SOAP in a Mobile Environment

Liam Dolan B.Sc.

A dissertation submitted to the University of Dublin,
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

September 2004

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: ________________

Liam Dolan

13th September 2004

Permission to Lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed:_________________

Liam Dolan

13th September 2004

Acknowledgements

I would like to thank my supervisor Mr. Mads Haahr for his guidance and advice throughout this project.

Thanks to my colleagues in the NDS class of 2004 for making a hard year enjoyable. Thanks also to my friends especially Kathy for being particularly supportive.

And finally last but by no means least I would like to thank my parents for their encouragement and support throughout the year. Thanks also to the other members of my family.

Abstract

Recent years has seen a dramatic rise in the number of mobile devices in use. These devices include PDAs, mobile phones and embedded devices such as those found in cars for satellite tracking. As the use of these devices increases so do the expectations of what functionality is offered by the devices. Customers now demand to be able to access the web and check their emails while on the move and in possession of a mobile device. However computing in a wireless environment must overcome many obstacles, such as variable bandwidth and unreliable network connectivity, which are not present in a wired environment.

Middleware is a term used to describe the software layer that sits between distributed systems and allows them to communicate. It aims to solve problems associated with distributed systems such as heterogeneity and distribution. The Common Object Request Broker Architecture (CORBA) and Java Remote Method Invocation (RMI) are two examples of middleware architectures. These architectures were designed for distributed systems operating on wired networks and when deployed on a wireless network can encounter several problems, such as frequent loss of network connectivity, which can seriously affect their operation.

The Architecture for Location Independent Computing Environments (ALICE) is an architecture developed by the DSG research group in Trinity College Dublin that allows mobility support to be added to any object-oriented middleware framework that satisfies a minimal set of requirements. This dissertation presents an instantiation of ALICE for the Simple Object Access Protocol (SOAP). SOAP is a lightweight communications protocol based on XML. SOAP will be used as the remote invocation protocol and will be combined with Java to provide an implementation of the abstract components that make up the ALICE Swizzling and Disconnected Operation Layers.

Contents

List of Figures

List of Tables

1 Introduction

As the level of computational power available on mobile devices (e.g. mobile phones and PDA's) increase, the amount of functionality provided by such devices is also increasing. This increased computational power combined with the increase in wireless networking capabilities provided by technologies such as Bluetooth and IEEE 802.11 has lead to a huge increase in the demand and use of such devices. This increase in usage of mobile devices has lead to a requirement for these devices to be connected to a network and to be capable of functionality that would previously have only been provided by desktop PC's. Such functionality ranges from email assess to online banking.

Middleware is the term used to describe the software layer that sits between distributed systems and allows them to communicate. It aims to solve issues like network heterogeneity and distribution. It also allows remote objects to perform remote procedure calls on each other. The Common Object Request Broker (CORBA) [2] and Java's Remote Method Invocation (RMI) [3] are examples of middleware architectures. CORBA allows remote objects developed using different programming language's to communicate while RMI allows distribute Java objects to invoke methods on each other. These two middleware architectures were not developed to operate in a mobile environment where network connections are frequently broken. When used in a mobile environment such architecture's can become unusable because of issues with server references changing frequently and network connections being broken for long periods of time.

The Architecture for Location Independent Computing Environments (ALICE) [1] was developed by the Distributed Systems Group (DSG) at Trinity College Dublin (TCD) to allow mobility support to be added to any object oriented middleware framework that satisfies a minimal set of requirements. ALICE is a layered architecture consisting of a number of abstract modular components that when instantiated for any given middleware framework will allow mobility support to be added to that framework. ALICE has already been instantiated for CORBA [4] and for RMI [5].

The Simple Object Access Protocol [6] is a communication protocol standardized by the W3C. It is based on the XML specification [10] and a SOAP message is basically a set of XML tags. SOAP is an XML based framework that enables communication between distributed objects. SOAP is not bound to any single programming language and can be used as a communication protocol by programming languages to carry out remote procedure calls. SOAP over comes many of the problems associated with other middleware frameworks by being based on open standards. The aim of this project is to provide an instantiation of ALICE for SOAP.

