specific Challenge	Minimum CORBA	Wireless CORBA	DOLMEN	$\prod^2$ with FATIMA	Object Migration Service	Jumping Beans
Mobile Devices (section 2.3)	Battery Power	○					
	Data Risks						
	User Interface						
	Storage Capacity	•					
	Processing Power	•					
Mobile Networking (section 2.4)	Network Heterogeneity		•		•		
	Disconnection		○	○	○		
	Low Bandwidth			•	•		
	Bandwidth Variability				•		
	Security Risks						
	Usage Cost			○	•		
Physical Mobility (section 2.5)	Address Migration		•	•	•	○	○
	Location-Dependent Information						
	Migrating Locality						

Table 3.4: Summary of CORBA Mobility Support Initiatives

With the exception of FATIMA, which is really a network-layer mobility support scheme, a common characteristic of the approaches reviewed in this section is that they are specific to CORBA. The schemes are specifically focused on providing mobility support within CORBA rather than addressing the general case of object-oriented middleware operating in mobile environments. The solutions reviewed here will not easily transfer to other object-oriented middleware architectures, such as Java RMI, DCOM or SOAP; and it is even unclear whether the same principles will apply to other architectures. We can hardly fault the designers of the various CORBA mobility support schemes for this;

they are after all working within the confines of CORBA. However, when possible, general solutions are more attractive, certainly because they facilitate component reusability but also because they offer additional insight into the nature of the problems. On this basis, we conclude that current approaches to CORBA mobility support are *architecture-specific*.

3.5 Java Mobility Initiatives

As discussed in section 1.1, a number of initiatives have adaptated existing object-oriented middleware architectures for use in mobile environments. Section 3.4 discussed the main initiatives in relation to CORBA. Another object-oriented middleware architecture is Sun Microsystems' Java language [71] and the environment constituted by the supporting tools and services. The Java environment addresses the two key challenges that most middleware aims to address, namely *heterogeneity* and *distribution*, as discussed in section 1.2.3. Java hides the heterogeneity of hardware platforms by introducing a homogenous execution environment called a Java Virtual Machine (JVM). Distribution is addressed in a number of ways, including a feature called Java Remote Method Invocation (RMI) [118] that allows Java objects to make object invocations across address spaces with a high degree of transparency. A number of efforts are currently ongoing towards making the Java architecture suitable for mobile environments. This section presents the state of the art in Java mobility support by reviewing one initiative by Sun Microsystems, the Java 2 Micro Edition (J2ME) and two independent efforts related to RMI support in mobile environments. As will be seen, each of the efforts targets a different set of the fourteen challenges discussed in chapter 2.

The discussion in this section requires some familiarity with key Java concepts, such as JVMs and the operation of Java RMI. A detailed introduction to the Java environment is beyond the scope of this thesis, and the reader is referred to the Java Language Specification [71] and Sun Microsystems' RMI white paper [118]. However, several of the approaches discussed in this section propose modifications to Java RMI, and its operation is therefore worth a special mention. In general, Java RMI was designed to simplify communication between objects in different address spaces by allowing methods of remote objects to be invoked transparently. When a client has obtained a reference to a remote object, the object can be invoked as if it was local. A computer that holds objects that can be invoked via Java RMI must run an instance of a software component called the RMI Registry. When a server wishes to make its methods available for invocation by remote clients, it registers the methods with its local RMI Registry. When a client wishes to invoke a remote object, it contacts the RMI Registry on that machine to obtain a server reference.

3.5.1 Java 2 Micro Edition

The Java 2 Micro Edition (J2ME) [119] is one of the three editions of Sun Microsystems' Java 2 Platform. Where the Java 2 Standard Edition (J2SE) is designed for use on desktop computers and workstations and the Java 2 Enterprise Edition (J2EE) for use on servers, J2ME is designed for use on devices with limited resources, such as mobile phones, set-top boxes, washing machines and car navigation systems. The intention is that J2ME will be employed on devices with 16- or 32-bit processors and with memory specifications ranging from 32 kilobytes to 10 megabytes [119, p.10]. For devices with better specifications, one of the other editions (J2SE or J2EE) would typically be more suitable. J2ME is part of the Pervasive Java standardisation efforts which, led by Sun Microsystems, aims to 'bridge the gap between disparate devices and platforms' [86, p.82]. J2ME plays a similar role in the context of Java technologies to that which Minimum CORBA, discussed in section 3.4.1, plays in the context of CORBA.

Description

J2ME consists of a set of virtual machines, libraries, APIs and tools that can be combined according to device capabilities and application requirements. The domain targeted by J2ME is divided through the idea of *configurations* and *profiles*. A J2ME configuration 'defines a minimum platform for a "horizontal" category or grouping of devices, each with similar requirements on total memory budget and processing power' [119, p.12]. By 'horizontal' is meant that configurations are related to the technical specifications of devices rather than what these devices are used for. In addition, profiles are used to group devices 'vertically' according to their intended purpose. A J2ME profile specifies a particular API that applies to a given type of device (e.g., a mobile phone). A device may implement several profiles. Sun explains that a profile 'is layered on top of (and thus extends) a configuration' and has as its main goal 'to guarantee interoperability within a certain vertical device family or domain by defining a standard Java platform for that market' [119, p.12]. Hence, a single profile would typically be shared by a number of devices with similar functionality, even though the devices might be made by different manifacturers and have different technical specifications. Applications are written 'for' a particular profile, and a profile is said to 'extend' a given configuration. The intention is that new configurations and profiles can be specified as required. This is done through the Java Community Process (JCP).

J2ME currently defines two configurations: the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). CDC is intended for use with '[s]hared, fixed connected information devices' which 'have a large range of user interface capabilities, memory budgets

in the range of 2 to 16 megabytes, and persistent, high-bandwidth network connections, most often using TCP/IP' [119, p.11]. Examples include TV set-top boxes, Internet-enabled screen-phones, high-end PDAs and entertainment/navigation systems found in cars. The CDC is based on the C Virtual Machine (CVM), a full J2SE-compliant virtual machine. In comparison, CLDC is a subset of CDC intended for use with '[p]ersonal, mobile, connected information devices' which 'have very simple user interfaces ..., minimum memory budgets starting at about 128 kilobytes, and low bandwidth, intermittent network connections ... often not based on the TCP/IP protocol suite' [119, p.11]. Examples include mobile phones, pagers and low-end PDAs. CLDC is based on the K Virtual Machine (KVM), a smaller and more limited virtual machine. While the majority of the functionality found in CDC and CLDC has been inherited from J2SE, the two configurations also contain some classes that are not part of J2SE. These classes reside within the `javax.microedition` namespace.

Currently, three profiles have been defined within the CDC and two within the CLDC. They are shown in table 3.5. In addition, a number of other profiles are being developed within the JCP. For example, the Java Game Profile extends CDC and the Foundation Profile for the purposes of game development targeting high-end consumer game devices and desktops. Of the two J2ME profiles, only CLDC is really targeted towards use on mobile devices and will therefore be the focus of the remainder of this review.

Configuration	Profile	Purpose
CDC	Foundation Profile	Supports limited-resource devices with some kind of network connection such as set-top boxes and web appliances. Used as the basis for other profiles.
	Personal Profile	Extends the Foundation Profile for those devices with a need for a high degree of Internet connectivity and web fidelity.
	Personal Basis Profile	Extends the Foundation Profile to provide an environment for network-connected devices that support a basic level of graphical presentation.
CLDC	Mobile Information Device Profile (MIDP)	Aims to enable application development for mobile information appliances and voice communication devices such as mobile phones.
	PDA Profile	Aims to provide user interface and data storage APIs for small, resource-limited handheld devices.

Table 3.5: J2ME Configurations and Profiles

CLDC Language and Virtual Machine Restrictions

CLDC omits some features from the Java language specification [71] that are available in the J2SE and J2EE editions. First, CLDC contains no support for floating point data types. The main reason for omitting this feature is that most hardware expected to host CLDC has no support for floating point arithmetic. Second, CLDC has no support for finalisation of class instances. This means that

Java objects cannot implement explicit operations to be executed just before the object is garbage collected. Third, CLDC contains limitations on error handling. CLDC contains full support for exceptions. While exceptions in Java are used for mild error conditions, 'errors' are used for more serious situations where recovery is typically not possible. Error recovery can be 'very device specific' and 'implementing error handling capabilities according to the full Java specification can be expensive and demanding in terms of overhead' [106, p.6].

CLDC contains a specification for a class of virtual machines of which KVM is Sun Microsystems' reference implementation. The entire implementation of CLDC (static size of the virtual machine plus libraries) should fit in less than 128 kilobytes of memory. Applications should run in as little as 32 kilobytes of heap space. A number of features were omitted from the CLDC virtual machine specification due either to strict memory limitations or because of potential security concerns (CLDC only implements a portion of the full J2SE security model) [119, p.22]: support for floating point data types, Java Native Interfaces (JNI), user-defined Java-level class loaders, reflection features, thread groups or daemon threads, finalisation of class instances and weak references. In addition, the virtual machine specification imposes restrictions on error handling corresponding to the omissions from the Java language mentioned above.

CLDC and Profile Libraries

Most of the classes in CLDC are inherited from J2SE. In addition, some J2ME-specific extensions have been made for network and I/O purposes. Embodied in the `javax.microedition.Connector` class, these extensions make it easier to deal with the diverse network interfaces of the devices [119, pp.26–27]. Examples of interfaces that could be addressed via the new CLDC features include sockets, serial ports, datagram transports and network file systems. CLDC, however, only facilitates the existence of multiple interfaces and does not actually define any protocol implementations.

The two CLDC profiles listed in table 3.5 extend CLDC with additional functionality. MIDP also includes features for dealing with a variety of transports, such as IR and serial lines, encrypted and otherwise. APIs also appear for display management (for user interface and game purposes), multimedia playback and recording and for MIDP applications (called MIDlets) to be managed, i.e., discovered, downloaded, verified, run and deleted. The PDA Profile specifies a subset of the Abstract Window Toolkit (AWT) library suitable for PDAs and also assumes with displays with a total resolution of at least 20,000 pixels, a pointing device, and character input. This profile also includes abstractions and APIs for typical PDA functionality, such as calendar entries, contact lists and todo lists.

Discussion

While J2ME as such does not focus exclusively on issues related to mobile environments, the device limitations addressed particularly by CLDC and the two profiles do constitute a significant part of the challenges related to such environments. Of the five challenges discussed in section 2.3, CLDC and the two profiles directly addresses issues related to user interface (section 2.3.3), storage capacity (section 2.3.4) and processing power (section 2.3.5). In addition, the reduced memory footprint and processing requirements can be said to partially address the issue of battery power (section 2.3.1). Arguably, the J2ME-specific extensions to the network API constitute partial support for network heterogeneity (section 2.4.1).

Perspective

J2ME offers not only one but several possible environments with significantly reduced complexity compared to a full-fledged J2SE or J2EE edition. The standards have already been embraced by industry, for example through MIDP-capable GSM/GPRS phones from Nokia, Siemens and Motorola. Perhaps the most significant advantage of J2ME is that it is highly adaptable which, compared to the fixed set of functionality offered by Minimum CORBA discussed in section 3.4.1, may possibly enable J2ME to reach adoption on the very wide range of devices that Sun Microsystems intends. However, at the moment industrial J2ME adoption is mainly limited to MIDP-capable mobile phones, and it still remains to be seen whether J2ME will also perform in the remaining domains.

3.5.2 Monads RMI

Campadello et al. observe that 'Java RMI works poorly in slow wireless environments' [32, p.114] and describe a solution, called Monads RMI [32, 31], which improves performance of Java over wireless links with low bandwidth and high latency, such as GSM. The main concern of Monads RMI is to address issues related to mobile networking as discussed in section 2.4. The approach does not attempt to address challenges related to mobile devices or physical mobility. Monads RMI supports clients but not servers on the mobile computer.

Description

Monads RMI is based on the gateway model described in section 2.2.2. In Monads RMI, mobile *clients* interact with remote *servers* in the fixed network through *proxies* residing on gateways on the periphery of the fixed network. The link between the client and the proxy is assumed to be a

low-bandwidth, high-latency link with high error rates, such as GSM. Campdadello et al. observe that a number of problems cause Java RMI to perform sub-optimally over such connections. First, Java RMI is performed over TCP connections which are notoriously problematic over links with high error rates, because TCP implementations interpret packet loss as a sign of congestion [30]. Second, the Java RMI protocol requires a significant number of round-trips which on a high-latency link translates into a considerable delay. For example, Java RMI requires clients to ping the RMI Registry to verify that the connection is still operational, and while the ping message does not consume much in terms of bandwidth, subsequent communication is dependent on its completion. In one example, 'six round-trips were necessary before the invocation was completed' [32, p.116]. Third, the RMI protocol has a significant overhead in terms of data not directly related to invocations. In one example, 'the actual invocation takes up only 5% of the total transmitted data while 69% was related to the DGC [distributed garbage collection] protocol' [32, p.116].

Monads RMI addresses these problems by using an optimised Java RMI protocol between the mobile computer and the gateway. An RMI Agent on the mobile computer is responsible for translating outgoing requests from the client into the optimised protocol. An RMI Proxy, residing on the proxy node, also implements the optimised protocol and is responsible for relaying outgoing requests to RMI Registries and servers as well as incoming responses to such requests. Four types of optimisations are proposed. First, data compression (gzip) is used to reduce the overhead in the Java format for serialised objects. Second, some protocol acknowledgements usually performed between the client and server are performed by the proxy without the client's involvement. This reduces traffic and round-trips required over the wireless link. Third, results of registry lookups are cached by the RMI agent on the client side. This reduces the time required for subsequent accesses to the same reference. Fourth, the protocol decouples the client and server in relation to distributed garbage collection. While a server reference is cached on the client side, the proxy will periodically renew the lease without involving the client. Only when the client explicitly notifies the proxy that no more references exist, will the proxy stop renewing the leases. This constitutes a loosening of the semantics of Java's distributed garbage collection but results in reduced traffic over the wireless link. Allowing the proxy to renew leases in this manner can be seen as a way of moving client functionality closer to the server in order to adapt the application to the mobile environment.

Performance

Campadello et al. [32, 31] present a number of experiments in which they show that the optimised protocol results in improved performance of Java RMI over GSM connections. The authors conclude

that '[u]sing the original GSM ... our implementation is more than four times faster than the normal RMI in a Windows environment, and more than five times faster in Linux' [32, p.119].

Of the four optimisations implemented, only data compression and reduction appeared in the experiments conducted by the authors. Of these, data compression accounted for the majority of the performance improvement. For non-empty invocations performed without data compression, the performance of Monads RMI was comparable to that of normal Java RMI [32, p.122–123]. For empty invocations without data compression, a speed-up of 8–9% was achieved, presumably attributable to the reduction in the number of round-trips required over the wireless link.

Discussion

The fact that data compression accounts for the significant performance improvement is somewhat surprising. As discussed in sections 3.2.2 and 3.4.3, most (if not all) GSM and analogue modems implement data compression in hardware. In the Monads RMI experiments, a Nokia Card Phone 2.0 GSM phone was used on the mobile computer in conjunction with a Multitech MT2834ZDXI analogue modem on the access node. Both of these implement V.42bis data compression in hardware [146] and should therefore be able to compress data at least as well as the data compression algorithm used in the implementation of Monads RMI. A possible explanation is that the authors, as mentioned in their paper [32, p.118], used a prototype of the GSM phone in the experiments. If this prototype did not come with V.42bis support, the results could have been expected to be as they appear in the paper.

Of the fourteen challenges discussed in chapter 2, Monads RMI addresses only low bandwidth (section 2.4.3), but it does so very well and with a considerable performance improvement. The proxy approach lends itself well to adding functionality to deal specifically with the wireless link, and it would seem relatively straightforward to add support for transparent reconnection of underlying TCP connections, broken as a result of dropped GSM connections. This would result in good support for short-term disconnections (section 2.4.2).

Perspective

Monads RMI addresses one of the fourteen challenges related to using object-oriented middleware in a mobile environment. The optimisations to the Java RMI protocol constitute a Java-specific solution to the Java-specific problem that RMI does not perform well over wireless links. As discussed in section 3.4.3, other sources have suggested similar (if not quite as comprehensive) optimisations for the CORBA remote invocation protocol. Hence, the characteristics of wireless links are generally problematic for remote invocation protocols designed for use in fixed network environments. In the

specific context of Java and Sun Microsystems' efforts in relation to Java on mobile devices, it would seem natural to integrate reduction in number of round-trips required in future versions of the Java RMI protocol. Support for optional data compression would be another candidate, if the protocol is expected to be used over interfaces that do not implement data compression in hardware. Despite the Java-specificity of Monads RMI, some of the optimisations, notably data compression and the migration of key functionality from the mobile host to the fixed network, would seem generally applicable to object-oriented middleware architectures and could perhaps be supported in a more generic manner.

3.5.3 Mobile RMI

Avvenuti et al. [11, 12] describe a system called Mobile RMI that supports mobility of server objects. Although Mobile RMI is a software mobility scheme, the mobility of server objects causes the same address migration problem as the physical mobility of hardware hosting (immobile) server objects. To tackle the address migration problem, Mobile RMI contains 'a reference updating mechanism, that makes references held by clients follow the migrating server' [11, p.100]. Despite the focus on software mobility, the authors of Mobile RMI do place their work in the context of mobile computing by observing that 'having control over dynamic relocation of components can greatly simplify the design of applications that deal with user mobility, device heterogeneity, disconnected operation and other mobile computing related issues' [11, p.98].

Description

In Java, a remote object that uses RMI's default sockets-based transport for communication is implemented by extending the `UnicastRemoteObject` (URO) class. Mobile RMI implements the `MobileUnicastRemoteObject` (MURO) class, an extension of the URO class with two additional methods `create` and `move` that allows MUROs to be created remotely and moved between different JVMs at runtime. Each JVM that can receive mobile objects hosts a mobility daemon, a remote object exporting two methods `MDcreate()` and `MDmove()`, used for implementing the MURO `create()` and `move()` operations.

When using Mobile RMI, objects can be created remotely by locally invoking the `MURO.create()` method, passing as parameters the relevant class name and the host name and port number where the mobility daemon for the remote JVM that will host the object is running. The `MURO.create()` method obtains a reference to the relevant mobility daemon and invokes that daemon's `MDcreate()` method to create the object within the remote JVM.

A MURO is moved from one JVM to another through invocation of its `move()` method which

takes as parameters the host name and port number of the destination JVM. The `move()` serialises the object, invokes the `MDmove()` method of the remote mobility daemon, passing itself as a parameter. Upon reception of the serialised MURO, the mobility daemon deserialises it into the new JVM's address space. If the class code is not available on the receiving JVM, the receiving mobility daemon fetches it from a URL included in the serialised object. The class code can also be serialised along with the object's state. The `MDmove()` method returns a new reference for the remote object. This reference is used to update the server reference held by the client invoking the MURO's `move()` method. Mobile RMI requires that none of the moving object's other methods (except for the `move()` method) are currently being invoked.

Management of Server References

The mobility of a server object means that references held by clients will become obsolete. Mobile RMI features three ways in which such references can be updated. To facilitate updating of references, Mobile RMI modifies the `UnicastRef` class. This class is used in the implementation of `RemoteRef`, the class that implements references to remote objects.

As explained above, the `MDmove()` method on the receiving mobility daemon returns the new reference. This allows the client invoking the MURO's `move` method to update its reference. The extended `UnicastRef` class makes this transparent to the client holding the reference. For clients that hold obsolete references but are not invoking the server, a different update technique is employed. When server movement has completed, a dummy object is created to replace the server object at its old location. This dummy object holds a forwarding pointer in the form of a new server reference. When a client attempts to invoke a server using an obsolete reference, the dummy object will be invoked instead. The dummy object returns the new reference to the client which updates the obsolete reference and retries the invocation. This approach relies on the modified `UnicastRef` class to update the reference and retry the invocation. If the server object has moved several times, a chain of dummy objects will form a path to the actual location of the server object.

The two means of updating references described above will update server references for clients that initiate the server movement and clients that subsequently invoke one of the server's methods. In addition, Mobile RMI extends the Java RMI Distributed Garbage Collector (DGC) to update client references that are not being used. The normal operation of the DGC requires clients holding references to remote objects to renew leases for these references at regular intervals. Mobile RMI extends the DGC to allow updated server references to be piggy-backed onto such lease renewals. When a client renews the lease for a server, the renewal may return not only the new lease but also

an updated reference. This feature also relies on the modified `UnicastRef` class to allow reference updates to take place transparently to clients. Each lease renewal will result in shortening the dummy object chain by one. The authors describe this approach as 'effective' [12, p.67] and claim it imposes 'almost no cost' [11, p.102]. While this seems plausible, no actual performance figures are presented.

Discussion

The chain of forwarding pointers formed by the dummy objects in Mobile RMI is not unlike the chain of references used in Wireless CORBA discussed in section 3.4.2. Where Wireless CORBA relied on a home location agent to be used in case of broken chains, Mobile RMI aims to reduce the risk of broken chains by updating even inactive server references. If the authors' assumption that the overhead is negligible is correct, the Mobile RMI solution could be potentially more scalable and require less administration, because no home location agent is required.

A potential impediment to adoption of a solution along the lines of Mobile RMI for use in mobile environments could be in relation to J2ME discussed in section 3.5.1. Mobile RMI relies on DGC to perform updating of inactive server references, and while J2ME does not forbid the use of DGC, this feature is included in very few profiles. Support for RMI is only included in the Personal Profile and the Personal Basis Profile (both of which extend the CDC), and none of these profiles contains support for DGC. To get DGC support with J2ME, the RMI Optional Package must be used. This package requires the CDC and the Foundation Profile, i.e., a configuration suitable only for use with relatively high-end mobile devices.

Perspective

Mobile RMI, being a software mobility support scheme, addresses only one of the fourteen challenges discussed in chapter 2: address migration (section 2.5.1). The proposed solution is elegant and promises to be quite efficient. It seems the functionality provided by Mobile RMI would be a natural candidate as a means to cache server functionality on mobile clients or vice versa. Combined with appropriate means for conflict detection and resolution, Mobile RMI could conceivably be used to implement good support for long-term disconnection. The modification of key Java classes, such as `UnicastRef`, and the idea of piggy-backing server reference updates onto existing DGC messages will not easily generalise to other object-oriented middleware architectures, and the approach is therefore highly Java-specific.

3.5.4 Summary

This section has reviewed the state of the art in mobility support for Java RMI by discussing the J2ME initiative from Sun Microsystems and two independent efforts. J2ME addresses nearly all of the challenges related to the nature of mobile devices by defining a set of Java environments suitable for devices with limited resources. Monads RMI addressed one of the six challenges (low bandwidth) related to mobile networking by improving the performance of Java RMI over low-bandwidth links. Mobile RMI addressed one of the three challenges (address migration) related to physical mobility by describing how changing server references could be updated in a manner transparent to clients.

Table 3.6 summarises the applicability of the systems and specifications reviewed in this section to the fourteen challenges discussed in chapter 2. The legend is that used in the previous summary tables. Table 3.6 shows that there is no overlap between the scope of the three efforts reviewed here; they address disjoint sets of challenges. The most comprehensive mobility support could be achieved by integrating the two RMI modifications in the context of J2ME. Such a solution would address seven of the fourteen challenges well, and two more partially. On this basis, we conclude that mobility support in Java is still *partial*.

General Category	Specific Challenge	Java 2 Micro Edition	Monads RMI	Mobile RMI
Mobile Devices (section 2.3)	Battery Power	○		
	Data Risks			
	User Interface	•		
	Storage Capacity	•		
	Processing Power	•		
Mobile Networking (section 2.4)	Network Heterogeneity	○		
	Disconnection			
	Low Bandwidth		•	
	Bandwidth Variability			
	Security Risks			
	Usage Cost			
Physical Mobility (section 2.5)	Address Migration			•
	Location-Dependent Information			
	Migrating Locality			

Table 3.6: Summary of Java Mobility Support Initiatives

A common characteristic of the efforts reviewed in this section is that they are very specific to Java. Certainly, J2ME deals with the definition of Java-specific functionality, for example in terms of APIs and libraries, but also the mobility support proposed by Monads RMI and Mobile RMI is implemented

through highly Java-specific means, such as modification to garbage collection messages and RMI protocol handshakes. These solutions will not easily transfer to other object-oriented middleware architectures, such as CORBA, DCOM or SOAP; and it is unclear to what extent even the principles employed here are generally applicable. On this basis, we conclude that current mobility support for Java RMI is highly *architecture-specific*.

3.6 Summary

This chapter has reviewed the state of the art in mobility support for distributed object applications. A number of systems were discussed and evaluated in relation to the fourteen challenges defined in chapter 2. The analysis divided the systems into three layers: network layer, transport layer and middleware layer. The most significant of the problems related to physical mobility, the address migration problem, was addressed by solutions at every single layer. The problems related to mobile networking were addressed only by transport- and middleware-layer solutions, and the problems related to mobile devices were addressed only by middleware approaches. Hence, the higher up in the protocol stack mobility support was provided, the more diverse was the set of challenges addressed. However, the higher-level solutions (such as the CORBA and Java RMI mobility schemes) were highly specific to their respective middleware architectures compared to lower-level mobility support schemes such as Mobile IP.

Section 3.1 reviewed three network-layer mobility support schemes and concluded that they were not in themselves sufficient to deal with the variety of challenges faced by applications operating in mobile environments. However, they could serve as a starting point. One of the transport-layer solutions discussed in section 3.2, the MOWGLI system covered in section 3.2.2, integrated a network-layer solution and added support to deal with the challenges related to mobile networking. Section 3.3 presented two middleware frameworks which focused on addressing challenges related to mobile networking. By adapting for use in mobile environments a number of mechanisms often used with distributed applications in fixed networks, significant improvements were made in relation to support for disconnected operation and adaptive application behaviour.

Sections 3.4 and 3.5 reviewed a number of efforts based on the adaptation of existing distributed object architectures for use in mobile environments. As opposed to the systems discussed in section 3.3, these systems preserved the semantics of existing architectures in order to retain compatibility. While a number of common techniques (e.g., data compression and the moving of client functionality onto the gateway) were used in the different solutions, each solution was highly tailored to a single

distributed object architecture. We therefore concluded that research in this category was highly *architecture-specific*. Since the problems related to the mobile environment affect any distributed object architecture, a more general approach would be to identify generic solutions applicable across several architectures.

The efforts reviewed in sections 3.4 and 3.5 addressed challenges from all of the three categories defined in chapter 2. The set of challenges addressed by CORBA mobility support schemes was generally more complete than that addressed by Java RMI mobility support initiatives. However, compared with the mobility support offered by Coda and Rover, both the CORBA and Java RMI efforts had significant gaps in relation to disconnected operation and flexible support for adaptive application behaviour. For the set of problems addressed, CORBA and Java RMI efforts generally adopted the approaches at the end of Satyanarayanan's range of adaptation strategies (section 2.6.3), i.e., either fully transparent support or no support at all. Although this may be suitable for some applications, the former tends to limit application flexibility (e.g., by hiding state related to mobility and connectivity) while the latter demands application-layer solutions that generally increase application complexity. Consequently, mobility support in current distributed object architectures is *partial*.

Chapter 4

Architecture

"The time has come," the Walrus said,
"To talk of many things:
Of shoes—and ships—and sealing-wax—
Of cabbages—and kings—
And why the sea is boiling hot—
And whether pigs have wings." [36]

This chapter describes the Architecture for Location-Independent Computing Environments (ALICE), an architecture that allows mobility support to be added to any object-oriented middleware framework (i.e., implementations of object-oriented middleware architectures, such as CORBA, Java RMI and SOAP) that supports a set of minimal requirements. ALICE is captured in a set of modular, reusable components that can be used to instantiate the architecture for different object-oriented middleware frameworks. Such modified frameworks have the attractive feature that mobility support remains completely transparent for those portions of distributed applications that do not reside on mobile hardware, while portions that do reside on mobile hardware can be aware or unaware of mobility as required for the application in question. Also, modified frameworks retain interoperability with unmodified frameworks implementing the same architecture.

This chapter describes the general ALICE architecture independently of specific object-oriented middleware frameworks. Chapter 5 extends this chapter by describing how the architecture can be instantiated to create mobility-enabled implementations of CORBA and Java RMI. Section 4.1 explains non-standard notation and terminology used to describe the architecture. Section 4.2 presents an overview of the architecture and outlines the roles of its different layers. Section 4.3 lists six

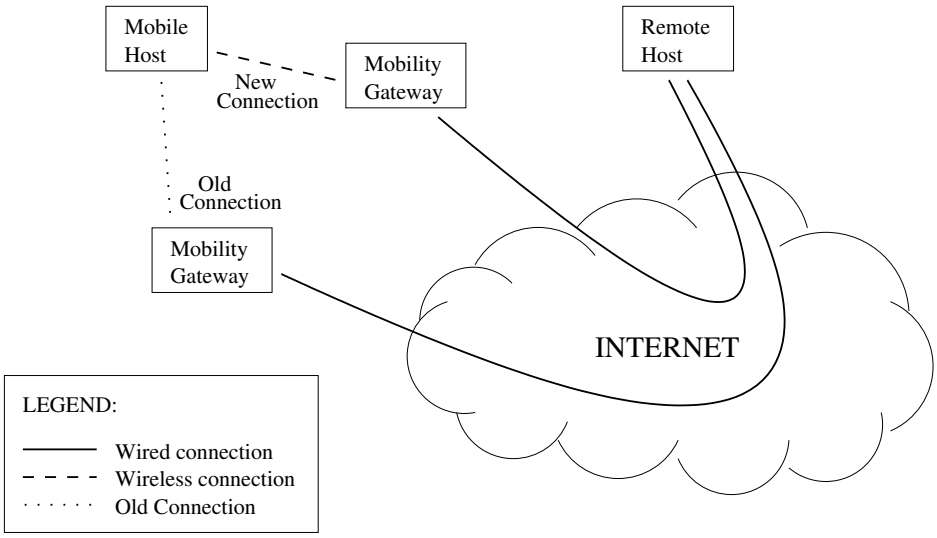


Figure 4.1: The ALICE Mobility Model

requirements on the object-oriented middleware for which ALICE can be instantiated. Next, sections 4.4, 4.5, 4.6 and 4.7 describe in detail the architecture's four layers. Section 4.8 gives the dependencies between six the requirements and the ALICE layers and discusses configuration of the architecture. Finally, section 4.9 summarises the chapter.

4.1 Conventions and Terminology

This section provides background information required to understand ALICE. The mobility model on which ALICE is based is defined and non-standard notation and terminology used to describe the architecture is explained.

4.1.1 Mobility Model

The model for communications used in ALICE is the gateway model discussed in section 2.2.2. In ALICE, mobile hosts connect to remote hosts via sophisticated proxies called mobility gateways. This is shown in figure 4.1. The terminology is as follows:

Mobile Host (MH): A device that moves between mobility gateways and can be disconnected while in transit. The connection between an MH and a mobility gateway may be wired or wireless. An MH can support programs that act as servers as well as clients.

Mobility Gateway (MG): A computer that acts as a proxy for MHs. It has at least one interface through which an MH can connect in addition to at least one wired network connection to the Internet or a LAN. The network addresses for an MG are assumed to change very rarely.

Remote Host (RH): A computer with which an MH communicates via an MG as mediator. An RH can contain client as well as server programs. It can be a fixed host or a mobile host. In the former case, it is assumed to be unaware that its corresponding party is a mobile host. In the latter case, it is assumed to communicate via an MG.

When it is preferable for an MH to connect directly to the network rather than go through a mediator, the MH can act as its own MG. An MG can relay connections from MHs to RHs and vice versa.

4.1.2 Mobility Awareness

Section 2.6.3 discussed the trade-off between transparency and awareness of the problems related to mobility. Both approaches had their merits depending on application requirements. While the simpler programming model offered by transparent mobility support was suitable for some (particularly, legacy) applications, awareness of mobility-related problems allowed other applications to adapt dynamically to changing conditions and thereby improve their performance and hence the usability of the MH.

ALICE provides support for both categories of applications by providing Application Programming Interfaces (APIs) that are compatible with industry standards (e.g., Berkeley Sockets) but which contain extra functions. Mobility-unaware applications can avail of some of the mobility support offered by the architecture, without sacrificing transparency, by using those portions of the ALICE API that correspond to the standard industry APIs, but not the extensions. In comparison, mobility-aware applications can use the extended ALICE APIs to monitor and affect the state of connectivity and the performance of individual communications interfaces on the MH.

On Satyanarayanan's range of adaptation strategies (figure 2.3), the support offered by ALICE would range from the application-transparent end to somewhere on the middle of the scale. This is shown in figure 4.2.

4.1.3 API Notation and Terminology

Each of the ALICE layers consists of a number of components. Each component has up to three different APIs: 'downcall,' 'upcall' and 'tuning' APIs. If a component performs services for layers

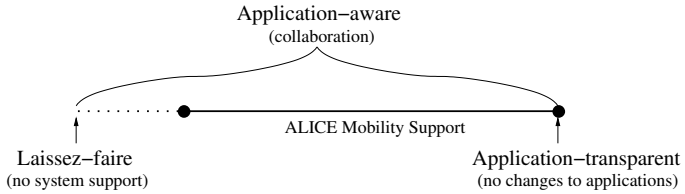


Figure 4.2: The Range of ALICE Adaptation Strategies

above it in the protocol stack, it will have a 'downcall' API through which invocations from upper layers are received. If a component receives upcalls (or callbacks) from layers below it, it also has an 'upcall' API through which this happens. If a component is runtime configurable (as some of the ALICE components are), it has an additional 'tuning' API for this purpose.

The notation used for describing the interaction between the ALICE components is an extension of the traditional protocol stack notation, where the components are placed on top of each other and communication takes place vertically between adjacent components. The notational extensions to this model include the addition of small boxes onto the edges of the components to illustrate the three types of APIs. This is shown in figure 4.3 where two components, each having all three types of APIs, interact.

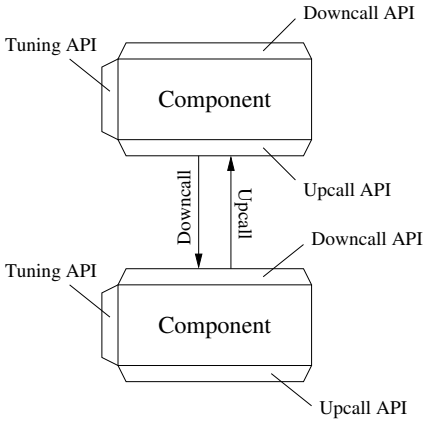


Figure 4.3: Downcall, Upcall and Tuning API Notation

4.1.4 Layer Notation and Terminology

In total, the ALICE protocol stack consists of five different layers, four of which are considered part of the architecture and one of which is considered part of the object-oriented middleware framework for which ALICE is being instantiated. Each of the five layers consists of a number of components

residing in different places. For example, one ALICE layer is called the Mobility Layer (ML) and consists of two components: one residing on the MH and one on the MG. When discussing an entire layer, it will be referred to either by its full name (e.g., the 'Mobility Layer') or its abbreviated name (e.g., the 'ML'). When discussing a single component of a layer, the location at which that component resides (MH, MG or RH) will be subscripted. For example, the part of the Mobility Layer that resides on the mobile host will be referred to as the ML_{MH} component. For components that are specific to the role of client or server rather than a particular location, the initial letter of the role (C or S) played is subscripted instead.

4.2 Architecture Overview

Figure 4.4 shows the ALICE architecture in its entirety. The majority of the ALICE components reside on the MH and MG because the RH is generally not assumed to be mobility-aware.

At the lowest level, a series of transports (TP1 to TP3 in figure 4.4) are used to facilitate communication between the MH and the MG. Each transport typically corresponds to a physical communications interface, such as Bluetooth or InfraRed. The transport modules are managed by the ML which offers a common sockets-like interface to the layers above it. In this way, the ML and the transport modules collaborate to address problems related to mobile networking (e.g., network heterogeneity and short-term disconnections) discussed in section 2.4 and one case of the address migration problem (section 2.5.1): the situation where an open connection between the client and server already exists. (For new connections, the address migration problem is dealt with by the Swizzling Layer discussed below.) The transport modules and the ML are described in sections 4.4 and 4.5 respectively.

On top of the ML sits the Remote Invocation Protocol (RIP) Layer. It has two components that reside on the MH and RH respectively. The RIP Layer implements the remote invocation protocol used in the object-oriented middleware architecture for which ALICE is being instantiated. The RIP Layer is not considered part of the ALICE architecture but rather a standalone software layer designed to interface to the network via the Berkeley Sockets API. On both the MH and the RH, the RIP Layer is typically part of a standard implementation of the object-oriented middleware architecture in question. On the MH, the RIP Layer is placed on top of the ML instead of a TCP/IP stack. The ML facilitates transparent proxying to the corresponding RIP Layer on the RH. Because it is not part of ALICE, this layer is not described further in this chapter. Examples of two RIP Layers are discussed in chapter 5.

Above the RIP Layer resides the Swizzling Layer. Its two components reside on the MH and

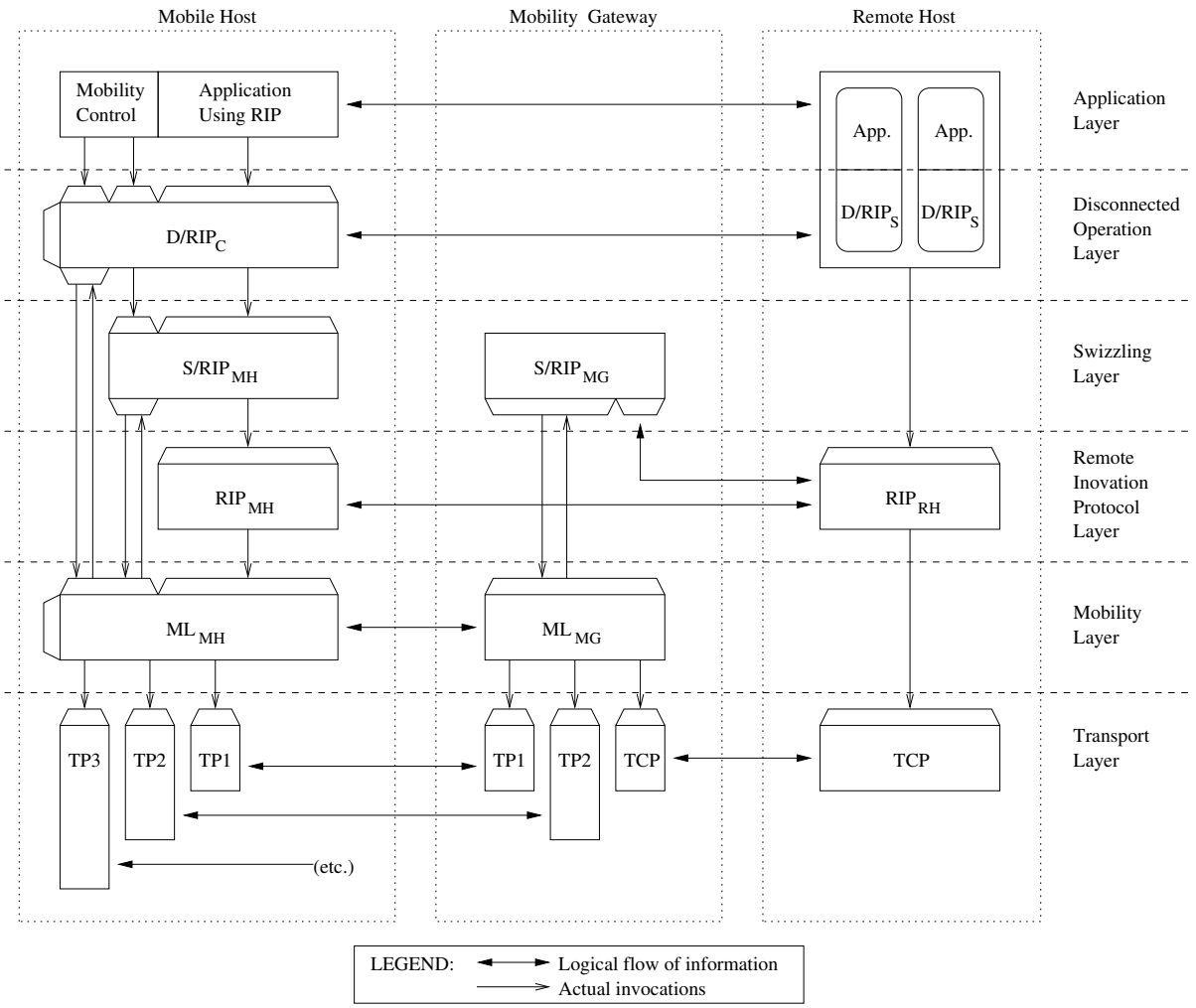


Figure 4.4: The Abstract ALICE Architecture

MG respectively. On the MH, the Swizzling Layer implements the same API as the RIP Layer and is therefore specific not only to the object-oriented middleware architecture in question but also to the specific implementation of that architecture. The role of this part of the Swizzling Layer is to intercept those downcalls from applications to the RIP Layer that have to do with references to server objects on the MH and to translate them at key points in time. On the MG, the Swizzling Layer receives invocations destined for servers on the MH and redirects them towards the server at its current location. This part of the Swizzling Layer is specific to the object-oriented middleware architecture in question but independent of the implementation being used on the MH. Together, the two components of the Swizzling Layer address one instance of the address migration problem (section 2.5.1): where a client holds a reference to a server but has yet to open a connection. (For existing connections, the

address migration problem is dealt with by the Mobility Layer discussed above.) The Swizzling Layer is described in section 4.6.

Above the Swizzling Layer resides the Disconnected Operation Layer. It has two components placed on the client and server side respectively. In figure 4.4, the client is shown to be on the MH and the server on the RH, but the reverse configuration is also possible. On the client side, the Disconnected Operation Layer implements the same API as the RIP Layer in a similar way to the Swizzling Layer. The client-side part of the Disconnected Operation Layer manages a cache of replicas of remote server objects and also has the ability to intercept outgoing invocations and redirect them to the replicas when the server object is not available. On the server side, the Disconnected Operation Layer facilitates replication of server objects and reconciliation of modified replicas with their authoritative server objects. Typically application-specific functions are required to perform replication, conflict detection and resolution, and for this reason, figure 4.4 shows the Disconnected Operation Layer on the server side (RH) as embedded in the application. The two parts of the Disconnected Operation Layer collaborate to address one of the problems related to mobile networking (the problem of long-term disconnection) discussed in section 2.4. The Disconnected Operation Layer of the architecture is described in section 4.7.

At the top of the figure resides the application that uses the mobility-augmented object-oriented middleware framework in question. Such applications may be aware or unaware of mobility as required. In neither case are they considered part of ALICE.

Table 4.1 summarises the responsibilities of the different ALICE layers by relating each layer to the fourteen mobility challenges. The notation is that also used in the summary tables in chapter 3 with the addition of the asterisk (∗) that marks three of the five challenges related to the nature of mobile devices. These challenges—battery power, storage capacity and processing power—require an actual implementation of the ALICE layers in order to be discussed meaningfully. Because this chapter describes the *abstract* ALICE architecture, it makes little sense to discuss the applicability of the abstract layers to these challenges. Instead, we postpone the discussion of how ALICE addresses these challenges to chapter 5 which presents specific instantiations of the ALICE layers.

4.3 Middleware Architecture Requirements

One of the most basic functions of object-oriented middleware is to allow clients to perform invocations on remote server objects. To assure interoperability between implementations of the same middleware architecture, each architecture specifies a protocol used to perform such remote invocations. The

General Category	Specific Challenge	Transport Modules	Mobility Layer	Swizzling Layer	Disconnected Operation Layer	Full ALICE Support
Mobile Devices (section 2.3)	Battery Power	*	*	*	*	*
	Data Risks					
	User Interface					
	Storage Capacity	*	*	*	*	*
	Processing Power	*	*	*	*	*
Mobile Networking (section 2.4)	Network Heterogeneity	○	○			●
	Disconnection		○		○	●
	Low Bandwidth		●			●
	Bandwidth Variability		●			●
	Security Risks					
	Usage Cost		○			○
Physical Mobility (section 2.5)	Address Migration		○	○		●
	Location-Dependent Information					
	Migrating Locality					

Table 4.1: Applicability of ALICE Layers to Mobility Challenges

protocol specification describes the format of the messages and how they are exchanged between the client and the server. Some of the ALICE layers work by transparently changing the dialogue between client and server and by translating server references at key points in time. In order for these changes to be possible, the remote invocation protocol and the server reference format in question must fulfil five requirements. In addition, a sixth requirement relates to the support for object mobility offered by the object-oriented middleware architecture in question. Table 4.2 lists the six requirements.

As a preliminary note, it can be observed that the six requirements are satisfied to different extents by different object-oriented middleware architectures. CORBA is an example of a object-oriented middleware architecture whose remote invocation protocol, the Internet Inter-ORB Protocol (IIOP), and server reference format, the Interoperable Object Reference (IOR), fulfil requirements R1–R5. However, CORBA offers poor support for object mobility and therefore does not easily satisfy requirement R6. In comparison, Java RMI is an example of an object-oriented middleware architecture that offers good support for object mobility (through its adoption of a homogenous execution environment) and therefore also easily satisfies requirement R6. Chapter 5 discusses how ALICE can be instantiated for both of these architectures. The following sections discuss the requirements on the remote invocation protocol (R1–R3), the server reference format (R4–R5) and the architectural support for object

#	Requirement
R1	The protocol must be client/server-oriented.
R2	The underlying transport must be TCP/IP.
R3	Redirection of client requests towards a different server location must be possible.
R4	A server reference must contain several endpoints at which the server can be found. Clients should try endpoints in order.
R5	It must be possible to store some extra information in a server reference. This information must be passed from client to server during invocation.
R6	The object-oriented middleware architecture must support at least weak object mobility.

Table 4.2: Requirements on the Middleware Architecture

mobility (R6) in more detail.

4.3.1 Requirement R1: Client/Server

The ALICE Mobility Layer works by providing an alternative to what is perhaps the most popular programming interface for network programming, the Berkeley Sockets API [175]. While not object-oriented, the Berkeley Sockets programming model is based on the client/server abstraction, and the Sockets interface allows applications to create client and server sockets. The latter distinguish themselves for example by the ability to receive incoming connections. To retain compatibility with the Berkeley Sockets programming model, the Mobility Layer requires the remote invocation protocol used by the object-oriented middleware architecture for which ALICE is being instantiated to be client/server-oriented.

This is not an unreasonable requirement. Object-oriented middleware is inherently client/server-oriented, and it is therefore natural to expect that the remote invocation protocols specified as part of such architectures will be based on the same paradigm. In principle, it would certainly be possible to conceive of an object-oriented middleware architecture based on a protocol that could not easily be called client/server (perhaps an asynchronous messaging protocol), but this appears somewhat contrived and we do not expect it to be the common case. As will be shown in section 6.4, requirement R1 is indeed satisfied by popular object-oriented middleware architectures.

4.3.2 Requirement R2: TCP/IP

As discussed above, the ALICE Mobility Layer acts as a replacement for the Berkeley Sockets layer. Generally, the Sockets API allows the client and server to send and receive data via the transport protocols UDP and TCP, both of which operate on top of IP. The three protocols (UDP, TCP and IP) constitute the Internet Protocol suite, which is by far the most dominant protocol suite for inter-

networking computers. It is therefore reasonable to expect the majority of object-oriented middleware architectures to support the transfer of invocations over either TCP or UDP.

UDP offers a connection-less, unreliable transport service and is often used for applications where real-time performance requirements are more important than guaranteed delivery of data. Examples include media streaming services and networked computer games. In comparison, TCP offers a connection-based, reliable transport service and is typically used for applications where data integrity is more important than timely delivery. Examples are the transfer of electronic mail and web content. For remote invocation protocols, data integrity is generally a high priority, and such protocols are therefore typically based on TCP rather than UDP. Hence, while they may also support other protocols, it seems reasonable to expect the majority of object-oriented middleware architectures to at least support the transfer of invocations over TCP. As section 6.4 will show, requirement R2 is also satisfied for popular object-oriented middleware architectures.

It is worth noting that there is in principle nothing wrong with ALICE supporting other transports than TCP, such as UDP or non-Internet transport protocols. However, requirement R2 simplifies the design of the Mobility Layer. By restricting ALICE to TCP-based remote invocation protocols, the Mobility Layer has to consider only connection-based communication and only protocols based on the Berkeley Sockets API. Support for UDP would require the Mobility Layer to also implement connection-less communication. While this would certainly be possible, it is doubtful whether it would add value to ALICE, since remote invocation protocols are generally easier to build on top of TCP than UDP. Alternatives to the Internet Protocol suite could also be supported, if the ML was made to implement industry standard APIs for such protocols. Adding support for UDP would relax requirement R2, and generalised support for other transport protocols would remove requirement R2 completely. Section 7.3 discusses how this could be done in future versions of ALICE.

4.3.3 Requirement R3: Redirection

Section 2.5.1 listed four general approaches to dealing with the address migration problem: selective broadcast, central services, home bases and forwarding pointers. The systems reviewed in chapter 3 were shown to employ variations of these approaches to allow mobile devices to be located. In relation to object-oriented middleware, the address migration problem must be solved in order to support servers on mobile devices. Clients must be able to locate a server whose address has changed.

The four approaches discussed in section 2.5.1 are generally used in one of two ways: through explicit lookup or through redirection. When using explicit lookup, the client holds an identifier for the server which does not specify the server's location. To invoke the server, the client must query

a location service to obtain the server's current location before it can attempt the invocation. The location service would map the identifier to an address. Such location services are typically based on one of the four approaches discussed in section 2.5.1. An example of a system using the explicit lookup approach is Snoeren's DNS extension discussed in section 3.2.1. In Snoeren's system, the client maps a location-independent identifer (the hostname of the machine where the server resides) through a lookup operation (the DNS resolution) to a server location (the current IP address of the server's machine).

When using redirection, the client holds an indentifier that contains a (possibly obsolete) location of the server. The client attempts to invoke the server with no preceding operation to find the server's current location. If the server does not reside at the expected location, an entity residing at that location supplies a new location to the client, i.e., performs a *redirection* of the client. The client is expected to retry the invocation at the new location. The redirecting entity would typically track the server based on one of the four approaches discussed in section 2.5.1. This type of approach was adopted in Wireless CORBA discussed in section 3.4.2. In Wireless CORBA, addresses of access bridges constitute server locations. Clients invoke the access bridges which have the ability to redirect the clients to other access bridges where the server may reside or which may have more recent information about the server's whereabouts.

In both the explicit lookup and the redirection case, an operation is performed to obtain the server's location, but the two approaches differ with respect to the client's awareness of this operation. The explicit lookup approach requires the client always to perform the lookup operation itself, while the operation in the redirection approach is performed transparently to the client as part of an invocation attempt. Both approaches require extra functionality on the client side compared to the ability to perform 'bare' remote invocations. This extra functionality takes the form of code to either perform explicit lookup operations or deal with redirections. In addition, both approaches would typically need to include a client-initiated retry facility in case the invocation fails. In particular, the explicit lookup approach is subject to a race condition between server location updates and client lookups, if no retry facility is provided. Hence, in terms of increased client complexity, the explicit lookup and the redirection approaches do not seem to differ significantly.

In order to solve the address migration problem, ALICE needs to adopt a tracking scheme based on the four approaches discussed in section 2.5.1. This scheme must be incorporated into the object-oriented middleware architecture for which ALICE is being instantiated either via an explicit lookup or a redirection mechanism. In general, redirection mechanisms have other uses than to support mobile computing and are often supplied in application-level protocols as general-purpose mechanisms. For

example, a load balancing cluster of web servers can be built based on the redirection mechanism found in the Hypertext Transfer Protocol (HTTP) [62]. A simple configuration would involve one dispatcher node to receive all requests and redirect them (e.g., using the 307 Temporary Redirect response) on a per-request basis to the least loaded of a set of identical web servers. Object-oriented middleware architectures are typically intended to support large distributed applications where such replicated services play a significant role, and we therefore expect most architectures to contain some sort of redirection mechanism. As will be shown in section 6.4, this is in fact the case for a number of popular object-oriented middleware architectures.

4.3.4 Requirement R4: Multi-Endpoint References

When using the redirection approach discussed above, clients will retain server references that contain (possibly obsolete) information about the server's location. In case the server no longer resides at the given location, a redirecting entity will supply a new location identifier to the client. The redirecting entity is critical because its *presence* and *availability* is required for the client to perform the invocation.