1.1 Mobile Computing

Mobile computing is a special case of distributed computing where some or all of the components of a system are mobile and their state of connectivity often changes. Examples of mobile computers are laptops, PDA's, and as their computational power continues to increase mobile phones. Mobile devises can connect to a network either by wireless or wired means.

The main advantage of mobile computing is the fact that people using the mobile computers are not bound to a specific location and can carrying out their work while on the move and away from the office. This is especially advantageous to people who previously would have had to leave their office to carry out their work and later return to the office to connected to their companies system and input data. An example would be a sale representative who had to leave his office to make sales and later return to update the companies system. With mobile computing the sales rep. could input his sales while away from the office by using a mobile computer to remotely connect to his company's network.

Mobile computing has obvious advantages. However to operate successfully mobile computers and the applications that run on them must overcome a number of obstacles. The mobile devices themselves are expected to be light and easily movable. This leads to the device having to carry out its functionality without a lot of the resources available to desktops computers such as a reduced user interface, battery power and data storage capacity. Connecting to a network remotely over a wireless network, while very convenient for the user of the device, leads to a range of obstacles that must be overcome. The most obvious obstacles to wireless connectivity are limited and variable bandwidth.

Middleware can be used to connect distributed systems and allow them to communicate. Like on wired networks, middleware can be used to allow a system operating on a wireless network to communicate with another system on the wireless network or a system on a wired network. However deploying a middleware framework that was designed to operate on a wired network on to a wireless network can lead to a lot of problems and it is these problems that ALICE was designed to overcome.

1.2 ALICE

The ALICE architecture was developed to allow mobility support to be added to any object-oriented middleware that satisfies a minimum set of requirements. ALICE provides mobility support in the form of a set of modular, reusable components and its architecture consists of a number of abstract components. An instantiation of ALICE for a given object-oriented middleware involves the creation of actual software components for some (or all) of the abstract components

ALICE consist of five layers, four of which are considered to be part of the Architecture and one that is considered to be part of the middleware framework for which ALICE is being instantiated.

The **Transport Layer** is responsible for the mode of communication used between two hosts. It consists of a number of transport modules each of which correspond to an actual communication interface e.g. Bluetooth, IEEE 802.11.

The **Mobility Layer** manages the state of connectivity between a mobile host and a mobility gateway. It also manages the transport modules and is responsible for transferring state information about a mobile host from one mobility gateway to another when the mobile host moves between gateways.

The **Swizzling Layer** is responsible for client redirection and server reference translation at key points in time.

The **Disconnected Operation Layer** allows clients to cache objects on the client-side so clients can continue operating during times of disconnection. It also provides functionality that allows the client to return replicas to the server where they are reconciled with the authoritative object on the server.

1.3 SOAP

SOAP is a lightweight communication protocol that allows heterogeneous systems to communicate. It is based on the XML protocol and is standardized by the W3C. SOAP is not intended for any specific style of distributed computing, but it can be used very easily for remote procedure calls (RPC). This is to be expected considering that one of the design goals of SOAP according its specification was to provide a convention for remote procedure calls and responses.

SOAP on its own is not a complete middleware framework as it provides no functionality for procedures such as reference dissemination or garbage collection. However SOAP can be used as the remote invocation protocol (RIP) around which a middleware architecture can be built. The real power of SOAP is exposed when it is combined with an application programming language such as Java or C++ and it is the language with which SOAP is combined with that can be used to complete the middleware architecture.

It is important to understand that even though SOAP stands for Simple Object Access Protocol it actually has no concept of what an object is. To SOAP everything is data encoded in XML. However when SOAP is combined with a SOAP parser objects specific to an application programming language can be transferred from one system to another using SOAP. The objects are serialized into a SOAP messages before they are sent to their destination and on arrival at their destination the objects are de-serialized back into their native format. It should be noted that it is the state of the object that is serialized not the actual behavior. SOAP also does not understand the concept of object references or server references. The only references used in SOAP are internal to SOAP messages and are done through the use of id and href attributes.

In the context of ALICE we are concerned with the use of SOAP as a remote invocation protocol. To provide an instantiation of ALICE for SOAP, SOAP itself will be used as the remote invocation protocol and will be combined with a programming language. The chosen programming language will provide extra functionality normally associated with middleware that SOAP cannot provide such as format of server references and server reference dissemination.