In relation to the *presence* of redirecting entities, a problem with the approach is that it requires a redirecting entity to reside at every past location of every mobile server for as long as server references containing that redirecting entity exist. For large mobile environments with a considerable number of server locations (in ALICE in the form of MGs), a large number of mobile servers (residing on MHs) and frequent mobility of hosts, we expect a considerable number of redirecting entities to exist at any point in time. These entities cannot exist indefinitely; it is important for the system's scalability that they be recycled when their presence is no longer required. The issue can be addressed in a number of ways. One approach is to track the number of references held by clients (through reference counting) and destroying redirecting entities for which no references exist. This approach requires communication between clients and redirecting entities every time references are created, destroyed and copied. Another approach is to limit the time for which a given reference is valid (leasing) and allow redirecting entities to be deleted when all references that refer to them are known to have expired. The approach requires less communication between the parties than the reference counting approach but instead requires clients to selectively renew server reference leases at various intervals. Hence, both approaches require considerable complexity in relation to reference management, especially on the client side. An alternative approach is to provide clients with a fallback mechanism in the form of an additional endpoint containing the location of a secondary entity that can perform the redirection. This approach allows redirection entities that are rarely used to be deleted and the clients automatically to fall back on a secondary entity which may still be able to perform the redirection

(possibly at reduced performance), for example by initiating a comprehensive search for the server in question. Compared to reference counting and leasing, this is a lazier approach (references are not tracked eagerly) that may result in more obsolete redirection entities being retained but at the cost of considerably less complexity and communication.

In relation to *availability* of the redirecting entities, the problem is similar. Redirecting entities may be temporarily unavailable for a variety of reasons, such as hardware, software or network failure. This may result in situations where servers that are functioning perfectly are effectively unavailable because of failed redirecting entities. Possible solutions to this problem include providing a successive retry mechanism in the client (assuring reasonable recovery time despite temporary unavailability) and/or a fallback mechanism, for example by including the locations of multiple redirecting entities in the server reference. The latter approach was adopted in Wireless CORBA (section 3.4.2) where a forwarding pointer scheme was combined with a home agent approach, resulting in two redirecting entities in each server reference. The multi-endpoint approach also has the advantage that it can address the problem of garbage collection of redirection entities discussed above, because several levels of fallback mechanisms can be specified as different endpoints.

As opposed to the additional complexity in relation to server reference management required for reference counting or leasing, we expect that the requirement for multiple endpoints in a server reference and the expectation that clients try the endpoints in sequence constitute a relatively simple requirement that is likely to be satisfied by most object-oriented middleware architectures. As discussed above, object-oriented middleware is typically designed with large-scale distributed applications in mind. To illustrate, we consider the web server cluster discussed in relation to requirement R3 as an example of a distributed application that uses replication to improve scalability. However, the cluster still has a single point of failure in the form of the entity performing the redirection. If fault tolerance is a higher priority than scalability for the application, a better approach may be to disclose the multiple server locations to the client, typically by including a series of server endpoints (rather than just one) in the server reference. Each endpoint represents a location where an object implementing the service can be found. The client holding a multi-endpoint reference can try the endpoints in sequence, and if one instance of the server is not available, the client can try the next until one succeeds. By moving the redirection task to the client, the single point of failure (the redirector) is eliminated, resulting in better fault tolerance. However, it is worth noting that there is a trade-off involved. For typical applications, the clients are unlikely to maintain details of the performance and availability of individual servers and will therefore not be able to choose the optimal server. Hence, while letting the clients perform the redirection may improve fault tolerance, the approach is unlikely to perform better

load balancing than the redirector-based approach. Scalability and fault tolerance are classic desirable qualities of distributed systems, and object-oriented middleware is typically intended to support applications where either characteristic is of primary concern. We therefore expect object-oriented middleware architectures to generally allow server references to contain multiple endpoints. As we will show in section 6.4, this is indeed the case for a number of popular architectures.

4.3.5 Requirement R5: Extra Information

A redirection scheme as discussed above requires the redirecting entities (the MGs in the case of ALICE) to receive invocations on behalf of mobile servers, to generate new server references and to return such references to invoking clients. As discussed above, it is important that the server references include several endpoints, on which clients can successively fall back in case of failures of redirecting entities. This characteristic applies for all references, irrespectively of how they were obtained by the client, e.g., from a naming service, in an invocation response, etc. However, particular attention must be paid to references that are generated by redirecting entities and returned to clients in redirection responses. For any redirection, the redirecting entity that generates a new reference must have sufficient information about all endpoints in the old reference to construct a new reference that contains the same endpoints. This is a problem because most object-oriented middleware architectures clients do not pass the full server reference to the server during an invocation. This information is typically not required by the server, because the server typically already knows its own reference.

There are two approaches to this problem. One would be to require mobile servers to supply information about their full references to any redirecting entity that they visit. Redirecting entities would then store this information such that it can later be used to construct new references. However, this is an unattractive solution, because it requires the redirecting entity either to maintain the details for past mobile servers indefinitely or implement a reference counting or leasing scheme as discussed above. Maintaining the details indefinitely is a clearly unscalable solution, and as discussed above, the other approaches come with a considerable increase in complexity, and it is far from obvious that they will harmonise well with typical object-oriented middleware architectures. The other approach is to require the client to supply the details during the invocation. This seems like a natural solution, because the client already holds a server reference containing the details and is already involved in communication with the redirecting entity at the time when the endpoint details are needed.

The endpoint details can be transmitted from the client to the server in two manners. One possibility is to let the MG request the endpoints (or the full server reference) explicitly from the client before the invocation goes ahead. This approach gives the redirecting entity access to any

details it might need. However, this behaviour essentially involves the client and server switching roles (the server is requesting information from the client) and therefore constitutes a significant extension to the standard client/server dialogue. Also, this type of functionality has few general applications (compared for example to the multi-endpoint reference formats discussed above) and would therefore not necessarily be expected to exist in the majority of object-oriented middleware architectures. The other possibility is to include the endpoint details in the request by embedding it in other data that is transmitted from the client to the server as part of the invocation. Typically, object-oriented middleware architectures use object and method identifiers supplied during an invocation that allow the server process receiving the invocation to identify the target server object and the method being invoked. Since this is standard behaviour in object-oriented middleware, this approach is more likely to be generally applicable than the idea of switching the two parties' roles. In this context, it is worth noting that both approaches were employed in Wireless CORBA (section 3.4.2), but that the feature which allowed the server to request its own reference from the client had only recently been introduced in the latest version of the CORBA remote invocation protocol.

Using the data fields already included in server references also has another advantage. Generally, a TCP-based server process is identified by the hostname and port number on which the server is running. The data supplied as part of an invocation request typically includes an object identifier that allows the server process to dispatch the invocation to the appropriate server object. However, object identifiers are not necessarily unique across server processes. For the purposes of redirection, this means that each mobile server must maintain a separate redirecting entity (running on its own port) at each location it has visited. While this is not necessarily a problem, it does seem somewhat wasteful. If the data fields are being modified for other purposes as discussed above, it seems natural to also include an identifier for the mobile server process such that a single redirecting entity can serve multiple mobile servers.

On the basis of this discussion, we require that it be possible to store some extra information in a server reference and this information must be passed from client to server during invocation. As we will show in section 6.4, this is a reasonable requirement, which is satisfied for a number of popular middleware architectures.

4.3.6 Requirement R6: Object Mobility

Chapter 3 discussed two systems—Coda (section 3.3.1) and Rover (section 3.3.2)—that feature good support for long-term disconnections. Both are based on the idea of transferring server functionality (files, in the case of Coda; objects, in the case of Rover) to the client side for use when the mobile

device is disconnected. Both also include a reconciliation step where modified server state (files or objects) is transferred back to, and reconciled with, the server. Given the positive experiences with the Coda and Rover systems, we have chosen to base ALICE support for disconnected operation on the same principles.

In the case of object-oriented middleware, this type of scheme requires that it be possible to replicate server objects and exchange them between the client and server. Compared to files, replicating and moving objects is typically more difficult, because objects can contain code and state that cannot easily be copied across machine boundaries. Examples of potentially difficult entities to copy include some types of attributes (e.g., pointers and references to objects in memory), member functions and computation (e.g., running threads). Object mobility is generally divided into weak and strong mobility. Weak mobility systems, such as Java [71], allow code and data to be moved. Strong mobility systems, such as Emerald [99], also allows computation (i.e., any running threads there may be in the object) to be moved.

In order to faciliate support for disconnected operation, ALICE requires the object-oriented middleware architecture in question to provide at least a weak object mobility model. It is also possible to instantiate ALICE for object-oriented middleware architectures that provide strong mobility, but strong mobility is not a requirement. This is not an unreasonable requirement. As will be shown in chapter 5, it is possible to provide weak object mobility within several popular object-oriented middleware architectures.

4.4 Transport Modules

As discussed in section 2.4.1, mobile computers despite hardware-imposed limitations generally offer a wide range of options for connecting to their surroundings. The types of interfaces vary between wired (e.g., Ethernet), wireless (e.g., 802.11), point-to-point (e.g., serial) and those which cover an area (e.g., Bluetooth). The purpose of the ALICE transport modules is to encapsulate each type of communications interface and provide a common programming interface, independent of interface-specific characteristics. Having a common programming interface to the different transports makes it is easier for the ML to manage and use communications interfaces with different characteristics. In addition, it allows new types of communications interfaces to be added to ALICE in the form of new transport modules without requiring modification of the ML or the layers above it.

4.4.1 Transport Services

The transport modules offer a common interface with identical semantics, although the underlying communications interfaces may be different. All transport modules allow connections to be established and data to be exchanged between the MH and MG. Connection attempts are always initiated by the MH. An open connection provides an unreliable datagram service. Transport modules verify the integrity of the datagrams received using a Cyclic Redundancy Check (CRC). Any datagram delivered from a transport module to a higher layer is guaranteed to have passed the CRC integrity check. Datagrams that fail the integrity check are dropped by the transport modules. The transport modules do not reorder datagrams or detect lost or duplicate datagrams. These tasks are performed by the ML in order to allow different datagrams to be sent over different transports. A transport module offers the following operations to the ML:

Start initialises the transport for operation.

Stop stops the transport.

Connect is invoked only on the MH and attempts to connect to an MG. This initiates a session between the MH and MG.

Disconnect ends the session.

CheckStatus $\rightarrow$ **Status** returns a status value (e.g., true or false) signifying the current state of the connection.

Send(Datagram) transmits a datagram over the transport.

Receive $\rightarrow$ **Datagram** receives a datagram over the transport.

4.4.2 Transport Configurations

Each transport module can support a number of actual transports of the type in question. For example, an instantiation of ALICE running on an MH with two serial ports might offer two different serial transports to the ML, each of which uses the code in the serial transport module. This is shown in figure 4.5 where ALICE transports 1 and 2 manage serial communications interfaces A and B respectively. The code segment in transport 2 is dotted, signifying that this transport uses the same code segment as that of transport 1. In comparison, transport 3, being of a different type (Bluetooth) has its own code segment. If the transport modules are implemented in an object-oriented language, a transport module maps naturally onto a class and an actual transport onto an instantiation of that

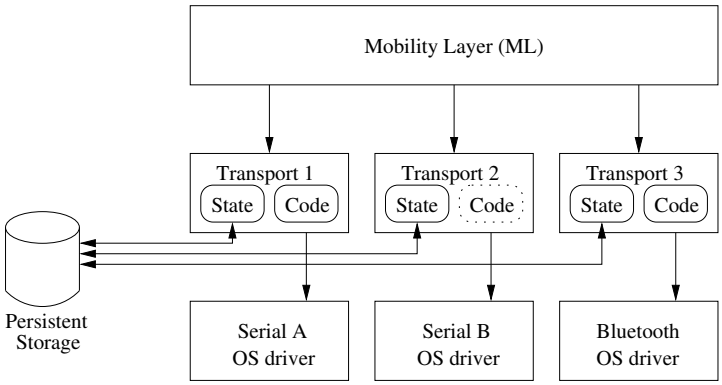


Figure 4.5: Three Communications Interfaces as Three Transports

class. Each of the different transports maintains its own state, which is read from and written to persistent storage as shown in the figure. The state is related to the past performance and current configuration of the transport in question and explained in detail in section 4.4.3. In practice, the majority of MHs have only one of each type of communications interface, and there is typically a one-to-one relationship between transport modules and actual transports.

ALICE also allows the mapping of several physical communications interfaces to a single logical transport. To the ML, the physical interfaces will appear as one single transport. This is shown in figure 4.6 where two physical serial communications interfaces are managed as one single logical transport. This type of configuration hides the network heterogeneity from the ML and prevents it from managing the transports separately. For this reason, it should generally be avoided, unless there is a very tight coupling between the two communications interfaces in question, i.e., the interfaces are generally expected to obtain and lose connections to the same MG at nearly the same time. This could for example be the case for an MH with two identical 802.11 interfaces, which would be expected always to be within coverage at the same time. In this case, presenting them as one transport to the ML can result in improved performance, because the ML does not need to discover (via reconnection heuristics) that the interfaces are related and should be used simultaneously. If two loosely coupled or completely uncoupled interfaces (e.g., a Bluetooth and an 802.11 interface) are presented to the ML as one transport, the ML's reconnection and performance heuristics will suffer, resulting in overall reduced performance.

4.4.3 Transport State

ALICE transport modules have the ability to maintain the state of the transports in question. The state is read from persistent storage when the transport is activated and written back when the

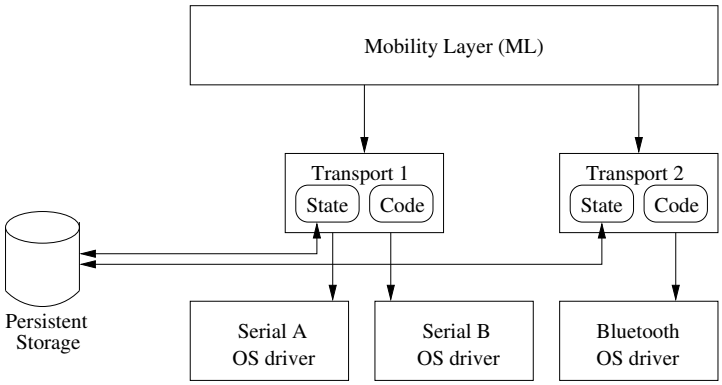


Figure 4.6: Three Communications Interfaces as Two Transports

transport is deactivated. This is shown as horizontal arrows in figures 4.5 and 4.6. The state consists of the following types of data:

- Interface-independent configuration details, such as the symbolic name of the interface and whether it is enabled or disabled. These details are used to manage the transport.
- Interface-specific configuration parameters, such as device names, data rates and error correction parameters. These are used to configure the transport during initialisation.
- Past experience about the performance of the interface in the form of latency, effective throughput and number of disconnections recorded. These are used to compute a quality index of the transport in question.

The state information is made available to the ML and (through the ML) to mobility-aware applications that may wish to control the usage of interfaces. Such applications can control the transport's current configuration via the ML's tuning interface. This is described in section 4.5.

4.4.4 Mobility Challenges

Of the fourteen mobility challenges discussed in chapter 2, the ALICE transport modules focus on one: the problem of network heterogeneity (section 2.4.1). This challenge is addressed by the ML and the transport modules which collaborate to hide the diverse characteristics of the network interfaces from upper layers.

4.5 The Mobility Layer

Connectivity management between the MH and the MG is handled by the Mobility Layer (ML). To layers above it, the ML offers services equivalent to a standard implementation of TCP, but with a number of mobility support extensions. These extensions address the challenges related to mobile networking discussed in section 2.4. The functionality of the ML falls in three parts: transport management, proxy operation and handoff/tunneling features. Section 4.5.1 describes how the ML manages the MH's different communications interfaces and performs reconnection of broken transport connections. Section 4.5.2 describes how the ML allows socket operations performed on the MH to be proxied by the MG. Section 4.5.3 explains how the ML enables an MH to move between MGs without losing connections through the use of handoff and tunneling.

To applications, the ML implements a superset of the standard Berkeley Sockets API. Mobile applications avail of the mobility support offered by the ML by using ML sockets instead of Berkeley sockets. Applications can remain aware or unaware of mobility as suits. Section 4.5.4 discusses how this flexibility is achieved, including how mobility-aware applications receive notification of changes in connectivity of the MH through callbacks from the ML. Section 4.5.5 discusses changes in semantics of ML sockets compared to those of standard Berkeley Sockets. Finally, section 4.5.6 summarises which of the fourteen mobility challenges are addressed by the ML and how.

4.5.1 Transport Management

The ML works by managing a set of transports on the MH and uses them to connect to one or more MGs at various points in time. Each transport typically corresponds to a physical communications interface, encapsulated as a transport module as described in section 4.4. The ML uses the services provided by the transport modules to implement a reliable stream-based service, comparable to TCP. Much like a TCP-implementation, the ML uses sequence numbering to detect lost and duplicate datagrams in combination with an acknowledgement scheme that allows missing datagrams to be retransmitted and received datagrams to be acknowledged.

Data Compression

The ML addresses the low bandwidth problem (section 2.4.3) by compressing the payload of datagrams sent over any transport it manages. A general-purpose lossless data compression library is used for this purpose.

Transport Statistics and Configuration

By using the tuning interface of the ML_{MH} component, an external component running on the MH can obtain information about and affect the operation of the transports managed by the ML. The programming interface allows the external component access to the names and current connectivity status of each interface, as well as details about the past performance of that particular interface in terms of average latency and throughput, number of dropped datagrams and number of disconnections. The programming interface also allows configuration of the interface, for example to enable or disable it, or to reset the collected statistics.

Reconnection Strategy

In normal circumstances, the ML will attempt to maintain the connection between the MH and the MG at all times. When a connection fails, the ML_{MH} component invokes the reconnection algorithm. The purpose of the reconnection algorithm is to identify the most suitable transport over which to reconnect. The ML implements a default reconnection algorithm that follows a simple round-robin scheme in which enabled transports are tried in turn.

Figure 4.7 shows the cyclic reconnection algorithm. The MH has $n+1$ transports named T_0 to T_n. The ML_{MH} component is connected to an MG via transport T_k where $0 \leq k \leq n$. When the connection through T_k fails, the ML_{MH} component will delay for a predefined period d and then try to reconnect through T_k. If that fails, another delay d is performed and the next transport $T_{(k+1) mod (n+1)}$ is tried. If no transport connects successfully, a longer delay D is performed before the initial interface T_k is retried. The motivation for performing the first reconnection attempt through the T_k is that reaching the same MG may prevent a handoff between MGs and thereby reduce overhead. This is for example the case if the transport in question is GSM or GPRS. However, the assumption depends on the transports in question, and in some cases it may be easier to reach the same MG over a different transport. For example, the MH may be communicating with the MG through a wireless interface (e.g., Bluetooth or 802.11) and have moved out of range. In this case, a different transport (e.g., GPRS) may be more successful reestablishing the connection to the MG.

The cyclic reconnection algorithm is rather simplistic and unlikely to be suitable for all scenarios. For this reason, the ML allows an external component (such as an MH configuration tool) to provide its own reconnection strategy. This is done through the ML_{MH} component's tuning interface which lets an application register a function which will be invoked whenever connectivity is lost. The function must select (and return an identifier for) one of the available transports through which the ML_{MH} component will then try to connect. This selection process is facilitated by statistics about the

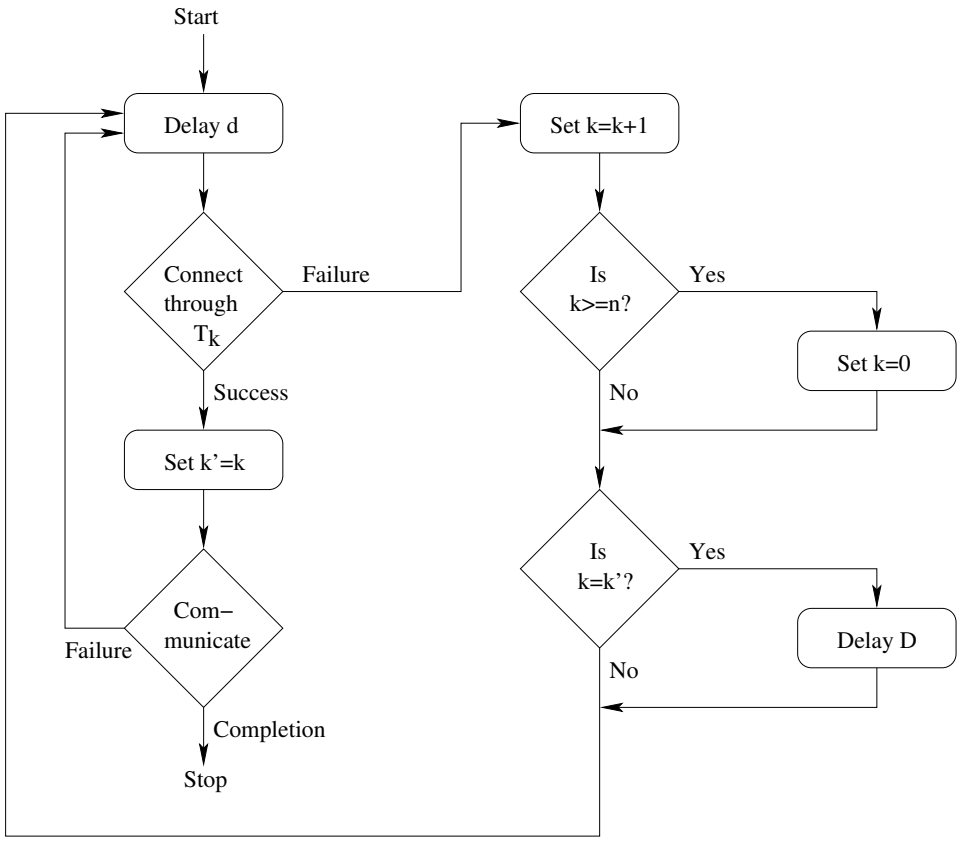


Figure 4.7: Cyclic Transport Selection Algorithm

past performance of the transports available through the tuning interface of the ML_{MH} component. An example of a possible approach to an alternative transport selection strategy based on the past performance of individual transports is given in appendix A. Other approaches could take into account other characteristics, such as the communications cost involved for each particular type of interface. Section 4.5.4 discusses a configuration tool that uses the tuning interface of the ML_{MH} component to register its own reconnection algorithm.

Operation During Disconnection

In some cases, it will not be possible to connect to an MG, and it is normal for an MH not be connected to an MG at all times. In this case, the default mode of operation for the ML is to synchronously queue data to/from the MH. Asynchronous queueing can be achieved by using application-level multi-threading in conjunction with synchronous queueing. Applications that require more complex semantics must implement their own. This is facilitated by the callback features described in section 4.5.4

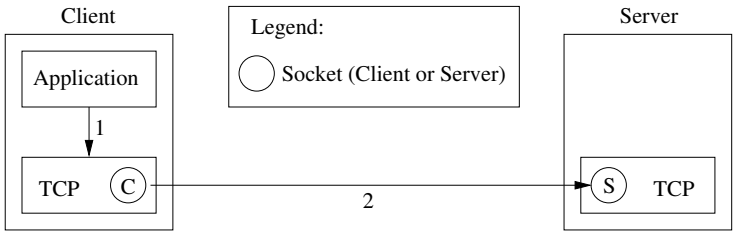


Figure 4.8: Standard Client/Server using Berkeley Sockets

which allow applications to maintain awareness of the MH's state of connectivity. An example of such an application is the Disconnected Operation Layer discussed in section 4.7.

4.5.2 Proxy Operation

Figure 4.8 shows standard client/server communication using TCP sockets. The client application first creates a socket using the `socket(2)`[1] system call. In response, the TCP layer creates a socket and returns a socket identifier to the client. This is shown with arrow 1. The client application then uses the `connect(2)` system call to connect the socket to a (possibly remote) server socket. This is shown with arrow 2.

When the MH is connected to an MG, the MG acts as a proxy for all socket operations performed by mobile applications using the ML. In effect, the ML_{MH} component overrides the socket operations from the standard Berkeley Sockets API and replaces them with functions that instruct the ML_{MG} component to perform the requested operations. For example, a mobile client may use the `socket(2)` system call to create a socket. If linked with a standard implementation of Berkeley Sockets, invoking this system call will create a socket on the MH. If linked with the ML, the socket will be created on the MG and any data sent from the MH will be relayed to the MG by the ML. Similarly, any data received by the MG on the proxy socket will be relayed to the MH.

The two ML components require a simple protocol that allows the MH to use the MG as a proxy for socket operations. The protocol must allow the MH to create client as well as server sockets on the MG and to send and receive data on those sockets. In addition, it must be possible for a server on the MH to receive incoming connections on proxied server sockets that reside on the MG. The following explains the proxying functionality of the ML in the case of a client and a server on the MH respectively. Chapter 5 gives a possible implementation of a protocol that allows this type of interaction between ML_{MH} and ML_{MG} components.

[1] We use the Unix convention of including the manual section in parentheses when discussing system calls.

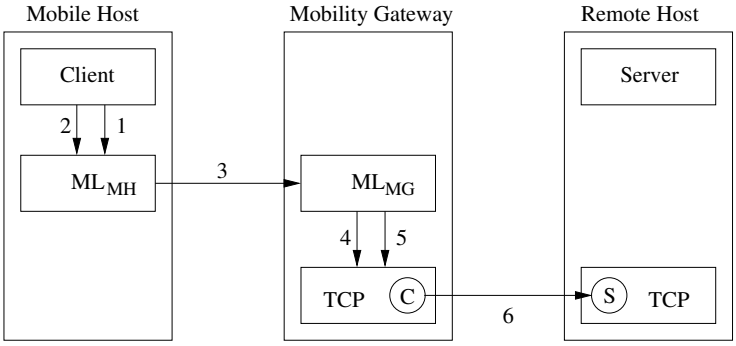


Figure 4.9: Mobile Client using ML Sockets

Client on the MH

Figure 4.9 shows a mobile client communicating with a remote (non-mobile) server through a ML socket. Because ML sockets implement the behaviour of standard Berkeley Sockets, the fact that the communication takes place via an intermediary (the MG) is transparent to the client application. Arrow 1 shows the application creating a socket using the ML's `socket(2)` call. The ML_{MH} component returns a socket identifier to the client application as would a normal TCP implementation. The `socket(2)` call is not relayed to the MG. The arrow marked 2 shows the application invoking the ML's `connect(2)` call to connect the socket to a remote server. Arrow 3 shows this call being relayed to the ML_{MG} component which creates a socket (arrow 4) and issues a `connect(2)` call (arrow 5) using the TCP stack on the MG. Arrow 6 shows the connection finally being made between the TCP stacks on the MG and RH.

Because the socket on the MG is only created when the client issues a `connect(2)` call, no MG is required for the creation sockets on the MH. However, calls to `connect(2)` on the MH (arrow 2 in figure 4.9) require the presence of an MG to complete. If no MG is available, the default behaviour of the the ML's `connect(2)` operation is to block until an MG becomes available.

After connection set-up has been completed, the client can send data to the server. The data follows the path shown by arrows 2, 3, 4 and 6 in figure 4.9. Data sent from the server to the client takes the same route in the opposite direction. The fact that the data is relayed through the MG remains transparent to both parties. No mobility support functionality (such as the ML) is required on the RH, and the server can therefore remain completely unaware that the client is mobile.

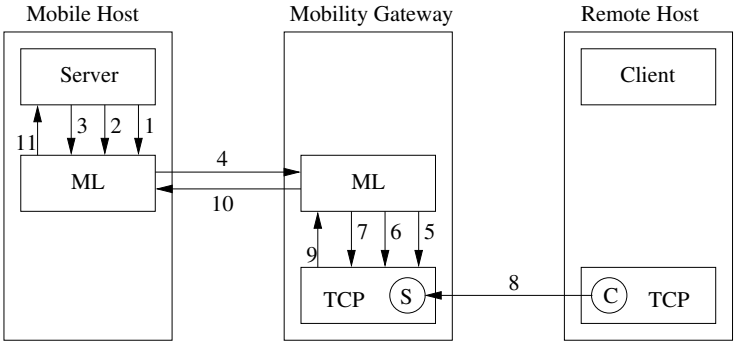


Figure 4.10: Mobile Server using ML Sockets

Server on the MH

Figure 4.10 shows a mobile server creating a socket and receiving a connection from a remote (non-mobile) client. The server invokes the ML_{MH} component's `socket(2)` function to create the socket (arrow 1) and `bind(2)` (arrow 2) to bind to allocate a port number to it. The ML_{MH} component performs these functions without involving the MG. Next, the server invokes the ML_{MH} component's `listen(2)` operation (arrow 3) to place the socket in listening mode. This call is relayed to the ML_{MG} component which creates the socket (arrow 5), binds it to a dynamically allocated port (arrow 6) and places it in listening mode (arrow 7) using the TCP stack on the MG. After the socket has been placed in listening mode, client connections can be received.

Arrow 8 in figure 4.10 shows the remote client connecting to the listening socket on the MG. This causes the file descriptor associated with the socket to change status. The ML_{MG} component will detect this (e.g., using the `select(2)` system call) and issue an `accept(2)` call to receive the connection (arrow 9). Arrow 10 shows the connection from the remote client being relayed to the ML_{MH} component where the ML socket descriptor changes state to reflect the new connection. Arrow 11 shows the mobile server receiving a file descriptor for the new connection as the result of an `accept(2)` call.

It is worth noting that the socket on the MG is only created, bound to a port and placed in listening mode when the mobile client issues a call to `listen(2)`. The effect of delaying these operations is that the use of mobile server sockets does not require the presence of an MG until the `listen(2)` operation is invoked.

An MG may serve multiple MHs at any point in time and may also run other applications than those required for ALICE mobility support. For this reason, proxy sockets are allocated on the MG dynamically. The ML_{MG} component can therefore make no guarantees to mobile applications

about the availability of specific port numbers. This has little impact on mobile clients because standard behaviour for TCP implementations is to allocate client port numbers dynamically and without involvement of the client application. However, for server sockets the port number is typically used to identify individual server processes, and there are conventions for which port numbers are used for common services. For example, port 25 is often used for mail and 80 for web servers. As opposed to a server using standard Berkeley Sockets, a mobile server using the ML can make no assumptions about the port number that will be allocated to it. Furthermore, the network interface on which the server will actually be listening will belong to the MG and not to the MH. The server's lack of control over its interface and port number constitutes a change to the semantics of Berkeley server sockets. This is further explored in section 4.5.3 and addressed through the Swizzling Layer discussed in section 4.6.

4.5.3 Handoff and Tunneling

As described in section 4.5.1, the ML_{MH} component will try to reconnect to an MG whenever the connection to the ML_{MG} component is lost. In some situations, it may be possible to reconnect to the same MG in which case communication can simply be resumed. For example, a given MH↔MG connection using a GSM modem transport may break but can be restored independently of the MH's location as long as GSM coverage is available. However, in many situations, it may not be possible to connect to the same MG. For example, short range Radio Frequency (RF) interfaces, such as Bluetooth or 802.11, have limited coverage, and connections may break as a result of the MH moving out of range of the MG. In such cases, it may be possible for the MH to obtain a connection to a new MG but not to the old. The migration of an MH from one MG to another is handled by the ML through a process called 'handoff.' Upon connection to the new MG, the MH identifies itself and requests that the new MG initiate a handoff between the old and the new MG. Similar approaches were adopted in DOLMEN (section 3.4.3) and Wireless CORBA (section 3.4.2).

An ML handoff has two components: the transfer of state related to sockets being proxied and the creation of tunnels to allow existing connections to persist despite host mobility. The following discusses each in turn.

Socket State Migration

An MG acting as a proxy for an MH maintains state for any sockets being proxied. For sockets that have active connections, this state includes unsent data and data that has been sent but not acknowledged. For server sockets, the state also includes information about the network interface and

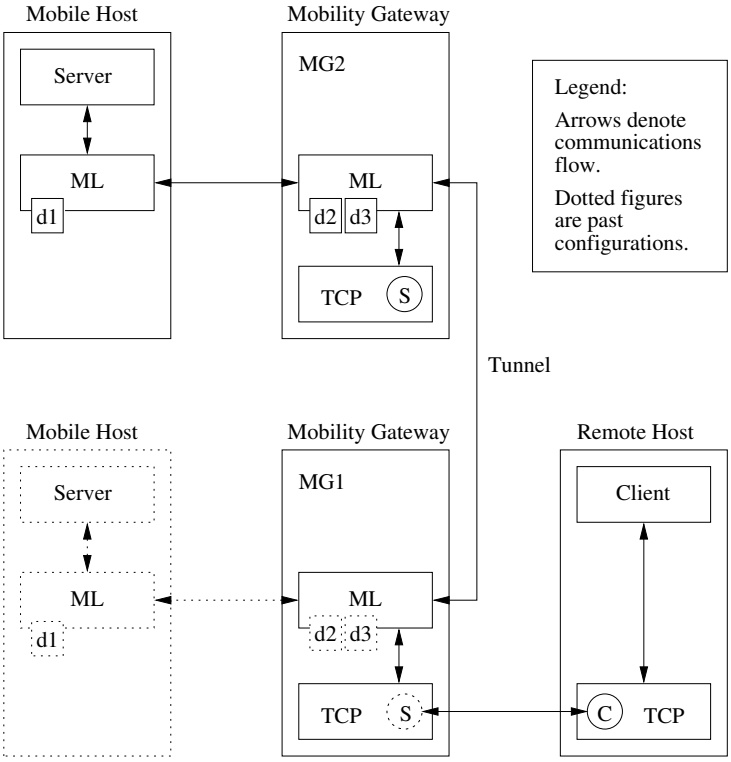


Figure 4.11: Handoff and Tunneling

port number on the MG and an identifier that identifies the server on the MH. During handoff, this state is transferred from the old to the new MG.

Figure 4.11 shows a scenario where an MH moves from MG1 to MG2. The MH has one open connection to an RH through MG1 and one server socket hosted on MG1. At the time when the connection between the MH and MG1 is broken, there is one unit of unsent data (d1) residing on the MH and two (d2 and d3) on MG1. When the MH connects to MG2, it requests a handoff between the MGs. This causes the unsent data units (d2 and d3) stored on MG1 to be transferred to MG2. On MG2, a server socket is created to replace the server socket on MG1. The server socket on MG1 is then destroyed. (The socket on MG1 that connects MG1 to the RH remains.)

As discussed in section 4.5.2, the mobile server's proxy port on the MG was allocated dynamically. In addition, the migration of the MH hosting the mobile server causes the server port to be reallocated dynamically on the new MG. Hence, mobile server ports are not only allocated without control by the mobile server but are also subject to potentially frequent reallocation. This transience is essentially the address migration problem (section 2.5.1), except it involves the mobility of server sockets rather

than hosts. The ML contain features to deal with the address migration problem for open connections (through tunneling, as explained below) but it contains no features that can assist a client with locating a server socket that has been moved. Hence, the ML does not attempt to solve the address migration problem for new connections. Instead, this instance of the address migration problem is left to be addressed by the Swizzling Layer described in section 4.6

Connection Tunneling

In addition to unsent data and server sockets that are passed between MGs during handoff, the MH may have open connections to RHs going through the old MG. Ideally, these connections would be migrated to the new MG as part of the handoff procedure. This is a viable approach if both the MGs and the RH are using IPv6, because IPv6 allows IP addresses to be rebound. In case one of the three parties is using IPv4, it is not possible to migrate endpoints to the new MG, and any open TCP connections will therefore still have to go through the old MG. For this reason, the ML allows tunnels to be set up between the old and the new MG. The tunnels allow the data to be forwarded transparently between applications on the MH and the RH at the cost of the extra leg of communication between the two MGs. A separate tunnel is set up for each connection between the MH and an RH, and each tunnel is maintained for as long as that connection persists. Figure 4.11 shows a tunnel being set up between two MGs as a result of the MH moving. After the tunnel has been set up, any data sent from the MH to the RH over the open connection will be received by the ML_{MG} component on MG2 and tunneled to the ML_{MG} component on MG1 from where it is forwarded to the RH. Data sent over the connection from the RH takes the same route in the opposite direction.

If the MH returns to MG1, the tunnel will be destroyed and the extra leg of communication eliminated. If the MH moves to a third mobility gateway MG3, this MG will initiate a handoff from MG2. If tunnels still exist between MG1 and MG2, MG2 will notify MG3 of this, and MG3 will initiate a tunneling request to MG1. This causes the MG1–MG2 tunnel to be destroyed and a new MG1–MG3 tunnel to be created. This assures that tunnels always have a length of 1.

4.5.4 Mobility Awareness

The mobility support provided by the ML can be used in a transparent or a non-transparent manner as preferred by the application. Applications that wish to remain unaware of problems related to mobility (such as legacy client/server applications) use the ML as they would an ordinary TCP layer. To such applications, the ML's mobility support features remain transparent and they use only the portion of the ML's downcall API that corresponds to standard Berkeley Sockets functions. A configuration

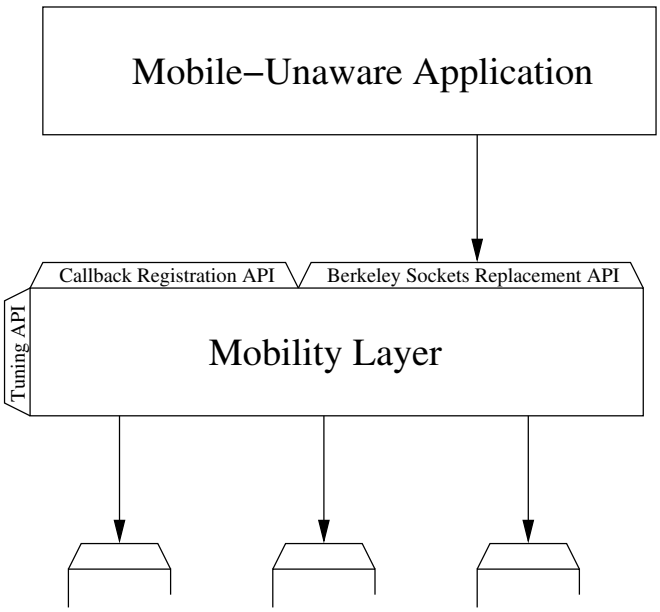


Figure 4.12: Mobility Layer with Mobility-Unaware Application

involving this type of application is shown in figure 4.12.

In contrast, some categories of applications may wish to maintain awareness of or affect the parameters related to mobility. The ML offers three interfaces for these purposes: the two callback interfaces (offered by the ML_{MH} and ML_{MG} components) and a tuning interface offered by the ML_{MH} component. The callback interfaces allow applications to track the state of connectivity for MHs through the reception of upcalls from the underlying ML. The tuning interface allows applications on the MH to control the configuration of the underlying ML, to query and configure individual transports and register custom reconnection algorithms. Figure 4.13 shows two applications on an MH that use the callback registration and tuning interfaces of the underlying ML. The following discusses in turn the callback interfaces on the MH and MG and the tuning interface on the MH.

Callbacks from the ML_{MH} Component

The callbacks used on MHs and MGs differ slightly. On the MH, mobile applications that wish to maintain awareness of the parameters related to mobility use a number of ALICE extensions to the Berkeley Sockets API. This allows the current state of connectivity of the MH to be monitored through the reception of upcalls from the ML. The application shown in figure 4.13 uses the portion of the ML's downcall API that constitutes the Berkeley Sockets replacement functions in the usual Sockets manner but also uses the extended functions that allow the registration of callback functions. The

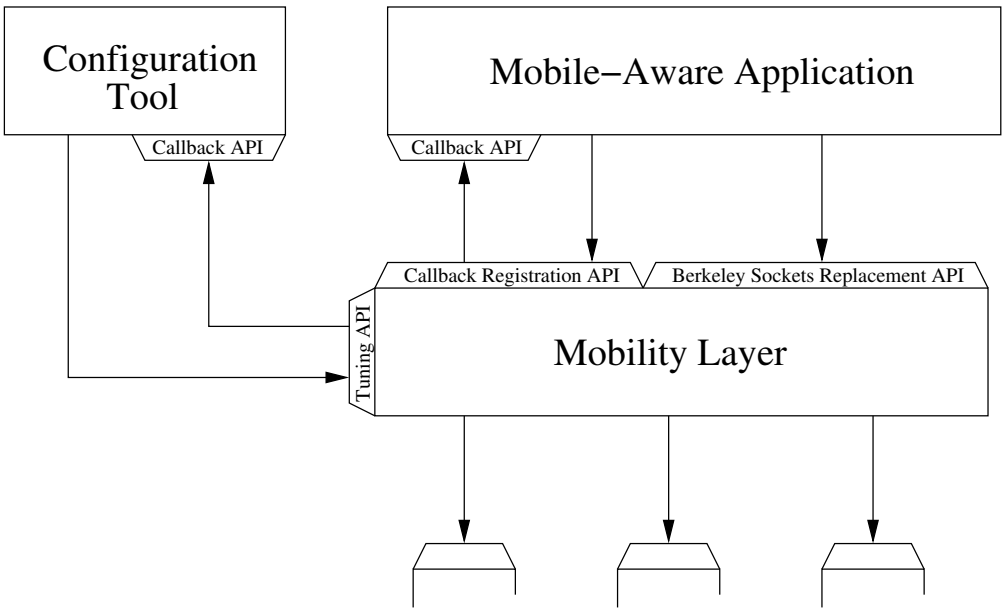


Figure 4.13: Mobility Layer with Mobility-Aware Applications

mobility-aware application must itself offer an upcall API in order to receive the callbacks.

Callbacks from the ML_{MH} component are registered on a per-socket basis and can be registered for client as well as server sockets. A typical client will invoke `socket(2)` to create a socket followed by `connect(2)` to connect the socket to a remote server. If mobility-aware, the client would also implement a callback function and register it with the ML at any point in time after the invocation of `socket(2)`. Subsequently, any change in the connectivity of the MH will cause the previously registered callback function to be invoked and the details of the new state of connectivity passed as parameters to the function. The parameters contain information about the current MG (if any) and the port number allocated on the MG. The ML_{MH} component also allows callbacks to be deregistered on a per-socket basis.

Mobility-aware servers use the ML callbacks in nearly the same way. A typical server application will invoke `socket(2)` to create the server socket followed by `bind(2)` to give the socket a specific address, i.e., network interface and port number. The server will then invoke `listen(2)` on the socket to place it in 'listening mode' where client connections can be accepted and later invoke `accept(2)` to receive and process the client connections one at a time. If mobility-aware, the server would register a callback function between the invocation of `bind(2)` and `listen(2)`. As for mobility-aware clients, any change in the MH's state of connectivity will cause an invocation of the registered callback function.

If the implementation language is C, the form of callback functions can be given using the following function prototype:

```
void callback(int sockfd, char *new_mg_name, int new_port);
```

The `sockfd` parameter identifies the socket to which the callback applies. The `new_mg_name` parameter contains the hostname[2] of the new MG to which the socket has been reallocated. This value may be `NULL`, in which case the MH has disconnected from an MG but not yet obtained a connection to a new MG. If `new_mg_name` is not `NULL`, the `new_port` parameter contains the port on the new MG that has been allocated for the socket in question. Otherwise, `new_port` is undefined.

Callbacks from the ML_{MG} Component

The purpose of the ML_{MG} component is to allow mobility support software (in the form of mobility-aware applications) to be placed on the MGs. The purpose of such software is to perform tasks related to MHs but which MHs may be unable to always perform because they may be disconnected or have moved to a different MG. Because the mobility support software resides on the MGs, it can perform operations related to MHs even when such MHs are not connected to the MG in question. In relation to object-oriented middleware, this type of mobility-aware application can be used to solve the address migration problem (sections 2.5.1 and 2.6.4) for new connections and thereby enable server mobility. In ALICE, this function is performed by the Swizzling Layer (described in section 4.6) which sits on the MG and selectively processes invocations targeted for MHs. The Swizzling Layer relays invocations for an MH that is currently being served by the MG directly to the MH, but redirects invocations for MHs that have moved to the new location. The behaviour of applications such as the Swizzling Layer depends on the current state of connectivity and the locations of MHs. Therefore, the ML_{MG} component allows mobile-aware applications on the MG to monitor key mobility events through the registration and reception of callbacks.

There are five key mobility events in which mobility-aware applications on MGs can register interest. Three of these have to do with the movements of MHs themselves and two with the behaviour of server processes on MHs. As described in section 4.5.1, an MH can be either connect to or disconnected from a given MG. An MH may also move to a different MG, triggering a handoff as described in section 4.5.3. These changes in the MH's state of connectivity constitute mobile events that need to be propagated to mobility-aware application, such that it can change behaviour. For example, upon learning that a given MH has disconnected from the MG in question, the Swizzling Layer may choose

[2] We generally use the term 'hostname' to refer to the Fully Qualified Domain Name (FQDN) of the host in question.

not to accept invocations or to notify clients that the server is temporarily unavailable. Upon learning that the MH has been handed off to a different MG, the Swizzling Layer can begin forwarding the connections to the new MG.

The ML provides good support for mobile clients through its tunneling and handoff features. Such clients will typically require no support from external mobility support components. Also, any open connections from remote clients to mobile servers are subject to tunneling by the ML. In this way, the ML solves the address migration problem for open connections. However, as opposed to mobile clients, mobile servers are also subject to the address migration problem for new connections. In relation to object-oriented middleware, an important use of mobility support components residing on MGs is to solve this instance of the address migration problem. This requires mobility support components to be aware of which mobile servers are currently being hosted by the MG. In the Berkeley Sockets API, an invocation of `listen(2)` signals the activation of a server. Invoking `close(2)` on the socket signals destruction of the socket and hence deactivation of the server. Mobile servers may be activated and deactivated independently of the MH's movements. For this reason, the ML_{MG} component allows mobility support components on the MG to monitor the activation and deactivation of individual server sockets.

If the implementation language is C, the form of callback functions offered by the ML_{MG} component could be given using the following function prototypes:

```
void connected(char *mh_name);
void disconnected(char *mh_name);
void handoff(char *mh_name, char *new_mg_name);
void listen_start(int mg_port, char *mh_name, int mh_port);
void listen_stop(char *mh_name, int mh_port);
```

The `connected()` and `disconnected()` functions are used to signal the arrival and departure of MHs at the MG in question. The `handoff()` function indicates that the MH in question has reconnected to a new MG and that a handoff has taken place. (This function is invoked on the old MG.) The `listen_start()` and `listen_stop()` functions are invoked when a mobile server starts and stops listening on a proxied server socket, respectively.

Tuning Interface for the ML_{MH} Component

Figure 4.13 also shows another mobility-aware application: a configuration tool that uses the ML's tuning API to monitor the performance and control the configuration of individual communications

interfaces on the MH. The tuning API, discussed in section 4.5.4, contains functions that let the configuration tool query and configure individual interfaces and to register custom reconnection algorithms to replace the cyclic reconnection algorithm discussed in section 4.5.1.

The tuning API of the ML_{MH} component is intended to allow an external (non-ALICE) component (such as a configuration tool) to monitor and control certain aspects of the ML's operation. The tuning API should allow the external component to inquire about the available interfaces on the MH, their names, current state of connectivity and past performance in terms of latency, error rates and average throughput. The tuning API should also allow individual interfaces to be enabled and disabled and allow the registration of an alternative reconnection algorithm than the default cyclic reconnection algorithm described in section 4.5.1. If the implementation language is C, the tuning interface of the ML_{MH} component could look as follows:

```
typedef int (*RA) (int old_itf);
int set_reconnection_algorithm(RA fa);
int get_interfaces(int *itfs[], int n)
int get_interface_name(int itf, char *itfname, int len)
int get_interface_mg(int itf, char *mgname, int len)
int get_interface_latency(int itf)
int get_interface_errors(int itf)
int get_interface_throughput(int itf)
enum { DISABLED = 0, ENABLED = 1 };
int get_interface_status(int itf, itf_status *st)}
int set_interface_status(int itf, itf_status st)
```

An implementation of the ML based on this tuning API is discussed in section 5.3.1.

4.5.5 ML Socket Semantics

With one exception, ML sockets are semantically equivalent to Berkeley Sockets. Effort has been made to assure as high compatibility as possible in order to minimise the adaptation required to existing applications. However, one modification to the standard sockets semantics was required because the ML cannot (as discussed in section 4.5.2) make the same guarantees concerning interface and port allocation as a normal implementation of TCP/IP.

As described in section 4.5.4, a server application typically uses the `bind(2)` operation to specify the interface and port number on which it wishes to receive client connections. When a server on the

MH performs a `bind(2)` using the ML_{MH} component, the operation is in reality performed on the MG rather than the MH. Consequently, it is impossible for the ML to honour a request for a specific local interface and a specific port that the server may expect on the MH. In addition, the endpoint obtained is subject to change when the MH changes MG. Thus, server endpoints are not only unpredictable but also short-lived. The problem is essentially a variation of the address migration problem discussed in section 2.5.1. As discussed in several of the approaches reviewed in chapter 3, the address migration problem can be solved at many different levels in the protocol stack. However, if transparency is to be retained for existing applications, modifications to the IP protocol as discussed in section 3.1 are required.

The approach taken by the ML is to silently ignore any requests for particular interfaces and ports specified in the `bind(2)` operation. This means that a mobility-unaware server may not be running on the interface and port number it expects. This is a significant change to the semantics of server sockets and is likely to complicate the operation of legacy client/server applications. While the operation of server processes themselves will be unaffected, it will be difficult for clients to continually locate such a mobile server, unless extra functionality is provided to address the address migration problem. The callback functions described in section 4.5.4 constitute the means to implement such mobility support in the form of mobility-aware applications residing on MGs. An example of such an application is the Swizzling Layer described in section 4.6.

4.5.6 Mobility Challenges

Of the fourteen mobility challenges discussed in chapter 2, the ML addresses five to various extents. First, the ML collaborates with the transport modules described in section 4.4 to address the challenge of network heterogeneity (section 2.4.1). Second, the ML provides support for short-term disconnections (section 2.4.2) through the transparent reconnection of broken transport connections and queueing of data during periods without connectivity. Third, the ML offers support for low bandwidth links (section 2.4.3) through the use of data compression. Fourth, the issue of usage cost (section 2.4.6) is partially addressed through the user-configurable transport selection algorithm which allow transports to be chosen on the basis of cost. Finally, the ML solves the address migration problem (section 2.5.1) for open connections through the transparent handoff and tunneling features. For new connections (i.e., server socket mobility), the address migration problem is not addressed by the ML.

4.6 The Swizzling Layer

As discussed in section 4.5.6, the ML's handoff and tunneling features solve the address migration problem for open connections. This applies for configurations where either the client or server resides on the MH. Also, the ML's proxying features make it possible for mobile clients to open connections to remote servers and retain such connections despite movement (and consequent change in address) of the MH. However, the ML leaves unsolved one important instance of the address migration problem: the situation where a remote client holds a reference to a mobile server but has yet to establish a connection. ALICE does not assume such clients to be aware that the server is mobile, and when the server's network address changes (when the MH moves to a new MG), the reference held by the client will be rendered obsolete.

The Swizzling Layer addresses this problem by performing runtime translation of server references and redirection of clients towards more recent server locations. The term 'swizzling' originates from research into persistent objects (e.g., [189, 125]) where the term 'pointer swizzling' describes 'the conversion of database objects between an external form (object identifiers) and an internal form (direct memory pointers)' [125]. The reference translation and client redirection performed by the ALICE Swizzling Layer is transparent to the client and server alike. The Swizzling Layer is dependent on the format of server references and on the remote invocation protocol used between the client and server. For this reason, a separate Swizzling Layer must be implemented for each protocol. We use the term 'S/RIP' to denote the abstract Swizzling Layer, and substitute 'RIP' with the name of the protocol when discussing an instantiation of the Swizzling Layer for a particular protocol. For example, the Swizzling Layer for Java RMI would be called 'S/RMI.' This section describes the general Swizzling Layer independently of particular object-oriented middleware technologies. Chapter 5 describes specific instantiations of the Swizzling Layer for CORBA and Java RMI. First, section 4.6.1 describes the general approach used by the Swizzling Layer to solve the address migration problem for new connections. Second, sections 4.6.2 and 4.6.3 explain in detail how the two key pieces of functionality required for the desired solution—server reference management and client redirection—can be provided within the context of object-oriented middleware. Finally, section 4.6.4 gives a summary of which of the fourteen mobility challenges are addressed by the Swizzling Layer and how.

4.6.1 Address Migration for New Connections

This section describes the approach taken by ALICE to support server mobility. First, we summarise the two main approaches to dealing with the address migration problem discussed in chapter 3.

Second, we present the solution adopted in the ALICE Swizzling Layer to solve the address migration problem for new connections. (Recall that the ML solves the address migration problem for existing connections.) Third, we discuss how server references are generally disseminated in object-oriented middleware and explain how the Swizzling Layer's approach to the address migration problem for new connections interacts with dissemination of server references.

Approaches to Address Migration

Chapter 3 showed that the address migration problem can be solved at several layers in the protocol stack. Common approaches were based on forwarding pointer schemes, such as the Mobile RMI system discussed in section 3.5.3, and home agent type schemes, such as Mobile IP discussed in section 3.1.1. Pure forwarding pointer schemes promise good performance and high availability, even in the face of frequent server movement. However, chains of forwarding pointers will need to be shortened in order to prevent the performance penalty associated with repeated redirection and to reduce the likelihood of broken chains.

The home agent type approach has the advantage that one designated entity (the home agent) is responsible for maintaining authoritative information about the location of the server. This makes it easy for servers to update their references and for clients to obtain fresh copies. However, in pure home agent solutions, the home agent typically constitutes a single point of failure whose availability is required for clients to invoke a mobile server. Unavailability of the home agent may cause a mobile server to be effectively unavailable, even though the server itself may be functioning perfectly. In relation to performance, the home agent may constitute a bottleneck in some scenarios. For example, if a client and a frequently moving server are close together (in terms of network distance) but far away from the home agent, it may be expensive for the client and server to repeatedly have to contact the home agent to refresh and update references.

Hybrid solutions, such as Wireless CORBA discussed in section 3.4.2, have been proposed in an attempt to get the best of both worlds. The intention is first to use forwarding pointers to locate the server and in case this fails fall back on a home agent. The advantage is that a server using a hybrid solution will not need to update its home agent as often as a server using a pure home agent solution, and that clients will rely less on the home agent because forwarding pointers can be used to locate the server in the majority of cases.

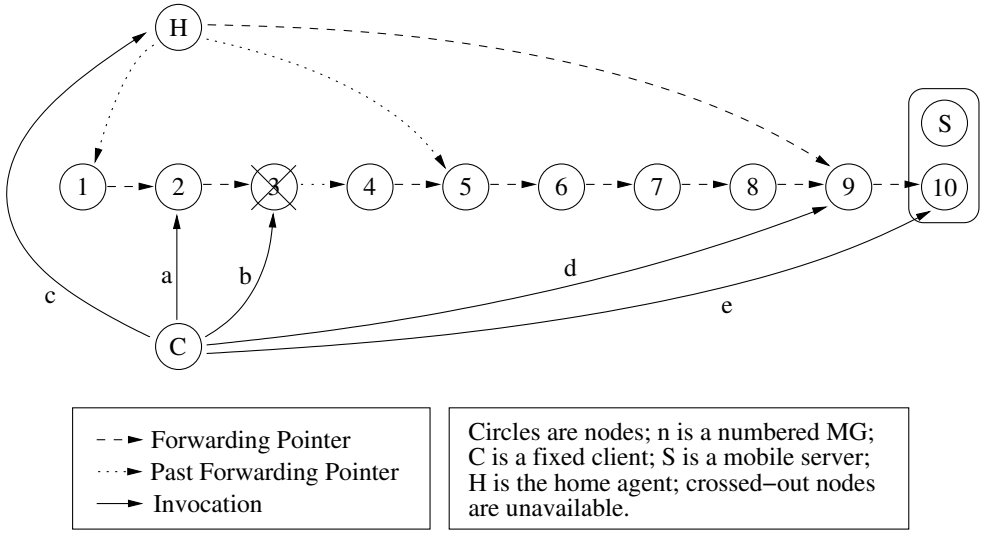


Figure 4.14: Forwarding Pointers and Home Agent

The ALICE Approach to Address Migration

The ALICE Swizzling Layer solves the address migration problem for open connections at the middleware level. A hybrid scheme is adopted that combines forwarding pointers with a home agent type approach. The intention is to facilitate frequent mobility of servers without requiring the constant involvement of a possibly distant home agent. Effectively, mobile servers 'leave traces' in their surroundings (in the form of forwarding pointers left at MGs) which clients can follow. If 'the trail is cold' (i.e., the forwarding pointers no longer exist, perhaps because the server reference held by the client is no longer valid), the client can fall back on the home agent to obtain a valid reference.

The scheme is depicted in figure 4.14. The mobile server (S) has moved from MGs 1 to 10 over a period of time. Each MG maintains a forwarding pointer to the next MG in the chain. Forwarding pointers are created during handoff (section 4.5.3). In the figure, the chain of forwarding pointers is broken because MG 3 is unavailable, for example due to a hardware or software failure. At the top of the figure is the mobile server's home agent (H) which also maintains a forwarding pointer to a recent (although not necessarily current) location of the MH. The server updates the pointer stored by the home agent at regular intervals, for example as a function of the time elapsed or the number of moves performed since the last update. In the scenario shown, the mobile server updates this pointer every four moves. The dotted arrows from the home agent show past forwarding pointers. Updating of forwarding pointers is not shown in the figure but would appear as invocations from the server to the home agent at the time when the server was hosted by MGs 5 and 9 respectively.

Figure 4.14 also shows a client (C) performing an invocation of the mobile server. The client holds an old reference to the server which points to MG 2. The invocation attempt (arrow a) results in a redirection to MG 3. However, MG 3 is unavailable which prevents the client from following the chain of pointers further (arrow b). Instead, the client invokes the server's home agent (arrow c) and obtains a more recent location of the server in the form of a pointer to MG 9. The client then invokes MG 9 (arrow d) and is redirected to MG 10 which hosts the mobile server. This scenario is reasonably complex because the reference held by the client is out of date. Had the reference been newer, fewer steps would have been required, and communication with the home agent would have been avoided. For example, had the reference held by the client been to MG 9 rather than MG 3, only the steps denoted by arrows d and e would have been required.

The Swizzling Layer's approach to the address migration problem shares a number of similarities with the approach adopted in Wireless CORBA discussed in section 3.4.2. Most notably, both use a hybrid redirection scheme where each server reference contains a forwarding pointer as well as a home agent address. However, the two approaches differ in a variety of respects. First, there is a difference in scope. Wireless CORBA deals with the address migration problem (section 2.5.1) but also the network heterogentity (section 2.4.1) and short-term disconnection (section 2.4.2) problems. In comparison, the Swizzling Layer is concerned only with address migration and leaves the other two problems to the Mobility Layer. Second, Wireless CORBA of course constitutes a CORBA-specific approach where ALICE (including the Swizzling and Mobility Layers) is intended for general use across object-oriented middleware architectures. Third, both Wireless CORBA and the Swizzling Layer specify extended server reference formats and define the rules for client redirections. However, Wireless CORBA is not concerned with translation or propagation of mobile server references; these issues are left to the mobile servers themselves. In this respect, the Swizzling Layer has a wider scope because it also maintains transparency for servers through the address translation scheme discussed in section 4.6.2.

Dissemination of Server References

An advantage of the scheme shown in figure 4.14 is that clients holding server references that have only recently become obsolete can avoid going through the home agent. However, the scheme assumes the client to be in possession of a server reference before the invocation is first attempted. A common feature of object-oriented middleware architectures is that they typically rely on directory services to disseminate server references. (Depending on the architecture in question, other methods may also possible, but directory services are generally the recommended option.) Servers are responsible for

creating their own references and registering them with a directory service. For example, CORBA specifies a Naming Service [82] that maps symbolic names to server references. The intention is that CORBA servers insert their references into the Naming Service and clients invoke the service to obtain the references. The CORBA Naming Service can be organised in a centralised or distributed manner. Java RMI uses an entity known as the RMI Registry (discussed briefly in section 3.5) that is essentially a distributed directory service for which the distribution criteria is fixed: each host runs an instance of the registry whose role is to map names to references for any servers running on that host. Java servers using RMI disseminate their references through registration with their local registry, and a client wishing to obtain a server reference must query the relevant registry instance.

In order to implement the redirection scheme shown in figure 4.14, mobile server references must contain the locations of MGs and home agents. For a given mobile server, the reference should contain the current MG to which the server's MH is connected. Hence, in order to assure mobile server references remain fresh, each reference should ideally be updated every time the MH holding the server moves to a new MG. While this issue could be left to mobile servers themselves (as proposed in Wireless CORBA), it is essentially a part of the address migration problem and it is therefore more attractive to solve it in mobility support software, preferably in a manner that is transparent to the server. Special care has to be taken in relation to reference dissemination, because the servers themselves are typically responsible for registering their references with directory services. Hence, if any transparent manipulation of a server reference is to take place, it should be done before the reference is registered with a directory service or otherwise distributed to clients.

Section 4.6.2 describes the ALICE approach to transparent server reference management. The idea is that a copy of a mobile server's reference held within the server itself can be maintained in an always updated state by transparently modifying it every time the MH moves. Any copy of that reference which the server choses to distribute (e.g., through a directory service) will be an updated reference.

4.6.2 Server Reference Management

In section 4.6.1, we described the ALICE approach to dealing with the address migration problem for new connections and proposed a redirection scheme in which clients would locate (and invoke) mobile servers primarily by following forwarding pointers but in case the pointer chain was broken could fall back on a home agent. We also discussed the typical means of disseminating server references in object-oriented middleware and argued for a transparent approach to keeping the reference that a mobile server holds to itself updated. This section describes the part of the Swizzling Layer that resides

on the MH, the S/RIP$_{MH}$ component, which is responsible for translating mobile server references such that they refer to the MGs and home agents in a manner that implements the redirection scheme described in section 4.6.1. The component works by keeping the copy of a mobile server reference that is stored in the server updated as the MH moves between MGs. The updates are performed transparently to the server. The effect is that whenever the server decides to publish its reference (e.g., through a directory service), it is always the updated (i.e., currently valid) reference that is published.

In order to maintain transparency for mobile servers, it is important that this functionality be added in a manner that does not interfere with the standard reference manipulation performed by server, e.g., the creation and dissemination of references. In ALICE, this is done by adding the functionality in form of a layer whose API is identical to that of the remote invocation protocol used on the MH (the RIP$_{MH}$ component). In the ALICE protocol stack shown in figure 4.4, the S/RIP$_{MH}$ component resides above the implementation of the remote invocation protocol used for the object-oriented middleware framework in question. The level of indirection allows the S/RIP$_{MH}$ component to transparently modify the semantics of operations related to the management of references to server objects on the MH. During normal operation, the server creates its reference and may register it with some directory service. However, if using ALICE, the reference will be transparently translated by the S/RIP$_{MH}$ component before registration takes place. Hence, it is the *translated* reference that makes it into the directory service. This allows clients to obtain the translated reference without explicit mobility-awareness of client or server or directory service. Parts of the API offered by the remote invocation protocol will be unrelated to the management of server references. Invocations of such functions are simply relayed by the S/RIP$_{MH}$ component to the underlying implementation of the remote invocation protocol in question.

It should be noted that for a given mobile server, the reference translation performed by the S/RIP$_{MH}$ component only affects the mobile server's reference to itself, i.e., the reference that is disseminated by the server. No attempt is made to track or update references stored on RHs. Such references are updated in a lazy manner through the redirection scheme discussed in section 4.6.3.

Server Reference Format

Server references are specific to object-oriented middleware architectures and each architecture describes its own reference format. In section 4.3 we defined two requirements (R4 and R5) related to server reference formats that the object-oriented middleware architecture must satisfy in order for it to be possible to instantiate ALICE for the architecture in question. R4 required the server reference

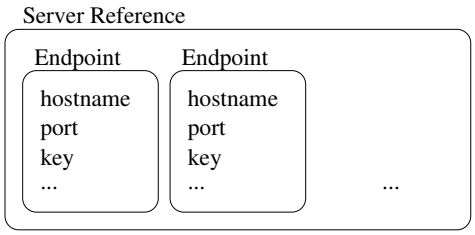


Figure 4.15: Minimal Server Reference

format to allow the server to be identified by multiple endpoints all included in the reference, and specified that clients holding a reference should try each endpoint in sequence. The fulfillment of this requirement allows us to store both the current MG and the home agent for a mobile server in the server's reference. If the endpoint referring to the MG is placed before the one that refers to the home agent, clients will attempt to invoke the server at the MG and fall back on the home agent, as described in section 4.6.1. R5 required that it be possible to store some extra information in the reference that would be passed from the client to the server during invocation. As described in section 4.3, this makes it possible to transfer information about endpoints in the server reference held by the clients to which the MG would not otherwise have access. On the MG, the information about these endpoints is required to construct new server references as part of redirection responses.

Using Extended Backus-Naur Form (EBNF) [91], a server reference fulfilling requirements R4 and R5 can be described as follows:

```
reference ::= endpoint { endpoint }
endpoint  ::= hostname port key
```

Recall that since requirement R2 specifies that the transport must be TCP/IP, we identify the server process by a hostname (or IP address) and a port number. Hence, a server reference that satisfies the requirements consists of one or more server endpoints, each of which is a triple. The host name specifies the physical machine on which the server resides, the port number specifies the server process in question, and the key specifies the server object within that process. Figure 4.15 shows an example of a minimal server reference that fulfills these requirements. The reference is minimal in the sense that it does not contain other data fields than those specified by the ALICE requirements. The server reference formats specified by most object-oriented middleware architectures generally conform to this format but are typically more complex than the one shown. The ellipses denote locations where other data fields would typically reside.

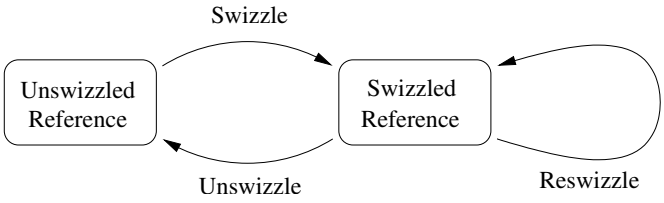


Figure 4.16: The Two States of Server References

The Two States of Server References

As explained above, the S/RIP$_{\text{MH}}$ component transparently updates a mobile server's reference to itself every time the MH where the server resides changes MG. The updates involve modifying the MG endpoint in the reference such that it always refers to the current MG. In addition, the S/RIP$_{\text{MH}}$ component must add an endpoint to the reference that refers to the server's home agent. This must be done when the server is created.

The process of reference translation is called 'swizzling,' a term borrowed from research into persistent objects [125, 189]. Server references can be in one of two states: swizzled or unswizzled. An unswizzled reference is the original, unmodified reference as created by a mobility-unaware server. It contains no extra information in the form of MG or home agent endpoints. A swizzled reference is one that has been modified by the S/RIP$_{\text{MH}}$ component to contain endpoints for the current MG and the home agent. Swizzled references follow the specification of server references given for the object-oriented middleware architecture in question. Hence, a swizzled server reference is indistinguishable from one that is not swizzled, except to the Swizzling Layer.

Three operations exist that allow server references to be transformed between the two states: *Swizzling* adds MG and home agent endpoints to an original, unmodified server reference; *reswizzling* updates the MG endpoint for a reference that is already in a swizzled state; and *unswizzling* restores the reference to its original state by removing the MG and home agent endpoints. Swizzling, reswizzling and unswizzling of server references takes place transparently to the mobile server, to the implementation of the remote invocation protocol used on the MH and to remote clients holding references to the server.

Figure 4.16 shows the two states (the rounded boxes) of server references and how the three operations (the arrows) transform the references between the two states. A reference initially created by a mobility-unaware server will be in unswizzled state. The following discusses swizzling, reswizzling and unswizzling in more detail.

Swizzling

The term 'swizzling' is applied to server references as well as the individual endpoints contained in the references. Swizzling a server reference occurs when a new reference is created while the MH is connected to an MG. The S/RIP_{MH} component intercepts the operation that causes the creation of the reference and performs the swizzling operation on the created reference. Hence, references are maintained in a swizzled state but the process of swizzling remains transparent to the application using the RIP_{MH} component. Swizzling a server reference consists of swizzling one or more of the endpoints contained in the reference and adding one extra endpoint that refers to the server's home agent.

Swizzling an individual endpoint involves changing the endpoint such that it no longer refers to the server process on the MH but to the S/RIP_{MG} component of the MG to which the MH is connected at the time of swizzling. Only endpoints referring to local interfaces on the MH are swizzled. Other endpoints appearing in the reference are left intact. Endpoint swizzling involves manipulating the three fields (hostname, port and key) contained in the endpoint. The hostname and port appearing in the unswizzled reference of the MH along with the name of the home agent are prepended to the key field, forming a composite key value. The hostname is set to the hostname of the MG to which the MH is connected at the time of swizzling. The port number of the mobile server is replaced with a well-known port on the MG, on which the S/RIP_{MG} component listens for incoming client connections. The use of a well-known (rather than dynamically allocated) port for this purpose allows swizzling to take place on the MH without involvement of the MG. When the mobile server starts listening on an endpoint, a proxy server socket is dynamically allocated on the MG by the ML_{MG} component. The ML_{MG} component also notifies the S/RIP_{MG} component of the socket creation through an upcall. This allows incoming connections from clients to the S/RIP_{MG} component to be forwarded to the server application on the MH through the dynamically allocated proxy on the MG.

Figure 4.17 shows a server endpoint being swizzled. The endpoint refers to a server running on an MH whose hostname is `mh1.dsg.cs.tcd.ie`. The server's home agent is `ha.dsg.cs.tcd.ie`. At the time of swizzling, the MH is connected to an MG called `mg1.dsg.cs.tcd.ie`. The mobile server is running on port 1234 on the MH, but because the MH is connected through the MG, it is not possible for clients to connect to the server directly through this port. (Such connections would break when the MH moved and changed its network address.) Within the server process, the server object in question is identified by the key `my_object`. After the endpoint has been swizzled, the endpoint's hostname has been replaced with that of the MG, and the port at which the server is running has been replaced with 5004 which is the well-known port on which the S/RIP_{MG} component is listening

Figure 4.17: Swizzling, Reswizzling and Unswizzling of a Server Endpoint

on the MG. The original hostname and port number (identifying the server process on the MH) are encoded in the key field from where they can later be retrieved. The home agent's hostname is also encoded in the key field, such that an MG who receives an invocation can create a new reference that includes an endpoint for the home agent. (As discussed in section 4.3.5, the S/RIP$_{\mathrm{MG}}$ component needs to know the hostname of the home agent to construct a new reference but the server endpoints are typically not included in invocation requests.)

When all endpoints referring to local interfaces on the MH have been swizzled, an extra endpoint is added to the reference. This endpoint identifies the home agent for that particular mobile server. Its presence in the reference allows a client to fall back on the home agent in case the MG specified in the first endpoint is unavailable. The hostname for the extra endpoint is set to the hostname of the home agent and the port number to the well-known port number for S/RIP$_{\mathrm{MG}}$ components. The hostname of the MH and the port number of the mobile server are encoded in the endpoint's key field. Figure 4.18 shows a server reference being swizzled in this manner. Each of the endpoints referring to local interfaces are swizzled as shown in figure 4.17.

A server reference may contain more than one endpoint, only some of which may refer to server processes on the MH. For example, references for replicated services would typically contain a series of endpoints, identifying server processes on physically distinct machines. When a server reference is swizzled, each endpoint that refers to a server process on the MH is subject to swizzling. Other endpoints are left untouched.

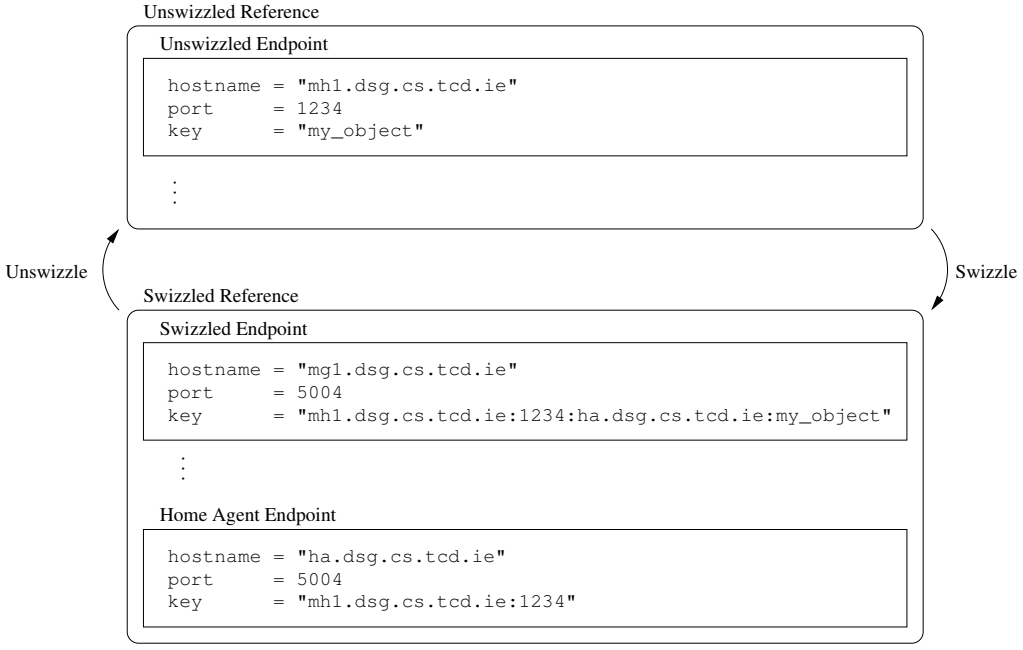


Figure 4.18: Swizzling and Unswizzling of a Server Reference

Reswizzling

Reswizzling of a server reference occurs when an MH moves from one MG to another. In this case, the S/RIP$_{\text{MH}}$ component receives a callback from the ML$_{\text{MH}}$ component, indicating that the MG has changed. When receiving the callback, the S/RIP$_{\text{MH}}$ component reswizzles all references to local servers held on the MH. Like swizzling, reswizzling is performed on a per-endpoint basis. Only endpoints that refer to server processes on the MH are affected; any others, including the home agent endpoint, are left unchanged. Reswizzling of an endpoint consists of replacing the hostname of the old MG with that of the new. The well-known port number and the key are left unchanged. This is shown in figure 4.17 where an MH has moved from `mg1.dsg.cs.tcd.ie` to `mg2.dsg.cs.tcd.ie`.

Unswizzling

Unswizzling of a server reference occurs if the ALICE mobility support layers are removed from the protocol stack, for example in case the MH obtains a direct connection to a LAN. In this case, all server references known to the S/RIP$_{\text{MH}}$ component are unswizzled. For each reference, this involves unswizzling all endpoints in the manner shown in figure 4.17. Each endpoint that refers to the S/RIP$_{\text{MG}}$ component is replaced by the local endpoint that was saved in the key field during swizzling. This restores each swizzled endpoint to its original state. In addition, the home agent

endpoint that was added to the reference during swizzling is removed. Any endpoints referring to remote interfaces are unchanged. This restores the reference to its original state.

Updating of Forwarding Pointers

A part of server reference management not directly related to swizzling is the ability to update forwarding pointers maintained by the S/RIP_{MG} components on the MGs and the home agent. To allow this, the S/RIP_{MG} components must implement the following function (given in IDL):

```
setLocation(in string mh_name, in string new_mg_name)
```

Both parameters are Internet hostnames. The home agent is simply a host that runs a copy of the S/RIP_{MG} component. It may or may not act as a full MG. The S/RIP_{MH} component or a mobility-aware server can use this function to update the forwarding pointer retained by the home agent as shown in figure 4.14 and to shorten chains of forwarding pointers between MGs. The `mh_name` parameter uniquely identifies the MH. The `new_mg_name` parameter specifies the MG to which forwarding pointers for any servers on the MH should point.

4.6.3 Client Redirection

Section 4.6.2 described how the part of the Swizzling Layer that resides on MH (the S/RIP_{MH} component) used the callback features of the ML_{MH} component to receive notification about the connectivity of the MH. In a similar fashion, the part of the Swizzling Layer that resides on the MG (the S/RIP_{MG} component) uses the callback features of the ML_{MG} component to receive notification about which MHs are currently hosted by the MG in question. This allows the S/RIP_{MG} component to receive incoming invocations from remote clients and redirect them according to the server's current location. This section describes how this function is performed. First, we review the requirements on the server reference format and the remote invocation protocol used by the object-oriented middleware architecture for which ALICE is being instantiated. Second, we describe how MHs are tracked. Third, we describe how the collected tracking information is used during client invocations. Finally, we discuss a problem related to dynamically generating new server references for MHs that have moved away.

Review of Requirements

As discussed in section 4.6.2, the part of the Swizzling Layer that resides on the MH requires R4 and R5 from section 4.3 to be satisfied. These requirements have to do with the format of server references used by object-oriented middleware architectures. Like server reference formats, remote invocation

protocols are also specific to object-oriented middleware architectures, and each architecture specifies a different protocol. In order for it to be possible to instantiate the part of the Swizzling Layer that resides on the MG, the middleware architecture must satisfy requirements R4 and R5 but also requirement R3.

The primary task of the part of the Swizzling Layer that resides on the MG (the S/RIP_{MG} component) is to receive and process invocations targeted for mobile servers. In case the mobile server for which the invocation is intended is connected to the MG in question, the S/RIP_{MG} component relays the invocation directly to the server. In case the MH hosting the server is currently connected to a different MG, the S/RIP_{MG} component constructs a new server reference and returns it to the client in a redirection response. In order for this to be possible, requirement R3 from section 4.3 must be satisfied by the object-oriented middleware architecture for which ALICE is being instantiated. Requirement R3 specifies that it should be possible to redirect clients towards other server locations.

The S/RIP_{MG} component plays a critical role in implementing the redirection scheme described in section 4.6.1 that combines forwarding pointers with a home agent approach. The hybrid scheme requires each server reference published to contain endpoints for the Swizzling Layers on an MG as well as a home agent. The order in which the client tries the endpoints is important, because the MG should be tried first and the home agent only as a last resort. For this reason, requirement R4 requires server references to contain multiple endpoints and specifies that clients should try the endpoints contained in a server reference in order.

In relation to requirement R5, the requirement of passing the key from the client to the server during invocation serves two purposes in relation to the S/RIP_{MG} component, both of which were discussed in section 4.3.5. First, the approach allows information about endpoints contained in the reference held by the client to be transmitted to the S/RIP_{MG} component. This makes it possible for the MG to create a new reference to the mobile server without requiring the MG to maintain information about home agents for individual mobile servers. Second, the approach allows an additional identifier for the mobile server process to be inserted into the key and thereby allow the S/RIP_{MG} component to serve as a redirector for multiple servers. By examining the mobile server identifier, the S/RIP_{MG} component can easily decide whether to relay the invocation to the server (if the MH is currently being hosted by that MG) or redirect the client either to another MG that has more recently hosted the MH or to the mobile server's home agent.

Mobile Server Tracking

The S/RIP$_{MG}$ component uses the ML$_{MG}$ component to keep track of mobile servers that use the remote invocation protocol. This happens via the upcalls described in section 4.5.4. The S/RIP$_{MG}$ component performs this function on an ongoing basis, independently of incoming client invocations. The mobile server state obtained via the upcalls is collected over time and used during invocations for redirecting clients towards the current server location.

The S/RIP$_{MG}$ component maintains two lists of mobile server sockets: the local and the remote list. The former contains entries for server sockets that are currently being proxied by the MG in question. Each entry has the following form: `(mh_name, mh_port, mg_port)`. The first two values identify the MH and the server process in question. The last value identifies the proxy port on the MG that corresponds to the mobile server process. When the S/RIP$_{MG}$ component receives an invocation of its `listen_start()` operation, it adds an entry to its local list. This happens when the mobile server has created a server socket and begins listening on it. When the component receives an invocation of its `listen_stop()` operation, it deletes the corresponding entry from its local list. This happens when the mobile server destroys its socket.

The remote list contains entries for server sockets that are currently being proxied by other MGs. This list effectively constitutes a set of forwarding pointers as shown in figure 4.14. Each entry has the following form: `(mh_name, mh_port, mg_name)`. As for the local list, the first two values identify a mobile server process. The last value identifies an MG to which the MH hosting the mobile server process was connected at a later point in time than the current MG. When the S/RIP$_{MG}$ component receives an invocation of its `handoff()` operation, it moves all entries for the MH in question from the local list to the remote list. For each of these entries, the `mg_port` value is replaced with the name of the new MG.

An MH may return to an MG that holds one or more forwarding pointers for servers on that MH. This is for example the case if an MH moves from MG1 to MG2 and back to MG1. During the move from MG1 to MG2, MG1 will be notified via an invocation of its `handoff()`. This will cause MG1 to move any server sockets belonging to the MH from the local to the remote list as described above. During the second move, MG2 will be notified (and act) in a similar manner. MG1 will be notified of the return of the MH through the invocation of `listen_start()` for each server socket on the MH. This allows any entries for the returning MH to be moved from the remote list back to the local list.

In the current design, the S/RIP$_{MG}$ component does not track the arrival and departure of MHs (only the creation and destruction of server sockets and the occurrence of handoffs) and therefore does not implement the `connected()` and `disconnected()` upcalls.

Operation During Invocation

A client may hold a swizzled reference identifying a mobile server. The swizzled reference will contain one or more swizzled endpoints. Each of these identifies a S/RIP$_{MG}$ component (rather than the server on the MH itself) as shown in figure 4.18. As mentioned in section 4.3, the client will try each of the endpoints in the reference in turn, starting with the first endpoint. Any non-swizzled endpoints contained in the reference are unaffected by the operation of ALICE and will be tried in the ordinary fashion. Any swizzled endpoints contain the details of the S/RIP$_{MG}$ component on the MG to which the MH was connected when the reference was published. When the client attempts to invoke the server using a swizzled endpoint, or the home agent endpoint, the invocation will be received by a S/RIP$_{MG}$ component.

During an invocation, the client passes along the key field from the endpoint in question. The S/RIP$_{MG}$ component receiving the invocation uses the MH name and port number (contained in the key field) identifying the server process to decide whether the mobile server is local, remote or unknown. If the server is local, a tunnel is set up to the local proxy server port. This effectively relays the invocation to the mobile server via the ML and allows the invocation result to be returned to the client. If the mobile server is remote, a new reference is constructed based on the forwarding pointer on file, and returns it to the client. This implements the redirection of clients towards more recent server locations. If there is no record of the server in question (i.e., the server does not appear in either the local or remote list), the S/RIP$_{MG}$ component constructs a new reference containing a single endpoint that identifies the S/RIP$_{MG}$ component on the home agent.

The functionality of a S/RIP$_{MG}$ component serving as a home agent is identical to that of any other S/RIP$_{MG}$ component. Upon receiving the invocation, the home agent's S/RIP$_{MG}$ component will return an updated reference containing an endpoint to the S/RIP$_{MG}$ component on an MG to which the MH has recently been connected. The updated reference follows the standard swizzled format and therefore also contains an endpoint that identifies the home agent's S/RIP$_{MG}$ component itself.

If the MH has disconnected but no reconnection or handoff has taken place, the S/RIP$_{MG}$ component cannot redirect the client. In this case, two behaviours are possible. One possibility is to tunnel the incoming connection to the server proxy anyway. This will cause the invocation to block while the ML waits for the MH to reconnect or initiate a handoff. Another possibility is to refuse new connections, or (if the middleware architecture allows it) to return an error message indicating that the service is temporarily unavailable.

Reswizzling During Redirection

When receiving an invocation for a server that resides on an MH, which is no longer hosted by the MG, an S/RIP_{MG} component will need to construct a new reference for the client. Given the reference held by the client, the S/RIP_{MG} component essentially needs to reswizzle the reference as described in section 4.6.2. While the client supplies the key belonging to the MG's endpoint as part of the invocation, the entire server reference is typically not passed as part of the invocation. This complicates the reswizzling operation for the S/RIP_{MG} component. Some, but not all, object-oriented middleware architectures (e.g., CORBA using IIOP 1.2 or newer) allow the server to request the full reference from the client. If ALICE is being instantiated for an architecture where this is possible, the S/RIP_{MG} component can request the server reference from the client, reswizzle it to contain the new MG and serve it back to the client in a redirection response. If ALICE is being instantiated for an object-oriented middleware architecture where this is not possible, the new server reference will have to be constructed on the basis of the key alone. This may result in the loss of any endpoints that cannot be generated on the basis of the key, such as endpoints referring to interfaces that do not belong to the MH.

4.6.4 Mobility Challenges

Of the fourteen mobility challenges discussed in chapter 2, the Swizzling Layer addresses only one case of the address migration problem (section 2.5.1): where a remote client holds a reference to a mobile server but has yet to establish a connection. The hybrid solution offered by the Swizzling Layer allows such references to be updated in a lazy fashion.

4.7 The Disconnected Operation Layer

Section 2.4.2 discussed the challenge of disconnection and identified the differences between long-term and short-term disconnection in terms of cause as well as viable solutions. The separation of concerns adopted in ALICE follows the distinction between these two types of disconnection. Hence, we consider disconnection to be the general problem, whose two subproblems are addressed by different ALICE layers. The short-term disconnection problem is addressed by the ML's ability to transparently restore broken transport connections and queue data during periods without connectivity, as discussed in section 4.5.6. During periods without connectivity, applications using the ML will experience synchronous blocking of send and receive operations. While suitable for hiding sporadic, short-term loss of connectivity, the blocking behaviour offered by the ML is unlikely to allow applications to

continue operating in the face of long-term disconnections.

The long-term disconnection problem is addressed by the Disconnected Operation Layer. The Disconnected Operation Layer allows server functionality to be cached on the client side during periods where the MH is not connected to an MG. Server objects can be copied and cached on the client side before disconnection, the replicas invoked during disconnection and finally reconciled with the original server objects when connectivity is regained. The Disconnected Operation Layer allows server functionality to be cached from a remote to a mobile host or vice versa. Like the Swizzling Layer, the Disconnected Operation Layer is specific to the object-oriented middleware architecture and framework for which ALICE is being instantiated. We refer to the general Disconnected Operation Layer as 'D/RIP' and substitute 'RIP' with the name of the remote invocation protocol, when discussing an instance of the layer for a given object-oriented middleware framework. For example, section 5.5 describes the D/IIOP Layer, an instantiation of the Disconnected Operation Layer for the CORBA Internet Inter-ORB Protocol (IIOP). Instantiations of the Disconnected Operation Layer are independent of applications but will have to be supplemented with application-specific code to perform replication and reconciliation of server objects.

This section describes the Disconnected Operation Layer. First, section 4.7.1 explains the general approach used to solve the long-term disconnection problem. Next, section 4.7.2 discusses in detail how cache management and invocation redirection are performed on the client side during periods with and without connectivity. Section 4.7.3 discusses how replication and reconciliation are performed on the server side within the context of object-oriented middleware. Some of the Disconnected Operation Layer's functions (e.g., redirection of invocations to replicas, reconciliation of replicas) depends on whether the MH is connected to an MG or not. Section 4.7.4 describes how the Disconnected Operation Layer monitors the MH's state of connectivity. Finally, section 4.7.5 summarises which of the fourteen mobility challenges are addressed by the Disconnected Operation Layer and how.

4.7.1 Long-Term Disconnection

This section describes the general approach taken by ALICE to solving the long-term disconnection problem. First, we summarise relevant work reviewed in chapter 3. Second, we present the solution adopted in ALICE. Third, we discuss matters arising in relation to the use of object mobility within object-oriented middleware architectures and to concurrency control for applications using such architectures in mobile environments.

Approaches to Long-Term Disconnection

The Coda (section 3.3.1) and Rover (section 3.3.2) systems implement support for long-term disconnected operation in a similar fashion. Each system addresses the problems of replication and mobility of the chosen type of data object (file or object, respectively) as well as the detection and resolution of potentially conflicting updates.

Coda operates in three modes: hoarding, emulation and reintegration mode. In hoarding mode, files are prefetched under the assumption that disconnection will occur later on. Compared to objects known from object-oriented middleware, plain files are relatively easy to replicate and move. The semantics for copying files—even across machine boundaries—are simple, because file contents do not have to be marshalled. (The writing of data to a file includes marshalling.) Coda recognises that the internal structure of files is application-specific and therefore treats files as opaque entities. It does not attempt to solve conflicts below the file granularity but instead allows applications to provide their own resolvers. Such resolvers are considered part of the application and can therefore be aware of the internal structure of files.

Rover allows server functionality to be moved temporarily to the client side in order to improve server availability during periods of disconnection. This is done by providing support for object replication, mobility and reconciliation in conjunction with an optimistic concurrency control scheme. The Rover object model is based on a homogenous execution environment (a Tcl/Tk virtual machine), which simplifies object mobility in heterogenous distributed environments. Like Coda, Rover treats application objects as opaque and does not attempt to solve conflicts below the granularity of individual objects. Instead, Rover allows applications to provide their own resolvers. Like Coda, Rover also considers such resolvers part of the application.

The ALICE Approach to Long-Term Disconnection

The ALICE approach to long-term disconnection is not unlike that proposed by the designers of Coda and Rover. The D/RIP Layer allows server functionality to be moved to the client side before disconnection takes place. During disconnection, any invocations made by the client are redirected to the local replica. This allows invocations to be performed in the absence of connectivity. When connectivity is restored, the replica is reconciled with the original server object. At this point, any conflicting updates are detected and resolved.

The D/RIP Layer has two components, residing on the client and server respectively. In this respect, the D/RIP Layer differs from the ML and the S/RIP Layer, for both of which the layer components were bound to host roles (the MH and MG) rather than application roles. The architectural

overview given in figure 4.4 shows the client residing on the MH and the server on the RH, a configuration that allows functionality from the RH to be cached on the MH. The reverse configuration is also possible, i.e., a client on the RH could cache functionality from a mobile server. The figure shows the client on the MH, because we expect this to be the common case. No functionality related to the D/RIP Layer resides on the MG. On both the client and server side, D/RIP components reside above the implementation of the remote invocation protocol and below the application layer.

The two parts of the Disconnected Operation Layer perform very different functions. The client-side component of the layer (the D/RIP_C component) manages a cache of replicated server objects under explicit application control and redirects invocations to the replicas on the client host during periods of disconnection. This happens transparently to the client. Section 4.7.2 discusses the operation of the D/RIP_C component in detail. The server-side component of the layer (the D/RIP_S component) handles requests for replication and reconciliation of server objects. It is integrated with the server application and is therefore non-transparent. Section 4.7.3 describes the D/RIP_S component in detail.

Object Mobility

As noted above, Coda and Rover relied upon well-specified runtime environments (Unix file systems and a Tcl/Tk virtual machine, respectively) to facilitate replication and mobility of their respective data objects. In comparison, ALICE is a general architecture designed to work with a range of object-oriented middleware architectures, each of which defines its own runtime environment. Any ALICE instantiation must be implemented using the runtime environment defined by the object-oriented middleware architecture for which ALICE is being instantiated. Because the Disconnected Operation Layer relies on the ability to replicate and move objects, the nature of this environment affects the ease with which the Disconnected Operation Layer can be instantiated. In particular, some object-oriented middleware architectures (e.g., Java RMI) provide better support for object mobility than others (e.g., CORBA). Not unlike Rover, Java RMI relies on a virtual machine to homogenize the distributed execution environment, whereas CORBA deals with the heterogeneity of the execution environment in a less transparent fashion.

Concurrency Control

As noted above, Coda and Rover did not implement fine-grained conflict resolution schemes. The authors of both systems recognised that while potential conflicts can be detected, the fine-grained resolution of conflicts eventually requires application-specific knowledge about the internal structure of data and the way it is used. The best that can be provided in terms of mobility support are

tools to facilitate the resolution of conflicts. ALICE takes a similar approach to Coda and Rover. Using ALICE, applications can implement their own concurrency control schemes. ALICE allows optimistic as well as pessimistic update policies to be implemented. However, it is worth stressing that experiences from Coda and Rover indicate that an optimistic approach can be expected to work better in many cases [162, 98].

4.7.2 Cache Management and Redirection

This section describes the component of the Disconnected Operation Layer that resides on the client side. As shown in figure 4.4, the D/RIP$_C$ component resides above the Swizzling and the Remote Invocation Protocol Layers. Like the part of the Swizzling Layer that resides on the MH, the D/RIP$_C$ component implements a superset of the Remote Invocation Protocol Layer's API. Applications designed to use this layer, can therefore use the D/RIP$_C$ component instead without any modifications to application code. This allows mobility-unaware applications to avail of the services of the Disconnected Operation Layer in a transparent manner (and without having to implement explicit cache control) and thereby allows support for disconnected operation to be inserted into the protocol stack in a transparent manner.

The D/RIP$_C$ component has two important functions: to maintain a cache of replicated server objects and to redirect invocations to these replicas. The two functions are independent and can be used separately by clients. For example, a simple, mobility-unaware client may not want to control which server objects are replicated but may still like to invoke a replica if one happens to be available. A more complex, mobility-aware client could be in explicit control of which server objects it wishes to cache. A third type of client could be a user-interactive 'hoarding-tool' which would interact with the cache even though it would never itself need to invoke any of the replicas. The following sections describe the two functions of the D/RIP$_C$ component in turn.

Cache Management

The D/RIP$_C$ component manages a collection of replicated server objects. The component has the ability to interact with D/RIP$_S$ components in order to start and stop caching of specific server objects. The D/RIP$_C$ component's cache management functions are made available through the component's tuning API, where it can be used by mobility-aware applications. The API is simple, consisting of only three functions:

Cache (ServerReference) instructs the D/RIP$_C$ component to obtain a copy of the server object

identified by the reference. This is done through invocation of the D/RIP$_S$ component for the original server object. This is described in section 4.7.3.

IsCached (ServerReference) → Boolean returns true if a cached replica of the server object identified by the reference is available on the client. If a replica is not available, the operation returns false.

Flush (ServerReference) instructs the D/RIP$_C$ component to remove any cached replicas of the server object in question. If a replica exists and has been modified, this will cause the D/RIP$_C$ component to return the replica to the D/RIP$_S$ component from which it came. The D/RIP$_S$ component will then initiate conflict detection and resolution mechanisms.

The three cache management functions can be used to implement mobility-aware applications that selectively cache application components, depending on anticipated loss of connectivity. They can also be used to implement a hoarding tool, along the lines of that used in Coda, which would allow users to explicitly prefetch a given series of objects before a period of expected disconnection. After connectivity has been restored, the replicas would be flushed, causing them to be reconciled with the original server objects.

Redirection

The downcall portion of the D/RIP$_C$ component's API is a superset of the API for the underlying RIP$_C$ component. Because this API depends on the object-oriented middleware framework for which ALICE is being instantiated, it cannot be given here.

To the application, the functionality offered by the D/RIP$_C$ component appears identical to that of the RIP$_C$ component. Both components offer a remote invocation service that allows remote server objects to be invoked. In case the client host does not hold a cached replica of the requested server object, the D/RIP$_C$ component simply passes the invocation to the underlying RIP$_C$ component (possibly through a S/RIP$_C$ component, if present) retaining the usual semantics. This is shown in figure 4.19.

In case the client host does hold a cached replica of the requested server object, the invocation is intercepted by the D/RIP$_C$ component which redirects the client application to the cached replica. Redirection can take place via the same mechanism for client redirection as used for the Swizzling Layer described in section 4.6.3. This is shown in figure 4.20. The client holds a reference to the remote server object S and makes an invocation on that object (arrow 1). The invocation is intercepted by the D/RIP$_C$ component, which detects that S is a remote object for which a replica S' is available

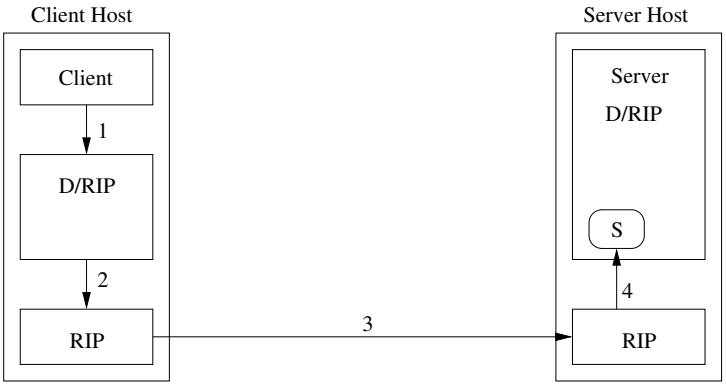


Figure 4.19: Invocation through D/RIP in Connected Mode

Figure 4.20: Invocation through D/RIP in Disconnected Mode

locally and responds with a redirection (arrow 2) to S'. The client then reissues the invocation on the local replica (arrow 3) using the new reference. An alternative to using a redirection scheme is to let the D/RIP_C component act as a proxy, relaying the invocation directly to the replica.

4.7.3 Replication and Reconciliation

This section describes the server-side component of the D/RIP Layer. Unlike the other layers in ALICE, the D/RIP_S component does not take the form of a full-fledged component but of an extension that must be implemented for any server objects that can be replicated and cached. The integration of these two components is required because the tasks performed by the D/RIP_S component—object replication and reconciliation—are highly application-specific. This means that server side support for disconnected operation is very far from transparent.

Replication of Server Objects

As described in section 4.7.2, the D/RIP$_C$ component caches replicas of remote server objects by invoking the D/RIP$_S$ component for the object in question. Any server objects that can be replicated must therefore implement a number of functions that allow this interaction between the two parts of the D/RIP Layer.

IsCacheable (ServerReference) → Boolean returns the value true if the server object identified by the reference can be replicated and returned to the client. Depending on the object-oriented middleware in question, a server object that does not support this operation may throw an exception or in some other way indicate an error condition. The D/RIP$_C$ component should interpret this as if the invocation had returned false.

Replicate (ServerReference) → MarshalledReplica instructs the D/RIP$_S$ component to replicate the server object in question and return the replica in marshalled form to the D/RIP$_C$ component. The D/RIP$_S$ component may record the occurrence of the replication and/or implement a leasing scheme, if this is necessary to ensure the application's consistency requirements.

Figure 4.21 shows a server object S being replicated and the replica returned to the client. Arrow 1 shows the client invoking the **Replicate** method, causing the creation of the replica object S'. Arrow 2 shows the marshalled replica object S' being returned as a result of the invocation.

It should be noted that the **ServerReference** parameters given above may or may not be required for any actual instantiation of the D/RIP$_S$ component. If the **IsCacheable** and **Replicate** operations are implemented as invocations on the server object (a natural solution), the server may already be aware of its own reference and there may therefore be no need also to pass it as a parameter. For instantiations of the D/RIP$_S$ component, it may be advantageous to implement the interaction between the two parts of the D/RIP Layer using the Remote Invocation Protocol itself. The specific techniques that can be used to facilitate object replication in ALICE also depend on the object-oriented middleware architecture for which ALICE is being instantiated. Some architectures (e.g., Java RMI) have better support for object replication and mobility than others (e.g., CORBA), and it will be easier to instantiate the D/RIP$_S$ component for such architectures. Chapter 5 offers detailed discussions of this issue in relation to Java RMI and CORBA and chapter 6 discusses general support for object replication and mobility in popular object-oriented middleware architectures.

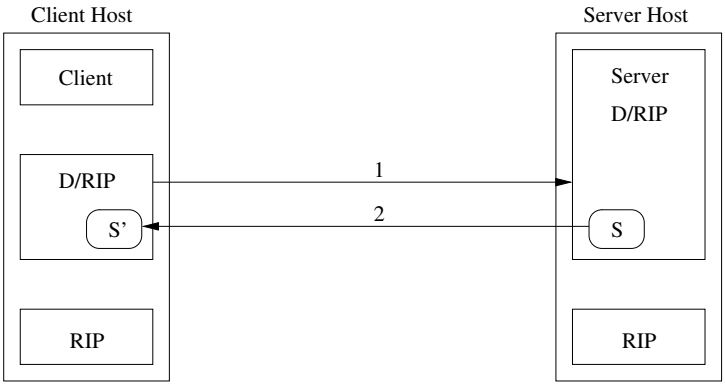


Figure 4.21: Replication of Server Object

Reconciliation of Replicas

As described in section 4.7.2, the D/RIP_C component allows a replica to be reconciled with the authoritative copy of a remote server object. This takes place through invocation of the D/RIP_S component for the object in question. The D/RIP_C component marshalls the object and passes it to the authoritative server object using the following invocation on the D/RIP_S component:

Reconcile (ServerReference, MarshalledReplica) instructs the D/RIP_S component to accept the supplied marshalled replica and reconcile it with the authoritative server object. This will trigger application-specific conflict detection and resolution.

Upon receipt of an invocation of the **Reconcile** method, the D/RIP_S component invokes an application resolver to detect and resolve any conflicts that may have occurred while the replica was being used in disconnected mode. Figure 4.22 shows such a client reconciling a replica S' with the original server object S. Arrow 1 shows the invocation of the D/RIP_S component's **Reconcile** operation. Arrow 2 shows the merging of the two objects S and S'. After the merge, the replica object S' has ceased to exist.

The specific techniques used to facilitate conflict detection and resolution in ALICE depend on the application in question. The approach taken by an instantiation of the D/RIP_S component would ideally be based on experience from systems such as Coda and Rover. A log could be used to record mutating operations performed on replicas, and this log could be transferred to the D/RIP_S component as part of the marshalled replica. If the log is managed from within the replica and returned as part of the marshalled replica, this approach would not affect the interfaces given above. The receiving D/RIP_S component would use this log to detect conflicts in the same manner as Rover. Rover also

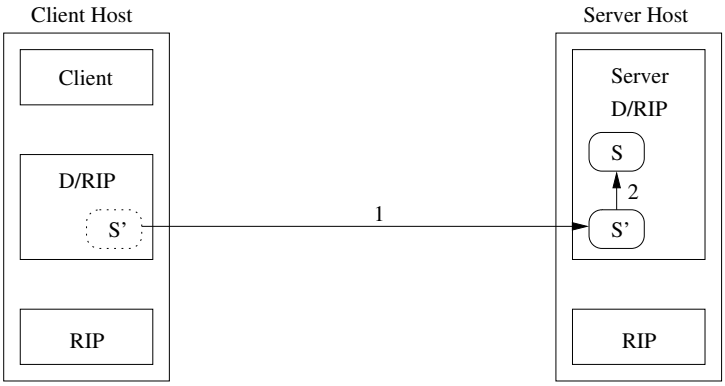


Figure 4.22: Reconciliation of Server and Replica

allows mobility-aware applications to distinguish between tentative (non-reconciled) and committed (reconciled) updates. This feature would also integrate well with an instantiation of ALICE.

4.7.4 Mobility Awareness

The previous sections have described the behaviour of the D/RIP Layer when the client and server are connected and disconnected respectively. The behaviour of the D/RIP Layer is client-driven, i.e., it is the D/RIP_C component that initiates server replication and reconciliation and also changes its behaviour according to whether the layer is operating in connected or disconnected mode. For this reason, it is important for the D/RIP_C component to track its state of connectivity, no matter whether it resides on the MH or the RH.

Like the S/RIP Layer, the D/RIP Layer uses the ML to monitor the MH's state of connectivity. However, where both the S/RIP Layer's components reside on top of the ML (on the MH and MG respectively), one of the D/RIP Layer's components always resides on the RH where no underlying ML is available. This component (whether it be the client or the server component) will not be able to monitor the MH's state of connectivity because there is no underlying ML to issue callbacks. This is not a problem when the server resides on the RH, because the D/RIP Layer is client-driven. In this case, the client-side part of the D/RIP Layer will be able to monitor the MH's state of connectivity through its underlying ML. This is sufficient to allow the client side of the D/RIP Layer to drive the dialogue with the server side.

However, when the client resides on the RH, there is no underlying ML from which it can receive notification about changes in connectivity of the MH. This is a problem, because the client needs to initiate the dialogue with the server as the result of such changes in connectivity. The solution

adopted in ALICE is to let the component of the D/RIP Layer that resides on the MH (and hence on top of an ML) subscribe to callbacks, no matter whether that component is the client or the server component of the D/RIP Layer. If the D/RIP Layer is configured with the server on the MH, the D/RIP_S component will subscribe to callbakcs from its underlying ML and relay such callbacks to the D/RIP_C component on the RH. The D/RIP_C component can then initiate the dialogue with the server.

4.7.5 Mobility Challenges

Of the fourteen mobility challenges discussed in chapter 2, the Disconnected Operation Layer addresses only one aspect of the disconnection problem (section 2.4.2): long-term disconnections. The other aspect of the disconnection problem—short-term disconnections—is addressed by the ML.

4.8 Architecture Configuration

The ALICE layers work by (transparently) modifying the behaviour of object-oriented middleware frameworks or the underlying transports used by the frameworks. In order for these modifications to be possible, each of the ALICE layers places requirements on the object-oriented middleware architecture for which ALICE is being instantiated. For each ALICE layer, the relevant requirements must be satisfied for it to be possible to instantiate the layer for the object-oriented middleware architecture in question. Table 4.3 (an extension of table 4.2) shows the six requirements as well as dependencies between the requirements and the ALICE layers.

In addition to the dependencies shown in table 4.3, there are also inter-dependencies between the ALICE layers. This section first motivates and describes the inter-dependencies between the ALICE layers. Next, we discuss partial instantiations of ALICE and finally map out a diagram describing the rules that govern the composition of the ALICE layers.

4.8.1 Inter-Layer and Requirement Dependencies

This section describes the inter-dependencies between the ALICE layers and summarises the dependencies between the layers and the six requirements discussed in section 4.3. There are no other dependencies than those given here.

The Transport Modules have no dependencies; each sits directly on top of the relevant device drivers included with the operating system of the MH and MG. In case multiple Transport Modules are deployed, there are no inter-dependencies between them.

#	Requirement	Transport Modules	Mobility Layer	Swizzling Layer	Disconnected Operation Layer
R1	The protocol must be client/server-oriented.	•	•		
R2	The underlying transport must be TCP/IP.	•	•		
R3	Redirection of client requests towards a different server location must be possible.			•	•
R4	A server reference must contain several endpoints at which the server can be found. Clients should try endpoints in order.			•	
R5	It must be possible to store some extra information in a server reference. This information must be passed from client to server during invocation.			•	
R6	The object-oriented middleware architecture must support at least weak object mobility.				•

Table 4.3: Requirements on the Middleware Architecture and ALICE Dependencies

The purpose of the Mobility Layer is to manage connectivity between the MH and MGs. The Mobility Layer depends on the Transport Modules for communication between the MH and MG for sending and receiving messages between MHs and MGs. At least one Transport Module is required for the Mobility Layer to work. The Mobility Layer has no other inter-layer dependencies but depends on requirements R1 and R2 from table 4.2 to be satisfied by the object-oriented middleware architecture for which ALICE is being instantiated. R1 and R2 are necessary because the Mobility Layer implements the Berkeley Sockets abstractions as discussed in section 4.3.

The purpose of the Swizzling Layer is to translate server references stored on the MH at key points in time and to redirect client invocations received by the MG. For this reason, the Swizzling Layer must be aware of the state of connectivity of the MH. The Swizzling Layer depends on the Mobility Layer for this purpose. The Swizzling Layer has no other inter-layer dependencies but depends on requirements R3–R5 from table 4.2, as discussed in section 4.3. R3 must be satisfied because the Swizzling Layer needs to redirect client invocations to new server locations. R4 must be satisfied such that the failure of the Swizzling Layer on a single MG (or that MG itself) does not cause the mobile server to become unavailable. R5 must be satisfied such that multi-endpoint references can be generated during redirection responses.

The purpose of the Disconnected Operation Layer is to manage a cache of replicated server objects on the client side for use when the MH is disconncted from the network. For this reason, the Disconnected Operation Layer needs to monitor the MH's state of connectivity, and it depends on

the Mobility Layer for this purpose. The Disconnected Operation Layer has no other inter-layer dependencies but requires R3 and R6 from table 4.2 to be satisfied, as discussed in section 4.3. R3 is required because the Disconnected Operation Layer needs to redirect client requests towards cached replicas instead of the real (unavailable) server objects. R6 is required because the replicas must be moved to the client side before disconnection takes place and back to the server side after connectivity is restored.

4.8.2 Partial Instantiation

Considered as a whole, the ALICE layers work together to address a considerable number of the challenges related to the mobile environment. However, it may be impossible to instantiate all ALICE layers for a given object-oriented middleware architecture, if that architecture fails to meet all of the six requirements. In such cases, it may still be possible to create partial instantiations of ALICE for the middleware architecture in question. For example, consider a hypothetical object-oriented middleware architecture that does not support redirection of client requests and therefore does not fulfil requirement R3. As shown in table 4.2, both the Swizzling and Disconnected Operation Layers depend on R3, and it is therefore not possible to instantiate these layers for the hypothetical middleware architecture. However, if the middleware architecture fulfils R1 and R2 it may still be possible to create a partial instantiation on the basis of the Mobility Layer and Transport Modules. Since it is lacking the Swizzling and Disconnected Operation Layers, the partial instantiation will only have partial support for address migration and disconnection, but it will still have support for the majority of problems related to mobile networking (as shown in table 4.1) through the Mobility Layer and Transport Modules.

While partial instantiations of ALICE may be created out of need, they can also be created voluntarily. Application requirements vary, and for any given application, it may or may not be necessary to avail of the functionality offered by the full range of ALICE layers. For example, a given application may not require support for server mobility and will therefore not need to deal with the address migration problem (section 2.5.1). If ALICE is being instantiated only with client mobility in mind, it is possible to leave out the Swizzling Layer (which deals with the address migration problem for new connections) from the instantiation of the architecture. Another example could be an application that does not require support for long-term disconnection. In this case, the Disconnected Operation Layer can be left out in a similar fashion. The reduction in complexity resulting from such partial instantiations can be expected to result in a better utilisation of mobile device resources.

4.8.3 Framework Configuration

ALICE is an architecture that is instantiated to form a framework. During instantiation, modular reusable software components are created for each of the abstract architectural components described in the remainder of this chapter. A collection of such instantiated components constitutes a framework that can be configured (through composition of the components) for particular application needs.

In general, we expect the configuration of an ALICE framework to be a static affair. The ALICE stack is configured (through the composition of components) before applications are started, and the stack remains unchanged while applications are running. Ideally, it would be possible to add and remove ALICE layers to and from the stack on an ongoing basis as application requirements changed. For example, the Transport Modules and the Mobility Layer could be removed from the stack of a given MH, in case the MH obtained a long-term wire network connection. If an MH that was otherwise well connected needed to be disconnected for an extended period of time, it would be desirable to insert a Disconnected Operation Layer into the stack in a transparent manner. While potentially useful, dynamic configuration of protocol stacks is not directly related to the subject of this thesis and therefore not considered in further detail. Section 7.3 presents some thoughts on potential future work in this area.

4.8.4 Compositional Rules

Instantiations of the ALICE layers can be composed according to a set of rules, governed by the requirements discussed in section 4.3 and the dependencies listed in section 4.8.1. Figure 4.23 shows these rules as a dependency diagram. The minimal instantiation of ALICE uses only the Mobility Layer and the Transport Modules and therefore only requires R1 and R2 to be satisfied. As the figure shows, R1 and R2 are critical; if these are not met by the object-oriented middleware architecture, none of the ALICE layers can be instantiated. The remaining requirements are optional but if satisfied allow a higher degree of mobility support in the resulting ALICE instantiation. For maximum mobility support, all ALICE layers must be instantiated which requires all six requirements to be satisfied.

4.9 Summary

This chapter has presented the ALICE architecture at an abstract level. The architecture allows mobility support to be added to implementations of object-oriented middleware architectures that satisfy a subset of six requirements. The chapter motivated and discussed the requirements and explained the rules that govern the composition of the ALICE layers depending on the requirements

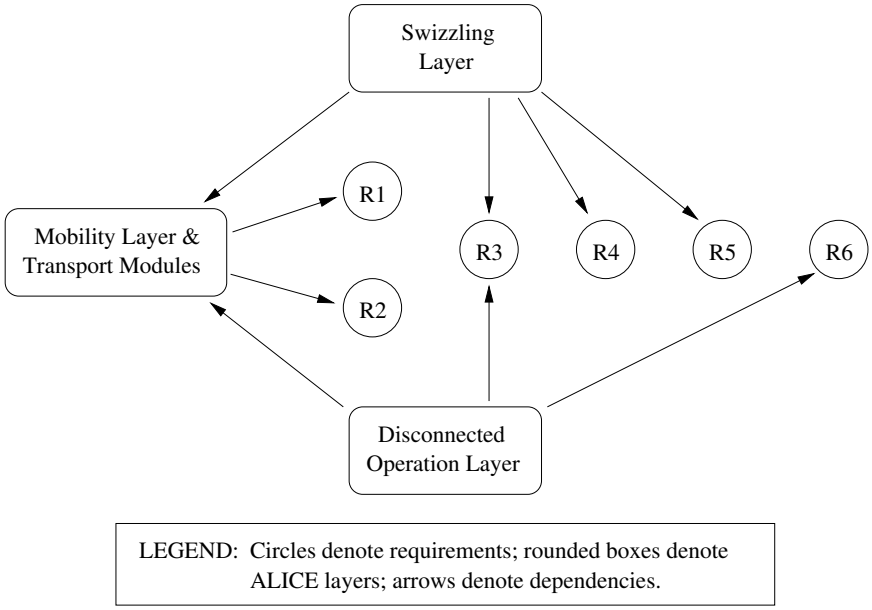


Figure 4.23: ALICE Dependency Diagram

and on interdependencies between the layers. A dependency diagram (figure 4.23) was given that summarised both types of dependencies. The chapter also presented an architectural overview and detailed descriptions of each of the ALICE layers. The exact purpose of each layer was discussed in terms of which of the fourteen mobility challenges (from chapter 2) it addressed, and each layer's responsibilities were given in terms of a functional description. The interfaces of individual layers were also described, as were the interactions with other layers.

The ALICE layers draw upon ideas proposed in related work to solve the problems of the mobile environment. Table 4.4 summarises which of the emerging themes listed in section 2.6 are adopted in which of the ALICE layers. The legend is the same as that used in previous summary tables. Analogously with previous notation, the asterisks in table 4.4 mark situations where an actual implementation of the architecture is required for a meaningful assessment. Such an assessment is given in chapter 5. In relation to table 4.4, it is worth noting that a flexible approach to transparency vs. awareness is a theme that is adopted by three of the four ALICE layers, reflecting a high degree of flexibility in the way the layers can be used. Perhaps the most important characteristics of the architecture are the *separation of concerns* between the different layers and the architecture's *configurability*, i.e., the flexibility with which the layers can be combined. The following sections discuss these in turn.

Emerging Theme	Transport Modules	Mobility Layer	Swizzling Layer	Disconnected Operation Layer	ALICE in Total
Resource Economy	*	*	*	*	*
Communications Economy		•			•
Transparency vs Awareness		•	•	•	•
Location Management			•		•
Disconnected Operation				•	•

Table 4.4: Applicability of ALICE Layers to Emerging Themes

Separation of Concerns

The ALICE architecture divides the support required for using object-oriented middleware in mobile environments into three general areas: connectivity management, location management and support for disconnected operation. The first of these is addressed by the Transport Modules and the Mobility Layer, the second by the Swizzling Layer and the third by the Disconnected Operation Layer. For each of the three areas, solutions were presented, some of which (e.g., support for disconnected operation) drew upon ideas presented in the literature discussed in chapter 3. In addition to the three areas of support offered by ALICE layers, the general support required for distributed object applications was provided by the Remote Invocation Protocol Layer.

The chapter explained how the five layers (the four ALICE layers and the Remote Invocation Protocol Layer) collaborate to address the challenges related to using object-oriented middleware in mobile environments. This discussion is summarised in table 4.5 (identical to table 4.1 but reprinted here for the sake of convenience) which relates each of the ALICE layers to the fourteen mobility challenges from chapter 2. (The Remote Invocation Protocol Layer does not address mobility challenges and therefore does not appear in the table.) A number of observations can be made from the table. First, it shows the Mobility Layer as being key to the architecture. ALICE addresses six out of the nine challenges related to mobile networking and physical mobility, and the ML is involved in all of them. The problem of network heterogeneity is addressed in collaboration with the Transport Modules. The problem of disconnection is addressed in collaboration with the Disconnected Operation Layer. The address migration problem is solved in collaboration with the Swizzling Layer. Hence, in three out of six cases, the ML works in conjunction with other layers to address the challenge in

General Category	Specific Challenge	Transport Modules	Mobility Layer	Swizzling Layer	Disconnected Operation Layer	Full ALICE Support
Mobile Devices (section 2.3)	Battery Power	*	*	*	*	*
	Data Risks					
	User Interface					
	Storage Capacity	*	*	*	*	*
	Processing Power	*	*	*	*	*
Mobile Networking (section 2.4)	Network Heterogeneity	○	○			●
	Disconnection		○		○	●
	Low Bandwidth		●			●
	Bandwidth Variability		●			●
	Security Risks					
	Usage Cost		○			○
Physical Mobility (section 2.5)	Address Migration		○	○		●
	Location-Dependent Information					
	Migrating Locality					

Table 4.5: Applicability of ALICE Layers to Mobility Challenges

question. The separation of concerns between the ML and the other layers is a key characteristic of the architecture. The central positioning of the ML can also be seen from the dependency diagram given in section 4.8.4 in which the ML forms the functional basis for the Swizzling and Disconnected Operation Layers.

While ALICE addresses six of the nine challenges related to mobile networking and physical mobility, it leaves three challenges unaddressed: security risks (section 2.4.5), location-dependent information (section 2.5.2) and migrating locality (section 2.5.3). Section 7.3 presents thoughts on future work on ALICE in relation to these challenges. In addition, the five challenges related to mobile devices (section 2.3) have not been discussed in this chapter. They require an actual implementation to be discussed meaningfully, and their treatment is therefore deferred until chapter 5 where an actual instantiation of ALICE is presented.

Configurability

The process of instantiating ALICE for a given object-oriented middleware framework involves the creation of actual software components for some (or all) of the ALICE layers. Some instantiated layers (the Transport Modules and the Mobility Layer) will be independent of the object-oriented middle-

ware architecture and framework in question, while others (the Swizzling and Disconnected Operation Layers) will be dependent not only on the object-oriented middleware architecture in question but also on the particular framework implementing that architecture being used. The result of instantiating ALICE for a given object-oriented middleware framework is a mobility-enabled version of that framework. This framework is configured through combination of the instantiated layers according to the rules described in section 4.8.4. Frameworks can be configured for a variety of purposes, for example to support particular applications or with particular mobile environments in mind. As will be shown in chapter 6, such mobility-enabled frameworks have the desirable feature that they remain compatible with unmodified frameworks implementing the same middleware architecture.

The six requirements given in section 4.3 determine the level of mobility support that can be achieved by instantiating ALICE for any object-oriented middleware architecture. For full mobility support, all six requirements must be satisfied, but partial mobility support can be provided even for middleware architectures that fail some of the requirements. Table 4.6 (a synthesis of tables 4.1 and 4.2) relates the six requirements to the fourteen mobility challenges. The notation in this table is slightly different from that used in previous summary tables. For any given challenge, the requirements marked by bullets (•) and those marked by circles (○) constitute separate groups. Groups marked by bullets contain mandatory requirements that must be satisfied in order for an ALICE instantiation to address the challenge in question. Groups marked by circles contain optional requirements that if also satisfied offer additional mobility support in relation to the challenge in question. For example, a partial solution to the problem of disconnection can be provided for a middleware architecture that satisfies R1 and R2 (marked with bullets) while a full solution also requires R3 and R5 (marked with circles) to be satisfied. Table 4.6 can also be read the other way. Given knowledge about the requirements satisfied for a given object-oriented middleware architecture, the table shows which of the fourteen mobility challenges will be addressed by a corresponding ALICE instantiation.

Chapter 5 extends this chapter by showing how ALICE can be instantiated for two popular object-oriented middleware architecutures (CORBA and Java RMI) in order to create mobility-enabled object-oriented middleware frameworks.

General Category	Specific Challenge	R1: Client/Server	R2: TCP/IP	R3: Redirection	R4: Multi-Endpoint References	R5: Extra Information	R6: Object Mobility
Mobile Devices (section 2.3)	Battery Power						
	Data Risks						
	User Interface						
	Storage Capacity						
	Processing Power						
Mobile Networking (section 2.4)	Network Heterogeneity	•	•				
	Disconnection	•	•	○			○
	Low Bandwidth	•	•				
	Bandwidth Variability	•	•				
	Security Risks						
	Usage Cost	•	•				
Physical Mobility (section 2.5)	Address Migration	•	•	○	○	○	
	Location-Dependent Information						
	Migrating Locality						

Table 4.6: ALICE Requirements Related to Mobility Challenges

Chapter 5

Instantiation

> *'It seems very pretty,' she said when she had finished it, 'but it's RATHER hard to understand!' (You see she didn't like to confess, ever to herself, that she couldn't make it out at all.) 'Somehow it seems to fill my head with ideas—only I don't exactly know what they are!* [36]

ALICE allows mobility support to be added to different object-oriented middleware frameworks through instantiation of the architecture described in chapter 4. Each such instantiation is specific to the object-oriented middleware framework in question. This chapter presents implementations of a number of the layers described in chapter 4 and shows how they can be used to instantiate ALICE for CORBA. To demonstrate generality, the chapter also outlines how the architecture can be instantiated for Java RMI. Appendix B contains further implementation details for some of the layers discussed here and is referenced where applicable.

Some of the layers presented in this chapter are generally applicable while others are specific to a given object-oriented middleware framework. The approach is to present the generally applicable layers first, which also corresponds to discussing the architecture's layers in bottom-up order. First, section 5.1 describes the software environment on which the implementation described in the following sections is based. Second, section 5.2 presents implementations of transport modules that support a number of wired and wireless communications interfaces commonly found on mobile computers. Third, section 5.3 describes an implementation of the Mobility Layer suitable for managing such transports. The transport and Mobility Layers are generally applicable and can be used for instantiating ALICE for CORBA as well as Java RMI. Sections 5.4 and 5.5 describe instantiations of the Swizzling and Disconnected Operation Layers for use with an implementation of the CORBA Internet Inter-ORB

Protocol (IIOP). A full implementation of the Swizzling Layer is presented, whereas the Disconnected Operation Layer is presented as an outline. Sections 5.6 and 5.7 describe Swizzling and Disconnected Operation Layers that could be implemented to work with a standard implementation of Java RMI. Both of these components are presented as designs. Finally, section 5.8 summarises the chapter.

5.1 Implementation Environment

The implementation language for the transport modules and ML is C, and the Swizzling Layer for CORBA is implemented in C++. The platform used is Linux 2.4 for Intel (Pentium) and iPAQ (StrongArm) processors. For Intel, the most recent development was done on the Debian 3.0 distribution. For iPAQ, the Familiar Linux distribution was used. The GCC 2.95.4 compiler was used for all development. The Intel-based hosts were equipped with Fast Ethernet (100 Mbps) network interfaces and the iPAQ with an 802.11 WLAN (11 Mbps) interface. This environment was chosen mainly because of the author's familiarity with it and because of the availability of suitable hardware of the mobile and stationary variety. The modules described in this chapter contain some Linux-specific code, mainly related to threads and shared libraries. However, this type of code has been isolated as much as possible and explicitly marked in order to facilitate portability of the system to different platforms.

5.2 Transport Modules

This section presents two instantiations of the abstract transport module described in section 4.4. The purpose of each transport module is to manage a connection and transmit data between an MH and an MG under control of the ML. The programming interface chosen for the instantiation is very similar to the abstract interface given in section 4.4 and is therefore not repeated here. A detailed description can be found in appendix B.1. Section 5.2.1 describes a transport module that allows the ML to communicate over serial lines, and section 5.2.2 describes one that uses the User Datagram Protocol (UDP). These particular transports were chosen because of their differences in relation to the way they are typically used as well as typical data rates and message ordering characteristics.

5.2.1 Serial Transport Module

Section 2.4.1 reviewed six mobile and stationary computers and observed that serial transports (in the form of RS232 and USB interfaces) were the only type of transport for which native support was in-

cluded in all devices. Serial interfaces are commonly used to connect mobile and stationary computers in a wired fashion, for example, through docking cradles typically supplied with PDAs. This type of serial communication is connection-based in a very physical manner: the two parties communiate over a fixed point-to-point link and it requires physical intervention to change the connection configuration. Serial communication is by nature ordered, i.e., the bits are guaranteed to arrive in the same order as they were transmitted. In addition, the serial port abstraction is often used for other types of wired and wireless interfaces, such as modems (GSM or analogue) and InfraRed interfaces. Data rates over serial interfaces vary from the 14.4 Kbps commonly used with GSM modems to 115 Kbps commonly used with standard serial cables to 1.5 Mbps which is the highest data rate supported by USB.

Because MGs (which are fixed) as well as MHs need to be equipped with suitable hardware for the two to communicate, the widespread availability of serial interfaces on mobile and stationary computers made the serial transport a natural candidate for implementation as an ALICE transport module. The serial transport module implemented is designed to work with a standard RS232 interface, such as a PDA docking cradle. The module accesses the serial device driver on the host machine and issues `ioctl(2)` calls to set configuration parameters, such as data rates, bit length and handshake parameters. The standard `read(2)` and `write(2)` system calls are used to receive and send data. A configurable timeout value is used to prevent indefinite blocking. Broken connections are detected via a drop in the Data Terminal Ready (DTR) signal on the serial line. A CRC check is invoked for all PDUs received but in the current prototype of the transport module has no effect.

The serial transport serves as an example of a minimal transport module for use with ALICE. It is simple in structure and contains only 382 lines of C code. Apart from being easy to understand, the serial transport module would be easy to use as a template for building transport modules to work with InfraRed transceivers or Hayes-compatible modems (GSM or analogue).

5.2.2 UDP Transport Module

UDP offers an unreliable, connectionless datagram service on top of IP. UDP is often used as an alternative to TCP for example for real-time data streaming. Because UDP runs on top of IP, it is available for a variety of transports, including wired networks (e.g., Ethernet and ATM), wireless networks (e.g., 802.11 WLAN) and even serial lines through the use of the Point-to-Point Protocol (PPP) [169]. UDP offers a point-to-point service between the sender and the receiver of datagrams, but unlike serial communiation, the physical communication path is not fixed: UDP requires the use of explicit source and destination addresses which can be changed in software. UDP communication is by definition unordered and unreliable, meaning that datagrams may arrive at their destination in a

different order from that in which they were sent, or they may not arrive at all. The typical data rate for UDP services depends on the underlying network technologies, but is typically higher than that obtained with serial communication. Common data rates based on UDP over conventional networks range from 10 Mbps (e.g., Ethernet or 802.11) to 1 Gbps (e.g., GigaBit Ethernet).

Like the serial ports practically ubiquitous on mobile computers, network interfaces that support UDP are extremely common on stationary computers and typically also available on mobile computers. As an underlying transport, UDP is different from serial lines in many respects, such as in relation to connection-orientedness, reliability and typical data rates. These characteristics makes UDP a strong candidate for implementation as an ALICE module. First, UDP can be used over a variety of network interfaces (e.g., Ethernet and 802.11 WLAN), and an ALICE transport module based on UDP can therefore be expected to have several uses. Second, a UDP transport module constitutes a good supplement to the serial transport module for the purposes of illustration, because the differences between the two modules demonstrate the variety of transports that can be supported as ALICE transport modules. Although it would have been easier to implement a transport module based on TCP than UDP, the idea was discarded because standard implementations of TCP have been shown to perform poorly over wireless links [30].

The UDP transport module accesses the network interface through the Berkeley Sockets interface in the conventional manner. On the MG, the configuration parameters include a local interface name and a port number on which to wait for incoming datagrams from MHs. On the MH, the configuration parameters include the hostname and port number of the corresponding transport module on the MG. The two parties must agree on which port on the MG is used for the purpose of communication. While the transport module allows the port number to be set on a per-host basis at runtime (using configuration files), it is convenient to use a well-known port for this purpose. Because UDP is connectionless, the transport uses the time delay between datagrams to decide whether the 'connection' between the MH and MG still exists. If no traffic is received for a certain duration (the timeout value), the 'connection' is assumed lost. The timeout value is set at compile time. A value of 30 seconds worked well during testing of the module. Both parties have the ability to send keepalive datagrams in the absence of other traffic. A CRC check is invoked for all PDUs received but in the current prototype of the transport module has no effect.

Like the serial transport module, the UDP transport module is relatively simple, containing only 448 lines of C code. It could be used as a template for other connectionless, unreliable transports.

5.3 The Mobility Layer

This section presents an instantiation of the Mobility Layer (ML). The instantiation follows the abstract description from section 4.5 closely, and some details of the layer's operation are therefore not repeated here. Additional implementation details, including specific interface and protocol definitions, are given in appendix B.2.

The ML and the transport modules described in section 5.2 cooperate to address the problems related to mobile networking (section 2.4). The problems are addressed through socket proxying and through connection tunneling and handoff, as described in section 4.5. The ML has two components that reside on the MH and MG, respectively. Implementations of these are described in sections 5.3.1 and 5.3.2. Section 5.3.3 describes how the two components interact in order to support socket proxying, and section 5.3.4 describes interaction between different ML_{MG} components in order to support handoff and tunneling.

5.3.1 The ML_{MH} Component

The ML_{MH} component takes the form of a shared library with which applications are linked and a daemon process that runs on the MH. The library contains a series of functions with the same signatures as the Berkeley Sockets functions. In addition, there are two functions to register and cancel callbacks from the library. These callback registration functions constitute an extension to the Berkeley Sockets API. During application development, an ML_{MH} header file is used instead of the standard Berkeley Sockets header file. During linking, the shared ML library replaces the Berkeley Sockets library. The Berkeley Sockets replacement functions communicate with the daemon process running on the MH. This daemon manages the communications interfaces on the MH and is responsible for interacting with the ML_{MG} component. The implementation works such that the Berkeley Socket operations performed through the ML_{MH} component take place on the MH but also result in extra functions being performed on the MG. For example, when a mobile application requests a socket to be created, a socket is first created on the MH and subsequently on the MG. The two sockets are linked via a logical identifier shared between the ML_{MH} and ML_{MG} components. Any change in state of one of the two sockets (e.g., a connection being created or the socket being closed) is relayed to the other socket. All the standard Berkeley Sockets functions can be used through the ML, but the implementation supports only the TCP portion of the Berkeley Sockets interface. There is no support for UDP sockets. Further implementation details are given in appendix B.2.1.

As discussed in section 4.5.4, the ML_{MH} component contains a tuning API with functions that

allow the component to be monitored and controlled by an external component. Although functions exist in the current ML_{MH} instantiation, they have not been implemented. The interface definitions are given in appendix B.2.2.

5.3.2 The ML_{MG} Component

The ML_{MG} component takes the form of a daemon process that runs on the MG. It manages a set of transports (i.e., instantiations of ALICE transport modules) through which it can receive connections from MHs. Communication between ML_{MG} daemons (e.g., for the purposes of handoff and tunneling) is not performed through these transports but directly through the MG's TCP/IP stack.

As described in section 4.5.4, applications running on the MG (such as Swizzling Layers) can register a callback function to receive notification through the ML_{MG} library when changes related to MHs occur. In the implementation presented here, this functionality has been implemented in a very simple fashion. An application that runs on the MG and wishes to register a callback function creates a TCP connection to a well-known port where the ML_{MG} component is listening. The application subsequently receives a stream of bytes and marshalled parameters that constitute a simple stream of mobility events. The ML_{MG} component can signal the arrival and departure of MHs, the creation and destruction of server sockets as well as the occurrence of handoffs. A detailed description of this interface is given in appendix B.2.3.

5.3.3 $ML_{MH} \leftrightarrow ML_{MG}$ Interaction

The ML_{MH} and ML_{MG} daemons communicate using a simple protocol that allows the MH to use an MG as a proxy in the manner described in section 4.5.2. This protocol allows the ML_{MH} daemon to perform the following functions: connect to and disconnect from an ML_{MG} daemon; to create and destroy client as well as server sockets on the MG; to make outgoing connections to RHs from client sockets; to accept incoming connections on MG server sockets and relay them to the mobile server; and to send and receive data over open connections.

The data and commands required for these operations are contained in nine types of messages or Protocol Data Units (PDUs). The messages are transmitted using the transport modules described in section 4.4. Each transport module implements an unreliable datagram service where datagrams may be lost or arrive out of order but where each datagram delivered to the ML is guaranteed to have passed an integrity check. The messages are equipped with sequence numbers that allow each message to be acknowledged individually and to allow the reordering of messages that arrive out of order. The sequence numbers also allow different messages belonging to the same stream of data to be sent over

different transports and be reordered by the receiving ML. Positive and negative acknowledgements are used to allow MLs to acknowledge the receipt of individual messages and request retransmission of particular messages. Each of the two parties stores transmitted messages until they have been acknowledged by the corresponding party. If a message marked as sent has not been acknowledged or requested retransmitted by the receiving party, the message, acknowledgement or retransmission request can be considered lost, and the sending party retransmits it.

Because the transport modules offer a point-to-point (between the ML_{MH} and ML_{MG} daemons) service, no routing at the ML level is required, and the source and receiver identities do not have to be contained in each message. However, for messages that are associated with a particular ML socket, an identifier is included to identify which ML socket on the receiving side the message is intended for.

Appendix B.2.4 gives a detailed description of the protocol used for communication between the ML_{MH} and ML_{MG} daemons.

5.3.4 $ML_{MG} \leftrightarrow ML_{MG}$ Interaction

A simple protocol is used for communication between MGs in order to facilitate handoff of MH sessions between MGs, including tunneling of open connections as described in section 4.5.3. The protocol contains only two messages: one to initiate handoff (and transfer connection state between MGs) and one to create a tunnel. Because the MGs are assumed to be connected via the Internet or a LAN, the protocol used for inter-ML_{MG} communication is built on top of TCP, which constitutes an abstraction that is easier to work with than an unreliable datagram service as offered by the ALICE transport modules. For this reason the inter-ML_{MG} protocol does not need to implement reordering or detection of lost or duplicate datagrams, because these functions are already being performed by TCP. Appendix B.2.5 gives a detailed description of the implementation of the inter-ML_{MG} protocol.

5.4 The IIOP Swizzling Layer

This section presents the S/IIOP Layer, a CORBA-specific instantiation of the abstract Swizzling Layer described in section 4.6. As described in section 4.6, the abstract Swizzling Layer consists of two components: an S/RIP_{MH} component that performs server reference management and an S/RIP_{MG} component that performs client redirection. Together, the two components collaborate to solve the address migration problem (section 2.5.1) for new connections. The S/IIOP Layer presented here follows this separation of concerns; it consists of two components that reside on the MH and MG respectively and perform CORBA-specific server reference management and client redirection.

We first introduce the server reference format and remote invocation protocols used in CORBA, then present the CORBA framework on which the Swizzling Layer is based and finally describe the two components of the S/IIOP Layer.

5.4.1 CORBA

CORBA server references are called Interoperable Object References (IORs). Clients invoke servers using the Internet Inter-ORB Protocol (IIOP). This section introduces the server reference format and remote invocation protocol used in CORBA and identifies a number of problems related to the use of IIOP in mobile environments.

CORBA Server References

The IOR format is specified in section 13.6 of the CORBA 3.0 specification [83] and reproduced in figure 5.1 in the OMG's IDL. As mentioned in section 3.4, an IOR constitutes a globally unique identifier for an object. IORs contain a series of profiles that identify endpoints at which the server can be found. CORBA 3.0 specifies three different types of profiles also shown in figure 5.1. Of these, `TAG_INTERNET_IOP` is of particular interest, since profiles of this type are used by TCP-based servers. (Recall that ALICE according to requirement R2 assumes the transport underlying the object-oriented middleware framework to be TCP/IP.) Servers with an IOR containing a `TAG_INTERNET_IOP` profile can be invoked via the Internet Inter-ORB Protocol (IIOP), which is the protocol used for performing CORBA invocations over the Internet.

The format for IIOP profiles is described in 15.7.2 of the CORBA 3.0 specification [83] and shown in figure 5.2. The figure shows that IIOP profiles contain the hostname of the host holding the server object, a port number identifying the server process within that host and an object key identifying the object within the server process. The `components` field contains additional information that may be used in making invocations on the object described by the profile. It is not required for the S/IIOP Layer and therefore not discussed further here.

CORBA Remote Invocation Protocols

In the CORBA standard, objects are hosted by servers known as Object Request Brokers (ORBs). ORBs perform a number of tasks, including the management of objects themselves and the routing of requests from clients to servers and responses in the opposite direction. Initially, CORBA made no provision for interoperability between ORBs supplied by different vendors. Later versions of the standard addressed this issue by defining a standard protocol for inter-ORB communication which

```
typedef unsigned long           ProfileId;

struct TaggedProfile {
  ProfileId                     tag;
  sequence <octet>              profile_data;
};

// an Interoperable Object Reference is a sequence of
// object-specific protocol profiles, plus a type ID.

struct IOR {
  string                        type_id;
  sequence <TaggedProfile>      profiles;
};

const ProfileId                 TAG_INTERNET_IOP = 0;
const ProfileId                 TAG_MULTIPLE_COMPONENTS = 1;
const ProfileId                 TAG_SCCP_IOP = 2;
```

Figure 5.1: CORBA 3.0 definition of IORs (IDL) [83]

```
struct Version {
    octet                       major;
    octet                       minor;
};

struct ProfileBody_1_0 { // renamed from ProfileBody
    Version                     iiop_version;
    string                      host;
    unsigned short              port;
    sequence <octet>            object_key;
};

struct ProfileBody_1_1 { // also used for 1.2 and 1.3
    Version                     iiop_version;
    string                      host;
    unsigned short              port;
    sequence <octet>            object_key;
    sequence <IOP::TaggedComponent> components;
};
```

Figure 5.2: CORBA 3.0 definition of IIOP Profiles (IDL) [83]

is known as the General Inter-ORB Protocol (GIOP) [83] and which can be mapped onto different underlying transports. The OMG also defined a mapping of GIOP onto TCP/IP known as the Internet Inter-ORB Protocol (IIOP) [83]. IIOP enables invocations to be relayed between different ORBs over TCP/IP and must be supported by all ORBs compliant with CORBA 2 and higher.

IIOP defines the minimum protocol necessary to transfer invocations between ORBs. IIOP makes a distinction between clients and servers in a request/reply interaction. A client creates an IIOP connection to a server and sends request messages to which the server typically responds with a corresponding reply message. The client is prohibited from sending reply messages as is the server from sending request messages. The server may act as a client by opening a different connection to another server.

IIOP specifies eight message types that provide the capability to transparently locate and invoke the methods of a server object. A method is invoked using an IIOP `Request` message. The `Request` message identifies the object being invoked and also contains the symbolic name of the method. In addition, the `Request` message contains any parameters passed to the method. Any results from the invocation are returned in an IIOP `Reply` message. A client can cancel an outstanding request by sending a `CancelRequest` message to the server. In case the server object no longer resides at the location specified in the IOR, the server can return a `LOCATION_FORWARD` reply containing a new IOR for the server object. IIOP also provides a `LocateRequest` message which can be used to check an object's location before proceeding to invoke its methods. The server replies with a `LocateReply` message containing the current location of the object in the form of a new IOR.

IIOP messages are transmitted using a well-defined transfer syntax called the Common Data Representation (CDR). CDR maps data types into a low-level representation for 'on the wire' transfer between clients and servers. Simple as well as complex data types (including IORs) can be marshalled into CDR format. IORs can also be 'stringified,' meaning that they are marshalled into a string form. A stringified IOR can be transmitted via non-CORBA means such as being written to a file, sent by email or published on a web page.

IIOP in Mobile Environments

IIOP is designed for a static environment and using it in a mobile setting is not straightforward. Specifically, IIOP suffers from the following problems related to various characteristics of mobile environments:

1. It is assumed that IIOP servers rarely (or never) change their transport connection endpoints, i.e., DNS names and IP addresses.

2. Both IIOP and transport connections are assumed to break very rarely. IIOP is heavily connection-oriented but has no support for resuming a broken IIOP connection over a different transport connection. When a transport connection breaks, the IIOP connection's state is irrevocably lost. This will typically result in the states of the client and server becoming inconsistent.

3. Because IIOP assumes a single underlying transport connection for the lifetime of an IIOP connection, there is no means of changing network interface (e.g., from GSM to Ethernet) during an IIOP connection without breaking it.

In addition, some sources have found that IIOP assumes transport connections to have a relatively high bandwidth. Notably, the DOLMEN project (discussed in section 3.4.3) observed that the IIOP encoding format is designed to be easy to use rather than to optimise bandwidth utilisation and consequently proposed to optimise the IIOP data format for use over wireless links [112]. However, Adwankar has examined GIOP/IIOP performance over wireless links and concludes that the message formats 'are already sufficiently optimized and further optimization will only result in reduced operability' [8].

5.4.2 IIOP Implementation

The S/IIOP Layer presented here is designed to work with a particular implementation of IIOP written by Cunningham [47] in 1998. Hence, the S/IIOP Layer is specific not only to CORBA but also to the particular implementation of CORBA used. Cunningham's IIOP implementation is called 'the IIOP Layer' and consists of a library written in C++ that implements IIOP 1.1. Applications using the IIOP Layer include a series of header files and are linked against the library in the usual manner. While designed for the purpose of supporting CORBA on mobile devices, the IIOP Layer contains no explicit mobility support features. Rather, it implements a minimal set of functionality intended to allow mobile applications to implement a subset of the functionality typically found in a full CORBA ORB. The intention is that the majority of mobile applications will not require full ORB functionality and that a considerable reduction in resource usage on the MH can be achieved by replacing the full ORB with a smaller toolkit on the MH. Hence, Cunningham's IIOP Layer precedes the OMG's work on MinimumCORBA (section 3.4.1) but builds on a similar philosophy.

The IIOP Layer implements the IIOP protocol independently of mobility and can be layered either above a standard implementation of TCP/IP for use in a traditional Internet environment or above the ML for use in a mobile environment supporting client objects on mobile devices. The S/IIOP Layer described here can be layered above the IIOP Layer in order to solve the address migration

problem for new connections and thereby also allow server objects to be hosted on mobile devices.

5.4.3 Server Reference Management

This section describes how CORBA IORs map on to the minimal server reference format discussed in section 4.6.2. First, we analyse the format for IORs and discuss how it satisfies requirements R4 and R5 from section 4.3 and how the processes of swizzling, reswizzling and unswizzling can be performed on them. Finally, we describe the implementation of the S/IIOP$_{\text{MH}}$ component, which is responsible for managing server references.

Swizzling IORs

The CORBA IOR format described in section 5.4.1 maps nicely onto the minimal server reference format given in section 4.6.2. The IORs correspond to the ALICE concept of a 'reference' and the IIOP profiles to ALICE 'endpoints.' The content of IIOP profiles correspond to the content of ALICE endpoints: the CORBA `host`, `port` and `object_key` fields map to the ALICE 'hostname,' 'port' and 'key' fields respectively. The fact that IIOP profiles also can contain an additional field (`components`) poses no problem.

The CORBA IOR format fulfils requirements R4 and R5 from section 4.3. R4 is fulfilled because the IOR format allows several profiles to appear in the IOR and because CORBA semantics specify that clients should try them in sequence. R5 is fulfilled because the `object_key` field can be used to store extra (ALICE-specific) information that is passed from client to server during an IIOP invocation. The `object_key` field is a string and therefore of flexible length.

Because the IOR format fulfils requirements R4 and R5, IORs can be swizzled, reswizzled and unswizzled. This is shown in figure 5.3, which corresponds closely to figure 4.17. Figure 5.4 shows an IOR being swizzled and unswizzled, corresponding closely to figure 4.18. The home agent endpoint appears as a separate IIOP profile, which is inserted into the IOR after the other profiles during swizzling and removed from the IOR during unswizzling.

Section 4.6.2 described that swizzling occurs when a server reference is created while the MH is connected to an MG. In the S/IIOP Layer, it turned out that it was easier to maintain IORs in an unswizzled state and swizzle them before they were exported from the IIOP Layer through marshalling. After an IOR has been exported, it is unswizzled. The result is equivalent to maintaining the IOR in the swizzled state at all times.

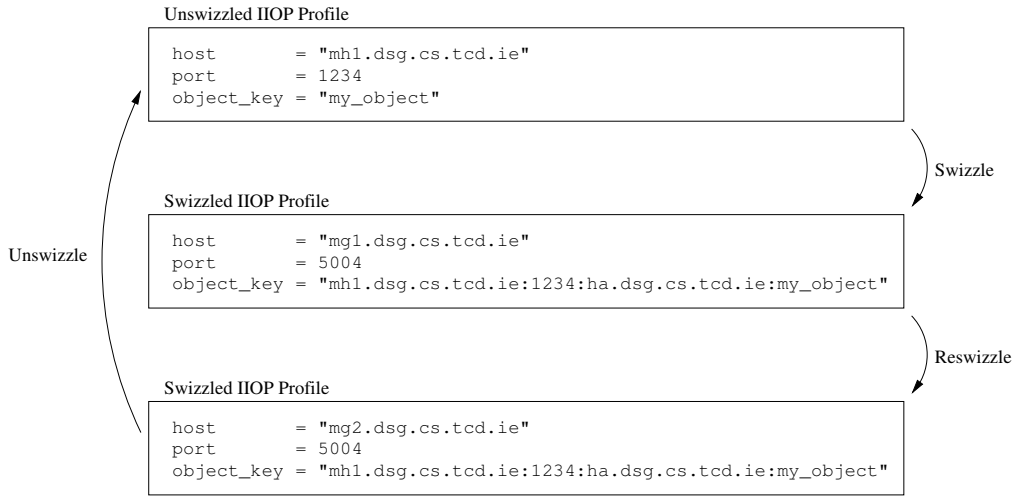


Figure 5.3: Swizzling, Reswizzling and Unswizzling of an IIOP Profile

Implementation of the S/IIOP$_{MH}$ Layer

As shown in the architectural overview in figure 4.4, the part of the Swizzling Layer that resides on the MH implements the same downcall API as the underlying protocol employed by the object-oriented middleware for which ALICE is being instantiated. Given an implementation of this protocol, a Swizzling Layer must be written to match. This makes it possible to insert a Swizzling Layer between an application and the protocol in a transparent fashion.

The full API for Cunningham's IIOP Layer is given in [47]. It is an object-oriented API that features C++ classes corresponding to key CORBA abstractions such as IORs, CDR streams and the eight IIOP messages. The S/IIOP$_{MH}$ component is implemented as a library, which is used by mobile servers instead of the normal IIOP library. The S/IIOP$_{MH}$ library has a downcall interface that mirrors the downcall interface of the IIOP Layer and works by overriding three methods in a total of two key classes from the IIOP library. In addition to the downcall interface, the S/IIOP$_{MH}$ library also has an upcall interface that is used to receive callbacks from the underlying ML$_{MH}$ library. Essentially, this API consists of a single callback function that is invoked by the ML$_{MH}$ library whenever a change in connectivity occurs. Such callbacks are used to trigger swizzling operations. Because the S/IIOP$_{MH}$ library uses ML callbacks, it constitutes a mobility-aware application. Detailed descriptions of the S/IIOP$_{MH}$ library's downcall and upcall interfaces (including overridden methods and callback functions) are given in appendices B.3.1 and B.3.2.

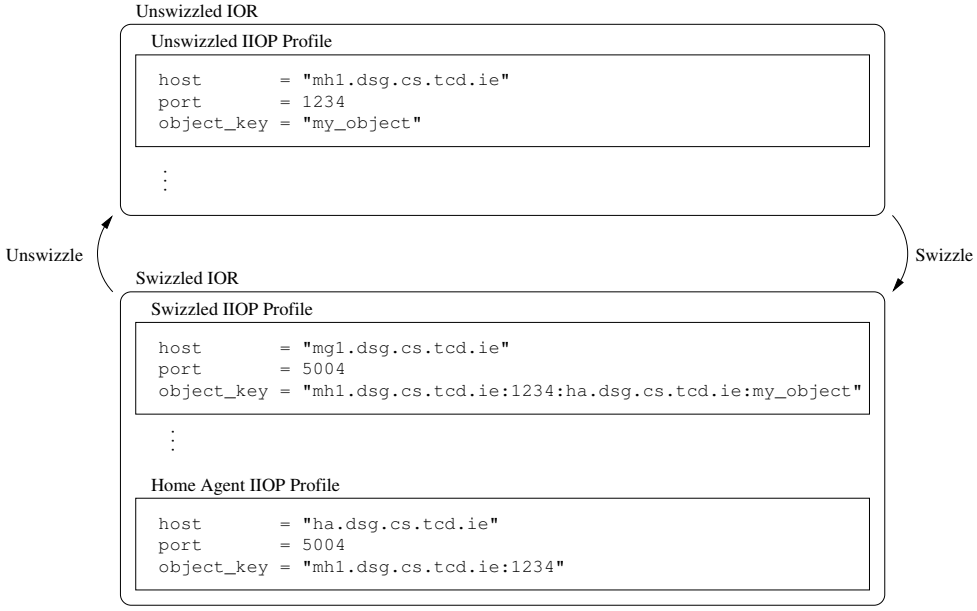


Figure 5.4: Swizzling and Unswizzling of an IOR

5.4.4 Client Redirection

This section describes how IIOP allows client redirection to be performed in the manner described in section 4.6.3. First, we discuss how the redirection features of IIOP satisfy requirement R3 from section 4.3. Next, we discuss how the forwarding pointers stored at MGs are managed. Finally, we describe the implementation of the S/IIOP$_{\mathrm{MG}}$ component, which is responsible for performing client redirection.

Redirection with IIOP

As discussed above, IIOP features two mechanisms through which clients can be redirected: the `LocateReply` sent in response to a `LocateRequest` and a `LOCATION_FORWARD` reply sent in response to an ordinary `Request` message. In both cases, a new IOR is returned to the client. These facilities are sufficient to satisfy requirement R3 from section 4.3. When the S/IIOP$_{\mathrm{MG}}$ component receives an invocation from a CORBA client, it examines the object key (supplied with the invocation) and decides whether the MH specified in the key is local to this MG. If so, the S/IIOP$_{\mathrm{MG}}$ component tunnels the connection to the locally proxied server port on the MG. This in turn triggers the connection to be forwarded (by the ML) to the mobile server, which completes the invocation. Since the connection takes place through the ML, the ML's support for tunneling and reconnection applies to the incoming

connection. If the mobile server has moved to a different MG, a forwarding pointer will exist, which allows the S/IIOP$_{MG}$ component to generate a new IOR that points to the new MG. This IOR is returned to the client in a `LocateReply` or `LOCATION_FORWARD` as appropriate.

The S/IIOP$_{MG}$ component is not aware of departures of MHs, only handoffs. If the MH has moved away but not reconnected, any incoming invocations will block until the MH reconnects to an MG. If the point of reconnection is a different MG, the blocked invocation will be tunneled to the server via the new MG. An alternative approach could be to let the S/IIOP$_{MG}$ component indicate that the server is temporarily unavailable, e.g., by raising a `CORBA::TRANSIENT` exception (described in section 4.12.3.18 of the CORBA 3.0 specification [83]). The client receiving such an exception is expected to reissue the invocation later. If the MH has connected to a different MG when the client reissues the invocation, the client could be redirected to the new MG instead of having the invocation relayed through a tunnel. Raising a `CORBA::TRANSIENT` exception therefore not only allows the client to take alternative action (such as alerting a user that the invocation is temporarily delayed), but also offers superior fault tolerance and performance by avoiding the tunnel. The cost of the solution is a slight increase in complexity in the S/IIOP$_{MG}$ component and the client in order to raise and interpret the exceptions. For reasons of simplicity, the current implementation of the S/IIOP$_{MG}$ component does not raise exceptions, but the ability to do so is clearly a desirable property. Section 7.3 offers thoughts on future work in this context.

Management of Forwarding Pointers

The S/IIOP$_{MG}$ component allows forwarding pointers to be updated as described in section 4.6.2. The `setLocation()` method is implemented as a CORBA server running on a well-known port. It constitutes the S/IIOP$_{MG}$ component's tuning interface and is described in appendix B.3.4.

Implementation of the S/IIOP$_{MG}$ Layer

The S/IIOP$_{MG}$ component is implemented as a daemon that runs on all MGs. In addition to the tuning interface that allows forwarding pointers to be updated, the daemon has an upcall interface through which it receives callbacks from the underlying ML. These upcalls allow the S/IIOP$_{MG}$ daemon to monitor the behaviour of MHs (in terms of arrivals, departures, handoffs and the creation and destruction of server sockets) and thereby keep its forwarding pointers updated. The upcall interface is very similar to the one given for the abstract Swizzling Layer discussed in section 4.6 and therefore not repeated here. A detailed description is given in appendix B.3.3.

5.5 The IIOP Disconnected Operation Layer

This section presents a detailed outline of how a Disconnected Operation Layer for CORBA could be implemented. The outline is based on the IIOP Layer that also served as the basis for the CORBA Swizzling Layer presented in section 5.4. The D/IIOP Layer presented here follows closely the generic Disconnected Operation Layer presented in section 4.7. A client (residing either on the MH or the RH) requests a replica from a server object, uses the replica for a period of time (typically in disconnected mode) and finally reconciles the replica with the original server object. The scheme requires object mobility because the replica needs to be exchanged between the client and server.

Section 5.5.1 discusses possible approaches to object mobility in CORBA. Sections 5.5.2–5.5.5 present a scheme for performing replication, cache management, redirection and reconciliation in the context of CORBA. Finally, section 5.5.6 describes the callback feature used to maintain the client-driven nature of the D/IIOP Layer dialogue when the server is on the MH.

5.5.1 Object Mobility in CORBA

Object mobility (requirement R6) is required to move server functionality to and from the client side at different points in time. Minimally, weak object mobility must be supported, meaning that it must be possible to transfer the code and state of objects. It is not necessary to support strong object mobility, i.e., to transfer computation such as running threads.

Like other object-oriented middleware architectures, CORBA addresses the problem of heterogeneity. CORBA allows any platform and any programming language to be used for implementing clients and servers. Hence, CORBA maintains the heterogeneous nature of the execution environment but hides it beneath standardised ways of describing interfaces and interacting with objects. This 'thin layer of homogenisation' can be contrasted with the more 'deeply homogenised' execution environment used with Java RMI. RMI clients and servers are written in the Java language, compiled into Java byte code and run inside JVMs. While CORBA offers developers flexibility in choice of platforms and languages, its shallow homogenisation of the execution environment makes object mobility problematic. It is difficult to offer general support for object mobility when objects can be written in any language and be running on any platform. Despite this problem, a number of OMG initiatives have dealt with supporting object mobility in CORBA, especially in relation to mobile agents. The following discusses the most significant of those and presents a possible approach on which an ALICE Disconnected Operation Layer for CORBA could be based. For a more detailed treatment of the initiatives, see the specifications cited below and a comparative report by Choy et al. [42].

Life Cycle and Externalization Services

The CORBA Life Cycle Service [80] defines conventions for creating, deleting, copying and moving objects. Moveable objects implement the `LifeCycleObject` interface which contains a `move` operation that allows the object to move. Locations are identified by `FactoryFinder` objects, and the `move` operation takes as parameter an object implementing this interface. The `FactoryFinder` object is also used to locate a `Factory` object, which is able to create the moving object on the receiving side.

The CORBA Externalization Service [78] defines protocols and conventions for marshalling ('externalizing') and unmarshalling ('internalizing') objects. An externalised object can be transmitted across the network as a stream of octets.

Choy et al. observe that it is possible to implement an object mobility scheme based on the CORBA Life Cycle, Externalization and Name Services [42, pp.12–14]. It is up to the application objects themselves to provide the code that implements the externalisation and internalisation of object state (using the interfaces provided by the Externalization Service), to distribute new IORs (using the Name Service) and to make sure the object's code is available on the receiving side. Choy et al. note that there is no CORBA service that facilitates this type of code management and therefore outline three possible approaches: (1) transmit the code as a parameter to the `Factory.create()` method, (2) make the code available in an implementation-specific way such as the Java class loader mechanism and (3) introduce a new CORBA service to facilitate code loading.

Objects By Value

Standard CORBA objects cannot be passed by value, only by reference. Objects By Value (OBV) [76, 83] is a CORBA initiative that aims to remedy this by providing functionality similar to the 'pass-by-value' semantics used in many programming languages. OBV is specifically designed to facilitate (although not directly provide) caching and replication. On this topic, the authors of the OBV specification write:

> We assume that the purpose of this feature [OBV] is NOT to implement replication and/or caching [...] but we believe that more complex caching and replication functionality can be built on the top of the feature set proposed here. [76, p.5–21]

An important goal of OBV was to provide good support for Java users of CORBA. To enable the pass-by-value semantics, OBV extends CORBA and IDL with the notions of 'valuetype' and 'abstract interface type.' Valuetypes allow the passing of an object by value. They are in many ways equivalent to regular IDL interface types but they are also an indication to the developer that they will have

some extra properties; they will have state associated with them and also an implementation that may be required to move. These properties put extra requirements on the valuetype beyond that of a normal IDL type. It is important to remember that valuetypes are not CORBA objects and that moving a valuetype is not the same as moving a CORBA object [182]. When invoking a valuetype (as opposed to a CORBA object), a local implementation is always used and invocations are not CORBA (i.e., IIOP-based) invocations. Abstract interfaces allow a developer to specify if an operation can explicitly support receiving either a valuetype or an interface at runtime.

As such, OBV does not support the movement of CORBA objects. Instead, objects are transformed into 'object values' that can be moved. At the destination, valuetypes are not converted back into CORBA objects but remain valuetypes. One could claim that the name 'Objects By Value' is a slight misnomer because, although it lets objects be passed by value, they do not remain objects in the CORBA sense.

D/IIOP Approach

The two OMG approaches reviewed above demonstrate the complexity of supporting object mobility in CORBA. OBV is unsuitable as a means of implementing mobility of CORBA objects and therefore cannot easily be used to instantiate D/RIP for CORBA. The method proposed by Choy et al. will allow CORBA objects to be moved with considerable support from three of the CORBA Services. These services are likely to make significant demands in terms of processing power, memory and bandwidth on both the sending and the receiving devices, and it is therefore questionable whether the level of support required will harmonise well with the type of mobile environment discussed in chapter 2.

For these reasons, the D/IIOP instantiation presented here is based on a simple scheme intended to require less runtime support from the environment than the one proposed by Choy et al. Other instantiations of D/IIOP may also be possible, based for example on the proposal by Choy et al. The scheme proposed here supports multiple platforms in a non-transparent manner, meaning that the heterogeneity of the execution environment (if any) is visible to applications. Text strings are used to identify different architectures, and a client that wishes to replicate a server object must specify its architecture as part of the replication request. The server in turn maintains a series of replicas suitable for execution on different architectures. These replicas are transmitted between the client and server as binary octet streams that form executable programs.

5.5.2 Replication

Replicas are full CORBA objects that run on the client side. They are returned by D/IIOP$_S$ components to D/IIOP$_C$ components as the result of replication requests. Replicas are transmitted in marshalled form as CORBA octet sequences constituting binaries that the D/IIOP$_C$ component can execute. A marshalled replica contains code and state, which are transmitted as two different result values in the replication response. The code may be a native binary that will run directly on the client host as a separate process, or it may be a binary that will run inside a virtual machine on the client. When the D/IIOP$_C$ component issues the replication request, it specifies an architecture for which the server will supply a compatible binary. The replica state returned in the replication response contains all the state the replica needs to operate on the client side. The state is marshalled in an application-specific manner.

The server maintains a code depository containing replica binaries suitable for different client architectures. Different binaries implementing the same replica must use the same format for storing state. Architectures are identified using text strings, such as `tcl` or `linux-i686`. It is possible for the server to raise an exception in case it does not support the architecture specified by the client in a replication request. In highly heterogenous environments (i.e., where many client architectures are used), maintaining separate binaries for different architectures will cause extra work for application developers. However, it has the advantage that it allows replication and reconciliation in homogenous as well as heterogenous execution environments. With this approach, the D/IIOP Layer can support a homogenous execution environment, such as a Java virtual machine, simply by offering Java byte code as one architecture. Appendix B.4.2 gives the detailed interface for the D/IIOP$_S$ component, which allows replicas to be obtained and reconciled.

5.5.3 Cache Management

Upon receiving a replica binary, the D/IIOP$_C$ component starts it, passing three parameters: an IOR$_D$ identifying the D/IIOP$_C$ component, the IOR$_S$ identifying the server object and the replica's marshalled state as received in the replication response. The parameters are passed in a manner specific to the execution environment on the client side. For example, if the execution environment is a Tcl virtual machine, the parameters could be passed as command line arguments.

When the replica starts, it goes through an initialisation phase where it unmarshalls its state, creates an IOR$_R$ identifying itself and prepares to start serving requests. When the initialisation phase is complete, the replica notifies the D/IIOP$_C$ component (using IOR$_D$). The notification is a CORBA invocation that includes IOR$_R$ and IOR$_S$ such that the D/IIOP$_C$ component can create a

mapping between the IORs for the server and the replica. From this point, the D/IIOP$_C$ component and the replica communicate using CORBA.

At any point in time, the D/IIOP$_C$ component maintains a table mapping server to replica IORs. Such a table is shown in table 5.1. The column labelled 'Server IOR' contains IORs for (remote) server objects. The column labelled 'Replica IOR' contains IORs for the (local) replicas. The column labelled 'Reconciling' contains a boolean value used to indicate whether reconciliation is currently taking place. If no replica is available for a given server object, no entry for that server appears in the table.

Server IOR	Replica IOR	Reconciling
`IOR:01...`	`IOR:78...`	`false`
`IOR:22...`	`IOR:a5...`	`false`
`IOR:13...`	`IOR:65...`	`true`

Table 5.1: D/IIOP$_C$ Cache Management Table

5.5.4 Redirection

The D/IIOP$_C$ component sits between the client application and the IIOP Layer and intercepts all outgoing invocations. A similar scheme was used with the S/IIOP$_{MH}$ component described in section 5.4.3. This section outlines how redirection takes place. Appendix B.4.1 gives the full interface for the D/IIOP$_C$ component, which is responsible for redirection.

The D/IIOP$_C$ component examines each outgoing invocation to see if an entry for the IOR used for the invocation exists in the Cache Management Table. If the table indicates that a local replica exists and is not currently undergoing reconciliation, the invocation is modified to use the replica's IOR rather than the IOR of the server. The modified invocation is passed to the IIOP Layer which invokes the replica instead of the server. If the table indicates that a replica exists but is currently undergoing reconciliation, the invocation is blocked until reconciliation is complete. After reconciliation, the replica may become available or unavailable, so the client will have to reexamine the table. If the table indicates that no replica is available, the D/IIOP$_C$ component can either relay the unmodified invocation to the underlying IIOP Layer in the usual manner (potentially causing the client to block) or raise a `CORBA::TRANSIENT` exception (described in section 4.12.3.18 of the CORBA 3.0 specification [83]), indicating that the object could not be reached.

The request translation performed by the D/IIOP$_C$ component when redirecting a request to a local replica simply involves replacing the details of the remote server (hostname, port number and

object key) with those of the local replica. In this respect, the translation is different from swizzling (as described in section 5.4), because it does not involve extra information to be encoded in the object key.

5.5.5 Reconciliation

When the D/IIOP$_C$ component wishes to stop the replica and return the replica state to the server for reconciliation, it first sets the 'Reconciling' field in the Cache Management Table to true. This causes invocations to the replica to block. The D/IIOP$_C$ component then invokes a stop method on the replica. In response, the replica stops serving requests, marshalls its state into a CORBA octet sequence and returns that state to the D/IIOP$_C$ component as a result of the stop invocation. Having received the marshalled state, the D/IIOP$_C$ component passes it to the server object as part of a reconciliation invocation. When reconciliation is complete, the D/IIOP$_C$ component either clears the 'Reconciling' field in the Cache Management Table (in case the replica continues to be available) or removes the replica's entry from the table completely (in case the replica is now unavailable). Finally, the D/IIOP$_C$ component unblocks any invocations that were blocked as a result of the replica being under reconciliation.

Appendix B.4.2 gives the full API for the D/IIOP$_S$ component, which allows reconciliation to take place. Appendix B.4.3 gives the API that allows the replica to be stopped as described here.

While the D/IIOP Layer presented here allows code and state to be transferred between clients and servers, it has no special features to assist the application with conflict detection and resolution on the server side. These are left to the application.

5.5.6 Mobility Awareness

Section 4.7.4 observed that mobility events would have to be forwarded from the D/RIP$_S$ to the D/RIP$_C$ component when the server is on the MH. In the D/IIOP instantiation presented here, this is done via a standard CORBA invocation. Hence, the server on the MH makes a callback to the client on the RH. This effectively turns the client into a server also. Hence, the mobile server must maintain an IOR for the remote client, such that the server can perform the callbacks. In case the remote client is also on an MH, it is assumed to have its own Swizzling Layer which will allow the callback to reach it even if it moves.

In order to facilitate this scheme, the client has the option of explicitly registering its callback function with the server. This would typically be done at the time when the client requests replication of a server object. However, it is not required when the server is on the RH and the client on the MH.

For this reason, the callback registration method has been maintained as a separate method (and not part of the replication method). This is shown in appendix B.4.2.

5.6 The Java RMI Swizzling Layer

This section presents a detailed outline of how a Swizzling Layer for Java RMI could be implemented. The outline is based on Sun Microsystem's implementation of Java RMI [118]. The work presented in this section is joint work with Greg Biegel, also of the Distributed Systems Group, Trinity College Dublin.

5.6.1 Java RMI

This section introduces Java RMI, with a focus on the client/server interaction and the management of server references. For a more detailed treatment, the reader is referred to the Java RMI specification [118] and to books by Pitt [147] and Grosso [73]. Java RMI trivially satisfies requirements R1 and R2 from section 4.3.

RMI Servers

Java RMI Servers (often called 'remote objects') must implement an extension of the `java.rmi.Remote` interface. Figure 5.5 shows Java code for such an RMI server based on the `UnicastRemoteObject` class. The `HelloImpl` class contains the code executed on the server side. The fact that the `HelloImpl` class extends the `UnicastRemoteObject` class indicates that `HelloImpl` implements a single (non-replicated) remote object and that it communicates via the default RMI transport, which is based on Berkeley Sockets. All the remote object's methods must throw the `RemoteException` to indicate network and messaging failures.

Java RMI Servers are identified by server references. Server references are maintained within stubs, i.e., objects that implement the same interface as the real server object. Instead of the real server functionality, each of the methods in the stub contains code that marshalls the method parameters, invokes the real (remote) server object over the network, unmarshalls the return value and returns it to the caller. This is shown in figure 5.6, where a `HelloClient` invokes a `HelloImpl` server through a stub and a skeleton (server side stub). The stub supports the same set of interfaces as the server object, and thus appears exactly as if it were the real server object. The stub code is generated by a stub compiler on the server at compile time.

```
import java.rmi.*;
import java.rmi.server.*;

public interface Hello extends Remote {
  public String say() throws RemoteException;
}

public class HelloImpl
       extends UnicastRemoteObject
       implements Hello {
  public String say() throws RemoteException() {
    return "Hello!";
  }
}
```

Figure 5.5: Simple Java RMI Server

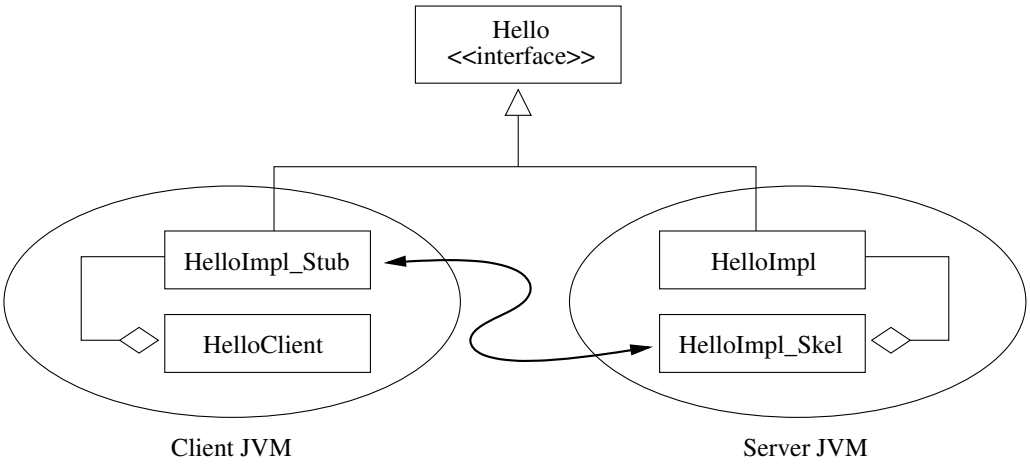


Figure 5.6: Stubs and Skeletons in Java RMI

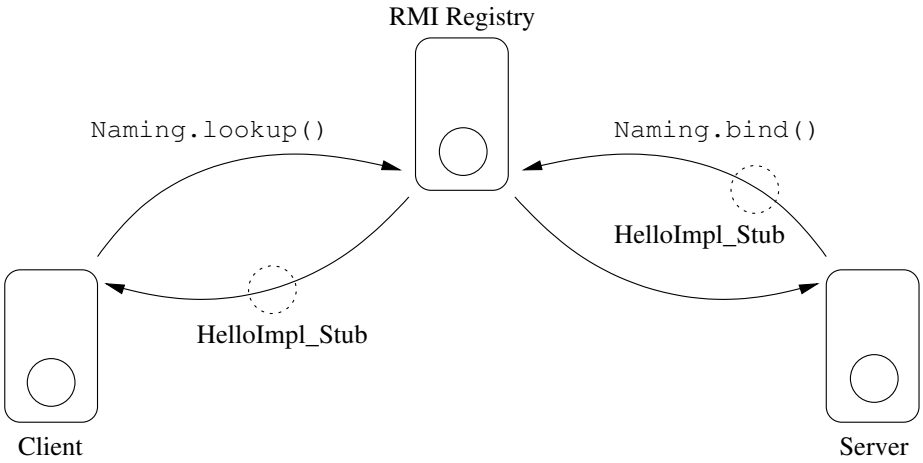


Figure 5.7: Stub Transfer in Java RMI

An entity known as the 'RMI Registry' runs on every host that holds RMI servers. This registry maintains name-to-stub mappings for all RMI servers on its host. The RMI Registry runs on a well-known port (1099). Effectively, the RMI Registry constitutes a non-replicated distributed name service where the distribution criterion is fixed. Servers are responsible for registering with their local RMI Registry, for example in the following manner:

```
Naming.bind("MyHello", server);
```

This causes a stub to be inserted into the registry, where it can later be retrieved by the client.

RMI Clients

An RMI client is a program that obtains a reference to and invokes an RMI server. There are no particular interfaces that clients must implement. Clients typically obtain stubs (and hence, references) from the RMI Registry but can also receive them as parameters to an incoming invocation or as return values from outgoing invocations. An operation on the `Naming` class called `lookup()` allows the client to obtain stubs from a given RMI Registry. Hence, the `bind()` operation allows the stub to be inserted into the registry by the server and the `lookup()` operation to allow the client to retrieve it from the registry. Figure 5.7 shows an RMI client and server interacting via an RMI Registry in this manner. Java RMI requires the registry and the server to be co-located (i.e., on the same host, though not necessarily in the same JVM), but the client could be running on a different host. The figure shows the `HelloImpl_Stub` object being transferred from the server to the registry during the invocation of `bind()` and from the registry to the client during the invocation of `lookup()`.

To find the right RMI Registry for a server object, the client must know the hostname of the machine where the server object resides. This allows it to find the RMI Registry, because the registry listens on a well-known port. Also, the client must know the symbolic name under which the server is registered. Both the hostname and the symbolic name are typically configured manually in the client. The following code fragment shows a typical lookup on the RMI Registry and subsequent server invocation:

```
Hello h = (Hello)Naming.lookup("rmi://localhost:1099/MyHello");
h.say();
```

The stub can either be statically or dynamically linked. In the former case, its code will be contained in the stub. In the latter case, the client will have to load the stub code either locally or remotely. The class loading path is configured on the client side and cannot be modified by a dynamically linked stub object or specified by the RMI Registry in response to a `lookup()` operation.

RMI Server Reference Internals

The format of server references is important for implementing swizzling for Java RMI. Figure 5.8 shows a class diagram for the relevant Java classes. (Attributes and methods not directly relevant for our purposes have been omitted for the sake of simplicity.) For the purposes of swizzling, the server hostname and port number are particularly relevant. As the figure shows, they are buried quite deeply within several layers of classes. The `RemoteStub` class extends the `RemoteObject` class, which contains an instance of the `UnicastRef` class, which implements the `RemoteRef` interface. The `UnicastRef` class contains an instance of `LiveRef`, which contains an instance of `TCPEndpoint`, which contains the server's hostname and port number. Despite the many classes, there are no fields that directly serve the purpose of the 'key' field discussed in section 4.6.2.

Once the stub is invoked on the client side, it delegates responsibility for invoking the method to the actual reference class, the `UnicastRef` class. This is shown in figure 5.9, which contains the stub code for the `Hello` server class. The `ref` object shown is the remote object reference inherited from `RemoteObject` (shown in figure 5.8) and is of type `UnicastRef`.

In the `UnicastRef` class, the `invoke()` method shown in figure 5.10 takes care of the invocation. The `invoke()` method contains code (not shown) to create a connection to the remote object, marshall the parameters and dispatch the call to the server. The S/RMI Layer described here works by extending the `UnicastRef` class.

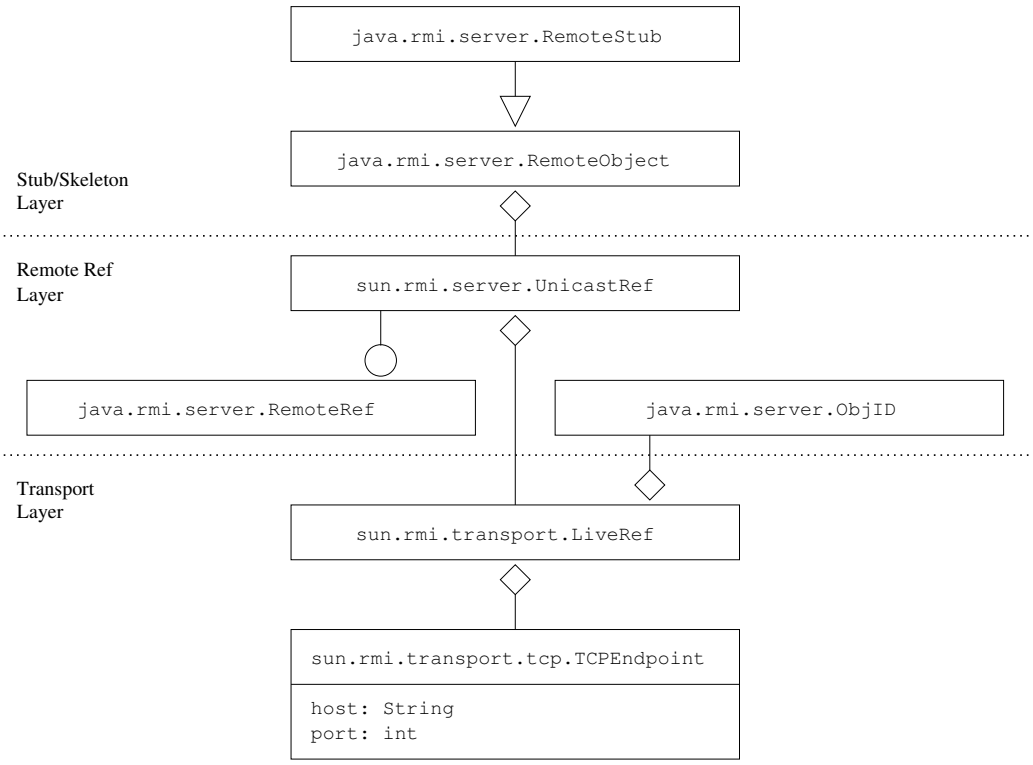


Figure 5.8: Class Diagram for Java RMI References (UML)

5.6.2 Server Reference Management

The process of swizzling as discussed in section 4.6 requires server references to be swizzled at creation time and reswizzled every time the MH arrives at a new MG. In the context of RMI, references should ideally be swizzled before they are entered into the RMI Registry. This section describes the S/RMI_{MH} component by discussing first how swizzling can be performed on Java RMI server references and then where the swizzling process can be inserted into the standard RMI server registration process.

Reference Swizzling

As discussed in section 5.6.1, the `UnicastRef` class (implementing the `RemoteRef` interface) contains not only the server endpoint but also the functionality to invoke the server. We propose to provide an extension to the `UnicastRef` class that includes the additional information required for swizzling. This class `SwizzledUnicastRef` is shown in figure 5.11. While the `LiveRef` attribute that in the `UnicastRef` class contains the mobile server's real endpoint is retained, new attributes have been added, containing details of the MG and the server's home agent. The `SwizzledUnicastRef` also

```
public final class HelloImpl_Stub
       extends java.rmi.server.RemoteStub
       implements Hello {
  public java.lang.String say() throws java.rmi.RemoteException() {
    Object $result = ref.invoke(this, $method_say_0, null,
                                -316483838299227514L);
    return ((java.lang.String) $result);
  }
}
```

Figure 5.9: Simple Java RMI Stub

```
public Object invoke(Remote obj,
                     java.lang.reflect.Method.method,
                     Object[] params,
                     long opnum) throws Exception {
  // Transmit invocation parameters and wait for result
}
```

Figure 5.10: The `invoke()` Method

contains a new implementation of `invoke()`. During invocation, this method first passes to the S/RMI$_{\mathrm{MG}}$ component the hostname for the MH and the port number on which the mobile server is listening. This allows the S/RMI$_{\mathrm{MG}}$ component to identify the MH for which the invocation is intended. This approach demonstrates that Java RMI satisfies requirement R5 from section 4.3.

Stubs can be statically or dynamically linked. In the former case, the code for the extended class is included with the stub object. In the latter case, a web server on the MG is required to serve stub class code to clients. Such a web server could be part of the S/RMI$_{\mathrm{MG}}$ component.

The approach of extending the `UnicastRef` class requires mobile servers to use the extended `SwizzledUnicastRef` class for creating their stub objects instead of the `UnicastRef` class. This is done in the server in the following manner:

```
HelloImpl hi = new HelloImpl();
SwizzledUnicastRef sur = new SwizzledUnicastRef();
HelloImpl_Stub stub = new HelloImpl_Stub(sur);
Naming.bind("MyHello", stub);
```

To keep stubs up to date, the `SwizzledUnicastRef` class needs to subscribe to callbacks from the ML. This will allow it to reswizzle the stub held on the MH (i.e., to update the `mg_name` and `ha_name` attributes) whenever the connectivity of the MH changes. (Copies of the stub held in the RMI registry or by remote clients will not be updated.) Since the ML is implemented in C, this is possible via Java Native Interface (JNI). An approach is given in Biegel et al. [23].

```
public class SwizzledUnicastRef extends sun.rmi.server.UnicastRef {

  private String mg_name, ha_name;

  public SwizzledUnicastRef(RemoteStub rs) {
    stub = rs;
  }

  public Object invoke(Remote obj,
                       java.lang.reflect.Method method,
                       Object[] params,
                       long opnum)
      throws Exception {

    // Open a socket to S/RMI on the MG
    // Transmit name of MH and port of mobile server (on MH)
    // Transmit invocation parameters and wait for result
  }

  // Other methods omitted for simplicity
}
```

Figure 5.11: `SwizzledUnicastRef` Class without Redirection

Server Registration

When performing the `bind()` invocation, the mobile server will attempt to contact its local RMI Registry, i.e., the one running on the MH. (Recall that Java RMI requires the server to bind to the registry on the same host.) However, because it resides on the MH, this registry is not generally available to remote clients. Even if it was, it would still be subject to the address migration problem discussed in section 2.5.1.

Instead, the server registration should be performed on the MG to which the MH is connected and on the RMI server's home agent. For this reason, the S/RMI$_{\text{MH}}$ component must catch the invocation of `bind()` and relay it to the MG and the home agent where the real `bind()` would be performed. The way in which the `bind()` operation is caught effectively constitutes the downcall API for the S/RMI$_{\text{MH}}$ component. The `bind()` operation can be caught by modifying the semantics of the `UnicastRemoteObject` class, which is responsible for creating the stubs and inserting them into the RMI Registry. We have identified two approaches to modifying the `UnicastRemoteObject` class:

1. Create a new class `AliceUnicastRemoteObject` which extends the `UnicastRemoteObject` class but which performs the operations described in section 5.6.2 instead of the ordinary stub operations. This approach requires that server code be modified to use the extended class rather than

the original `UnicastRemoteObject` class. Also, it requires installation of the extended class on the MH.

2. Modify the Java runtime on the MH such that the `UnicastRemoteObject` class is replaced with an ALICE version of the same. The functionality of the original class can be retained by renaming it `OrigUnicastRemoteObject`. This approach requires no modifications to server code but requires modification of the runtime classes.

While the changes to server code proposed in approach 1 are relatively small, they do break transparency for mobile servers using the S/RMI Layer. In comparison, approach 2 requires no modifications to source or byte code for mobile servers and is therefore a more attractive solution. By modifying the runtime environment, the modifications are hidden completely. However, the approach may cause problems for non-ALICE applications on the MH that may wish to use the original `UnicastRemoteObject` class. Such applications would have to be modified to use the original `UnicastRemoteObject` class, or the ALICE replacement would need a way of detecting whether the invoking application is looking for the original or the ALICE version of the class.

5.6.3 Client Redirection

This section describes the S/RMI$_{\mathrm{MG}}$ component by discussing how client redirection within Java RMI is possible. Java RMI does not include builtin features for redirection towards new server locations and at a first glance may therefore seem not to satisfy requirements R3 and R4 from section 4.3. However, it is possible to extend the `SwizzledUnicastRef` class given in figure 5.11 further to include functionality to handle redirection. A similar approach was used by Baratloo et al. [19] in a system for building replicated fault-tolerant servers. Java RMI is designed to include such extensions.

Figure 5.12 shows such an extended `SwizzledUnicastRef` class that has the ability to negotiate with the S/RMI$_{\mathrm{MG}}$ component that is being invoked. Effectively, the `SwizzledUnicastRef` object makes a preliminary invocation on the S/RMI$_{\mathrm{MG}}$ component to examine whether the MH in question is present, before it proceeds with the real invocation. This allows the S/RMI$_{\mathrm{MG}}$ component to redirect a client to another MG (in case the MH has moved and left a forwarding pointer) or to the home agent (in case the S/RMI$_{\mathrm{MG}}$ component has no record of the MH in question). If the MH is being hosted by the MG on which the S/RMI$_{\mathrm{MG}}$ component resides, the invocation proceeds as planned.

The S/RMI$_{\mathrm{MG}}$ component must be able to receive and act upon the preliminary invocations issued by the `SwizzledUnicastRef` class. Figure 5.13 outlines code for a S/RMI$_{\mathrm{MG}}$ daemon. The three possible outcomes of the preliminary invocation issued by the code in figure 5.12 corresponds

```
public class SwizzledUnicastRef extends sun.rmi.server.UnicastRef {

  private String mg_name, ha_name;

  public SwizzledUnicastRef(RemoteStub rs) {
    stub = rs;
  }

  public Object invoke(Remote obj,
                       java.lang.reflect.Method method,
                       Object[] params,
                       long opnum)
      throws Exception {

    // Open a socket to mg_name at S/RMI well-known MG port
    //   If fail, try HA instead for new forwarding pointer
    //   If okay, send mh_name and mh_port and wait for reply
    //     If redirect to new MG, update mg_name and try again
    //     If redirect to HA, try HA for new forwarding pointer
    //     If okay, transmit invocation parameters and wait for result
  }

  // Other methods omitted for simplicity
}
```

Figure 5.12: `SwizzledUnicastRef` Class with Redirection

```
public class SRMI_MG_Daemon extends Thread {

  // Create a new server socket
  // Listen for requests from SwizzledUnicastRef objects
  // Read in the mh_name and mh_port for the mobile server
  //   If MH is local, relay invocation
  //   If MH has a forwarding pointer, redirect client to new MG
  //   If MH is unknown, redirect client to its HA
}
```

Figure 5.13: The `SRMI_MG_Daemon` Class

to those received by the code in figure 5.13. In case a forwarding pointer is present for the MH, the S/RMI$_{\mathrm{MG}}$ component returns the forwarding pointer as the result of the preliminary invocation. In case the client is forwarded to the home agent, no parameters are returned, because the home agent's hostname already appears in the `SwizzledUnicastRef` object.

The redirection scheme described here demonstrates that Java RMI satisfies requirements R3 and R4 from section 4.3. To redirect clients, the S/RMI$_{\mathrm{MG}}$ component must subscribe to mobility events from the ML$_{\mathrm{MG}}$ component. This will allow it to obtain notification about MH arrivals, departures and handoffs. With the implementation of the ML$_{\mathrm{MG}}$ component discussed in section 5.3.2, this is relatively simple and merely involves creating a socket connection to the ML$_{\mathrm{MG}}$ component and reading the mobility events one at a time.

5.7 Java RMI Disconnected Operation Layer

This section outlines how a Disconnected Operation Layer for Java RMI could be implemented. The outline is based on Sun Microsystem's implementation of Java RMI [118]. First, section 5.7.1 discusses support for object mobility and disconnected operation in Java. Second, sections 5.7.2–5.7.5 discuss how replication, cache management, redirection and reconciliation can be performed in the context of Java RMI. The interfaces for the client, server and replica are also given in these sections. Finally, section 5.7.6 describes the callback feature used to maintain the client-driven nature of the D/RMI dialogue when the server is on the MH.

5.7.1 Disconnected Operation in Java

As opposed to CORBA, the Java runtime environment has inherent support for weak object mobility. Objects that implement the `java.io.Serializable` interface can be marshalled and unmarshalled

```
public class NotCacheableException extends java.lang.Exception {
  ...
}
public interface DRMI_Server {
  public boolean IsCacheable();
  public String Replicate() throws NotCacheableException;
  public void Reconcile(String marshalled_replica);
  public void RegisterCallback(RemoteUnicastServer client);
}
```

Figure 5.14: The D/RMI$_S$ API

to and from stream form and thereby moved between JVMs. Hence, Java RMI trivially satisfies requirement R6 from section 4.3.

The implementation of the D/RMI Layer proposed here follows the abstract specification given in section 4.7 closely. The D/RMI Layer is integrated with application server objects that have the ability to create replicas and to reconcile with modified replicas. Because the Java execution environment is homogenous, clients and servers do not need to concern themeselves with differences in architecture. This simplifies the design of the D/RMI Layer compared to the D/IIOP Layer described in section 5.5.

Like the D/IIOP Layer, the D/RMI Layer allows code and state to be transferred between clients and servers but has no special features to assist the application with conflict detection and resolution on the server side. However, guidelines exist that advise application developers on how to design Java applications that can operate effectively when disconnected from the network. For example, a June 2003 white paper published by Sun Microsystems [183] describes rules for selecting data suitable for migration to the client and also provides guidelines for performing conflict detection and resolution.

5.7.2 Replication

The D/RMI$_S$ component is integrated into the application as shown in the architectural overview in figure 4.4. Server objects that can be replicated must implement the `DRMI_Server` interface shown in figure 5.14. The `DRMI_Server` interface constitutes the D/RMI$_S$ component's tuning API. It allows clients to examine whether server objects can be replicated, to obtain replicas and to reconcile replicas with the original server objects. Figure 5.15 outlines a cacheable version of the `Hello` server from section 5.6.1 which implements the `DRMI_Server` interface.

Replicas are implemented as objects that run on the client side. They must implement the same application methods as an ordinary server object, i.e., must offer the same services to the client as the real server object. Instead of the `DRMI_Server` interface, replica objects must implement the `DRMI_Replica`

```
public class HelloImpl extends UnicastRemoteObject
                       implements Hello
                       implements DRMI_Server {

  public boolean IsCacheable() {
    ...
  }

  public String Replicate() throws NotCacheableException {
    // Create replica object.
    HelloReplica rep = new HelloReplica (...);

    // Serialize it to a file.
    FileOutputStream fos = new FileOutputStream("replica.tmp");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(rep);
    oos.close();

    // Read file contents into string (not shown).
    String marshalled_replica = ... ;

    // Return string to client.
    return(marshalled_replica);
  }

  public void Reconcile(String marshalled_replica) {
    // Write parameter contents to file "replica.tmp" (not shown).

    // Deserialize from file.
    FileInputStream fis = new FileInputStream("replica.tmp");
    ObjectInputStream ois = new ObjectInputStream(fis);
    HelloReplica rep = (HelloReplica) ois.readObject();
    ois.close();
  }

  public void RegisterCallback(RemoteUnicastServer client) {
    ...
  }

  // Other methods omitted for simplicity
}
```

Figure 5.15: Hello Server Implementing the D/RMI$_S$ API

```
public interface DRMI_Replica extends Serializable {
  public void Start();
  public void Finish();
}
```

Figure 5.16: The D/RMI$_R$ API

```
public class HelloReplica implements Hello
                          implements DRMI_Replica {
  public void Start() { ... }
  public void Finish() { ... }
  // Other methods omitted for simplicity
}
```

Figure 5.17: Hello Replica Implementing the D/RMI$_R$ API

interface shown in figure 5.16. Replica objects are created by a D/RMI$_S$ component and returned to a D/RMI$_C$ component as part of a replication request. Replicas can be serialized, i.e., marshalled into string form. To replicate a server object, the D/RMI$_C$ component invokes the `Replicate()` method on the server object. If the server can not be replicated, it throws a `NotCacheableException`. Otherwise, the serialized replica is passed in the form of a string back to the D/RMI$_C$ component. Figure 5.17 outlines a replica class for the cacheable version of the `Hello` server.

5.7.3 Cache Management

The D/RMI$_C$ component is implemented as a daemon running on a well-known port on the client side and an extension to server references that gives them the ability to communicate with the daemon. Although the daemon resides on the same host as the client, it may reside in a different address space. The daemon is responsible for managing a cache of replicas on the client side. The tuning interface, shown in figure 5.18, allows mobility-aware applications on the client side to control the cache contents. The D/RMI$_C$ daemon can also implement its own caching heuristics based, for example, on the past usage of server objects. The `Cache()` method shown in figure 5.18 allows an external (non-ALICE) component to request that a replica of a given server object be obtained. This can be used to prefetch replicas before an anticipated period of disconnection. The `IsCached()` method can be used to query the status of a given server object. It returns `true` if a local replica is present and `false` if not. The `Flush()` method is used to request that the D/RMI$_C$ daemon reconcile a replica of a given server object with the original server.

```
public class NotCacheableException extends java.lang.Exception {
  ...
}

public class NotCachedException extends java.lang.Exception {
  ...
}

public class DRMI_C_Daemon implements Runnable {

  // Tuning API used to control cache
  public void Cache(UnicastRemoteObject ref)
    throws NotCacheableException { ... }
  public boolean IsCached(UnicastRemoteObject ref) {
    ...
  }
  public void Flush(UnicastRemoteObject ref)
    throws NotCachedException {
    ...
  }

  // Tuning API invoked by the server, when server is on the MH
  public void Callback(UnicastRemoteObject ref,
                       bool connection_status) {
    ...
  }

  // Downcall API
  public Object Invoke(Remote obj,
                       java.lang.reflect.Method method,
                       Object[] params,
                       long opnum)
    throws Exception {
    // Perform a lookup in the cache management table
    //   If replica exists, perform invocation on replica
    //   If not, throw a NotCachedException
  }
}
```

Figure 5.18: The D/RMI$_C$ Tuning and Downcall API

Server Ref	Replica Ref
...	...
...	...
...	...

Table 5.2: D/RMI$_C$ Cache Management Table

To obtain and reconcile replicas, the D/RMI$_C$ daemon interacts with the D/RMI$_S$ component using the API shown in figure 5.14 and with locally cached replicas through the API shown in figure 5.16. A server is replicated by invoking its `Replicate()` method. If the server allows replication (i.e., does not throw an exception), a replica is returned. Upon receiving a replica, the D/RMI$_C$ daemon deserializes it and invokes the replica's `Start()` method, shown in figure 5.16, causing the replica to initialise itself. When the invocation has completed, the replica is ready to serve requests, and the D/RMI$_C$ daemon records the replica's reference in the cache management table shown in table 5.2.

If running on the MH, the D/RMI$_C$ daemon subscribes to callbacks from the underlying ML in order to monitor the state of connectivity of the MH. This allows it to perform cache management in a heuristic manner, prefetching and reconciling replicas without necessarily requiring an external cache management tool. If running on the RH, the D/RMI$_C$ daemon relies on callbacks from the D/RMI$_S$ component (on the MH) to retain awareness of the server's state of connectivity. This is described in section 5.7.6.

5.7.4 Redirection

The D/RMI$_C$ daemon must intercept invocations from the client in order to redirect them to locally stored replicas. We propose to do this by providing an alernative to the `UnicastRef` class in a similar manner to that described in section 5.6.2 for the S/RMI Layer. The new class, called `CachingUnicastRef`, is shown in figure 5.19. When receiving an invocation targeted for a remote server, the `CachingUnicastRef` class attempts to invoke the requested method through the local D/RMI$_C$ daemon instead of contacting the remote RMI server directly. This gives the daemon the option of redirecting the invocation to a replica. The method invoked on the daemon is the `Invoke()` method shown in figure 5.18. Because the daemon may be running in a different address space, the `Invoke()` method is invoked via RMI. If a replica of the requested server object exists, the daemon performs the requested invocation on the replica and returns the result to the caller. If no replica exists, the D/RMI$_C$ daemon throws a `NotCachedException`. This causes the `CachingUnicastRef` class to relay the invocation to the ordinary `UnicastRef` class from where it will be relayed in the usual

manner. This may cause the client to block in case no connectivity is available. Because replicas are application servers, they may raise exceptions in the usual manner. These are caught and rethrown by the $DRMI_C$ daemon, such that they can be caught by the caller. This preserves existing semantics for exceptions. Hence, if the $DRMI_C$ daemon's `Invoke()` method returns any other exception than `NotCachedException`, the exception can be assumed to have been thrown by the replica as part of normal application behaviour.

It is possible to combine the S/RMI and D/RMI Layers in cases where the server resides on the MH. (In case the server resides on the RH, no S/RMI Layer is present.) By combining the two layers, a client on the RH can replicate a mobile server, use the replica while the MH is disconnected and reconciliate with the server upon reconnection. A client invoking a server in this configuration should attempt to invoke a local replica first and if none is present attempt to invoke the mobile server through a swizzled reference, subject to redirection as discussed in section 5.6.3. For this reason, mobile servers that can be replicated should use an implementation of `CachingUnicastRef` that is derived not from the standard `UnicastRef` class as shown in figure 5.19 but from the `SwizzledUnicastRef` shown in figure 5.12. Apart from the change in inheritance, such a `CachingUnicastRef` class is identical to that shown in figure 5.19 and therefore not shown in a separate figure.

In order for the client to use the extensions provided in the `CachingUnicastRef` class, an instantiation of this class must be included in the server stub instead of an instantiation of the `UnicastRef` class. (Recall that a `UnicastRef` object is included in the stub as shown in figure 5.8.) Since stubs are created on the server side, this requires extra functionality in the D/RMI_S component. Hence, when the stub for a cacheable server object is created, an instantiation of the `CachingUnicastRef` class must be passed as a parameter to the stub constructor. A similar approach was used by the S/RMI Layer in relation to server reference management as described in section 5.6.2. This is a straightforward modification to the server because the D/RMI_S component is already mobility-aware. Hence, where we for the S/RMI Layer endeavoured to create the modified stubs in a manner that was transparent to the server, this is not required for the D/RMI Layer. For this reason, there is no need to modify the implementation of the `UnicastRemoteObject` class as proposed in section 5.6.2 in relation to the S/RMI Layer.

As noted in section 5.7.3, the D/RMI_C daemon subscribes to ML callbacks in case it is running on the MH. In relation to redirection, this allows it to redirect invocations to replicas based on the MH's current connectivity. In case the D/RMI_C daemon is running on the RH, callbacks from the D/RMI_S component allow the client to retain awareness of the server's state of connectivity. This is described in section 5.7.6.

```
public class CachingUnicastRef extends sun.rmi.server.UnicastRef {

  public CachingUnicastRef(RemoteStub rs) {
    stub = rs;
  }

  public Object invoke(Remote obj,
                       java.lang.reflect.Method method,
                       Object[] params,
                       long opnum)
      throws Exception {
    // Invoke the Invoke() method on the local DRMI_C_Daemon
    //   If it throws a NotCachedException, invoke the
    //     invoke() method on the real UnicastRef implementation
    //   If it throws another exception, return that exception
    //     to the caller
    //   If no exception was thrown, return the return value
    //     to the caller
  }

  // Other methods omitted for simplicity
}
```

Figure 5.19: The `CachingUnicastRef` Class

5.7.5 Reconciliation

The `Flush()` method (shown in figure 5.18) can be used to advise the D/RMI$_C$ daemon to stop caching a particular replica. The original server reference is passed as a parameter to the method. When the D/RMI$_C$ daemon receives such an invocation, it first removes the replica's entry from the Cache Management Table. This causes redirections to the replica to cease. The D/RMI$_C$ daemon then invokes the replica's `Stop()` method, causing the replica to wait for any ongoing client invocations to complete. When control returns to the D/RMI$_C$ daemon, the replica is ready to be reconciled. The D/RMI$_C$ daemon then invokes the D/RMI$_S$ component's `Reconcile()` method (shown in figure 5.14), passing the replica as a parameter. The server object unmarshalls the replica and reconciles its state with the server object's own state. This involves application-specific resolvers.

If running on the MH, the D/RMI$_S$ component subscribes to ML callbacks in order to monitor the MH's state of connectivity. Those of the callbacks related to new connections being obtained are relayed to the D/RMI$_C$ daemon as described in section 5.7.6. Callbacks related to loss of connectivity typically cannot be relayed because no connection exists.

5.7.6 Mobility Awareness

Like the abstract D/RIP Layer discussed in section 4.7, the actions of the D/RMI Layer are client-driven. As discussed in section 4.7.4, the server must relay mobility events to the client, in case the server resides on the MH. This is necessary to maintain the client-driven nature of the interaction. For the D/RMI Layer, the problem is solved in a similar manner as for the D/IIOP Layer. If running on the MH, the D/RMI_S component subscribes to callbacks from the underlying ML. These callbacks are relayed to the D/RMI_C daemon on the RH using standard RMI. The method invoked is called `Callback()` and is shown in figure 5.18. If the RH is also mobile, it is assumed to be running an instantiation of the S/RMI Layer such that the client can be found, even if it moves. In order to receive callbacks from a mobile server, the client must register its callback function with the server using the `RegisterCallback()` method shown in figure 5.14.

5.8 Summary

This section presented instantiations of a number of ALICE components. First, two transport modules were presented, suitable for operation with very different types of communications interfaces. Implementations of both of these are nearly complete, lacking only integrity checks of PDUs and facilitites for collecting statistics related to data error rates. Second, an instantiation of the Mobility Layer was presented. This instantiation was also nearly complete, lacking only features for shortening of inter-MG tunnels and support for data compression. Although data compression would be relatively easy to add, it is questionable (as discussed in sections 3.2.2, 3.4.3 and 3.5.2) whether it will result in actual performance improvements. Third, we presented a fully functional implementation of a Swizzling Layer and a detailed outline of a Disconnected Operation Layer for CORBA. Swizzling and Disconnected Operation Layers were also outlined for Java RMI.

The two object-oriented middleware architectures treated here are different in a number of respects, in particular in relation to their format for server references and the ways such references are managed and propagated. Another considerable difference was related to the architectures' support for object mobility. Despite these differences, both CORBA and Java RMI were shown to satisfy requirements R1–R6 from section 4.3, and it was possible to fully instantiate ALICE for implementations of both architectures. Hence, we have not only shown that that the six requirements are reasonable but also that ALICE is sufficiently general to be applicable to two major object-oriented middleware architectures.

Chapter 6

Evaluation

> *Alice watched the White Rabbit as he fumbled over the list, feeling very curious to see what the next witness would be like, '—for they haven't got much evidence YET,' she said to herself.* [35]

This thesis presented the ALICE architecture which allows mobility support to be added to any object-oriented middleware framework whose architecture supports a set of minimal requirements. Chapter 4 presented ALICE at an abstract level and chapter 5 presented a set of components that could be used to instantiate the architecture for CORBA and Java RMI. As discussed in chapter 1, the objective of the thesis is to show how mobility support can be added to existing object-oriented middleware architectures in a manner that is both *transparent* and *general*. In relation to transparency, we distinguish between *application-level* and *framework-level* transparency. Application-level transparency is the extent to which applications using a mobility-enabled object-oriented middleware framework can remain unaware of the fact that the framework is different from a framework that has not been mobility-enabled. Framework-level transparency is the extent to which object-oriented middleware frameworks that are not mobility-enabled can remain unaware that they are interacting with one that has been mobility-enabled. Hence, both types of transparency are properties of *instantiations* of the ALICE architecture rather than the architecture itself. In relation to generality, we are concerned with the general applicability of ALICE across object-oriented middleware architectures. Hence, generality is a property of the ALICE architecture rather than individual instantiations.

This chapter evaluates ALICE in relation to the two types of transparency and in relation to the architecture's generality across object-oriented middleware architectures. First, section 6.1 presents an experimental application and configuration used to evaluate the CORBA instantiation of ALICE in

terms of transparency. Second, section 6.2 evaluates the CORBA instantiation of ALICE in relation to application-level transparency by comparing the functionality and performance of the mobility-enabled CORBA framework and a commercially available framework that has not been mobility-enabled. Section 6.3 evaluates the CORBA instantiation of ALICE in relation to framework-level transparency by testing interoperability between the mobility-enabled CORBA framework with a commercially available framework that has not been mobility-enabled. Section 6.4 evaluates the generality of the ALICE architecture presented in chapter 4 by examining a number of popular object-oriented middleware architectures and discussing the ease with which they satisfy the six requirements that define the degree to which mobility support can be added through the use of ALICE. Finally, section 6.5 summarises the chapter.

6.1 Experimental Configuration

A client/server application called the 'grid application' is used for the purposes of evaluating the CORBA instantiation of ALICE in terms of application-level and framework-level transparency. This section describes first the grid application and explains why it was chosen, and then gives the configurations in which it was used. Six configurations are presented. The first two do not feature ALICE support and are used as a baseline against which the overhead of the CORBA instantiation of ALICE can be measured. The next two configurations feature ALICE support with one MG involved. The last two feature ALICE support with two MGs and mobility of the MH from one MG to the other. In configurations 1, 3 and 5, the client resides on the MH. In configurations 2, 4 and 6, the server resides on the MH. The configuration of the ALICE stack differs depending on whether the client or server resides on the MH, because a mobile client does not require the presence of the S/IIOP Layer.

6.1.1 Experimental Platform

Three computers were used in the experiments: one MH and two MGs, one of which also served as RH. Table 6.1 shows the hardware and OS configuration for each machine. The two MGs were connected via a Fast Ethernet (100 Mbps) LAN. The MH was connected to the MG using a Lucent 802.11 card and a Compaq WL400 access point. On all platforms, the ALICE and grid application components were compiled with the GCC 2.95.4 compiler.

For all experiments, ALICE was configured with UDP and serial transport modules, but only the UDP module was used in the experiments. Two CORBA frameworks were used: the IIOP Layer discussed in chapter 5 and version 4.1.2 of the commercially available ORBacus. ORBacus was chosen

Host	Brand and Model	CPU	Clock Speed	Memory	OS
MH	Compaq iPAQ 3650	StrongArm	206 MHz	32 MB	Familiar Linux
MG1/RH	Dell Dimension XPS T800r	Pentium III	800 MHz	256 MB	Debian Linux 3.0
MG2	Siemens Scenic 300	Pentium III	500 MHz	384 MB	Debian Linux 3.0

Table 6.1: Hardware and OS Configuration for Experimental Platform

from the many commercial and open source ORBs available because it is a popular ORB in fixed (non-mobile) network environments and therefore suitable for testing interoperability with the CORBA instantiation of ALICE. ORBacus was also available for free under an educational license.

6.1.2 The Grid Application

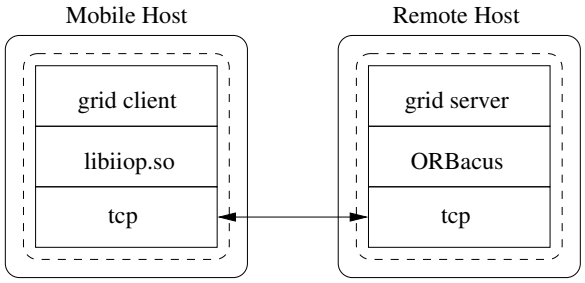
The grid application features a centralised server that maintains a matrix of integers. Clients can read and write individual cells in the matrix by invoking methods on the server and can also examine the dimensions of the matrix by reading server attributes. Figure 6.1 shows the interface to the server in Interface Definition Language (IDL).

```
interface grid {
  readonly attribute short height;  // height of the grid
  readonly attribute short width;   // width of the grid

  // set the element [n,m] of the grid, to value:
  void set(in short n, in short m, in long value);

  // return element [n,m] of the grid:
  long get(in short n, in short m);
};
```

Figure 6.1: Interface to the Grid Server (IDL)

The grid application was chosen with a number of considerations in mind. The instantiated ALICE components collaborate to allow distributed object applications to perform invocations despite the difficult conditions of the mobile environment. For this reason, it was important to choose an application that was intensive in its requirements to communications (i.e., the number of invocations performed) rather than its requirements to storage or processing. The grid application lends itself well to configuration in this manner. As a highly invocation-intensive distributed object application, it belongs to the class of applications for which we expect the CORBA instantiation of ALICE to exhibit its worst performance.

The grid server used in the experiments is configured with a small grid (size 10 by 10) and the



LEGEND: Solid rounded boxes denote node boundaries; dashed rounded boxes denote process boundaries; arrows denote communication; box sizes reflect layer correspondence across protocol stacks.

Figure 6.2: Configuration 1: Mobile Client without ALICE

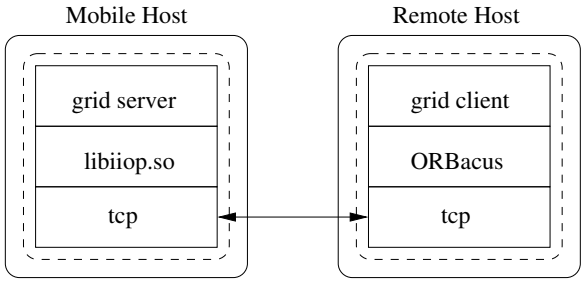
client operates on individual cells one at a time. This results in low memory requirements for both parties. Although grids are often used to perform processing-intensive tasks (e.g., large-scale matrix arithmetics), the client and server used here were configured to perform only minimal arithmetic operations in order to minimise processing requirements for both parties.

The grid server maintains the 10×10 grid of integers, while the grid client cycles through a loop. On each iteration, the client first invokes the server to obtain the height and width of the grid, then increments a given cell (cell $(4, 5)$ was used, but any cell would have worked) by invoking `get` followed by a `set`. Finally, the client invokes `get` on the modified cell to obtain the new value. The loop is executed 5,000 times in order to obtain meaningful figures. In order to minimise processing requirements on the client, it was chosen to increment the same cell repeatedly rather than, for example, a random cell every time. Also, using the same cell was useful as a debugging aid, because the number of invocations performed could easily be examined. No caching of requests, responses or cell values was used.

Because of its simplicity, the grid application is easy to implement using a given implementation of CORBA. The client and server were implemented for each of the two CORBA frameworks used. The functionality of the two implementations is identical.

6.1.3 Configuration 1: Client on the MH without ALICE

Figure 6.2 shows the grid application configured without ALICE support. The grid client resides on the MH and the server on the RH. There is no MG because no ALICE support is present.



LEGEND: Solid rounded boxes denote node boundaries; dashed rounded boxes denote process boundaries; arrows denote communication; box sizes reflect layer correspondence across protocol stacks.

Figure 6.3: Configuration 2: Mobile Server without ALICE

6.1.4 Configuration 2: Server on the MH without ALICE

Figure 6.3 shows the grid application configured without ALICE support. The grid server resides on the MH and the client on the RH. As in configuration 1, there is no MG because no ALICE support is present.

6.1.5 Configuration 3: Client on the MH with ALICE and One MG

Figure 6.4 shows the grid application configured with the client on the MH and the server on the RH. On the MH, the grid client and the ML daemon run as separate processes. The grid client is linked directly with the IIOP Library. The S/IIOP Library is not required, because clients do not require address translation. Because no S/IIOP Library appears on the MH, ML callbacks are not required, and the client is therefore linked with a mobility-unaware version of the ML Library (`libmlmh-mu.so`). This version of the ML Library has a smaller footprint than the mobility-aware version (`libmlmh-ma.so`) but cannot issue callbacks. On the MG, the ML daemon mediates between the MH and RHs in the usual manner. No S/IIOP redirection daemon appears on the MG, because no mobile server exists. On the RH, the grid server is built with the ORBacus framework. It accepts incoming invocations via the ORBacus implementation of IIOP over TCP.

6.1.6 Configuration 4: Server on the MH with ALICE and One MG

Figure 6.5 shows the grid application configured with the server on the MH and the client on the RH. On the MH, the grid server and the ML daemon run in separate processes. Usually, a server using the IIOP Layer would be linked with the IIOP Library (`libiiop.so`) but because the mobile server requires address translation, it is linked with the S/IIOP Library (`libsiiiop.so`) instead. The

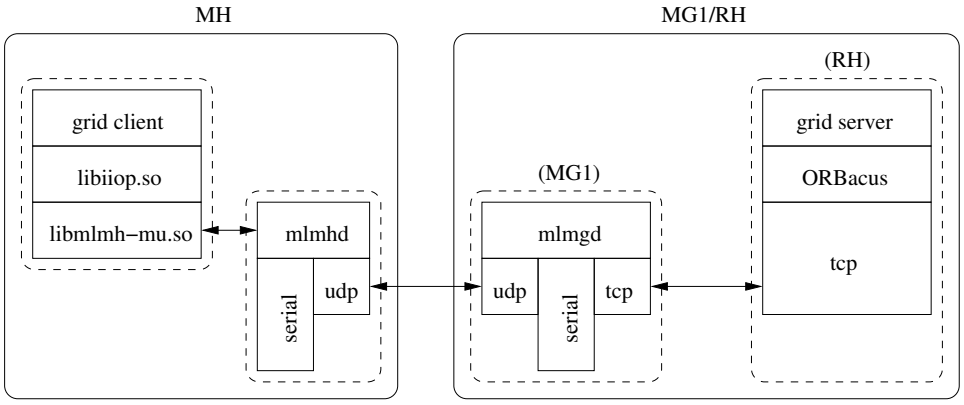


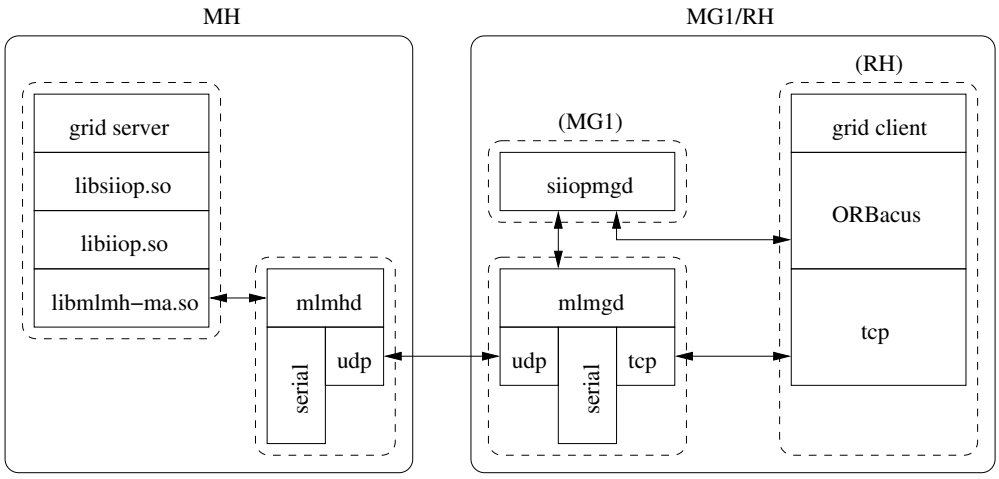
Figure 6.4: Configuration 3: Mobile Client with ALICE and One MG

S/IIOP Library is a mobility-aware application and is therefore linked with a version of the ML Library that can issue callbacks (`libmlmh-ma.so`). This library communicates with the ML daemon. On the MG, the S/IIOP redirection daemon (`siiopmgd`) and the ML daemon run as separate processes. The latter communicates with the ML daemon on the MH via the UDP transport module and with RHs via TCP. It issues upcalls to the S/IIOP redirection daemon. On the RH, the grid client is built with the ORBacus framework. It uses the ORBacus implementation of IIOP to communicate with the grid server over TCP. This happens through the S/IIOP redirection daemon in a manner that is transparent to ORBacus and the client.

6.1.7 Configuration 5: Client on the MH with ALICE and Two MGs

Figure 6.6 shows the grid application configured with the client on the MH and the server on the RH. The configuration is identical to configuration 3 with the addition that the MH moves from MG1 to MG2 after the client has opened the connection to the server. This results in a tunnel being set up between the two MGs. No S/IIOP redirection daemons are shown on the MGs, because they are not required for client mobility.

Mobility of the MH from MG1 to MG2 is triggered by using `ipchains(8)` on MG1 to prevent communication with the MH. In this way, a broken connection between the MH and MG1 is simulated. When it discovers that MG1 has become unreachable, the MH connects to MG2 instead.



LEGEND: Solid rounded boxes denote node boundaries; dashed rounded boxes denote process boundaries; arrows denote communication; box sizes reflect layer correspondence across protocol stacks.

Figure 6.5: Configuration 4: Mobile Server with ALICE and One MG

6.1.8 Configuration 6: Server on the MH with ALICE and Two MGs

Figure 6.7 shows the grid application configured with the server on the MH and the client on the RH. The configuration is identical to configuration 4 with the addition that the MH has moved from MG1 to MG2 after the server reference was obtained by the client. When the client attempts to invoke the server (arrow 1), it is redirected to MG2 (arrow 2).

As for configuration 5, `ipchains(8)` are used on MG1 to simulate a broken connection between the MH and MG1 by preventing MG1 from communicating with the MH. This causes the MH to connect to MG2 instead of MG1.

6.2 Application-Level Transparency

Application-level transparency is the extent to which the application can remain unaware that the object-oriented middleware framework being used is an ALICE instantiation, i.e., a mobility-augmented framework. This section examines two kinds of application-level transparency: functional and performance transparency. The former has to do with the extent to which application developers can remain unaware of the mobility support provided by the ALICE instantiation. The latter has to do with the runtime operation of the application, i.e., how the mobility support framework affects the application's resource usage and performance. Three types of resources are affected: memory, processing and network resources. This section looks first at the functional issues and then in turn at the three types of

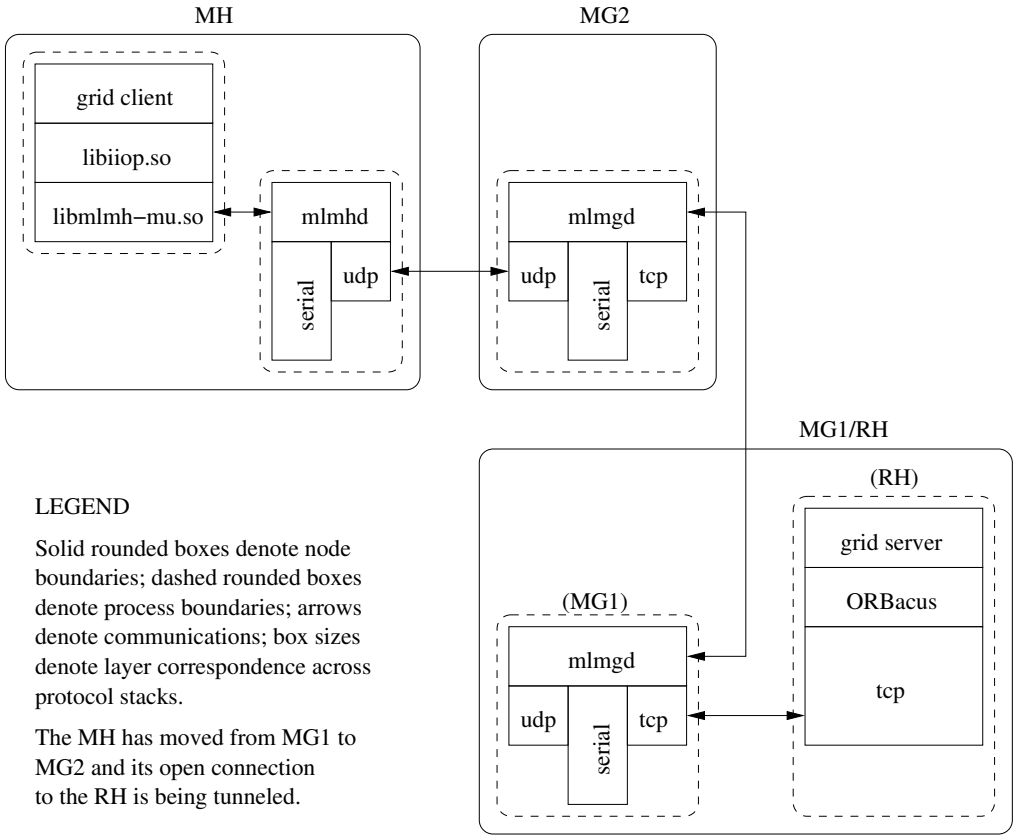


Figure 6.6: Configuration 5: Mobile Client with ALICE and Two MGs

performance transparency. Because *transparency* is a property of a particular instantiation of ALICE, rather than of the architecture itself, this section assesses the level of application-level transparency offered by the CORBA instantiation presented in chapter 5. The instantiation is configured with two Transport Modules, the Mobility Layer and the S/IIOP Layer. The D/IIOP Layer was presented as an outline only and is therefore not evaluated in relation to application-level transparency. As a preliminary note to the performance analysis, it is worth clarifying what is meant by overhead in terms of *memory, processing* and *network* resources:

Memory overhead is the amount of Random Access Memory (RAM) consumed by the ALICE components compared to that consumed by an object-oriented middleware framework that has not been mobility-enabled. Memory overhead is measured in bytes or kilobytes. Retaining performance transparency through minimising memory consumption of ALICE components is important in order to address the limited storage capacity (section 2.3.4) of mobile devices.

Processing overhead is the CPU time used by the ALICE components to perform their functions

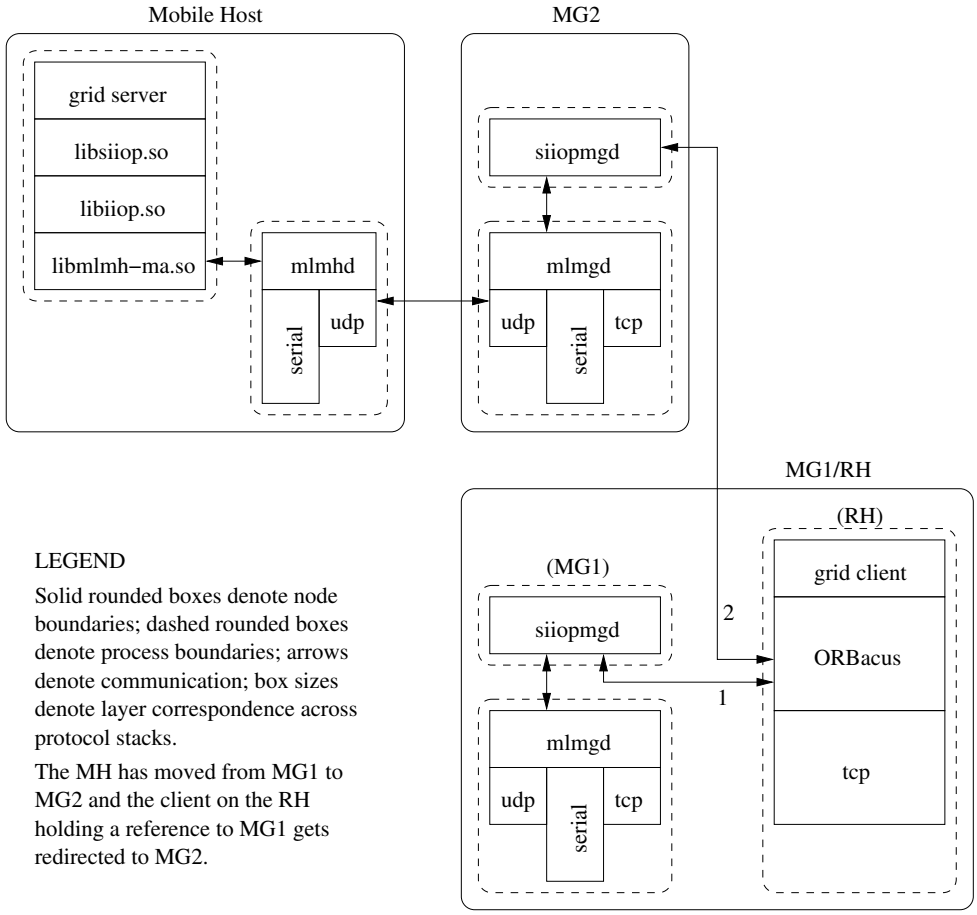


Figure 6.7: Configuration 6: Mobile Server with ALICE and Two MGs

during the operation of an application. We measure processing overhead as a time delay in seconds or milliseconds. Retaining performance transparency by minimising the processing power used by ALICE components is important in order to address the limited processing power (section 2.3.5) of mobile devices.

Network overhead is the amount of traffic caused by the ALICE components during the operation of an application compared to the traffic caused by an object-oriented middleware framework that has not been mobility-enabled. This 'ALICE traffic' is due to PDU headers, acknowledgements, etc. Network overhead is measured in bytes or kilobytes. Minimising the additional traffic caused by ALICE components is important in order to address the low bandwidth (section 2.4.3) challenge.

6.2.1 Functional Transparency

This section discusses the extent to which the mobility support provided by ALICE remains transparent to the application developer. This is a purely analytical evaluation. The instantiation of the Mobility and S/IIOP Layers presented in chapter 5 is examined and compared to the Berkeley Sockets and IIOP Layers, which are the layers whose functionality the ML and S/IIOP Layers are designed to replace.

Mobility Layer

The ML is intended to act as a replacement for a standard implementation of Berkeley Sockets that provides transparent mobility support to mobility-unaware applications. In this section we examine the transparency of this approach by discussing compatibility with the Berkeley Sockets API and the source code modifications required to replace Berkeley sockets with ML sockets for two sample applications. This discussion can be seen as an extension of the discussion of ML socket semantics presented in section 4.5.5.

In general, the ML implements the most basic socket functions to work in the usual manner. ML sockets are opened and closed as with a standard implementation of Berkeley Sockets. Making outgoing connections and performing read and write operations have the usual semantics, except that the operations will block indefinitely when the MH is not connected to an MG. (With a standard Sockets implementations, such connections will time out.) This is a feature of the ML. However, there are a number of differences from standard Socket semantics:

1. The ML only supports TCP sockets. UDP sockets are not supported. Since the ML is designed for the purpose of supporting object-oriented middleware, we do not consider this compatibility problem a big issue. Remote invocation protocols are typically easier to implement over TCP than UDP.

2. The `bind(2)` operation has no effect on the MH, i.e., does not actually bind the socket to a given port and network interface. In this respect, the ML is not compatible with the standard Berkeley Sockets API. For client/server applications using the ML (not necessarily object-oriented middleware), this means the server will not be runing at the port that it expects, and that clients will not be able to find them. This problem cannot be solved within the ML. The ALICE architecture allows Swizzling Layers to be implemented to solve this problem.

3. The `getsockname(2)` operation is generally used to obtain the binding of a given socket. When invoked for an ML socket, it will not return the actual values for client or server sockets, because

the call is not relayed to the MG. While, this could be changed in future versions of the ML, it is unlikely to improve compatibility with existing applications that expect the binding returned to refer to a local network interface and a local port.

4. The `getsockopt(2)` and `setsockopt(2)` calls are used to read and write configuration parameters for individual sockets, such as buffer sizes and timeouts. In the current implementation of the ML, these calls are not relayed to the MG and the calls therefore have no effect. This will break compatibility with applications that have special requirements on their socket configurations.

In addition to the preservation of socket semantics, the second aspect of functional application-level transparency is the ease with which a mobility-unaware application can be adapted to use the ML instead of the standard Berkeley Sockets library. This is measured as the number of changes required to source code, makefiles, etc. To assess the level of transparency offered by both approaches, two small case studies were performed.

The first case study was performed for a simple application called the 'talk program' written by C. David Shaffer. The talk program consists of a client and a server written in C that communicate via TCP using the Sockets API. The server implements a chat room. Clients connect to the server and exchange messages by sending them to the server which relays each message to all other clients. The talk program constitutes a minimal client/server application using TCP via the Berkeley Sockets API. It is representative of simple client/server programs. It does not use UDP or any of the functions mentioned in 3–4 above, but it does use the `bind(2)` operation. In order to modify the talk application to use the ML instead of the Berkeley Sockets implementation, one change was required to the makefile. The change involved adding three options (with parameters giving library names and locations) to the compilation flags in order to link the talk application against the ML instead of the Sockets library. No modifications were required to the client or the server source code. The result was a mobility-unaware (but mobility-enabled) application whose connections between client and server took place through the MG. Broken connections were reconnected and host mobility caused connections to be tunneled by the ML in a completely transparent manner. The mobility-enabled talk program was tested and found to perform identically to the umodified talk program.

The second case study was performed for an implementation of the grid client built with the ORBacus framework. The grid client uses the Berkeley Sockets through the ORBacus libraries. The ORBacus libraries were not examined to assess which of the functions discussed in 1–4 above were used during the client's operation, but it seems reasonable to assume that only TCP was used and that `bind(2)` was not used, because the program is a client. It is not known whether the ORBacus

libraries used the functions mentioned in 3–4. A detailed analysis of the ORBacus source code (which is available under an educational license) would reveal whether this is the case, but is left as future work. As for the talk program, one change was required to the grid application's makefile. Three options (with parameters) were added to the compilation flags in order to link the client against the ML instead of the Sockets library. No modifications were required to the client's source code. The result was a mobility-unaware (but mobility-enabled) client that could invoke a remote server from an MH through an MG in a manner that was completely transparent to the client as well as the server. The mobility-enabled ORBacus grid client was tested and found to perform identically to a standard grid client built with the ORBacus framework.

Swizzling Layer

On the MH, the Swizzling Layer provides transparent address translation for mobile servers. On the MG, the Swizzling Layer redirects clients towards new server locations. In this section, we evaluate the transparency of the S/IIOP$_{\mathrm{MH}}$ component. Because the redirection performed by the S/IIOP$_{\mathrm{MG}}$ component occurs at the remote invocation protocol level, the transparency in relation to this component's dialogue with the client is considered framework-level transparency. For this reason, the discussion of the level of transparency offered by the S/IIOP$_{\mathrm{MG}}$ component is deferred to section 6.3 which deals with framework-level transparency.

On the MH, the S/IIOP$_{\mathrm{MH}}$ component is designed to be inserted above the IIOP Layer in the protocol stack in a manner that is transparent to applications. Hence, the intention is that a (mobility-unaware) server written to work with the IIOP Layer should work with the S/IIOP Layer with as few modifications as possible. In order to insert the S/IIOP$_{\mathrm{MH}}$ component on top of the IIOP Layer, two modifications were required to the makefile for the application. First, two compilation options were added to compile the server with the headers for the S/IIOP Library before those of the IIOP Library. This allowed operations declared in the S/IIOP Library to override those declared in the IIOP Library. Second, five linking options were added to link the server against the S/IIOP and ML libraries. No modifications to the source code of the server or the IIOP Library were required.

During operation, the grid server publishes its IOR by stringifying it and writing it to a shared file system. The client subsequently reads the IOR from the file system and uses it to invoke the server. It is common behaviour for CORBA servers to publish stringified references out of band using for example a shared file system. The S/IIOP$_{\mathrm{MH}}$ component overrides the marshalling operation that converts the IOR to CDR format, as described in appendix B.3.1. The marshalling operation is used to export IORs for example as return values or out parameters of incoming invocations, or as in

parameters of outgoing invocations. The marshalling operation is also used to stringify IORs. This assures that any IOR published by the server either through stringification or as part of an invocation is the *swizzled* IOR. No modification to the server was required to implement this behaviour.

Summary

This section discussed functional application-level transparency. For the ML, small case studies of two applications were used to examine the effort related to replacing the Berkeley Sockets implementation with the ML. The impact of this modification on the applications was also examined. For both applications, the only change required was the addition of three compilation options to each application's makefile. This can be considered a minimal modification. The resulting mobility-enabled, mobility-unaware applications retained their original behaviour. For the S/IIOP Layer, a total of five options were required to two sections of the grid server's makefile. This can also be considered a minimal modification. On this basis, the conclude that the CORBA instantiation of ALICE presented in chapter 5 offers a high level of functional application-level transparency.

6.2.2 Performance Transparency: Memory

This section examines the memory overhead involved in adding mobility support to an application using the CORBA instantiation of ALICE presented in chapter 5 by analysing memory footprints of instantiated ALICE components. All memory footprints listed pertain to binaries and libraries that have been compiled with minimum debugging information and which have been stripped using the Unix `strip(2)` command. For reasons of simplicity, only memory related to code segments is considered. A more thorough analysis involving also memory allocated at runtime for application state is left as future work. All footprint figures are given in bytes. Footprints between components compiled for the two processor architectures differ slightly. We give figures for iPAQ (`armv5tel`) in relation to components used on the MH and for Pentium (`i686`) in relation to components used on MGs and RHs. Table 6.2 presents memory footprints of the components presented as part of the CORBA instantiation of ALICE presented in chapter 5 and also lists the memory footprint for the IIOP Library. To add mobility support to a client or a server, the relevant ALICE components must be used, as shown in configuration 3 and 4, respectively. Table 6.3 shows the memory footprints for the grid client and server built with the IIOP Library.

File	Location	Purpose	Component	Arch	Footprint
libiiop.so	MH	Generic IIOP Support	IIOP Layer	iPAQ	72,860 bytes
libsiiop.so	MH	Reference Swizzling	$S/IIOP_{MH}$	iPAQ	22,184 bytes
libmlmh-ma.so	MH	ML Mobility-aware interface	ML_{MH}	iPAQ	21,644 bytes
libmlmh-mu.so	MH	ML Mobility-unaware interface	ML_{MH}	iPAQ	12,480 bytes
mlmhd	MH	ML daemon	ML_{MH}	iPAQ	43,788 bytes
mlmgd	MG	ML daemon	ML_{MG}	Pentium	47,644 bytes
siiopmgd	MG	Redirection daemon	$S/IIOP_{MG}$	Pentium	29,068 bytes

Table 6.2: Memory Footprints for CORBA Instantiation of ALICE (Linux)

Component	Footprint
Client	11,248 bytes
Server	15,964 bytes

Table 6.3: Memory Footprints for Grid Application (Linux on iPAQ)

Mobile Client

In order to add mobility support to a mobile client, the CORBA instantiation of ALICE must be configured with the IIOP Layer, the mobility-unaware version of the ML_{MH} library and the ML_{MH} daemon. This configuration corresponds to the grid application as shown in configuration 3. In comparison, a client without mobility support will be using only the IIOP Layer. This configuration corresponds to the grid application as shown in configuration 1. We measure the size of the CORBA instantiation of ALICE compared to the size of the IIOP Layer, which is our standard implementation of IIOP. Using the figures from table 6.2, the overhead is calculated as follows:

$$\frac{|\texttt{libmlmh} - \texttt{mu.so}| + |\texttt{mlmhd}|}{|\texttt{libiiop.so}|} = \frac{12,480 + 43,788}{72,860} = 77\%$$

Using figures from tables 6.2 and 6.3, the total footprint of the mobility-enabled grid client can be calculated as follows:

$$|\texttt{client}| + |\texttt{libiiop.so}| + |\texttt{libmlmh} - \texttt{mu.so}| + |\texttt{mlmhd}| =$$
$$11,248 + 72,860 + 12,480 + 43,788 \text{ bytes} =$$
$$140,376 \text{ bytes}$$

Mobile Server

In order to add mobility support to a mobile server, the CORBA instantiation of ALICE must be configured with the IIOP Layer, the $S/IIOP_{MH}$ component, the mobility-aware version of the ML_{MH} library and the ML_{MH} daemon. This configuration corresponds to the grid application as shown in configuration 4. In comparison, a server without mobility support will be using only the IIOP Layer.

This configuration corresponds to the grid application as shown in configuration 2. As for the client, the size of the CORBA instantiation is compared to the size of the IIOP Layer. Using the figures from table 6.2, the overhead is calculated as follows:

$$\frac{|\texttt{libsiiop.so}| + |\texttt{libmlmh} - \texttt{ma.so}| + |\texttt{mlmhd}|}{|\texttt{libiiop.so}|} = \frac{22,184 + 21,644 + 43,788}{72,860} = 120\%$$

Using figures from tables 6.2 and 6.3, the total footprint of the mobility-enabled grid server can be calculated as follows:

$$|\texttt{server}| + |\texttt{libiiop.so}| + \texttt{libsiiop.so}| + |\texttt{libmlmh} - \texttt{ma.so}| + |\texttt{mlmhd}| =$$

$$15,644 + 72,860 + 22,184 + 21,644 + 43,788 \text{ bytes} =$$

$$176,120 \text{ bytes}$$

Perspective

Compared to the IIOP Layer, the CORBA instantiation of ALICE causes an increase in memory footprint of 77% and 120% on the MH for clients and servers respectively. These figures are independent of individual applications and pertain to any client and server built with the CORBA instantiation of ALICE. Whether this is an acceptable overhead, depends on the mobile devices in question. Comparing the total footprints given above to the mobile computers discussed in section 2.3 shows that the grid application built with the CORBA instantiation of ALICE could operate on most current mobile computers. Of the six computers listed in table 2.2, only the two sensor nodes do not have sufficient storage to run the mobility-enabled grid application. The extremely limited storage capacity of these two devices (8 and 32 KB respectively) mean that even if built with the standard IIOP Library (i.e., no mobility support), the grid application is too big. The two PDAs listed in table 2.2 have 16 and 64 MB of memory respectively, which is sufficient for both the mobility-enabled client and server. Although both devices have sufficient memory, it should be noted that only one (the iPAQ) supports our chosen platform (Linux) and will actually be able to run distributed applications built with the CORBA instantiation of ALICE.

6.2.3 Performance Transparency: Processing

This section examines the processing overhead on the MH caused by ALICE mobility support in an application. The grid application presented in section 6.1.2 is used for this purpose. First, the performance for client mobility is examined through experiments with configurations 1, 3 and 5 (mobile client without ALICE and with ALICE with one or two MGs respectively). Second, the performance

of server mobility is examined through experiments with configurations 2, 4 and 6 (mobile server without ALICE and with ALICE with one or two MGs respectively). This allows us to assess the processing overhead caused by the ALICE components both in the case of mobile clients and servers. The Unix `time(1)` command is used to collect CPU usage information for the programs. Experiments were performed 100 times for all configurations and average values were calculated. No other CPU intensive processes were running on the hosts during the experiments other than those directly related to the grid application and the ALICE components.

The grid application is distributed and requires a network to operate meaningfully. Because this section is concerned with *processing* overhead on the MH, we attempt to minimise the impact that network traffic has on the experiments. First, for all of configurations 3–6, MG1 and RH are configured to be the same host. This means the MG1–RH connection will take place internally in the TCP stack of the MG1/RH. This corresponds more closely to configurations 1 and 2 where only one leg of communication exists, because no MG is present. Second, in all configurations the MH and MGs are connected using an 802.11 WLAN network connection, which has a data rate of 11 Mbps. As discussed in section 2.4, this is one of the fastest (if not the fastest) wireless network technologies supported by mobile devices currently on the market. While this type of link can not be expected to be available to mobile devices at all times, it is used for the experiments presented here, because it minimises the performance penalty due to communiation between the client and server and is therefore more likely to reveal performance penalties due to processing requirements than would slower network technologies.

Overhead for Mobile Client

Table 6.4 shows CPU consumption for configuration 1 (mobile client without ALICE support). For each row, the table's first column lists the component for which the figures in that row apply. The first section of the table gives figures for the client (residing on the MH) and the second for the server (residing on the MG). In configuration 1 there is no ALICE support and the only components residing on the hosts are the grid client and server. The table's second column lists the total run time (wall clock time) that each component took to execute. These differ because the server was started before the client and given grace time to initialise before the client was started. Except for initialisation, the server would only have been active while the client was running. The third column lists the number of CPU seconds each component used directly (in user mode). The fourth column lists the number of CPU seconds used by the system on behalf of each component (in kernel mode). The fifth column (the sum of columns three and four) gives the total number of CPU seconds spent by each component. No other CPU intensive processes than those listed in the table were running during the experiments.

Component	Host	Run Time	User Time	System Time	Total CPU Time
client	MH	91.7s	15.0s	9.1s	24.1s
server	MG1/RH	97.5s	0.03s	0.09s	0.12s

Table 6.4: CPU Consumption for Configuration 1 (Linux)

Table 6.5 shows the CPU consumption for configuration 3 (mobile client with ALICE support) in the same format as that used in table 6.4. Table 6.5 contains two sections: one for components residing on the MH and one for components residing on the MG1/RH. The last row in each section summarises the CPU consumption for all involved components on that host. The components ran for different lengths of (wall clock) time, because they needed to be started in order and short delays were inserted to allow each component to initialise. For example, the ML daemon on each host was given time to initialise before the client and server were started. The run time figures for several components running on the same host are not added up, because the run time is wall clock time and the components were run in parallel.

Component	Host	Run Time	User Time	System Time	Total CPU Time
client	MH	144s	14.4s	9.1s	23.5s
mlmhd	MH	155s	28.2s	8.4s	36.7s
=	MH	–	42.6s	17.5s	60.2s
server	MG1/RH	150s	0.03s	0.09s	0.12s
mlmgd	MG1/RH	158s	2.4s	1.1s	3.5s
=	MG1/RH	–	2.4s	1.2s	3.6s

Table 6.5: CPU Consumption for Configuration 3 (Linux)

A number of observations in relation to processing performance transparency on the MH can be made from comparing tables 6.4 and 6.5. First, using the CORBA instantiation of ALICE with a mobile client causes an overhead of $42.6 - 15.0 = 27.6$ CPU seconds spent by the client directly (in user mode) and $17.5 - 9.1 = 8.4$ CPU seconds spent by the system (in kernel mode) on behalf of the client. This corresponds to relative overheads of 184% and 92% for user and kernel time respectively. The total overhead is $60.2 - 24.1 = 36.1$ CPU seconds (user and kernel), which corresponds to an overall 150% overhead in terms of extra CPU time required for ALICE mobility support.

Table 6.6 shows the CPU consumption for configuration 5 in a similar manner to the two previous tables. This configuration involves three hosts (MH, MG1/RH and MG2) and features the MH moving from MG1/RH to MG2 after it has connected to the server and begun performing invocations. The movement causes a tunnel to be established between the two MGs. Table 6.6 has three sections showing the CPU consumption of each of the three hosts on a per-component basis and also gives the total amount of CPU resource spent by all components on that host. Comparing tables 6.4

and 6.5 shows that the additional overhead compared to configuration 1 (without mobility support) amounts to $61.3 - 24.1 = 37.2$ CPU seconds, corresponding to a relative increase of 154%. Comparing tables 6.5 and 6.6 shows that when the ALICE components are used to add mobility support to the client, the additional tunnel between MGs has little performance impact on the client. Compared to configuration 3, the additional overhead required for configuration 5 amounts to only $61.3 - 60.2 = 1.1$ CPU seconds, a relative increase of 1.8%.

Component	Host	Run Time	User Time	System Time	Total CPU Time
client	MH	196s	14.7s	9.3s	24.0s
mlmhd	MH	208s	28.6s	8.7s	37.3s
=	MH	–	43.3s	18.0s	61.3s
server	MG1/RH	202s	0.02s	0.09s	0.12s
mlmgd	MG1/RH	211s	1.8s	0.5s	2.3s
=	MG1/RH	–	1.8s	0.6s	2.4s
mlmgd	MG2	209s	3.9s	1.6s	5.6s
=	MG2	–	3.9s	1.6s	5.6s

Table 6.6: CPU Consumption for Configuration 5 (Linux)

Comparing the percentage of CPU used in the three configurations shows that processing power was not a bottleneck on the MH. In all of the configurations, the total CPU time spent is considerably lower than the total run time, indicating that in none of the experiments was the CPU fully loaded. The grid client does not perform disk I/O during its operation, so the constraining factor is not related to the speed of local storage devices. As all of the three tables show, the processing performed on the MGs/RH was minimal both without and with ALICE support. This rules out the performance of the server or of ALICE components on the MGs as a constraining factor. Hence, it seems safe to conlude that the performance of configurations 1, 3 and 5 was limited by the network used in the experiments (802.11 WLAN).

Overhead for Mobile Server

Table 6.7 shows the CPU consumption for configuration 2 (mobile server without ALICE support) in the same format as the tables given for the mobile clients. Similarly, tables 6.8 and 6.9 show CPU consumption for configurations 4 and 6 (mobile server with ALICE support with one and two MGs respectively) in the same fashion.

Component	Host	Run Time	User Time	System Time	Total CPU Time
server	MH	99.0s	15.5s	12.7s	28.2s
client	MG1/RH	93.4s	1.3s	4.6s	5.8s

Table 6.7: CPU Consumption for Configuration 2 (Linux)

Component	Host	Run Time	User Time	System Time	Total CPU Time
server	MH	164s	16.8s	15.7s	32.5s
mlmhd	MH	174s	28.4s	8.4s	36.8s
=	MH	–	45.2s	24.1s	69.3s
client	MG1/RH	155s	1.4s	6.1s	7.5s
mlmgd	MG1/RH	191s	2.5s	1.2s	3.7s
siiopmgd	MG1/RH	186s	1.2s	0.2s	1.3s
=	MG1/RH	–	3.7s	1.3s	5.0s

Table 6.8: CPU Consumption for Configuration 4 (Linux)

Component	Host	Run Time	User Time	System Time	Total CPU Time
server	MH	242s	16.9s	15.8s	32.7s
mlmhd	MH	248s	28.9s	8.7s	37.6s
=	MH	–	43.3s	18.0s	70.3s
client	MG1/RH	155s	1.3s	4.6s	6.0s
mlmgd	MG1/RH	253s	0.0s	0.0s	0.0s
siiopmgd	MG1/RH	252s	0.0s	0.0s	0.0s
=	MG1/RH	–	1.3s	4.6s	6.0s
mlmgd	MG2	217s	4.6s	1.9s	6.4s
siiopmgd	MG2	215s	2.9s	0.2s	3.1s
=	MG2	–	7.5s	2.1s	9.5s

Table 6.9: CPU Consumption for Configuration 6 (Linux)

The level of processing performance transparency offered by ALICE configured with the server on the MH can be assessed by comparing tables 6.7 and 6.8. In configuration 4, ALICE causes an overhead of $45.2 - 15.5 = 29.7$ CPU seconds spent by the server directly (in user mode) and $24.1 - 12.7 = 11.4$ CPU seconds spent by the system (in kernel mode) on behalf of the server. This corresponds to relative overheads of 192% and 90% for user and kernel time respectively. The total overhead is $69.3 - 28.2 = 41.1$ CPU seconds (user and kernel), which corresponds to an overall 146% overhead in terms of extra CPU time required for ALICE mobility support in a mobile server configuration.

Table 6.9 shows the CPU consumption for configuration 6 in a similar manner to the two previous tables. This configuration involves three hosts (MH, MG1/RH and MG2) and features the MH moving from MG1/RH to MG2 after the server has published its reference, but before the client starts. This causes the client to be redirected from MG1 to MG2. The difference between CPU consumption in configuration 6 and 2 is is $70.3 - 28.2 = 42.1$ CPU seconds on the MH, corresponding to a total performance overhead of 149% compared the mobile server without mobility support. Comparing tables 6.8 and 6.9 shows that the server receiving invocations from the redirected client performs nearly identically to the server receiving invocations from the client that has not been redirected. Compared to configuration 4 where no client redirection was used, configuration 6 uses $70.3 - 69.3 = 1$ extra CPU second, a relative increase of approximately 1% compared to configuration 4. Hence, the

redirection of the client had little impact on the CPU usage on the MH.

As for the mobile client discussed above, processing power was not a constraining factor on the MH in the experiments. For all of the three configurations, the total CPU time spent was considerably lower than the total run time, indicating that the MH's CPU was far from fully loaded. Like the grid client, the grid server does not perform disk I/O during its operation, so the constraining factor is not related to the speed of MH storage devices. As all of the three tables show, the processing performed on the MGs was minimal both without and with ALICE support. This rules out the performance of the server or of ALICE components on the MGs as a constraining factor. Hence, it seems safe to conlude that the performance of configurations 2, 4 and 6 was limited by the network used in the experiments (802.11 WLAN).

Perspective

This section has examined the processing overhead involved in adding mobility support to a distributed object application using the CORBA instantiation of ALICE described in chapter 5. It was found that the processing overhead attributable to ALICE varied between 149% and 154% on the MH. The overhead for mobile servers was slightly lower (146% and 149%) than for mobile clients (150% and 154%). A possible explanation of this difference is the way in which PDUs are cached by the ML. Every PDU sent is cached until it has been acknowledged by the receiving party. During operation, the client sends more data than the server, because client requests are bigger than server responses. (The exact figures are given in section 6.2.4.) This leads to more data being cached by the ML_{MH} component in the case of the mobile client than in the case of the mobile server and could explain the slight difference in processing overhead.

While the processing overheads found were considerable, the CPU requirements for the application using the ALICE instantiation were still well within the capabilities of a typical mobile device, such as the iPAQ used in the experiments. This was the case, even though the experiments were performed over a high-end wireless network interface with the intention of maximising the use of CPU resources for the duration of the experiments. Of the three groups of mobile devices discussed in section 2.3, the iPAQ belongs in the high end of the middle (PDA) section. Given the good margin with which the iPAQ used in the experiments was able to support the ALICE components and the distributed application, it seems likely that also devices with somewhat more constrained processing capabilities (e.g., the Handspring Visor discussed in section 2.3) will have sufficient processing power to support a CORBA instantiation of ALICE. With regards to highly constrained mobile devices (e.g., the two sensor nodes listed in table 2.2), it is unclear whether the CORBA instantiation of ALICE is too

processing intensive to operate on such devices. However, as discussed in section 6.2.2, these devices are too constrained in terms of storage to support the CORBA instantiation of ALICE, so the question is largely academic. High-spec mobile devices (e.g., the laptop and the wearable computer listed in table 2.2) are considerably more powerful than the iPAQ in terms of processing power and therefore trivially able to support the CORBA instantiation of ALICE.

6.2.4 Performance Transparency: Network

The presence and interactions of the ALICE components cause an increase in network traffic over the MH–MG link. This overhead is due to PDU header fields, such as sequence numbers and checksums, and to acknowledgement messages exchanged between the communicating parties. We refer to this overhead as 'ALICE traffic' to distinguish it from the payload of PDUs which constitutes 'IIOP traffic.' The overhead is the price paid for the mobility support offered by any instantiation of ALICE. This section examines the network traffic caused by the CORBA instantiation of ALICE described in chapter 5. A low level of ALICE traffic corresponds to a high degree of network performance transparency and a good utilisation of the MH's networking resources.

ALICE is designed to work with a variety of transports between the MH and the MGs. As discussed in section 2.4, the majority of networking options typically used with mobile devices offer slower data rates than those used with stationary computers. Also, current mobile devices often rely on power-intensive wireless communication but at the same time are subject to battery constraints. These are incentives to maximise network performance transparency over the link between the MH and MG. In this section, the level of network performance transparency offered by the CORBA instantiation of ALICE is assessed by measuring the traffic caused by the grid application and the ALICE components over the MH–MG link. Inter-MG overhead, such as additional traffic caused by tunneling and handoff between MGs, is not measured. Instead, it is assumed that the fixed network which connects the MGs is not subject to the same constraints (e.g., cost, bandwidth, failure and power issues) as the MH–MG links.

The grid application presented in section 6.1.2 is used for the purpose of measuring network performance transparency over the MH–MG link. Experimental configurations 3–6 are used to measure the amount of data sent and received during the operation of the grid application in different circumstances. 100 identical experiments were conducted with each configuration and average values computed. Data was counted by ALICE code residing in the transport modules. All outgoing PDUs were counted, no matter whether they arrived on the receiving side or not, and any PDUs resent were counted twice. Incoming PDUs were only counted upon successful reception. The amounts of

ALICE and application data were measured separately. This allows the amount of traffic that can be attributed to the ALICE components compared to the grid application to be assessed for mobile clients as well as servers.

Overhead for Mobile Client

Table 6.10 shows the number of bytes sent and received during the operation of experiments with configuration 3 (mobile client with ALICE support and one MG). The bytes sent and received by the grid client are shown separately from those sent and received by ALICE components. The table shows that ALICE traffic accounts for $\frac{750,110}{2,910,110} = 25.8\%$ of the total data sent by the client and $\frac{750,102}{1,410,102} = 53.2\%$ of total data received by the client. The difference in these two figures can be attributed to the fact that most of the requests issued by the grid client (in particular the `get` and `set` operations) include more parameters, and can therefore be expected to be bigger than, the server replies. The amount of ALICE data required to facilitate the transmission of IIOP replies is about the same as that required to transmit IIOP requests.

Component on MH	Link	Bytes Sent	Bytes Received
Grid Client	MH–MG1	2,160,000	660,000
ALICE Components	MH–MG1	750,110	750,102
Total	–	2,910,110	1,410,102

Table 6.10: Network Traffic for Configuration 3 (Linux)

Table 6.11 shows the number of bytes sent and received by the MH during experiments with configuration 5 (mobile client with ALICE support and two MGs). The table contains separate rows for the MH's dialogue with MG1 and MG2. The figures for MG1 pertain to communication that occurred before the MH lost the connection to MG1. For the dialogue with MG1, ALICE-related traffic constitutes $\frac{11,920}{(11,920+37,588)} = 24.1\%$ of the data sent and $\frac{11,894}{11,894+12,049} = 49.8\%$ of the data received. For the dialogue with MG2, ALICE-related traffic constitutes $\frac{688,206}{(688,206+2,172,614)} = 24.1\%$ of the data sent and $\frac{688,188}{(688,188+698,009)} = 49.6\%$ of the data received. Of the total traffic over the critical MH–MG link, ALICE traffic accounts for $\frac{(11,920+688,206)}{2,910,328} = 24.1\%$ of the data sent and $\frac{(11,894+688,188)}{1,410,140} = 49.6\%$ of the data received.

A number of observations can be made from tables 6.10 and 6.11. First, the traffic related to the ALICE components constitutes a considerable portion of the total data exchanged between the client and the server in both configurations. Especially, the server to client case, where ALICE-related data accounts for more than half of the total traffic, leaves plenty of room for improvement. Second, it may seem somewhat surprising that the figures for data sent and received by the ALICE components

Component on MH	Link	Bytes Sent	Bytes Received
Grid Client	MH–MG1	37,588	12,049
Grid Client	MH–MG2	2,172,614	698,009
ALICE Components	MH–MG1	11,920	11,894
ALICE Components	MH–MG2	688,206	688,188
Total	–	2,910,328	1,410,140

Table 6.11: Network Traffic for Configuration 5 (Linux)

are nearly identical. A likely explanation is that the protocol used between the ML_{MH} and ML_{MG} components is symmetrical in the sense that each PDU sent in one direction triggers a response PDU from the other party. Since all PDUs have the same header size, the amount of ALICE data sent in both directions can be expected to be similar.

Overhead for Mobile Server

Table 6.12 shows the number of bytes sent and received during the operation of experiments with configuration 4 (mobile server with ALICE support and one MG). The bytes transmitted by the grid application are shown separately from those transmitted by ALICE components. The table shows that ALICE traffic accounts for $\frac{750,272}{3,510,389} = 21.4\%$ of the total data received by the server and $\frac{750,272}{1,410,453} = 53.2\%$ of the total data sent by the server. As in the mobile client configuration, the difference in these two figures can be attributed to the fact that the client requests are bigger than the server replies, but that the amount of data exchanged between the ALICE components for the requests and replies are nearly the same. As for the mobile client configuration, the incoming and outgoing ALICE traffic can be expected to be nearly identical, because the ML protocol used between the MH and MG is symmetric.

Component on MH	Link	Bytes Sent	Bytes Received
Grid Server	MH–MG1	660,171	2,760,117
ALICE Components	MH–MG1	750,282	750,272
Total	–	1,410,453	3,510,389

Table 6.12: Network Traffic for Configuration 4 (Linux)

Table 6.13 shows the number of bytes sent and received during the operation of experiments with configuration 6 (mobile server with ALICE support and two MGs). The table contains separate rows for the MH's dialogue with MG1 and MG2. The figures for MG1 pertain to communication that occurred as part of the MH's attempt to initiate a connection to MG1 before giving up and obtaining a connection to MG2. No application data is transmitted from the MH to MG1 and the ALICE-related data never arrives. For the dialogue with MG2, ALICE-related traffic constitutes

$\frac{750{,}370}{(750{,}370+660{,}171)} = 53.2\%$ of the data sent and $\frac{750{,}272}{(750{,}272+2{,}760{,}117)} = 21.4\%$ of the data received by the server. This corresponds to the figures obtained for configuration 4.

Component on MH	Link	Bytes Sent	Bytes Received
Grid Server	MH–MG1	0	0
Grid Server	MH–MG2	660,171	2,760,117
ALICE Components	MH–MG1	88	0
ALICE Components	MH–MG2	750,370	750,272
Total	–	1,410,629	3,510,389

Table 6.13: Network Traffic for Configuration 6 (Linux)

As in the case of the mobile client, the experiment conducted with the mobile server shows that ALICE traffic accounts for a considerable portion of the total traffic between the MH and the MG, especially in the server to client direction. A more detailed analysis is required to identify the most suitable optimisations, but a natural starting point would be to improve the protocol used between the ML_{MH} and ML_{MG} components. Possible avenues for reducing ALICE-related traffic would be to employ a header compression scheme, an improved buffering scheme (the current ML instantiation simply sends outstanding data immediately) and an acknowledgement scheme based on bundling or piggy-backing rather than the individual acknowledgements used in the current protocol.

In addition to ALICE-related traffic, the amount of application data exhanged between the client and server could be reduced by adding data compression capabilities to the ML instantiation, as discussed in section 4.5. This could reduce application traffic at the cost of increased processing on the MH and MG. In light of the level of perfomance transparency discussed in section 6.2.3, this is likely to be a useful trade-off.

Other Network Overheads

The preceding sections have characterised the network overhead on the critical MH–MG link. In addition, the ALICE components utilise network resources in the fixed network, i.e., between MGs and RHs involved in the communication. This evaluation assumes these hosts to be 'resource-rich' (see section 2.4.3) compared to mobile devices and therefore does not offer a detailed characterisation of ALICE network performance transparency between these hosts. Instead, this section outlines the types of overheads on the inter-MG and MG–RH links on which a detailed analysis should focus but does not offer a quantitative assessment. Section 7.3 presents some thoughts on how a more detailed performance analysis could be conducted as part of future work.

On inter-MG links, there will be an overhead related to *handoff* when MHs move. One TCP connection is set up for each handoff and persists until the handoff is complete. The amount of data

transmitted between MGs during handoff depends on the number of server sockets that are being proxied by the MGs on behalf of servers residing on the MH. There will also be an overhead related to *tunneling* on inter-MG links when mobile clients or servers migrate with open connections. The CORBA instantiation of ALICE described in chapter 5 uses TCP connections between MGs for this purpose. For each connection being tunneled, there is a setup phase where state related to that connection is transferred. For the remainder of the open connection's life time, only application data is transmitted through the tunnel. In addition, there will be an overhead related to *forwarding pointer management* on the inter-MG links, including links to the MG serving as home agent. The rate of server movement and the frequency at which the home agent is updated determines the level of this overhead. Each home agent update consists of an IIOP invocation, i.e., a TCP connection followed by an exchange of data between the parties.

On MG–RH links, there will be a *redirection overhead* in the case where a client invoking a mobile server follows a chain of forwarding pointers. Each redirection can be expected to be comparable to an IIOP invocation. The number of redirections depends on the length of the path of forwarding pointers that the client has to follow. Because it depends on the age of the reference held by the client, the server's rate of movement and the rate at which the server updates its home agent, the lengths of forwarding pointer chains are difficult to characterise without making specific assumptions about application behaviour and device mobility.

Perspective

This section has examined the network overhead involved in using the CORBA instantiation of ALICE with the grid application described in section 6.1.2. The evaluation took the form of a characterisation of traffic over the critical MH–MG link and left a more comprehensive analysis (including traffic over the inter-MG and MG–RH links) as future work. The data presented here showed that the CORBA instantiation of ALICE described in chapter 5 comes at a considerable cost in terms of ALICE-related traffic over the MH–MG link, and that there is plenty of room for improvement in relation to this aspect of the instantiation. A more detailed analysis is required to identify the most suitable optimisations, but a number of possible candidates were discussed.

The experiments used to examine the level of network performance transparency used 802.11 WLAN, one of the fastest wireless networking technologies in general use. Similar figures can be expected to be obtained for other networking technologies popular with mobile devices, such as the Bluetooth or InfraRed technologies discussed in section 2.4, allowing for minor variations related to the reliability of the network, such as PDUs requiring retransmissions, etc. Because most mobile

networking technologies are slower than 802.11 WLAN, the relatively large network performance overhead of the ALICE instantiation demonstrated in this section can be expected to have a bigger impact on the usability of mobile devices using slower technologies.

On this basis we conclude that while the abstract ALICE architecture presented in chapter 4 in principle offers a high degree of performance transparency in relation to network traffic, the instantiation presented in chapter 5 does come with a considerable overhead and offers a lower degree of transparency than desirable. The main cause appears to be the somewhat simplistic protocol between the MH and MG and that the instantiation of the ML presented in chapter 5 does not implement data compression as described in chapter 4.

6.2.5 Summary

This section has examined the ALICE instantiation presented in chapter 5 in relation to application-level transparency and found that it offers a very high degree of functional transparency and varying degrees of performance transparency for the CORBA application used in the experiments.

In relation to functional transparency, it was shown that the ALICE ML is not quite fully compatible with the Berkeley Sockets API, but that the level of compatibility offered was fully sufficient to add the mobility support offered by the ML (transport management, proxy operation and handoff/tunneling) to the object-oriented middleware framework (the IIOP Layer) in a highly transparent manner. In a similar fashion, the mobility support offered by the S/IIOP Layer (reference swizzling and client redirection) was added to the IIOP Layer also with a high degree of transparency. In particular, the ALICE approach to reference swizzling can be contrasted with Wireless CORBA (section 3.4.2) where mobile servers have to perform their own address translation. The ALICE approach demonstrates how address translation can be performed in a manner that is independent from and transparent to servers. Section 6.4 extends these findings by examining the extent to which the approach can be expected to be generally applicable across object-oriented middleware architectures.

In relation to performance, the CORBA instantiation of ALICE showed varying levels of transparency. In relation to memory, performance transparency is good. Configured for full mobility support, the CORBA instantiation of ALICE constitutes only a 120% increase in the size compared to the IIOP layer. Considering that the IIOP Layer constitutes a very small CORBA implementation, this level of memory overhead seems acceptable for the level of mobility support provided. Also, the total memory requirements were within range of all but the most constrained mobile devices by a good margin. In relation to processing, performance transparency is relatively good. The ALICE layers did result in a considerable overhead (on the order of 150%) for the processing requirements of the grid

application, but the mobile device used in the experiments could easily perform the extra processing required. As discussed in section 6.1.2, the grid application is not processing intensive, and for more applications that are more demanding in this respect, the ALICE processing overhead will play a comparatively smaller role. In terms of network performance transparency, the CORBA instantiation of ALICE presented in chapter 5 is not particularly efficient. As discussed in section 6.1.2, the grid application is intensive in terms of communication, but in some cases the ALICE components even more so. In this respect, the CORBA instantiation of ALICE presented in chapter 5 leaves considerable room for improvement.

6.3 Framework-Level Transparency

Framework-level transparency is the extent to which an object-oriented middleware framework that interacts with an ALICE instantiation can remain unaware that the ALICE instantiation is not an ordinary but a mobility-enabled framework. If the mobility support offered by the ALICE instantiation remains hidden from such other frameworks, we say the ALICE instantiation offers a high level of framework-level transparency. Hence, framework-level transparency is a property of individual ALICE instantiations rather than of the architecture as such. Framework-level transparency essentially has to do with interoperability between different frameworks implementing the same object-oriented middleware architecture. In general, it is critical that different implementations of the same middleware architecture are fully interoperable. This section examines the extent to which the CORBA instantiation of ALICE presented in chapter 5 retains framework-level transparency. The approach is to examine interoperability with ORBacus, a popular commercial ORB, by reviewing experiments conducted with the six experimental configurations presented in section 6.1. ORBacus is not designed specifically for use in mobile environments but its popularity in wired networks makes it suitable for the purposes of examining interoperability.

It should be noted that the interoperability tests performed here build upon the assumption that the IIOP Layer [47] used in the CORBA instantiation of ALICE is fully CORBA-compliant.[1] For this reason, the tests presented here focus on functionality for which the CORBA instantiation of ALICE is different from that of the IIOP Layer. Each of the functions performed by the Transport Modules and Mobility Layer (transport management, proxy operation and handoff/tunneling) and the S/IIOP Layer (reference translation and client redirection) discussed in chapter 4 are considered in turn. For

[1] The documentation for the IIOP Layer raises no concerns about this [47], so the assumption seems reasonable. A fully exhaustive interoperability test of the CORBA instantiation of ALICE remains future work. Several frameworks [108, 110] exist which could be used to assist with this task.

each function, experiments with the six configurations are discussed to show how the function affects interoperability at the framework level. Finally, a summary is presented.

6.3.1 Transport Management

The Transport Modules (section 4.4) and the Mobility Layer (section 4.5) collaborate to manage the communications interfaces on the MH and offer a Berkeley Sockets type abstraction suitable for supporting object-oriented middleware applications.

In configuration 5, the two MGs constituted different transports using the same (UDP) transport module. Experiments with the configuration showed that the CORBA instantiation of ALICE was able to change transport (and MG) without disrupting an ongoing IIOP invocation to the remote server. The remote server was built using an uncustomised version of the ORBacus framework, which had no notion of ALICE mobility support. Apart from a time delay while the connection to MG1 timed out and a connection to MG2 was established, the transport management features were completely transparent to the mobile client as well as the remote server. Configuration 6 featured a server on the MH and mobility from MG1 to MG2, but the move was performed before (rather than during) the invocation in order to allow client redirection. A variation of configuration 6 in which the move was performed after the invocation had begun was tested and worked also as expected. Apart from a delay while the connection to MG1 timed out and a connection to MG2 was established, the move remained transparent to the remote client as well as the mobile server.

6.3.2 Proxy Operation

The ML (section 4.5) allows mobile server sockets to be proxied by MGs. Such sockets can also be relocated to new MGs when MHs move.

Experiments with configuration 4 showed that the CORBA instantiation of ALICE allowed a socket belonging to a mobile server to be proxied by the MG in such a fashion that a client built using the ORBacus framework could connect to it and invoke the server residing on the MH. The fact that the server did not reside on the MG remained completely transparent to the client. Experiments with configuration 6 showed that such proxied server sockets could be relocated to new MGs as the MH changed its point of connectivity. A relocated socket could be used to invoke the mobile server in the same manner as the original proxy socket. Transparency was retained for the client as well as the server.

6.3.3 Handoff/Tunneling

The ML (section 4.5) allows state pertaining to open connections to be transferred between MGs during handoff and also allows inter-MG tunnels to be created for use while such connections persist.

Experiments with configuration 5 showed that an open connection between a mobile client and a remote server could be tunneled between MGs without loss of transparency of either client or server. Experiments with configuration 6 showed that the handoff procedure allowed forwarding pointers to be maintained on MGs.

6.3.4 Reference Translation

The Swizzling Layer (section 4.6) allows server references to be translated as the MH moves between MGs. The effect is that any reference published by the server will always be the current reference.

Experiments with configuration 4 showed that the CORBA instantiation of ALICE allowed server references to be translated in a manner that remained transparent to the server. The server published the translated reference through a shared file system from where it was obtained by the client in the usual manner. Neither client nor server were aware that the reference had been translated. References published as return values or out parameters from incoming invocations or as in parameters to outgoing invocations would also be published in translated state.

6.3.5 Client Redirection

The Swizzling Layer (section 4.6) allows clients holding outdated server references to be forwarded to an MG that has more recently hosted the MH where the server resides.

Experiments with configuration 6 showed that when a translated reference referred to an MG which was no longer hosting the MH, the MG could construct a new reference containing a more recent location (in the form of a different MG) and return it to the client. The client would invoke the server at its new location. The client used for the experiments with configuration 6 was built with an unmodified version of the ORBacus framework, which had no notion of ALICE mobility support. The redirection (and hence, server mobility) remained completely transparent to the client.

6.3.6 Summary

It was found that clients and servers built with the IIOP Layer and used with the CORBA instantiation of ALICE presented in chapter 5 remained fully interoperable with clients and servers built with the

ORBacus framework. On this basis, we conclude that the CORBA instantiation of ALICE offers the same (high) level of framework-level transparency as the IIOP Layer [47].

6.4 Generality

This section evalutes ALICE in relation to *generality*. As discussed in chapter 1, the primary motivation for a general approach to mobility support in object-oriented middleware is that general solutions allow the reuse of ideas, expertise and software components across object-oriented middleware architectures. While we cannot easily evaluate whether ALICE makes it possible to reuse expertise, this section demonstrates that the ideas for solving mobility-related problems adopted in ALICE are generally applicable across object-oriented middleware architectures and that some of the software components presented in chapter 5 can be reused for instantiating ALICE for several such architectures. In the following sections we evaluate the applicability of ALICE to three popular object-oriented middleware architectures—CORBA, Java RMI and SOAP—by discussing the extent to which each middleware architecture satisfies the six ALICE requirements and the level of mobility support offered by an instantiation of ALICE for that architecture. For CORBA and Java RMI, we also compare the level of mobility support offered by an ALICE instantiation to that offered by the mobility-enabled CORBA and Java RMI solutions reviewed in chapter 3.

Figure 6.14 gives an overview of the degree to which the four middleware architectures satisfy the six requirements. A bullet (•) denotes a requirement that is easily satisfied by the middleware architecture in question. A circle (◦) denotes a requirement that can be satisfied with considerable difficulty or only partially. A blank denotes a requirement that cannot be satisfied. A question mark denotes a requirement that is subject to future work. The following sections discuss the table in more detail.

6.4.1 CORBA

As discussed in section 5.4, CORBA satisfies requirements R1–R5 relatively easily. CORBA's remote invocation protocol, the Internet Inter-ORB Protocol (IIOP), is client/server-based and operates over TCP/IP. CORBA therefore easily satisfies requirements R1 and R2. IIOP also allows redirection of clients to new server locations, so CORBA also satisfies requirement R3. CORBA's server reference format, the Interoperable Object Reference (IOR) format, allows server references to contain multiple endpoints in the form of 'profiles' that each identify a server location. A client holding a server reference tries the profiles contained in the reference in sequence when making an invocation. Hence, CORBA

#	Requirement	CORBA	Java RMI	SOAP
R1	The protocol must be client/server-oriented.	•	•	•
R2	The underlying transport must be TCP/IP.	•	•	•
R3	Redirection of client requests towards a different server location must be possible.	•	•	•
R4	A server reference must contain several endpoints at which the server can be found. Clients should try endpoints in order.	•	•	?
R5	It must be possible to store some extra information in a server reference. This information must be passed from client to server during invocation.	•	•	•
R6	The object-oriented middleware architecture must support at least weak object mobility.	○	•	?

Table 6.14: ALICE Requirements and Popular Middleware Architectures

also satisfies requirement R4. Each profile contains a field called the 'object key' which is a string of variable length that is passed from the client to the server during invocation. Hence, CORBA also satisfies requirement R5. However, as noted in section 5.5.1, the thin layer of homogenisation offered by CORBA makes object mobility problematic. The solution described in section 5.5 demonstrated that it is possible to move CORBA objects (code as well as state) between the client and server but with considerable difficulty. The proposed approach required both client and server to retain awareness of differences in platform, and the server needed to retain a repository of binaries for all possible client platforms. Hence, CORBA can be said to satisfy requirement R6 but only with considerable difficulty.

Challenges Related to Mobile Networking and Physical Mobility

Table 6.15 (a synthesis of tables 3.4 and 4.1) compares the level of mobility support offered by the six CORBA mobility support initiatives reviewed in section 3.4 to that offered by a full CORBA instantiation of ALICE and by the prototype CORBA instantiation of ALICE presented in chapter 5. The notation is that also used in previous summary tables. The table shows that in relation to mobile networking and physical mobility, a full CORBA instantiation of ALICE will address the same number of challenges as the most comprehensive CORBA mobility support initiative reviewed in section 3.4: the $\prod^2$ Proxy Platform used in conjunction with FATIMA. The two approaches differ in relation to the extent to which they address the disconnection and usage cost problems. $\prod^2$ with FATIMA offers a superior solution to the usage cost problem through its profile-based transport selection scheme. ALICE for CORBA offers a superior solution to long-term disconnection through the mobility of server replicas.

The prototype CORBA instantiation of ALICE presented in chapters 5 does not address the full set of challenges that a full CORBA instantiation of ALICE would address. While the prototype does

provide full support for dealing with network heterogeneity and address migration, it does not feature an instantiation of the Disconnected Operation Layer and therefore only offers support for short-term disconnections. In addition, the considerable overhead of the ALICE protocols documented in section 6.2.4 and the fact that the instantiation does not currently perform compression of application data means that the instantiation currently does not offer support for dealing with low bandwidth links. Also, the instantiation of the ML presented in chapter 5 does not currently collect usage statistics and does therefore not offer support for dealing with bandwidth variability.

General Category	Specific Challenge	Minimum CORBA	Wireless CORBA	DOLMEN	Π^2 with FATIMA	Object Migration Service	Jumping Beans	ALICE for CORBA (full)	ALICE for CORBA (prototype)
Mobile Devices (section 2.3)	Battery Power	○						○	
	Data Risks								
	User Interface								
	Storage Capacity	●						●	●
	Processing Power	●						●	○
Mobile Networking (section 2.4)	Network Heterogeneity		●		●			●	●
	Disconnection		○	○	○			●	○
	Low Bandwidth			●	●			●	
	Bandwidth Variability				●			●	
	Security Risks								
	Usage Cost			○	●			○	○
Physical Mobility (section 2.5)	Address Migration		●	●	●	○	○	●	●
	Location-Dependent Information								
	Migrating Locality								

Table 6.15: Comparison of CORBA Mobility Support Initiatives

Challenges Related to Mobile Devices

Table 6.15 also summarises the extent to which the CORBA instantiation of ALICE presented in chapter 5 addresses the five challenges related to mobile devices discussed in section 2.3. As discussed in section 6.2.2, the instantiation offers good transparency in relation to memory consumption on the MH and therefore addresses the challenge of limited storage capacity well. However, as discussed in section 6.2.3 the instantiation does come with considerable processing overhead (although well within the capabilities of current mobile devices) and therefore only partially addresses the challenge of limited processing power. In relation to battery power, the considerable network traffic consumption

over the power-consuming wireless link means that the instantiation does nothing to improve battery life on mobile devices.

Based on these experiences, table 6.15 also offers an assessment of the extent to which challenges related to mobile devices could be addressed by a full version of ALICE. Given a better level of network performance transparency would not only result in good support for low bandwidth links (a challenge related to mobile networking) but also address the challenge of restricted battery power by reducing the power requirements over the (commonly wireless and therefore power-intensive) MH–MG link. Reducing the amount of data transmitted over the MH–MG link could potentially also lead to a reduction in processing power required and therefore a higher level of processing performance transparency than offered by the instantiation presented in chapter 5.

6.4.2 Java RMI

As discussed in section 5.6, Java RMI trivially satisfies requirements R1 and R2 but does not have built-in support for multi-endpoint references, keys that are passed from clients to servers during invocation or for redirection of clients towards different server locations. Hence, at a first glance, Java RMI seems not to satisfy requirements R3, R4 and R5. However, section 5.6.3 showed how it was possible to add these features by supplying an extension to the `UnicastRef` class. Hence, Java RMI was shown also to satisfy requirements R3, R4 and R5. In relation to requirement R6, Java RMI relies on the standard Java runtime environment that implements weak object mobility. Hence, Java trivially satisfies requirement R6.

Table 6.16 (a synthesis of tables 3.6 and 4.1) compares the three Java RMI mobility support initiatives discussed in section 3.5 to the mobility support offered by the Java RMI instantiation of ALICE presented in chapter 5. The notation is that also used in previous summary tables. The table shows that an instantiation of ALICE for Java RMI will solve many more of the challenges related to mobile networking and physical mobility than any of the Java RMI mobility initiatives reviewed in section 3.5. Since chapter 5 did not present implementations of the Swizzling and Disconnected Operation Layers for Java RMI, we cannot discuss whether the instantiation addresses the problems related to mobile devices. Those challenges that could possibly be addressed by an ALICE instantiation are marked with question marks in table 6.16. Challenges that would not be addressed are left blank.

6.4.3 SOAP

The Simple Object Access Protocol (SOAP) [29, 170] is being specified by the World Wide Web Consortium (W3C). It is a 'lightweight protocol for exchange of information in a decentralized, distributed

General Category	Specific Challenge	Java 2 Micro Edition	Monads RMI	Mobile RMI	ALICE for Java RMI
Mobile Devices (section 2.3)	Battery Power	○			?
	Data Risks				
	User Interface	•			
	Storage Capacity	•			?
	Processing Power	•			?
Mobile Networking (section 2.4)	Network Heterogeneity	○			•
	Disconnection				•
	Low Bandwidth		•		•
	Bandwidth Variability				•
	Security Risks				
	Usage Cost				○
Physical Mobility (section 2.5)	Address Migration			•	•
	Location-Dependent Information				
	Migrating Locality				

Table 6.16: Comparison of Java RMI Mobility Support Initiatives

environment' [29]. SOAP is a general messaging protocol that can be used to implement a remote procedure call abstraction. The SOAP specification [29] includes a convention for representing remote procedure calls and responses. In the context of ALICE, we are concerned with the use of SOAP as an object-oriented middleware architecture. Hence, the discussion here applies to SOAP for the purposes of remote procedure calls, rather than as a general messaging protocol.

SOAP can operate over a number of transports. Protocol bindings for different transports define how SOAP messages are exchanged over that transport. The first SOAP binding specified by the W3C binds SOAP to the Hypertext Transfer Protocol (HTTP) [62]. Another binding is to the Simple Mail Transfer Protocol (SMTP) [153]. The capabilities of the transport protocols determine the interaction that is possible between the client and the server. For this reason, the choice of transport affects the extent to which SOAP satisfies the six ALICE requirements. To discuss SOAP in relation to the six requirements, we therefore need to consider a particular SOAP binding. In the following, we consider SOAP over HTTP. It should be noted that instantiating ALICE for SOAP is the subject of future work and the results presented here are preliminary. Section 7.3 offers thoughts on future work in this area.

The conventions for using SOAP for remote procedure calls are based on the traditional client/server abstraction. Hence, SOAP trivially satisfies requirement R1. Because HTTP can operate over TCP/IP, SOAP also trivially satisfies requirement R2. In relation to requirement R3 (redirection),

HTTP supports redirection of client requests through the use of the 3xx series status codes. For example, the '301 Moved Permanently' status code allows redirection to a new location. Hence, SOAP also satisfies requirement R3.

In relation to requirement R4 (multi-endpoint references), the situation is less clear. HTTP uses Uniform Resource Identifiers (URIs) [21] to describe the locations of resources. Each URI can refer to only a single location. In the context of SOAP, server endpoints are represented as URIs. Future work on instantiating ALICE for SOAP is expected to reveal whether it is possible to construct multi-endpoint SOAP references, i.e., references with more than one URI. Hence, it is currently unclear to what extent SOAP satisfies requirement R4.

In relation to requirement R5 (extra information), the remote procedure call conventions for SOAP specify that requests must be sent to the server using HTTP POST. Any method parameters are encoded in the HTTP request body, as is this type of HTTP request requires. When using HTTP POST, extra information can also be added to the path portion of the URI, possibly in the form of extra parameters (using '?' notation) or by extending the URL path. Since parameters do not usually appear in the URI for HTTP POST, extra functionality will have to be implemented on the server side to decode both types or parameters. This extra functionality could be implemented in the Swizzling Layer. In this manner, SOAP satisfies requirement R5.

In relation to requirement R6, it is currently unclear to what extent SOAP facilitates object mobility. Like CORBA, SOAP provides homogeneity through standardised ways of describing interfaces and interacting with objects rather than a homogenous execution environment, such as that used with Java RMI. Future work on instantiating ALICE for SOAP is required to reveal the extent to which SOAP satisfies requirement R6.

6.5 Summary

This chapter has evaluated the ALICE architecture presented in chaper 4 in terms of transparency and generality, the two key objectives identified in section 1.5. First, it was shown that a mobility-enabled CORBA framework could be produced by instantiating ALICE for a CORBA implementation (the IIOP Layer), and that the mobility support offered by the resulting instantiation, except for performance issues in particular related to network traffic, remained highly transparent to applications as well as unmodified CORBA frameworks. It was also possible to produce a partial (i.e., mobile client support only) instantiation of ALICE for ORBacus. ALICE mobility support remained highly transparent to the application built with the modified ORBacus framework.

Second, it was shown that the ALICE architecture can improve the operation of object-oriented middleware in mobile environments at least to the level of state of the art systems (for CORBA) and sometimes beyond (for Java RMI) in a manner that is general across middleware architectures. Strong cases for ALICE applicability to CORBA and Java RMI were made, and promising (albeit partial) evidence of applicability to SOAP was presented. In this respect, the ALICE approach to mobility support distinguishes itself from all of the CORBA and Java RMI initiatives reviewed in chapter 3, which provided partial and architecture-specific solutions to mobility challenges.

Seen in a greater context, the thesis shows what is possible in terms of mobility support within the confinements of current models of distributed object computing. ALICE shows how it is possible to retain a high degree of application-level transparency (and therefore a good degree of compatiblity with existing applications) as well as framework-level transparency (and therefore a high degree of interoperability with existing middleware), and at the same time address the specific problems of distributed object applications in mobile environments in a general manner. ALICE also shows where current models for distributed object computing are problematic (e.g., in relation to support for long-term disconnected operation), and it therefore seems natural that the experience with ALICE should feed back into the development of new object-oriented middleware architectures where support for mobile computing is a priority. A natural starting point would be to use the six ALICE requirements as a checklist against which candidate architectures can be evaluated.

Chapter 7

Conclusion

> *Alice looked round her in great surprise. 'Why, I do believe we've been under this tree the whole time! Everything's just as it was!'*
>
> *'Of course it is,' said the Queen, 'what would you have it?'*
>
> *'Well, in OUR country,' said Alice, still panting a little, 'you'd generally get to somewhere else—if you ran very fast for a long time, as we've been doing.'* [36]

This chapter summarises the achievements of the thesis, places them in a greater context and outlines promising directions for future work.

7.1 Achievements

This thesis addressed the general area of middleware for mobile computing. The motivation for the work presented here arose from the observation of two significant developments in relation to distributed systems: the wide acceptance of the distributed object paradigm as a suitable abstraction for building distributed applications and the increasing popularity of mobile computing environments as operating environments for such applications. Perhaps surprisingly, these two developments are not very compatible.

The thesis presented a comprehensive review of the challenges faced by systems that aim to offer support for distributed object applications operating in mobile environments. Fourteen challenges were identified and categorised in three groups related to mobile devices, mobile networking and physical mobility. The relevance of each individual challenge to the problem of supporting distributed

object applications was assessed. The analysis of the mobile environment identified five emerging themes that constituted a high-level synthesis of possible approaches to addressing the challenges.

The thesis reviewed the state of the art in solutions that address these challenges at a variety of levels. Network-layer solutions, such as Mobile IP (see section 3.1.1), were found to solve a very small portion of the fourteen challenges and needed to be used in connection with other solutions in order to be suitable for supporting object-oriented middleware. Most transport-layer solutions were found to address a larger subset of the challenges (some through the adoption of a network-layer solution) but still provided only partial support for object-oriented middleware.

At the middleware layer, research systems such as Rover (see section 3.3.2) have explored the type of support required for distributed object applications operating in mobile environments, but such initiatives typically introduce new middleware architectures and therefore do not easily integrate with existing architectures. The most widely used existing object-oriented middleware architectures were developed with non-mobile distributed computing environments in mind, and as a result lack features required by distributed applications operating in mobile environments. A number of initiatives were reviewed that attempted to address this problem. Of the CORBA initatives, $\prod^2$ with FATIMA (see section 3.4.4) shows how a considerable portion of the fourteen challenges can be addressed in a manner that is highly specific to CORBA. The Java RMI mobility support initiatives are generally less comprehensive than their CORBA counterparts but similarly specific to Java RMI. Until now, little work had been done in relation to integrating support for mobile computing into existing object-oriented middleware architectures in a general manner.

The work presented here is based on the assumption that current object-oriented middleware architectures are basically sound but that they only solve a portion of the problems caused by the mobile environment, and that the solutions tend to be architecture-specific. The ALICE architecture shows how mobility support can be added to existing object-oriented middleware frameworks in a *general* and *transparent* manner. The software components presented (some as implementations, others as designs) could be used to instantiate the ALICE architecture for different object-oriented middleware frameworks, resulting in mobility-augmented frameworks. By demonstrating that ALICE could be instantiated for two popular middleware architectures, it was shown that the approach is generally applicable. The detailed examination of one of the two instantiations showed that mobility support could be added in a highly transparent manner at the application as well as the framework level.

7.2 Perspective

Over the six years that the author has worked with object-oriented middleware in mobile environments, it has become increasingly clear that mobile environments are coming to play an important role as operating environments for distributed applications. Chapter 2 identified two such types of environments. The first was formed by portable computers in the PDA-range and by wireless (and wired) communications technologies currently used with such devices. This type of enviroment has been the focus of the implementation work presented in this thesis. The second environment is that formed by sensor nodes and networks, which were identified as first steps of a gradual realisation of Mark Weiser's vision of ubiquitous computing [187]. Although the current generation of sensor nodes are very constrained in terms of their processing capabilities, this could change within a few years.

An example of a mobile environment that has recently undergone this sort of change is the computational environment constituted by mobile phones. When work on the first version of ALICE began in 1997, mobile handsets were mainly phones with little notion of applications and only one type of communications interface (GSM voice). Six years later, many mobile handsets now have sufficient computational power and wireless communications capabilities that a collection of handsets can be considered a mobile computing environment capable of supporting distributed applications. For example, the author's current phone (a Siemens S55) has 400 KB of memory, supports four types of wireless transports (GSM data/voice, GPRS, Bluetooth and InfraRed) and one type of wired interface (serial) and in addition has the ability to run Java (J2ME) applications. Modern mobile phones also come with an increasing range of applications, many of which are typical of PDAs, such as games, calendars and email clients. It seems likely that environments formed by mobile phones will be host to an increasing number of distributed applications. MP3 players and digital cameras are examples of other types of devices whose increasing processing and communications capabilities mean that they could potentially partake in the formation of mobile environments.

Seen as a whole, the computing environment formed by PDAs, mobile phones, sensor nodes and other devices (including conventional stationary computers) interacting via wired and wireless networks forms a 'post-Internet' of which the current Internet is merely a part. If current research trends continue, the wired (i.e., traditional Internet) part of this environment can be expected to grow slower than that formed by the mobile devices. Chapter 2 showed that many of the assumptions that hold for wired computing environments, such as the current Internet, do not hold for mobile computing environments. As the number of networked mobile devices increases, the assumptions made by current object-oriented middleware architectures become increasingly invalid. The question is: How can distributed applications be built for the post-Internet? What kind of middleware is required? This

thesis has taken the view that the distributed object paradigm despite the difficult nature of mobile environments is still a suitable abstraction for building distributed applications and that current object-oriented middleware architectures remain valid. A bird's eye view of the work described in this thesis is therefore that it shows what can be done in terms of mobility support within the confinements of existing models for distributed object computing. Time will show whether that is sufficient for building future applications.

7.3 Future Work

The detailed mapping of a terrain always results in the discovery of new ground worthy of exploration and often also of particular features of the terrain being mapped that seem to deserve further attention. While pursuing these tasks straight away can be compelling, the explorer must restrain himself to proceed with mapping the main terrain and simply mark such features clearly on the map in order that further explorations can be undertaken. The following sections describe a number of areas in relation to support for mobile computing in object-oriented middleware in general and to ALICE in particular that the author deems suitable for further study.

7.3.1 Completion of Components

A number of the components presented in chapter 5 are partially implemented and lack features described in chapter 4. For example, the implementation of the Mobility Layer does not perform data compression, does not collect statistics related to the performance of the different transports and cannot shorten tunnels between MGs. The Swizzling Layer for CORBA cannot raise `CORBA::TRANSIENT` exceptions when mobile servers are unavailable, even though section 5.4.4 identified this as a desirable ability. Completing the implementations of these components would be a natural first task on a list of future work.

7.3.2 Completeness of Instantiation

The components presented in chapter 5 can be used to instantiate ALICE for CORBA, but because some of the components are presented as designs rather than implementations, they can not be used to build actual instantiations. For this reason, and as discussed in section 6.4, the CORBA instantiation of ALICE does not address the full set of mobility challenges possible with the architecture. An important task would be to implement the two Disconnected Operation Layers (for CORBA and Java RMI respectively) and the Swizzling Layer for Java RMI as specified in the designs. The Disconnected

Operation Layer for CORBA would be a natural first choice, since it would complete the ALICE instantiation for CORBA and allow it to address the full set of mobility challenges possible with the architecture.

Further instantiations of transport modules would allow CORBA and Java RMI instantiations of ALICE to support a greater range of the network technologies discussed in section 2.4. Natural first choices would be InfraRed and Bluetooth which are popular wireless communications interfaces on current mobile devices.

7.3.3 Comprehensive Performance Analysis

The performance analysis presented in chapter 6 characterised the performance of the ALICE instantiation for CORBA but did not present a comparative evaluation. A more thorough analysis would compare the ALICE overhead to the overhead caused by other transport protocols. TCP forms the basis for many remote invocation protocols and would therefore be a natural baseline against which to evaluate the ML. It has been shown that TCP can be implemented very efficiently [43] but also that it performs poorly over wireless links [30]. For this reason, such a performance comparison should be conducted using several (wired as well as wireless) communications interfaces.

The performance analysis presented in chapter 6 focused on the MH and the MH–MG link where resources were most constrained. A more thorough performance analysis would also estimate resource consumption on the MGs, RHs and the links that connect them. As mentioned in section 6.2.4, ALICE has an inter-MG network overhead as a result of traffic related to tunneling and handoffs. Also, an overhead on the MG–RH link can be expected due to redirection of clients towards new server locations. While the performance of ALICE on these hosts and links may not be critical for small scale scenarios, a low resource consumption on these hosts and links would allow the instantiation to scale well in terms of support for many mobile hosts and invocations.

7.3.4 Improved Network Performance Transparency

As discussed in chapter 6, the current instantiation of the ML comes at a considerable overhead in terms of network traffic and therefore offers a lower level of network performance transparency than desirable. As a result, the current ML cannot be said to address low bandwidth (section 2.4.3) or battery power (section 2.3.1) constraints. Better utilisation of the MH–MG link in the form of reduced ALICE traffic would allow the ML to also address these challenges.

7.3.5 Comprehensive Interoperability Study

A full CORBA interoperability test could be conducted in order to show that the CORBA instantiation of ALICE is indeed 100% interoperable and therefore prove more rigorously that the CORBA instantiation of ALICE offers a high level of framework-level transparency. The current instantiation of the S/IIOP Layer relies on the interoperability of the IIOP Layer [47].

7.3.6 Relaxing Requirement R2

The current implementation of the ML could be extended to support UDP in addition to TCP. This would relax requirement R2 to also support protocols over UDP. However, UDP-based remote invocation protocols are not very common. A better approach could be to further generalise the abstract ML to be independent of the Berkeley Sockets interface. This would completely remove requirement R2 and make ALICE even more generally applicable.

7.3.7 Instantiations for Other Middleware Architectures

Chapter 5 made a strong case for ALICE applicability to CORBA and Java RMI. Chapter 6 presented a preliminary discussion on the applicability of ALICE to SOAP. A natural extension of the work described in this thesis is to build a SOAP instantiation of ALICE. This is the subject of ongoing work within the Distributed Systems Group.

Other avenues would be to investigate the applicability of the ALICE architecture to Microsoft's DCOM middleware and/or .NET framework.

7.3.8 Dynamic Configuration

As noted in section 4.8, we expect an ALICE framework to be statically configured through the composition of instantiated components. Because mobile devices are subject to changing conditions, it would be desirable to be able to reconfigure the ALICE stack without interrupting applications. This would allow mobility support layers to be added and removed as required. For example, a Swizzling Layer could be added if a server was created on the MH or a Mobility Layer could be removed if a MH obtained a permanent network connection.

7.3.9 Remaining Challenges

Of the fourteen challenges identified in chapter 2, a full ALICE instantiation for a given object-oriented middleware architecture can address a maximum of nine. Of the remaining five challenges, user

interface (section 2.3.3) is not of specific concern to distributed object applications and therefore not a candidate for future work. Data risks (section 2.3.2) can be reduced through the use of distributed object applications, so while the ALICE architecture does not reduce data risks directly, it does allow such risks to be addressed by offering improved support for applications that reduce data risks. Security risks (section 2.4.5) have received little consideration in this thesis and is a natural topic for future study. A starting point could be to examine how security models used with the different object-oriented middleware architectures could be integrated into the ALICE architecture in a general manner. Some level of security could be provided in the Mobility Layer if it was extended to include a general-purpose encryption layer, such the Secure Sockets Layer (SSL) [65]. This would require some Public Key Infrastructure (PKI) and the presence of Certificate Authorities (CA). As noted in section 2.4.5, this may be difficult in ad hoc environments, but since ALICE is based on the gateway model, this may be a possible extension. The level of support offered by ALICE instantiations in relation to usage cost (section 2.4.6) could be improved by adding a profile-based transport selection scheme along the lines of that found in the $\prod^2$ approach. Such a scheme could be added to the ML or introduced as a new ALICE component. Chapter 2 identified the challenges of location-dependent information (section 2.5.2) and migrating locality (section 2.5.3) as relevant for, but not specific to, distributed object applications. Future work could be to further examine the extent to which these challenges could be addressed within the ALICE architecture.

Appendix A

Educated Guess Algorithm

The cyclic reconnection algorithm described in section 4.5.1 attempts to reconnect through the available transports in a round-robin fashion. It is unlikely, however, that this rather naïve approach will be suitable in all cases, and more advanced algorithms may perform better in many situations. This section describes a possible alternative based on the idea of recording the past reconnection events. The algorithm has not been implemented but is presented here as an example of the type of transport selection strategy could be provided through the ML's tuning interface.

A.1 Description

As an example, assume that a MH is equipped with an Ethernet, a 802.11 WLAN and a GSM phone interface. When the user unplugs the host from the Ethernet in his/her office (without switching on the GSM phone), it is more likely that communication can be resumed over the corporate 802.11 WLAN than on any of the other two transports. The basic assumption of the educated guess algorithm is that user movement patterns will cause certain transitions between the failing and the subsequently successful transport to appear more often than others.

The approach is to maintain a matrix of experience data E for the transports. The matrix, indexed in both dimensions from 0 to $n-1$, is depicted in figure A.1. Each cell holds a value $E_{a,b}$ giving the likelihood (based on past events) that a connection will succeed on transport T_b after it has failed on T_a. The sum of all the cells in a row is 1, i.e., for the row a,

$$\sum_{i=0}^{b} E_{a,i} = 1$$

It should be noted that the matrix does not hold probabilites but rather numeric values representing

experience. It is therefore not guaranteed that connection can indeed succeed after a failure, even when the sum of the cells in a row equal 1.

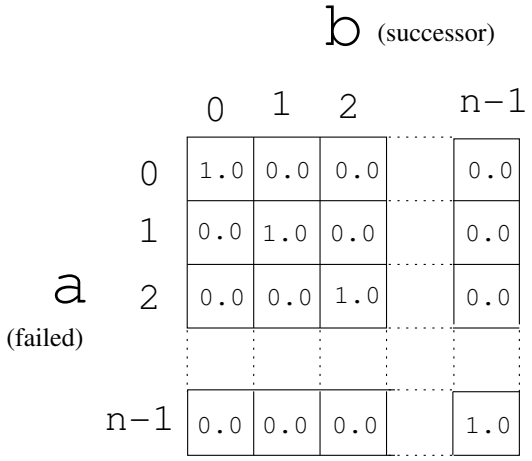


Figure A.1: Experience Matrix E with Initial Values

Given E, the algorithm is depicted in figure A.2. Initially, all cells of the matrix are set to zero, except the diagonal ($E_{i,i}$ where $0 \leq i < n$) which is set to one. Also, $k = 0$ and $k' = 0$. We assume the existence of an update function f for adding information to the matrix. This function is discussed below. Note that in figure A.2, the notation $E(a, b)$ denotes the single cell $E_{a,b}$ whereas $E(a)$ denotes the row E_a.

The intention of the educated guess algorithm is to summarise those geographical movement patterns that cause network failures in an 'experience matrix.' Since this information is gathered dynamically and individually on each mobile host, it reflects that particular host's movement patterns. In other words, the data in the experience matrix which is used to make the decisions is slowly personalised as time progresses. If host movement patterns are regular, the algorithm is intended to recognise it and offer better transport selection than the cyclic retry algorithm. If patterns are irregular, performance of the educated guess algorithm is intended to degrade to that of the cyclic reconnection algorithm.

A.2 Update Function

As mentioned, the educated guess algorithm uses a function for updating a particular row in the experience matrix. The exact nature of the update operation remains to be specified. If connection

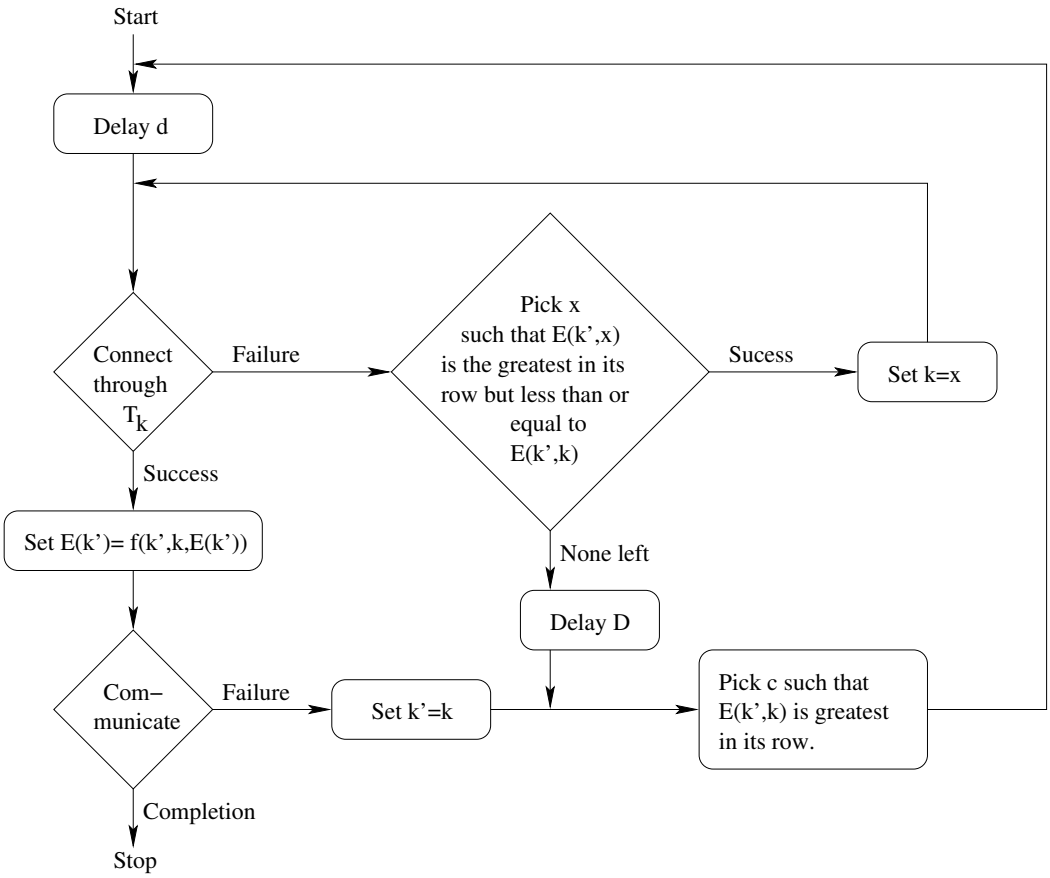


Figure A.2: Educated Guess Algorithm

succeeded on transport T_k after having failed on $T_{k'}$, the function's type would be,

$$f(k', k, P_{k'}) \rightarrow P'_{k'}$$

where $P'_{k'}$ is the replacement row for $P_{k'}$ in the experience matrix E.

A.3 Predefined Network Preferences

The educated guess algorithm does not include a predefined or builtin preference of the transports. This could be a useful feature, since the user would probably prefer, e.g., Ethernet to 802.11 WLAN, if it is at all available. Perhaps such preferences could be supported by initialising the matrix properly, but it would of course require knowledge of the user preferences themselves. On the other hand, user patterns may show that explicit preferences are unnecessary, since fast network connections also tend to be reliable (i.e., have few broken connections), and experience would automatically prefer them.

A possible extension would be to make the experience matrix reflect the quality of the transport with regards to the amount of data passed over it. The algorithm as described here only records information about connection failures and subsequent successes. It would not be difficult to also update the matrix according to how quickly data is transferred over a transport.

Appendix B

Implementation Details

Chapter 5 presented a number of components that could be used to instantiate the ALICE architecture for CORBA and Java RMI. This appendix offers further details in the form of specific interface and protocol descriptions for these components. Whereas the purpose of chapter 5 was to show how ALICE could be instantiated for a variety of middleware architectures, the purpose of this appendix is to serve as a reference for developers who are working with actual ALICE instantiations.

B.1 Transport Modules

Section 5.2 presented instantiations of two ALICE transport modules. This section describes the API used for the transport modules in detail. This API must be implemented by any transport modules intended to be used with the ALICE instantiation discussed in chapter 5.

As explained in section 4.4, the roles of the transport modules on the MH and MG are essentially identical, with the exception that connections are always initiated by the MH. Each transport module implements the same downcall API, given by the C function declarations shown in figure B.1. Each function returns zero on success and non-zero on failure. Transport modules are used only by the ML and have no tuning or upcall APIs. Each of the functions shown in figure B.1 takes as its first parameter a transport to which the operation applies. The functions to save and restore state for the transport take a pointer to an open file where the state information is written and read. The functions to send and receive data take as parameter a Protocol Data Unit (PDU) created by the ML. Except for a Cyclic Redundancy Check (CRC) to verify data integrity, the transport modules treat the ML PDUs as opaque entities that are exchanged between the ML components. The types and formats of the ML PDUs are described in section B.2.4.

```
/* Transport modules have no upcall or tuning API;
   all functions are part of the downcall API. */

int tp_readstate (transport *tp, FILE *fh);
int tp_writestate (transport *tp, FILE *fh);

int tp_start (transport *tp);
int tp_stop (transport *tp);

int tp_connect (transport *tp);
int tp_disconnect (transport *tp);
int tp_check_status (transport *tp);

int tp_send (transport *tp, pdu *p);
int tp_receive (transport *tp, pdu *p);
```

Figure B.1: Interface for Transport Modules (C)

`int tp_readstate (transport *tp, FILE *fh)`

This function reads the transport's state from the supplied file handle. The file must be open and ready for reading. The state is described in section 4.4.3. This function must be called before the transport is started. It returns zero on success and non-zero if the state could not be read.

`int tp_writestate (transport *tp, FILE *fh)`

This function reads the transport's state from the supplied file handle. The state is described in section 4.4.3. This function must be called after the transport has been stopped. It returns zero on success and non-zero if the state could not be written.

`int tp_start (transport *tp)`

This function initialises the transport in question. This involves opening the device and/or initialising the driver for the underlying communications interface. It may also involve setting communications parameters (such as speed and error correction parameters) based on the state of the transport. If the transport was started successfully, the function returns zero, and the transport is ready to be used for sending and receiving PDUs. If the transport could not be started, the function returns non-zero.

`int tp_stop (transport *tp)`

This function shuts down the transport in question. This involves closing the device and releasing any resources related to the communications interface in question. After the transport has been closed, it

can no longer be used to send or receive PDUs. If the transport was stopped successfully, the function returns zero, otherwise a non-zero value is returned.

`int tp_connect (transport *tp)`

This function only exists on the MH. When invoked, the transport attempts to connect to a MG. How this is done depends on the characteristics of the underlying communications interface. If the interface is of the point-to-point variety (e.g., a serial link), a connection attempt simply involves checking if there is an MG at the other end of the link. If the interface gives access to a number of devices within an area (e.g., Bluetooth), connecting to a MG may involve using a discovery type protocol to scan the vicinity for a host that is willing to serve as a MG. The transport may also attempt to connect to a particular (named) MG through the interface. This is useful when the interface (e.g., Ethernet or GSM) gives access to a potentially large number of MGs. The function returns zero if a connection was established, after which PDUs can be sent to the correspondent MG. The function returns non-zero if no connection was established.

`int tp_disconnect (transport *tp)`

This function disconnects from the MG in question. The function returns a zero value if disconnection was successful and non-zero in case the transport was not in a connected state. In either case, no connection is assumed to exist after the function returns.

`int tp_check_status (transport *tp)`

This function queries the current state of the transport. It returns zero if the transport is currently in connected state and non-zero if it is not.

`int tp_send (transport *tp, pdu *p)`

This function sends a PDU over the transport in question. The transport must be in connected state for this function to succeed. The function returns zero if the PDU was sent successfully and non-zero if the connection was lost. Note that since the service offered by the transport is unreliable, a zero return value is no guarantee that the PDU arrived successfully on the receiving side.

`int tp_receive (transport *tp, pdu *p)`

This function receives a PDU via the communications interface in question. The function implements synchronous (blocking) read but may use a timeout to prevent indefinite blocking. If no PDU (or

only a partial PDU) was available within the timeout window, a non-zero value is returned, indicating failure. An error is also returned if a PDU was received which failed the integrity check. No PDU is returned in the case of error conditions.

B.2 Mobility Layer

This section describes the ML instantiation presented in section 5.3 in further detail. Figure B.2 shows the interface to the ML_{MH} component. The first section of the API shows the replacement functions for the Berkeley Sockets API and the functions that allow a layer above the ML_{MH} component to register callbacks for individual sockets. Only those functions for which the ML implements a different behaviour than Berkeley Sockets are shown in the figure. These functions and the differences in behaviour from standard Berkeley Sockets are discussed in appendix B.2.1. The second section of the API shown in figure B.2 shows the tuning API that allows the operation of the ML_{MH} component to be examined and configured at runtime. A detailed description of these functions is given in section B.2.2. Section B.2.3 describes the messages exchanged between the ML_{MG} and mobility-aware applications on the MG. Section B.2.4 describes the protocol used for communication between the ML_{MH} and the ML_{MG} components. Section B.2.5 the protocol used for communication between different ML_{MG} components residing on different MGs.

B.2.1 ML_{MH} Downcall Interface

Section 5.3.1 described the ML_{MH} component. This section describes the API for this component in further detail. The reader is assumed to be familiar with the Berkeley Sockets API [175], of which the ML_{MH} component's API is an extension.

```
int socket(int domain, int type, int protocol)
```

This function first creates a socket on the MH as would an ordinary implementation of Berkeley Sockets. It is simply a wrapper for the standard implementation of the `socket(2)` operation. No proxy socket on the MG is created at this stage.

```
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen)
```

This function establishes a socket connection from the ML_{MH} library to the ML_{MH} daemon and instructs the ML_{MH} daemon to create a socket and outgoing connection to the RH and port number

```
/* Downcall API: Sockets replacement and callback registration functions. */
int socket(int domain, int type, int protocol);
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);
int bind(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int close(int fd);
typedef void (*CBF) (int sockfd, char *new_mg_name, int new_port);
int add_callback(int sockfd, CBF cbf);
int delete_callback(int sockfd);

/* Tuning API. */
typedef int (*RA) (int old_itf);
int set_reconnection_algorithm(RA fa);
int get_transports(int *itfs[], int n);
int get_transport_name(int itf, char *itfname, int len);
get_transport_mg(int itf, char *mgname, int len);
int get_transport_latency(int itf);
int get_transport_errors(int itf);
int get_transport_throughput(int itf);
enum { DISABLED = 0, ENABLED = 1 };
int get_transport_status(int itf, itf_status *st);
int set_transport_status(int itf, itf_status st);
```

Figure B.2: Interface for $\mathrm{ML_{MH}}$ Component (C)

specified in the `serv_addr` parameter. A logical connection identifier is created between the $\mathrm{ML_{MH}}$ and $\mathrm{ML_{MG}}$ daemons to identify the connection.

`int bind(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen)`

This function is used by a server process after a socket has been created to bind that socket to a given interface and port number specified in the `sockaddr` parameter. The ML replacement performs the binding in the standard fashion but also saves the address requested to a temporary data structure where it can be accessed later if the socket is placed in listening mode. This function has no other effect. In particular, it is not relayed to the $\mathrm{ML_{MG}}$ daemon because the ML makes no guarantees as to the port number allocated on the MG.

`int listen(int sockfd, int backlog)`

This function places the socket in listening mode, such that it can accept incoming connections. The ML version of this function passes the call to the underlying Berkeley Sockets implementation but also informs the $\mathrm{ML_{MH}}$ daemon about the port number (on the MH) on which the server is listening. The $\mathrm{ML_{MH}}$ daemon then requests that the $\mathrm{ML_{MG}}$ daemon place the proxy socket in listening mode

to reflect the state of the local socket. When an incoming connection (e.g., from an RH) is received on the proxy socket (by the ML_{MG} daemon), it results in a connection being created from the ML_{MH} daemon to the socket (on the MH) on which the mobile server is listening.

```
int accept(int sockfd, struct sockaddr *serv_addr, socklen_t *addrlen)
```

This function accepts an incoming connection on a socket which has previously been placed in listening mode. The ML version of this function is simply a wrapper for the standard implementation of the `accept(2)` operation.

```
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen)
```

This function obtains the local interface address and port number for a socket. The ML version of this function is a wrapper for the standard implementation of the `getsockname(2)` operation.

```
int getpeername(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen)
```

This function obtains the remote interface address and port number for a connected socket. The ML version of this function is a wrapper for the standard implementation of the `getsockname(2)` operation.

```
int close(int fd)
```

This function closes the socket in question. It can be used on sockets and other file descriptors. If the descriptor identifies a server socket, the ML version of this function requests the ML_{MH} daemon to request the ML_{MG} daemon to close the proxy socket on the MG. After that, the standard implementation of `close(2)` is invoked to close the server socket on the MH. If the descriptor does not identify a server socket, the call is passed to the standard implementation of `close(2)`.

```
typedef void (*CBF) (int sockfd, char *new_mg_name, int new_port);
int add_callback(int sockfd, CBF cbf)
```

This function is not found in the Berkeley Sockets API. It registers a callback function for a given socket. If there is a change in connectivity (i.e., a connection between the MH and MG is lost or created) that affects the socket in question, the ML_{MH} library will invoke the callback function, passing the new connectivity state as parameters.

```
int delete_callback(int sockfd)
```

This function removes a previously registered callback function.

B.2.2 ML_{MH} Tuning Interface

As discussed in section 4.5.4, the ML's tuning API is intended to allow an external (non-ALICE) component (such as a configuration tool) to monitor and control certain aspects of the ML's operation. Although function declarations are provided for the tuning API, they have not been implemented and the ML implementation does not currently gather statistics about the performance of the different transports.

```
typedef int (*RA) (int old_itf);
int set_reconnection_algorithm(RA fa)
```

This method takes as parameter a pointer to a function. This function will be invoked by the ML_{MH} library to decide the transport which to reconnect during subsequent disconnections. If a null value is passed as the `fa` parameter, the ML_{MH} daemon defaults to the standard round-robin policy described in section 4.5.1.

```
int get_transports(int *itfs[], int n)
```

This method returns a sequence of transport identifiers that the ML_{MH} daemon controls. The caller can subsequently invoke other functions on the ML_{MH} library to read and modify the status of each transport.

```
int get_transport_name(int itf, char *itfname, int len)
```

This method takes as parameter a transport identifier and returns a textual, user-readable name associated with the transport, such as 'Serial Port COM2' or 'XirCom GSM Modem.' Transport names are meant to be presented to the user (e.g., as part of a GUI) for easy identification of different transports and not meant to be used for automatic classification of transports. In particular, there is no standardisation scheme for transport names. The `len` parameter specifies the length of the buffer used to store the transport name.

```
int get_transport_mg(int itf, char *mgname, int len)
```

This method returns the hostname of the mobility gateway to which the given transport is currently connected. If the transport is not connected to a mobility gateway, a null value is returned.

```
int get_transport_latency(int itf)

int get_transport_errors(int itf)

int get_transport_throughput(int itf)
```

These functions are intended to give access to statistics about the performance of the individual transports on the MH. The ML does currently not collect statistics, and the functions therefore have no effect.

```
enum { DISABLED = 0, ENABLED = 1 };
int get_transport_status(int itf, itf_status *st)
```

This method queries the status of the given transport, returning either `ENABLED` or `DISABLED`. The ML_{MH} daemon will only attempt to connect to mobility gateways through transports that are `ENABLED`. If none of its transports are enabled, the ML_{MH} daemon will remain passive until at least one transport has been enabled.

```
int set_transport_status(int itf, itf_status st)
```

This method sets the status of the given transport to `ENABLED` or `DISABLED`.

B.2.3 ML_{MG} Downcall Interface

As described in section 5.3.2, the ML_{MG} component can deliver mobility events to any mobility support components (such as Swizzling Layers) that reside on the MG. Each such component creates a TCP connection to a well-known port on which the ML_{MG} component is listening. The mobility support component subsequently receives a stream of bytes and marshalled parameters that constitute a stream of mobility events of the types given below. Because they unlike the ML_{MH} downcall and tuning interfaces are not implemented as C functions, the notation used is that of parameterised messages.

`MSG_MH_CONNECTED(mh_name)` signals arrival of an MH to the MG in question. The hostname of the MH is passed as a parameter to the message.

`MSG_MH_DISCONNECTED(mh_name)` signals the departure of an MH from the MG in question. The hostname of the MH is passed as a parameter to the message.

`MSG_MH_LISTEN(mg_port, mh_name, mh_port)` signals the creation of a new mobile server socket. The port number allocated for the proxy socket is passed as a parameter along with the hostname of the MH and the port number on the MH allocated to the mobile server.

`MSG_MH_CLOSE(mh_name, mh_port)` signals the closing of a mobile server socket. The hostname of the MH is passed as a parameter along with the port number on the MH that is allocated to the mobile server.

`MSG_MH_HANDOFF(mh_name, new_mg_name)` signals that an MH has reconnected to another MG and that a handoff has taken place. The hostnames of the MH and the new MG are passed as parameters. There may or may not previously have been sent a `MSG_DISCONNECTED` message for the MH in question.

B.2.4 $\text{ML}_{\text{MH}} \leftrightarrow \text{ML}_{\text{MG}}$ Interaction

Section 5.3.3 mentioned the protocol used for communication between the ML_{MH} and ML_{MG} daemons. The purpose of this protocol is to allow the socket operations performed on the MH to take place on the MG. This section describes the protocol in more detail. We first present the nine message types and then describe the operation of the protocol.

Message Types

The protocol used to communicate between the ML_{MH} and ML_{MG} daemons uses nine types of messages, described as follows. The function signatures are given using IDL. Except for the positive and negative acknowledgement messages, each message has a corresponding reply message which contains any `out` parameters appearing in the function signatures. The MH and MG maintain a local and a remote socket identifier (`socket_id`) for each ML socket. The former is valid to the party sending the message and the latter to the party receiving it. When a message containing a socket identifier is created, the sender includes the receiving party's socket identifier in the message.

`hello (in string mh, in string last_mg, out string new_mg)`

This message is sent from an MH to an MG to initiate a session between the the two parties. The `mh` parameter is the MH's hostname, and the `last_mg` parameter is the hostname of the previous MG to which the MH was connected. If the previous MG is different from the MG

receiving the `hello` message, the latter will initiate a handoff from the former. The MH may supply an empty string as the name of its previous MG if it was not previously connected to an MG or does not wish to have a handoff initiated. After the MH has received the response message containing the new MG's hostname in the form of the `new_mg` parameter, the remaining eight types of messages can be exchanged.

`ack (in integer msg_id[])`

This message is a positive acknowledgement containing an array of sequence numbers. The receiver of the acknowledgement can be assured that the corresponding party has received the messages with the given sequence numbers and will not request them to be retransmitted in the future. Acknowledgement messages are not themselves subject to acknowledgement and are not sequence numbered.

`nack (in integer msg_id[])`

This message is a negative acknowledgement containing an array of sequence numbers. The receiver of the negative acknowledgement message should resend the messages with the given sequence numbers. Negative acknowledgements can be sent for the same message repeatedly, in case a message gets lost or fails integrity checks several times. Negative acknowledgements are not themselves subject to acknowledgement and are not sequence numbered.

`connect (in string rh_name, in integer rh_port, out mg_sid)`

This message is sent from an $\mathrm{ML_{MH}}$ to an $\mathrm{ML_{MG}}$ daemon where a session is currently in progress. The message instructs the $\mathrm{ML_{MG}}$ daemon to create a client socket and attempt to connect to the given RH and port number. A socket identifier is returned.

`data (in socket_id sid, in sequence<octet> data)`

This message is sent between two $\mathrm{ML_{MH}}$ and $\mathrm{ML_{MG}}$ daemons where a session is currently in progress. It can be sent in either direction. It contains data to be sent over a client or server socket that is currently in connected mode.

`listen (in integer mh_port, out mg_port)`

This message is sent from an $\mathrm{ML_{MH}}$ to an $\mathrm{ML_{MG}}$ daemon. It instructs the latter to create a server socket and start listening. The port number on the MG is allocated dynamically and returned to the $\mathrm{ML_{MH}}$ daemon.

`close (in socket_id sid)`

This message is sent between an $\mathrm{ML_{MH}}$ and an $\mathrm{ML_{MG}}$ daemon in either direction. If originating

at the MH, the message causes the MG to close and destroy the socket associated with `sid` after any outstanding data has been sent. If originating at the MG, the message means that the remote party has closed the connection associated with the socket.

`sclose (in integer mg_port)`

This message is sent from an ML_{MH} to an ML_{MG} daemon as a request for the MG to stop listening on the server socket associated with `mg_port`.

`accept (in socket_id mg_sid, in integer mg_port, out socket_id mh_sid)`

This message is sent from an ML_{MG} to an ML_{MH} daemon when a new connection has been accepted on the port identified by `mg_port`. The ML_{MG} daemon also supplies a socket identifier `mg_sid` for the socket created as a result of the new connection. The ML_{MH} daemon responds with a socket identifier for the new connection.

Operation

Figure B.3 shows a scenario in which an MH is acting as a client and the RH as a server. In the left half of the figure, each arrow corresponds to a message as described in the previous section. Sequence numbers (seqno) and socked identifiers (sid) are shown. Different sequences are maintained for messages exchanged in the $ML_{MH} \rightarrow ML_{MG}$ direction and $ML_{MG} \rightarrow ML_{MH}$ direction. In the right half of the figure, the arrows denote invocations of the Berkeley Sockets API. Hence, the figure shows how the ML messages are translated into invocations of the Berkeley Sockets API by the MG. In the scenario, the MH begins a session with the MG (seqno 0 going right), opens an outgoing connection to a server on an RH (seqno 1 going right), sends a request (seqnos 2, 3 and 4 going right) and receives a reply (seqno 2 going left) and finally closing the socket (seqno 5 going right). The figure also shows one message (seqno 3 going right) getting lost, creating a negative acknowledgement (nack with no sequence number) and being retransmitted. For the sake of simplicity, the diagram does not show positive acknowledgements.

Figure B.4 shows a scenario with a mobile server creating a session with an MG (seqnos 0), creating a server socket on the MG (seqnos 1), accepting a connection from a client on the RH (seqnos 2), receiving data from the RH (seqnos 3, 4 and 5 going left) and sending data in response (seqno 3 going right). After this, the remote client closes the connection (seqno 4 going right). Finally, the MH destroys the listening server socket (seqno 4 going right and 7 going left). For reasons of simplicity, acknowledgements are not shown.

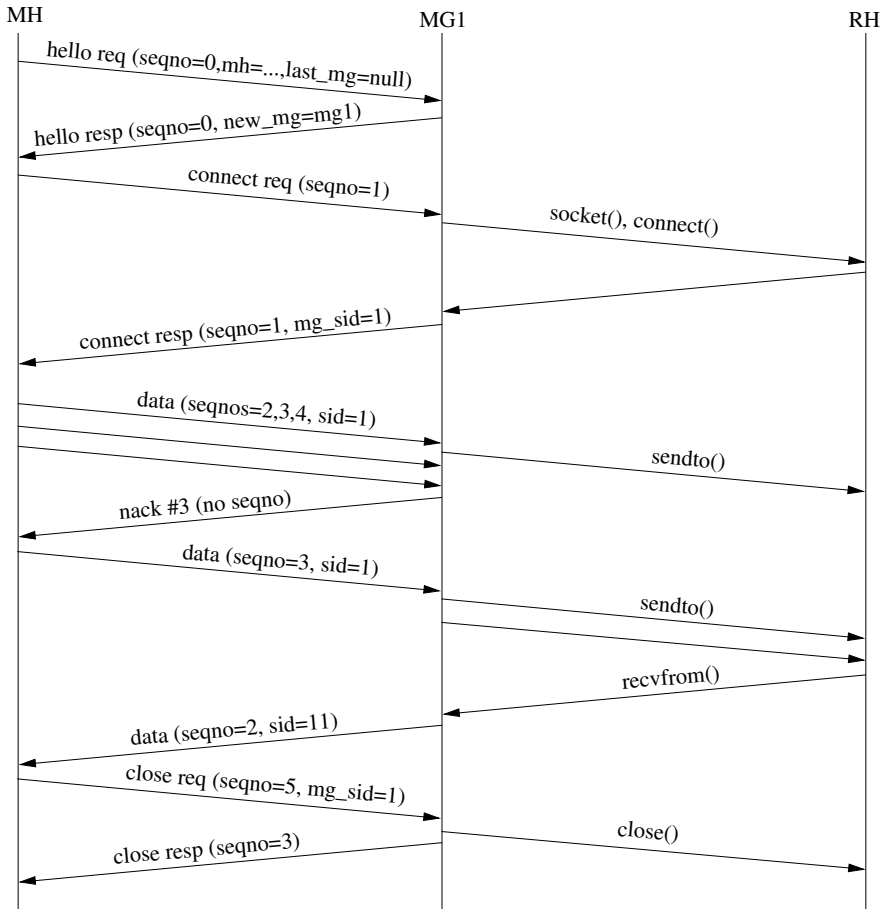


Figure B.3: $\mathrm{ML_{MH}}\leftrightarrow\mathrm{ML_{MG}}\leftrightarrow$RH Interaction with Mobile Client

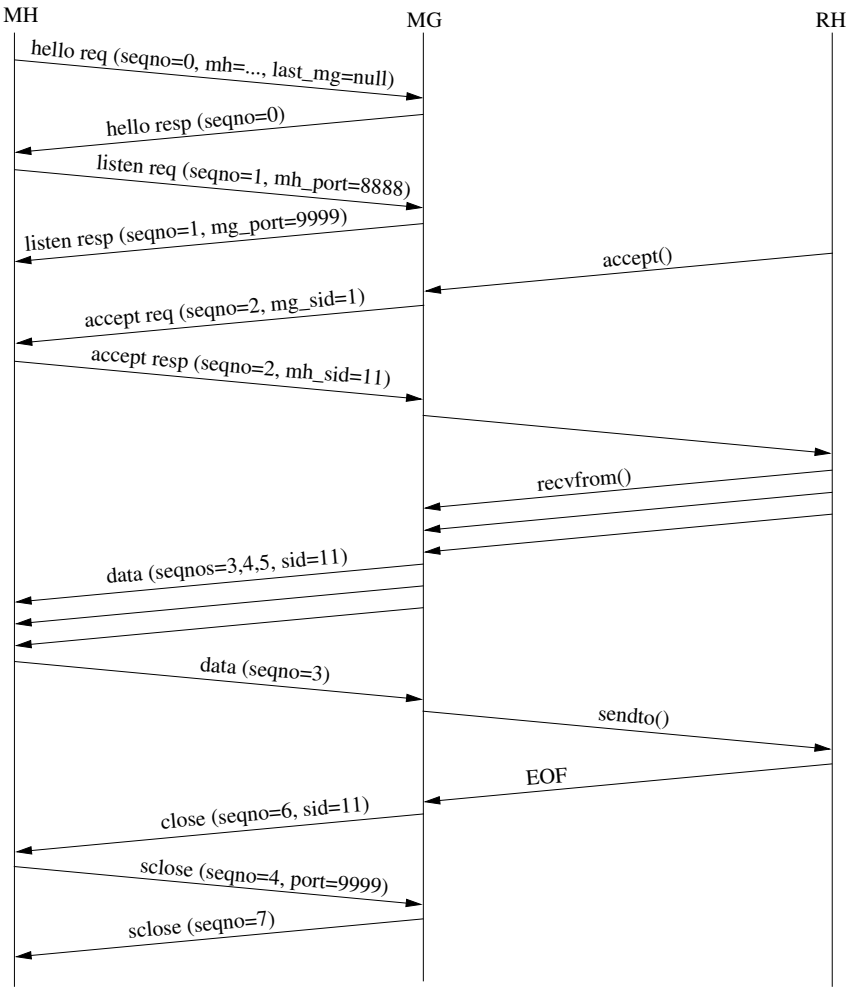


Figure B.4: $\mathrm{ML_{MH}}\leftrightarrow\mathrm{ML_{MG}}\leftrightarrow$RH Interaction with Mobile Server

B.2.5 $ML_{MG} \leftrightarrow ML_{MG}$ Interaction

This section describes in detail the inter-ML_{MG} protocol presented in section 5.3.4. It is a simple protocol that facilitates handoff between MGs and the creation of inter-MG tunnels. We first present the two message types and then describe the operation of the protocol.

Message Types

Two different message types are used between MGs to facilitate handoffs. Each message is sent over a TCP connection.

`handoff (in string mh_name, in string new_mg_name, out message unsent[], out message sent[])`

When an MH arrives at a new MG and needs to initiate a handoff, the new MG sends a handoff message to the old MG. The parameters identify the MH for which handoff is being performed. The new MG also supplies its own hostname to the old MG, such that the old MG can record it and use it to generate upcalls to any applications on the MG, as described in section 4.5.4. Upon receiving the handoff message, the old MG returns any messages queued for the MH and any messages that have been sent but not acknowledged.

`tunnel (in string mh_name, in socket_id mh_sid)`

During handoff, the old MG sets up tunnels to the new MG for any open connections that exist to RHs. This is done using a `tunnel` message for each such connection. The message contains the hostname for the MH and the socket identifier for the connection in question. This allows the new MG to connect the tunnel to the appropriate connection on the appropriate MH. After MH and socket identifier parameters have been sent, the TCP connection becomes a tunnel as described in section 4.5.3. Because the tunnel is a TCP connection, which offers a reliable stream-based service, the data is sent through the tunnel (in either direction) simply as a stream of application data and not as ML PDUs.

Operation

Figure B.5 shows a scenario in which handoff is performed between an old (MG1) and a new (MG2) mobility gateway. The MH requests the handoff by sending a hello message to MG2. MG2 then initiates the handoff by sending a handoff message to MG1. The handoff response includes an unsent message (with seqno 12) which is relayed to the MH immediately. The MH also has two outstanding connections to RHs. For each of these, MG1 completes the handoff by setting up tunnels to MG2. Data

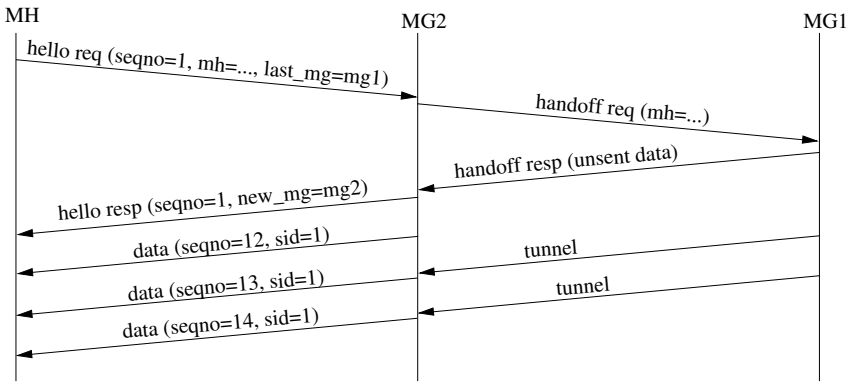


Figure B.5: $ML_{MH} \leftrightarrow ML_{MG} \leftrightarrow ML_{MG}$ Interaction during Handoff

received by MG1 over these connections is then relayed to MG2 through the tunnels and forwarded to the MH as ordinary data messages.

B.3 Swizzling Layer for CORBA

This section describes the instantiation of the S/IIOP Layer presented in section 5.4 in further detail. We give the details for four different interfaces. Sections B.3.1 and B.3.2 describe the downcall and upcall interfaces for the $S/IIOP_{MH}$ component, respectively. Sections B.3.3 and B.3.4 describe the upcall and tuning interfaces for the $S/IIOP_{MG}$ component.

B.3.1 $S/IIOP_{MH}$ Downcall Interface

As shown in figure 4.4, the part of the Swizzling Layer that resides on the MH implements the same downcall API as the underlying protocol employed by the object-oriented middleware for which ALICE is being instantiated. Given an implementation of this protocol, a Swizzling Layer must be written to match. This makes it possible to insert a Swizzling Layer between an application and the protocol in a transparent fashion.

The full API for Cunningham's IIOP Layer is given in [47]. It is an object-oriented API that features C++ classes corresponding to key CORBA abstractions such as IORs, CDR streams and the eight IIOP messages. The $S/IIOP_{MH}$ library works by overriding three methods in a total of two key classes. The following describes the classes and methods affected.

```
int IOR::putProfiles(CDR &cdr)
```

The `IOR` class implements CORBA IORs. It offers basic IOR manipulation facilities, such as creation and destruction of IORs and marshalling and unmarshalling of IORs to and from CDR format. The `IOR` class also allows profiles to be inserted into and removed from existing IORs. A method called `putProfiles()` is used to marshall IORs into CDR form. The S/IIOP$_{\mathrm{MH}}$ library overrides this method, such that the IOR is swizzled before marshalled into CDR form and unswizzled after marshalling has been performed.

```
Status tcpEndpoint::bind(SOCKET &sock, char hostname, unsigned short port)
```

The `tcpEndpoint` class implements a front-end to Berkeley Sockets for use with CORBA servers. After a server has created its IOR, it starts listening on it. This triggers the creation of a number of `tcpEndpoint` objects corresponding to the IIOP profiles that exist in the IOR. The `bind()` method is used to bind sockets to server ports as specified in the profiles. The S/IIOP$_{\mathrm{MH}}$ library overrides the implementation of `bind()` to register a callback function with the underlying ML for the socket in question. The reception of callbacks allows the S/IIOP$_{\mathrm{MH}}$ library to keep continual track of the MH's current state of connectivity and thereby perform swizzling in an up-to-date fashion.

```
int tcpEndpoint::shutdown(SOCKET sock, int mode)
```

The `shutdown()` method works analogously to the Berkeley Sockets function of the same name. In the IIOP Layer, it is invoked on a per-profile basis for the relevant IIOP profiles in a server's IOR when the server stops listening. The S/IIOP$_{\mathrm{MH}}$ library overrides the `shutdown()` method to remove the callbacks registered during the previous invocation of the `bind()` method.

B.3.2 S/IIOP$_{\mathrm{MH}}$ Upcall Interface

In addition to the downcall interface described above, the S/IIOP$_{\mathrm{MH}}$ library also has an upcall interface that is used to receive callbacks from the underlying ML$_{\mathrm{MH}}$ library. Essentially, this API consists of a single callback function that is invoked by the ML$_{\mathrm{MH}}$ library whenever a change in connectivity occurs. Registration and deregistration of callback functions is performed as described in section B.3.1. The callback function has the following form:

```
void siiop_callback(int sockfd, char *mgname, int mgport)
```

This method is invoked after having been passed to the ML_{MH} library as parameter to an `add_callback()` method, as described in section 4.5.4. When the ML_{MH} library subsequently detects a change in connectivity (either the disconnection from or the successful connection to an MG), it signals this by invoking the registered callback functions for all affected socket file descriptors.

In case the callback signifies that a connection to a mobility gateway was lost and no new connection has been established, the `mgname` string contains a null value and the value of `mgport` is undefined. When receiving such an upcall, the $S/IIOP_{MH}$ library knows that the mobile host is now operating in disconnected mode. In case the callback signifies that a connection to a new MG was obtained, the `mgname` string contains the hostname of that MG, and the `mgport` parameter holds the port (on the MG) at which the server process associated with the `sockfd` file descriptor has been relocated. Receipt of such a callback causes the $S/IIOP_{MH}$ library to update the parameters used to swizzle the IORs.

B.3.3 $S/IIOP_{MG}$ Upcall Interface

The upcall interface for the $S/IIOP_{MG}$ daemon is different from that of the $S/IIOP_{MH}$ library. The latter receives an upcall for each server socket for which it has registered a callback, and a socket file descriptor is passed as a callback parameter to let the $S/IIOP_{MH}$ library identify the socket that has been relocated. In comparison, the $S/IIOP_{MG}$ daemon does not need to know about socket file descriptors, but only that the server previously residing at a particular port number at the MG is now located at a different port at a different MG. The $S/IIOP_{MG}$ daemon's upcall interface is implemented to match the ML_{MG} daemon's interface described in section B.2.3. The ML_{MG} daemon generates five types of mobility events that are sent to the $S/IIOP_{MG}$ daemon over a socket connection. The $S/IIOP_{MG}$ daemon deals with them as described in the following.

```
MSG_MH_CONNECTED (mh_name)
```

```
MSG_MH_DISCONNECTED (mh_name)
```

These messages are currently not used by the $S/IIOP_{MG}$ daemon.

```
MSG_MH_LISTEN (mg_port, mh_name, mh_port)
```

After receiving this message, the $S/IIOP_{MG}$ daemon will relay incoming IIOP requests destined for the mobile server identified by `mh_name` and `mh_port` to the local port `mg_port`.

`MSG_MH_CLOSE (mh_name, mh_port)`

After receiving this message, the S/IIOP$_{MG}$ daemon will stop relaying incoming IIOP requests destined for the mobile server identified by `mh_name` and `mh_port`.

`MSG_MH_HANDOFF (mh_name, new_mg_name)`

When receiveing this message, the S/IIOP$_{MG}$ daemon knows that any mobile servers residing on the MH identified by `mh_name` now need to be reached at the MG identified by `new_mg_name`. This allows the S/IIOP$_{MG}$ daemon to create a forwarding pointer for all servers on the MH in question.

B.3.4 S/IIOP$_{MG}$ Tuning Interface

The S/IIOP$_{MG}$ daemon implements a tuning interface that allows the forwarding pointers kept at any MG to be adjusted. This can be used by MHs or MGs to update an MH's home agent and to shorten paths of forwarding pointers. The tuning interface is as defined in section 4.6.2 in IDL:

```
void setLocation(in string mh_name, in string new_mg_name)
```

As discussed in section 5.4.1, the operation is implemented as a standard CORBA invocation. Upon receiving an invocation of the `setLocation` method, the S/IIOP$_{MG}$ daemon updates the forwarding pointer for the MH identified by `mh_name` to point to the MG identified by `new_mg_name`. There are currently no security features implemented; updates are allowed from any parties.

B.4 Disconnected Operation Layer for CORBA

This section describes the instantiation of the D/IIOP Layer presented in section 5.5 in further detail. We give the details for three different interfaces. Section B.4.1 describes the API for the D/IIOP$_{C}$ component, section B.4.2 that for the D/IIOP$_{S}$ component and section B.4.1 the interface implemented by replicas.

B.4.1 D/IIOP$_{C}$ Interface

The D/IIOP$_{C}$ component has the same downcall API as the implementation of IIOP used for the ALICE instantiation. This makes it possible to insert a Disconnected Operation Layer into an existing protocol stack without changing existing code. The part of the D/IIOP Layer that resides on the MH may avail of ML callbacks to monitor the state of connectivity on the MH. Depending on the

```
module DIIOP {

  exception NotCacheable { string reason; };
  exception NotCached { string reason; };

  interface Client {

    interface Downcall {
      // same as underlying IIOP implementation; see below
    }

    interface Upcall {
      // invoked by underlying ML, if client is on MH
      void Callback(in int sockfd, in string mg_name, in int mg_port);
    }

    interface Tuning {
      // invoked by e.g., cache management tool
      void Cache(in IOR ref) raises (NotCacheable);
      void Flush(in IOR ref) raises (NotCached);
      bool IsCached(in IOR ref);

      // invoked by a newly created replica on the client
      void RegisterReplica(in IOR ref);

      // invoked by server on MH, if client is on RH
      void Callback(in IOR ref, bool connection_status);
    }
}
```

Figure B.6: D/IIOP$_C$ Interface (IDL)

application's configuration, this may be the D/IIOP$_C$ or the D/IIOP$_S$ component. For this reason, both parts of the D/IIOP Layer have upcall interfaces. The tuning interface of the D/IIOP$_C$ component allows an external (non-ALICE) component, such as a cache management tool, to control the behaviour of the layer at runtime. CORBA exceptions are thrown as appropriate. This functionality can be used by a cache management tool to implement its own caching policy. The full interface for the D/IIOP$_C$ component is shown in figure B.6 in OMG IDL. The individual operations are discussed in detail below.

`Status ClientEndpoint::Connect(IOR & in)`

This interface is part of the D/IIOP$_C$ component's downcall interface and overrides a function from the IIOP Layer's API. A client using the IIOP Layer works by creating an instance of a class called `ClientEndpoint`. The client then connects the endpoint to a server using the `Connect()` method.

Subsequently, the client can send and receive IIOP messages over the connection.

The D/IIOP$_{\mathrm{C}}$ component modifies the `Connect()` method such that it examines the IOR supplied in the `in` parameter. If the IOR appears in the first column in the Cache Management Table, the `in` parameter is replaced with the IOR appearing in the second column. In both cases (i.e., no matter whether the `in` parameter was modified or not), the `in` parameter is passed to the `Connect()` method of the underlying IIOP Layer. In this way, all outgoing invocations are redirected to remote replicas when available.

```
void Upcall::Callback(in int sockfd, in string mg_name, in int mg_port)
```

This callback method is used when the D/IIOP$_{\mathrm{C}}$ component resides on top of the ML$_{\mathrm{MH}}$ library, i.e., when the client is on the MH. It is invoked by the ML$_{\mathrm{MH}}$ library in the standard fashion described in section 4.5.4. In case the callback signifies that connection to an MG was lost and no new connection has been established, the `mg_name` parameter is null and the value of `mg_port` is undefined. When receiving such an upcall, the D/IIOP$_{\mathrm{C}}$ component knows that the MH is now operating in disconnected mode.

In case the callback signifies that a connection to an MG was obtained, the `mg_name` parameter contains the hostname of the MG and the `mg_port` parameter holds the port to which the server process associated with the `sockfd` file descriptor has been relocated. The D/RIP$_{\mathrm{MH}}$ component may then choose to stop redirecting client invocations to replicas (see section 5.5.4) and to reconcile replicas with the original server objects (see section 5.5.5).

```
void Tuning::Cache(in IOR ref) raises (NotCacheable)
```

The `Cache` operation constitutes a CORBA mapping of the abstract ALICE function with the same name given in section 4.7.2. It is invoked by an external component (e.g., a cache management tool) and signals the D/IIOP$_{\mathrm{C}}$ component to request the server (through the D/IIOP$_{\mathrm{S}}$ component's interface described in section B.4.2) to create a replica that can be transferred to the client, as described in section 5.5.2. As long as the replica is cached on the client side, the D/IIOP$_{\mathrm{C}}$ component will intercept invocations of the real server object and redirect them to the replica, as described in section 5.5.4. The `Cache` operation throws a `NotCacheable` exception in case the server object could not be replicated.

```
bool Tuning::IsCached(in IOR ref)
```

Similarly to the `Cache` operation, the `IsCached` operation is a CORBA mapping of the generic ALICE operation with the same name given in section 4.7.2. The `IsCached` operation queries the D/IIOP$_{\mathrm{C}}$

component as to whether a replica of the given server object currently exists on the client side. It returns `true` if a replica exists and `false` if one does not.

```
void Tuning::Flush(in IOR ref) raises (NotCached)
```

The `Flush` operation is a CORBA mapping of the identically named ALICE operation. It takes as parameter an IOR for a server object for which a replica currently exists on the client side. An invocation of `Flush` signals the D/IIOP$_C$ component to stop caching the replica and reconcile it (via the D/IIOP$_S$ component's API described in section B.4.2) with the original server object. The `Flush` operation throws a `NotCached` exception in case no replica of the specified server object exists on the client side.

```
void Tuning::RegisterReplica(in IOR ref)
```

This method is invoked by a newly created replica on the client side and allows the replica to register itself with the D/IIOP$_C$ component as described in section 5.5.3. The parameter contains an IOR identifying the replica on the client side. After registration, the D/IIOP$_C$ component may start redirecting invocations to the replica.

```
void Tuning::Callback(in IOR ref, bool connection_status)
```

This callback method is used when the D/IIOP$_S$ component resides on top of the ML$_{MH}$ library, i.e., when the client is on the RH. In this case, the D/IIOP$_S$ component relays the mobility events to the client as described in section 4.7.4. This is necessary because the operation of the D/RIP (and hence the D/IIOP) Layer is client-driven. The operation is invoked when there is a change in connectivity status of the MH. The `connection_status` parameter indicates whether the server object identified by `ref` has become available (`true`) or unavailable (`false`). The D/IIOP$_S$ component may not always be able to anticipate impending disconnections and the D/IIOP$_C$ component may therefore not always receive invocations indicating that a server has become unavailable. The D/IIOP$_C$ component receiving notification of a server object's (un)availability changes behaviour as described in sections 5.5.4 and 5.5.5.

B.4.2 D/IIOP$_S$ Interface

The D/IIOP$_S$ component's tuning interface contains functions used to replicate and reconcile server objects for use on the client side during periods of disconnection. Because replication and reconciliation strategies for distributed objects are highly dependent on application semantics, this part of ALICE is

```
module DIIOP {

  exception NotCacheable { string reason; };
  exception NotCached { string reason; };

  typedef sequence <octet> ReplicaCode;
  typedef sequence <octet> ReplicaState;

  interface Server {
    interface Tuning {
      // invoked by clients
      bool IsCacheable(in string arch);
      Replicate(in string arch, out ReplicaCode rc, out
                ReplicaState rs) raises (NotCacheable);
      Reconcile(in ReplicaState rs) raises (NotCached);

      // invoked by clients when server is on the MH
      RegisterCallback(IOR client_ior);
    };
  };
};
```

Figure B.7: D/IIOP$_S$ Interface

contained within the application. The interface is shown in figure B.7. The operations are implemented by server objects that can be replicated. Hence, for each operation, the server object to which the function pertains is implicit.

`bool Tuning::IsCacheable(in string arch)`

This operation allows a client to query whether the server object in question can be replicated. The client passes a string identifying its architecture. The function returns `true` if it is possible to replicate the server object for the architecture in question and `false` it is not. In case the server knows nothing about D/IIOP or ALICE, it raises a `CORBA::BAD_OPERATION` exception (described in [83, 4.12.3.13]). The invoking entity (typically a D/IIOP$_C$ component) treats this exception as if the operation had returned `false`, indicating that the server object cannot be replicated.

`Tuning::Replicate(in string arch, out ReplicaCode rc, out ReplicaState rs)`

When the D/IIOP$_C$ component wishes to replicate a particular server object, it invokes that server object's `Replicate` operation, passing a string identifying the client host's architecture. In return, it receives an object implementing the `Replica` interface given in section B.4.3. The object consists of two portions: code and state. The server may explicitly raise the `NotCacheable` exception if the

server object cannot be replicated, for example because it does not support the client's architecture. The replica passed to the D/IIOP$_C$ component implements the same interfaces as the server except that the `Server` interface shown in figure B.7 is replaced by the `Replica` interface shown in figure B.8. As for the `IsCacheable` operation, the server will raise a `CORBA::BAD_OPERATION` exception if it does not include a D/IIOP$_S$ component. The invoking entity (typically a D/IIOP$_C$ component) treats this exception as it would a `NotCacheable` exception.

`Tuning::Reconcile(in ReplicaState rs)`

This operation is used to reconcile a replica's state (as returned by the replica's `Finish` operation) with the original server object. The D/IIOP$_C$ component invokes the original server object's `Reconcile` operation, passing the replica state as a parameter. It is then up to the original server object to detect and resolve conflicts based on the state information included in the replica. The replica's code is assumed not to have changed and is therefore not passed back to the server.

`Tuning::RegisterCallback(IOR client_ior)`

This operation is invoked by a client on the RH when caching a replica from a server on the MH. As explained in sections 4.7.4 and 5.5.6, the client has no access to ML callbacks in this scenario, because no ML support exists on the RH. The client-driven interaction between the D/IIOP$_S$ and D/IIOP$_C$ components require the callbacks to be relayed from the server to the client. The `client_ior` parameter contains an IOR where the client can receive callbacks. The callback function is given by the `Callback` method discussed in section B.4.1.

B.4.3 D/IIOP$_R$ Interface

The D/IIOP$_S$ component can return a marshalled replica of a server object. Such replicas are transferred to the client side where they are first unmarshalled, then operate for a period of time and then are transferred back to the originating server where they are reintegrated. Replicas implement an interface that allows the D/IIOP$_C$ component to shut them down cleanly and produce a marshalled object that can be transferred back to the server. This interface is shown in figure B.8 and discussed below.

`Replica::Finish(out ReplicaState rs)`

This function is implemented by a running replica hosted by the D/IIOP$_C$ component. It is invoked by the D/IIOP$_C$ component when the replica's state needs be transferred to the server side for rec-

```
// IDL
module DIIOP {

  typedef sequence <octet> ReplicaState;

  interface Replica {
      Finish(out ReplicaState rs);
  };
};
```

Figure B.8: D/IIOP$_{\mathrm{R}}$ Interface

onciliation. The replica implementing this operation must shut down its operation completely and produce marshalled state that can be sent from the D/IIOP$_{\mathrm{C}}$ to the D/IIOP$_{\mathrm{S}}$ component.

Bibliography

[1] Psion Organiser 1 Pocket Computer Home Page. http://members.surfeu.at/org2/psion1. Viewed 27 September 2002.

[2] Java 2 Platform, Standard Edition, v1.2.2, API Specification, 1999.

[3] What's It All About: An i-mode Primer. *BusinessWeek Online*, 2000-01-17.

[4] Bluetooth Specification Version 1.1, February 2001.

[5] Charter of the Multicast-Address Allocation (malloc) Working Group, October 2002.

[6] Vodafone Pay Monthly Options, 2002.

[7] Ayman Abdel-Hamid and Hussein Abdel-Wahab. Local-Area Mobility Support through Cooperating Hierarchies of Mobile IP Foreign Agents. In *Proceedings of the Sixth IEEE Symposium on Computers and Communications*, pages 479–484, Hammamet, Tunisia, July 2001. IEEE.

[8] Sandeep Adwankar. Mobile CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA '01)*, pages 52–63, Rome, Italy, September 2001.

[9] Oguz Angin, Andrew T. Campbell, Michael E. Kounavis, and Raymond R.-F. Liao. Open Programmable Mobile Networks. In *Proceedings of the Eight International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Cambridge, England, July 1998.

[10] Ken Arnold. The Jini Architecture: Dynamic Services in a Flexible Network. In *Proceedings of the 36th Design Automation Conference*, pages 157–162, New Orleans, LA, USA, June 1999. IEEE.

[11] Marco Avvenuti and Alessio Vecchio. Embedding Remote Object Mobility in Java RMI. In *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '01)*, pages 98–104, Bologna, Italy, October 2001.

[12] Marco Avvenuti and Alessio Vecchio. Supporting Remote Reference Updating Through Garbage Collection in a Mobile Object System. In *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing*, pages 65–70, Mantova, Italy, February 2001.

[13] Jean Bacon. *Concurrent Systems.* Addison-Wesley, 2nd edition, 1998.

[14] B. R. Badrinath, Arup Acharya, and Tomasz Imielinski. Impact of Mobility on Distributed Computations. *Operating Systems Review*, 27(2):15–20, 1993.

[15] Arno Bakker, Maarten van Steen, and Andrew S. Tanenbaum. From Remote Objects to Physically Distributed Objects. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47–52, Cape Town, South Africa, December 1999.

[16] Ajay V. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 136–143. IEEE Computer Society Press, 1995.

[17] Ajay V. Bakre and B. R. Badrinath. M-RPC: A Remote Procedure Call Service for Mobile Clients. In *Proceedings of the 1st International Conference on Mobile Computing and Networking (MobiCom'95)*, pages 97–110, 1995.

[18] Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman. A Case for Message Oriented Middleware. *Lecture Notes in Computer Science*, 1693, 1999.

[19] Arash Baratloo, P. Emerald Chung, Yennun Huang, Sampath Rangarajan, and Shalini Yajnik. Filterfresh: Hot Replication of Java RMI Server Objects. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 65–78, Santa Fe, New Mexico, April 1998.

[20] Bill Beckwith. MinimumCORBA Tutorial Notes. OMG Real-Time and Embedded Workshop, July 2002.

[21] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifiers (URI): Generic Syntax. Technical Report RFC2396, IETF, August 1998.

[22] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, 1996.

[23] Greg Biegel, Vinny Cahill, and Mads Haahr. A Dynamic Proxy-Based Architecture to Support Distributed Java Objects in a Mobile Environment. In *Proceedings of the International Symposium of Distributed Objects and Applications*, October 2002.

[24] Kenneth Black, Jon Currey, Jaakko Kangasharju, Jari Länsiö, and Kimmo Raatikainen. Wireless Access and Terminal Mobility in CORBA. White Paper, 2001.

[25] Gordon S. Blair, Geoff Coulson, Philippe Robin, and Michael Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE 1998)*, pages 191–206, London, 1998. Springer-Verlag.

[26] Christian Blum and Refik Molva. A CORBA-based Platform for Distributed Multimedia Applications. In *Proceedings of Multimedia Computing and Networking (MMCN'97)*, San Jose, CA, February 1997.

[27] Gaetano Borriello. The Challenges to Invisible Computing. *IEEE Computer*, 33(11):123–125, November 2000.

[28] Jim Bound, Charles E. Perkins, Mike Carney, and Ralph Droms. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). Internet draft, IETF, January 2001.

[29] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. Technical report, W3C, May 2000.

[30] Ramón Caceres and Liviu Iftode. Improving the performance of reliable transport protocols in mobile computing environments. *IEEE Journal on Selected Areas in Communications*, 13(5):850–857, June 1995.

[31] Stefano Campadello, Heikki Helin, Oskari Koskimies, and Kimmo Raatikainen. Performance Enhancing Proxies for Java2 RMI over Slow Wireless Links. In *Proceedings of the Second International Conference on the Practical Application on Java*, pages 76–89, Manchester, UK, April 2000.

[32] Stefano Campadello, Oskari Koskimies, Kimmo Raatikainen, and Heikki Helin. Wireless Java RMI. In *Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC 2000)*, pages 114–123, Makuhari, Japan, September 2000.

[33] Andrew T. Campbell, Javier Gomez, Sanghyo Kim, Zoltan Turanyi, Chieh-Yih Wan, and Andras G. Valko. Comparison of IP Micro-Mobility Protocols. *IEEE Wireless Communications Magazine*, 9(1):72–82, February 2002.

[34] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Middleware for Mobile Computing: Awareness vs. Transparency (Position Summary). In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001. IEEE Computer Society Press.

[35] Lewis Carroll. *Alice's Adventures in Wonderland.* Macmillan and Co., London, 1866.

[36] Lewis Carroll. *Through the Looking-Glass, and What Alice Found There.* Macmillan and Co., London, 1872.

[37] Kaushik Chakraborty, Archan Misra, Subir Das, Anthony McAuley, Ashutosh Dutta, and Sajal K. Das. Implementation and Performance Evaluation of TeleMIP. In *Proceedings of the IEEE International Conference on Communications (ICC 2001)*, pages 2488–2493, Helsinki, Finland, June 2001. IEEE.

[38] Stuart Cheshire and Mary Baker. Internet Mobility 4x4. In *ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 318–329, 1996.

[39] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences. In *Proceedings of CHI 2000*, pages 17–24, Netherlands, April 2000.

[40] Imrich Chlamtac and Jason Redi. Mobile Computing: Challenges and Potential. In Anthony Ralston, Edwin D. Reilly Jr., and David Hemmendinger, editors, *Encyclopedia of Computer Science.* International Thomson Publishing, 4th edition, 1998.

[41] Hoon Choi and Nader Moayeri. A Fast Handoff Scheme for Packet Data Service in the CDMA 2000 System. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM '01)*, pages 1747–1753. IEEE, November 2001.

[42] Sang Choy, Markus Breugst, and Thomas Magedanz. A CORBA Environment Supporting Mobile Objects. *Lecture Notes in Computer Science*, 1597:168–180, 1999.

[43] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[44] Michael Clarke, Tom Fitzpatrick, and Geoff Coulson. Adaptive System Support for Multimedia in Mobile End-Systems. In *Proceedings of the 3rd Communications Networks Symposium*, July 1996.

[45] M. Scott Corson and Joseph Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. Technical Report RFC2501, IETF, January 1999.

[46] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 2nd edition, 1994.

[47] Raymond Cunningham. Architecture for Location Independent CORBA Environments. Master's thesis, University of Dublin, Trinity College, Ireland, September 1998. Also published as Technical Report TCD-CS-1999-29.

[48] Nigel Davies, Gordon S. Blair, Keith Cheverst, and Adrian Friday. Supporting Adaptive Services in a Heterogeneous Mobile Environment. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, USA, 1994.

[49] Nigel Davies, Stephen P. Wade, Adrian Friday, and Gordon S. Blair. Limbo: a Tuple Space Based Platform for Adaptive Mobile Applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pages 291–302, Toronto, Canada, May 1997.

[50] Jesse M. Davis. *An Ambient Computing System.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Kansas, 2001.

[51] Prasan de Silva and Harsha Sirisena. A Mobility Management Protocol for IP-Based Cellular Networks. In Jenny Li, Ronald Luijten, and Eun Kyo Park, editors, *Proceedings of the Tenth International Conference on Computer Communications and Networks*, pages 476–482, Scottsdale, AZ, USA, October 2001. IEEE.

[52] Stephen E. Deering. ICMP Router Discovery Messages. Technical Report RFC1256, IETF, September 1991.

[53] Stephen E. Deering and Robert M. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Draft Standard RFC2460, IETF, December 1998.

[54] Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. Technical Report RFC2246, IETF, January 1999.

[55] Asuman Dogac, Cevdet Dengi, and M. Tamer Öszu. Distributed Object Computing Platforms. *Communications of the ACM*, 41(9):95–103, 1998.

[56] Ralph Droms. Dynamic Host Configuration Protocol. Technical Report RFC1541, IETF, October 1993.

[57] Dan Duchamp. Issues in Wireless Mobile Computing. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 2–10, Key Biscayne, Florida, U.S., 1992. IEEE Computer Society Press.

[58] Ad Astra Engineering. Jumping Beans White Paper. http://www.jumpingbeans.com/, December 1998.

[59] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 263–270, 1999.

[60] Victor Hayes et al. ANSI/IEEE Std. 802.11, 1999 Edition.

[61] Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Technical Report RFC2267, IETF, January 1998.

[62] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Technical Report RFC2616, W3C, June 1999.

[63] George H. Forman and John Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, 1994.

[64] Robert J. Fowler. Decentralized Object Finding Using Forwarding Addresses. Technical Report 85-12-1, University of Washington, Seattle, WA (USA), 1985.

[65] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol, Version 3.0, November 1996.

[66] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[67] Qiang Gao and Anthony S. Acampora. Connection Tree Based Micro-Mobility Management for IP-Centric Mobile Networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC2002)*, pages 30–35, Orlando, FL, USA, March 2002. IEEE.

[68] Kurt Geihs. Middleware Challenges Ahead. *IEEE Computer*, 34(6):24–31, June 2001.

[69] Eugene Gluzberg and Stephen Fink. An Evaluation of Java System Services with Microbenchmarks. Technical Report RC 21715 2/3/2000, IBM, 2000.

[70] Aniruddha S. Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 306–317, 1996.

[71] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[72] António Grilo, Pedro Estrela, and Mário S. Nunes. Terminal Independent Mobility for IP (TIMIP). *IEEE Communications Magazine*, 39(12):34–41, December 2001.

[73] William Grosso. *Java RMI*. O'Reilly, November 2001.

[74] Object Management Group. The Common Object Request Broker: Architecture and Specification, V2.0, February 1997. formal/97-02-25.

[75] Object Management Group. CORBA Finance: Financial Domain Specifications, V1.0, December 1998. formal/98-12-02.

[76] Object Management Group. Objects By Value, January 1998. 98-01-18.

[77] Object Management Group. CORBA Med: Healthcare Domain Specifications, V1.0, March 1999. formal/99-03-01.

[78] Object Management Group. Externalization Service Specification, v1.0, April 2000. 00-06-16.

[79] Object Management Group. Telecom Wireless CORBA Specification, June 2001. 01-06-02.

[80] Object Management Group. Life Cycle Service Specification, v1.2, September 2002. 02-09-01.

[81] Object Management Group. Minimum CORBA Specification, V1.0, August 2002. formal/02-08-01.

[82] Object Management Group. Naming Service Specification, v1.2, September 2002. 02-09-02.

[83] Object Management Group. The Common Object Request Broker: Architecture and Specification, V3.0, July 2002. formal/02-06-01.

[84] Mads Haahr, Raymond Cunningham, and Vinny Cahill. Supporting CORBA Applications in a Mobile Environment. In *Proceedings of the 5th International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 36–47. ACM, August 1999.

[85] Frederick Hayes-Roth. Architecture-Based Acquisition and Development of Software: Guidelines and Recommendation from the ARPA Domain Specific Software Architecture (DSSA) Program Version 2, 1994.

[86] Sumi Helal. Pervasive Java. *IEEE Pervasive Computing*, 1(1):82–85, January 2002.

[87] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[88] Akihiro Hokimoto and Tatsuo Nakajima. Handling Continuous Media in Mobile Computing Environment. In *Proceedings of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 175–182, 1996.

[89] Russell Housley, Warwick Ford, Tim Polk, and David Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Technical Report RFC2459, IETF, January 1999.

[90] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[91] International Organization for Standardization. International Standard ISO/IEC 14977, 1996.

[92] John Ioannidis and Gerald Q. Maguire Jr. The Design and Implementation of a Mobile Internetworking Architecture. In *USENIX Winter*, pages 489–502, 1993.

[93] Ken Jackson and Maarten Boasson. The Benefits of Good Architectural Style in the Engineering of Computer Based Systems. In *Proceedings of the 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems*, Tucson, Arizona, March 1995.

[94] David B. Johnson. Mobile Host Internetworking Using IP Loose Source Routing. Technical report, School of Computer Science, Carnegie Mellon University, USA, 1993.

[95] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In Tomasz Imieliński and Henry F. Korth, editors, *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[96] David B. Johnson, Charles E. Perkins, and Jari Arkko. Mobility Support in IPv6. Internet draft, IETF, June 2002.

[97] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and Frans M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 156–171, Copper Mountain, Co., 1995.

[98] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, 46(3):337–352, 1997.

[99] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[100] Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". In *Proceedings of the 5th International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 271–278, Seattle, USA, 1999.

[101] Georgios Karagiannis. Mobile IP: State of the Art Report. Technical Report 3/0362-CFP NB 102 88 Uen, Ericsson, July 1999.

[102] Steven S. King. Middleware! Making the Network Safe for Application Software. *Data Communications Magazine*, 21(4):58–67, 1992.

[103] Markku Kojo, Kimmo Raatikainen, and Timo O. Alanko. Connecting Mobile Workstations to the Internet over a Digital Cellular Telephone Network. Technical report, University of Helsinki, Helsinki, Finland, 1994.

[104] Markku Kojo, Kimmo Raatikainen, and Timo O. Alanko. Connecting Mobile Workstations to the Internet Over a Digital Cellular Telephone Network. In Tomasz Imieliński and Henry F. Korth, editors, *Mobile Computing*, pages 253–270. Kluwer Academic Publishers, 1996.

[105] Markku Kojo, Kimmo Raatikainen, Mika Liljeberg, Jani Kiiskinen, and Timo O. Alanko. An Efficient Transport Service for Slow Wireless Telephone Links. *IEEE Journal on Selected Areas in Communications*, 15(7):1337–1348, September 1997.

[106] Pradip Lamsal. J2ME™Architecture And Related Embedded Technologies. Unpublished technical report, University of Helsinki.

[107] Sean Landis and Silvano Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[108] Eric Leach. Validating Multi-Vendor CORBA Conformance and Interoperability in Heterogenous Environments. White Paper, December 2001.

[109] Marcus Leech, Matt Ganis, Ying-Da Lee, Ron Kuris, David Koblas, and LaMont Jones. SOCKS Protocol Version 5. Technical Report RFC1928, IETF, March 1996.

[110] Mang Li and Arno Puder. A Test Framework for CORBA Interoperability. In *Fifth IEEE International Enterprise Distributed Object Computing Conference*, pages 152–163, Seattle, Washington, September 2001. IEEE Computer Society Press.

[111] Joshua Lifton, Deva Seetharam, Michael Broxton, and Joseph Paradiso. Pushpin Computing System Overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networks. In Friedemann Mattern and Mahmoud Naghshineh, editors, *Proceedings of the First International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002. Springer-Verlag.

[112] Mika Liljeberg, Kimmo Raatikainen, Michael P. Evans, Steven M. Furnell, Nicholas Maumon, Eric Veldkamp, Bernie C. F. Wind, and Sebastiano Trigila. Using CORBA to Support Terminal Mobility. In *Proceedings of Global Convergence of Telecommunications and Distributed Object Computing (TINA 97)*, pages 59–67, Santiago, Chile, November 1997.

[113] Steve Litchfield. The History of Psion (from 1980 to 1997). *Palmtop Magazine*, 1998.

[114] David A. Maltz and Pravin Bhagwat. MSOCKS: An Architecture for Transport Layer Mobility. In *Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'98)*, pages 1037–1045, San Francisco, CA, USA, March 1998.

[115] Steve Mann. 'Smart Clothing': Wearable Multimedia Computing and 'Personal Imaging' to Restore the Technological Balance Between People and Their Environments. In *Proceedings of ACM Multimedia*, pages 163–174, November 1996.

[116] Seapahn Meguerdichian, Farinaz Koushanfar, Miodrag Potkonjak, and Mani B. Srivastava. Coverage Problems in Wireless Ad-hoc Sensor Networks. In *INFOCOM*, pages 1380–1387, 2001.

[117] Paul Meller. EU sees IPv6 Adoption as Crucial to 3G Success, April 27 2001.

[118] Sun Microsystems. Java Remote Method Invocation: Distributed Computing for Java. White Paper.

[119] Sun Microsystems. Java™2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices. White Paper, May 2000.

[120] Brent A. Miller and Chatschik Bisdikian. *Bluetooth Revealed.* Prentice-Hall, 2001.

[121] Gary J. Minden and Joseph B. Evans. Ambient Computational Environments. In P. Sadayappan, editor, *Proceeding of the 2000 International Workshop on Parallel Processing*, pages 39–42, Toronto, Canada, August 2000.

[122] Stefan Mink, Frank Pählke, Günter Schafer, and Jochen Schiller. FATIMA: a Firewall-Aware Transparent Internet Mobility Architecture. In *Proceedings of the Fifth IEEE Symposium on Computers and Communications (ISCC 2000)*, pages 172–179, Antibes-Juan les Pins, France, July 2000.

[123] Gabriel E. Montenegro. Reverse Tunneling for Mobile IP. Technical Report RFC2344, IETF, May 1998.

[124] Gabriel E. Montenegro and Vipul Gupta. Sun's SKIP Firewall Traversal for Mobile IP. Technical Report RFC2356, IETF, June 1998.

[125] J. Eliot B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[126] Amy L. Murphy and Gian Pietro Picco Gruia-Catalin Roman. Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21 st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 524–536. IEEE Press, April 2001.

[127] Computer History Museum. Timeline of Computer History. http://www.computerhistory.org/. Viewed 18 October 2003.

[128] Jayanth Mysore and Vaduvur Bharghavan. A New Multicasting-Based Architecture for Internet Host Mobility. In *Proceedings of the 3rd International Conference on Mobile Computing and Networking (MobiCom'97)*, pages 161–172, 1997.

[129] Thomas Narten, Erik Nordmark, and William Allen Simpson. Neighbor Discovery for IP Version 6 (IPv6). Draft Standard RFC2461, IETF, December 1998.

[130] B. Clifford Neuman, Steven Seger Augart, and Shantaprasad Upasani. Using Prospero to Support Integrated Location-Independent Computing. In *Proceedings USENIX Symposium on Mobile & Location-Independent Computing*, pages 29–34, August 1993.

[131] Brian D. Noble, Mahadev Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[132] Hutchinson Norman C, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language. Technical Report 87-10-07, University of Washington, Seattle, WA (USA), 1987.

[133] Bill Nowicki. NFS: Network File System Protocol Specification. Technical Report RFC1094, Sun Microsystems, Inc., March 1989.

[134] Thor Olavsrud. MobileStar Cuts the Wires. *InternetNews*, 2001-08-17.

[135] Frank Olken, Hans-Arno Jacobsen, Chuck McParland, Mary Ann Piette, and Mary F. Anderson. Object Lessons Learned from a Distributed System for Remote Building Monitoring and Operation. In *Proceedings of the 1998 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 284–295, October 1998.

[136] Frank Pählke, Rainer Ruggaber, and Jochen Schiller. Providing Nomadic CORBA Servers by Integrating $\prod^2$ and FATIMA. In *Proceedings of European Wireless 2000*, pages 171–174, Dresden, Germany, September 2000. Verlag VDE.

[137] Vincent D. Park and M. Scott Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proceedings of the 3rd IEEE Conference on Computer Communications (INFOCOM'97)*, pages 1405–1413, 1997.

[138] Chales E. Perkins and Andrew F. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, pages 415–419, 1994.

[139] Charles E. Perkins. IP Encapsulation within IP. Technical Report RFC2003, IETF, May 1996.

[140] Charles E. Perkins. IP Mobility Support. Technical Report RFC2002, IETF, October 1996.

[141] Charles E. Perkins. Ad-hoc On-demand Distance Vector Routing. In *MILCOM '97 Panel on Ad Hoc Networks*, November 1997.

[142] Charles E. Perkins. Mobile IP. *IEEE Communications Magazine*, 35(5):84–99, May 1997.

[143] Charles E. Perkins. Mobile IP. *IEEE Communications Magazine*, 40(5):66–82, May 2002.

[144] Charles E. Perkins and David B. Johnson. Mobility Support in IPv6. In *Mobile Computing and Networking*, pages 27–37, 1996.

[145] Nokia Mobile Phones. M2M: New Opportunity in Wireless Data Business. White Paper, 2001.

[146] Nokia Mobile Phones. Nokia Card Phone 2.0 Specifications, 2002.

[147] Esmond Pitt and Kathy McNiff. *Java.RMI: The Remote Method Invocation Guide.* Addison Wesley, June 2001.

[148] Andreas Polze, Janek Schwarz, Kristopher Wehner, and Lui Sha. Integration of CORBA Services with a Dynamic Real-Time Architecture. In *Proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium (RTAS 2000)*, pages 198–206. IEEE, 2000.

[149] Jonathan B. Postel. User Datagram Protocol. Technical Report RFC768, IETF, August 1980.

[150] Jonathan B. Postel. Internet Control Message Protocol. Technical Report RFC792, IETF, September 1981.

[151] Jonathan B. Postel. Internet Protocol. Technical Report RFC791, IETF, September 1981.

[152] Jonathan B. Postel. Transmission Control Protocol. Technical Report RFC793, IETF, September 1981.

[153] Jonathan B. Postel. Simple Mail Transfer Protocol. Technical Report RFC821, IETF, August 1982.

[154] Arno Puder and Kay Römer. *MICO: An Open Source CORBA Implementation.* Morgan Kaufmann, March 2000.

[155] Ramachandran Ramjee, Kannan Varadhan, Luca Salgarelli, Sandra R. Thuel, Shie-Yuan Wang, and Thomas F. La Porta. HAWAII: a Domain-Based Approach for Supporting Mobility in Wide-Area Wireless Networks. *IEEE/ACM Transactions on Networking*, 10(3):396–410, June 2002.

[156] Rainer Ruggaber and Jochen Seitz. Using CORBA Applications in Nomadic Environments. In *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications*, pages 161–170, Los Alamitos, CA, USA, December 2000.

[157] Rainer Ruggaber and Jochen Seitz. A Transparent Network Handover for Nomadic CORBA Users. In *Proceedings of 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 499–506, Mesa, AZ, USA, April 2001.

[158] Mahadev Satyanarayanan. Accessing Information on Demand at Any Location: Mobile Information Access. *IEEE Personal Communications*, 3(1):26–33, February 1996.

[159] Mahadev Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Symposium on Principles of Distributed Computing*, 1996.

[160] Mahadev Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.

[161] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[162] Mahadev Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. Experience with Disconnected Operation in a Mobile Computing Environment. In *Proceedings USENIX Symposium on Mobile & Location-Independent Computing*, pages 11–28, 1993.

[163] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 2001.

[164] Alexander Schill and Sascha Kümmel. Design and Implementation of a Support Platform for Distributed Mobile Computing. *Distributed Systems Engineering Journal*, 2(3):128–141, 1995.

[165] Douglas C. Schmidt, David L. Levine, and Chris Cleeland. Architectures and Patterns for Developing High-performance, Real-time ORB Endsystems. *Advances in Computers*, 48, 1999.

[166] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4), 1998.

[167] Srinivasan Seshan, Hari Balakrishnan, and Randy H. Katz. Handoffs in Cellular Wireless Networks: The Daedalus Implementation and Experience. *Kluwer Journal on Wireless Personal Communications*, 1996.

[168] Dong-yun Shin and Ki-soo Chang. An Efficient Handoff Method to Support Real-Time Services in Mobile IP Environment. In Y. X. Zhong, S. Cui, and Y. Wang, editors, *Proceedings of the International Conferences on Info-tech and Info-net (ICII 2001)*, pages 615–620, Beijing, China, October–November 2001. IEEE.

[169] William Allen Simpson. The Point-to-Point Protocol (PPP). Technical Report RFC1661, IETF, July 1994.

[170] James Snell, Doug Tidwell, and Pavel Kulvhenko. *Programming Web Services with SOAP.* O'Reilly, Sebastopol, California 95472, 2002.

[171] Alex C. Snoeren and Hari Balakrishnan. An End-to-End Approach to Host Mobility. In *Proceedings of the 6th International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 155–166, 2000.

[172] Hesham Soliman, Claude Castelluccia, Karim El-Malki, and Ludovic Bellier. Hierarchical Mobile IPv6 Mobility Management (HMIPv6). Draft standard, IETF, October 2002.

[173] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software Architecture in Industrial Applications. In *International Conference on Software Engineering*, pages 196–207, 1995.

[174] Mark D. Spiteri and John Bates. An Architecture to Support Storage and Retrieval of Events. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE 1998)*, pages 443–458, London, 1998. Springer-Verlag.

[175] W. Richard Stevens. *Unix Network Programming.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1990.

[176] Aaron Striegel, Ranga S. Ramanujan, and Jordan Bonney. A Protocol Independent Internet Gateway for Ad-Hoc Wireless Networks. In *Proceedings of 26th IEEE Conference on Local Computer Networks (LCN 2001)*, Tampa, Florida, November 2001.

[177] Fumio Teraoka, Yasuhiko Yokore, and Mario Tokoro. A Network Architecture Providing Host Migration Transparency. In *ACM SIGCOMM '91 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 209–220, 1991.

[178] Susan Thomson and Thomas Narten. IPv6 Stateless Address Autoconfiguration. Draft Standard RFC2462, Internet Engineering Task Force, December 1998.

[179] Mark C. Torrance. Advances in Human-Computer Interaction: The Intelligent Room. In *Working Notes of the CHI 95 Research Symposium*, Denver, Colorado, May 1995.

[180] Vertel. Vertel Announces e*ORB, the First Carrier-grade Object Management Platform for Telecommunications, December 1999.

[181] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), 1997.

[182] Steve Vinoski. New Features for CORBA 3.0. *Communications of the ACM*, 41(10), October 1998.

[183] Thierry Violleau and Ray Ortigas. Supporting Disconnected Operation in Wireless Enterprise Applications. White Paper (Preliminary), June 2003.

[184] W3C. SOAP Version 1.2 Part 1: Messaging Framework, July 2002.

[185] W3C. SOAP Version 1.2 Part 2: Adjuncts, July 2002.

[186] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[187] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3):94–104, September 1991.

[188] Mark Weiser and John Seely Brown. Designing Calm Technology. *PowerGrid Journal*, 1(1), July 1996.

[189] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *1992 International Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), 1992. IEEE Computer Society Press.

[190] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *Proceedings of the 2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[191] Jiang Xie and Ian F. Akyildiz. A Distributed Dynamic Regional Location Management Scheme for Mobile IP. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1069–1078. IEEE, 2002.

[192] Lidong Zhou and Zygmunt J. Haas. Securing Ad Hoc Networks. *IEEE Network*, 13(6):24–30, 1999.