1.4 Project Goal

The main goal of this project is to provide an instantiation of ALICE for SOAP and by doing so, build on the "Supporting Mobile Computing in Object-Oriented Middleware Architectures" thesis undertaken by Mr. Mads Haahr [1]. The process of instantiating ALICE involves the creation of actual software components for some of the ALICE layers. The result of an instantiation of ALICE for SOAP will be to show how SOAP can be a used as a remote invocation protocol in a mobile environment.

The main focus of this instantiation is on the Swizzling and Disconnected Operation Layers of the ALICE architecture. The aim is to provide a complete instantiation for both of these layers and once completed to integrate the layers with the existing Mobility Layer of the ALICE architecture.

As mentioned previously this project will build on Mr. Mads Haahr thesis and aims to investigate further some of the issues outlined in the future work section of that thesis. In particular focus will be placed on the possibility of construction multi-endpoint SOAP server references and to investigate to what extent SOAP facilities object mobility.

1.5 Project Achievements

The majority of the goals outlined for this project have been achieved. A project has been designed and built that provides an implementation of the Swizzling Layer and an implementation of the Disconnected Operation Layer using SOAP. The project was implemented using SOAP as the communication framework. Functionality required for an instantiation of ALICE that could not be provided by SOAP was implemented using Java.

The process for integrating the Swizzling and Disconnected Operation Layers with the existing Mobility Layer has been designed. The implementation of the design purposed for integration with the Mobility Layer is a source of future work.

1.6 Dissertation Roadmap

The remaining chapters of this dissertation are laid out as follows:

Chapter 2 – Background

This chapter gives a background to the project by providing an overview of ALICE, SOAP and J2ME. The ALICE requirements are discussed and the challenges facing devices operating in a mobile environment are also outlined.

Chapter 3 - State of the Art

This chapter reviews a number of toolkits that use SOAP for their communication protocol and are designed to operate on mobile devices. The toolkits are reviewed in terms of the fourteen mobility challenges outlined in chapter 2.

Chapter 4 – Design

This chapter details the design of the SOAP Swizzling and Disconnected Operation Layers. The design outlined resembles the design of ALICE as closely as was possible. A design for the process of integrating the Swizzling and Disconnected Operation Layer with the existing Mobility Layer is also outlined.

Chapter 5 – Implementation

This chapter describes in detail how the project was implemented. The implementation of the two Swizzling Layer components and the two Disconnected Operation Layer components is described. Also it was necessary to implement a SOAP client, server and registry architecture that could be used as the basis for the instantiation of ALICE for SOAP. The implementation of this architecture is also described.

Chapter 6 – Evaluation

An evaluation of the project is presented in this chapter. The Swizzling Layer is evaluated in terms of performance, code size and transparency. The Disconnected Operation Layer is evaluated in terms of performance. A performance evaluation of both layers was carried using a number of different configurations which are also presented in this chapter.

Chapter 7 – Conclusion

This chapter concludes the dissertation. In this chapter the completed work and the work yet to be completed are summarized. Possible future work on the project is also presented.

2 Background

2.1 Introduction

This chapter provides a background to the project through a brief introduction to the Architecture for Location-Independent Computing Environments (ALICE), the Java 2 Micro Edition (J2ME) and the Simple Object Transfer Protocol (SOAP). The requirements ALICE places on the remote invocation protocol and server references used by the middleware architecture for which it is being instantiated are also outlined. The challenges of computing in a heterogeneous mobile environment are first discussed.

2.2 Challenges of Mobile Environments

Devices that operate in a mobile environment are significantly more constrained than those that operate in a fixed network. This is due to the characteristics of a mobile environment such as variable bandwidth and connectivity. The devices themselves also have more constraints due to the fact that they are expected to be mobile and therefore must be much lighter and smaller than a fixed device such a desktop computer. Mads Haahr [1] outlined fourteen challenges to computing in a mobile environment. These challenges are divided into three related groups: mobile devices, mobile networking and physical mobility. The challenges as outlined in [1] are summarized here:

2.2.1 Mobile Devices

Mobile devices are generally described as those ranging from mobile phones, PDA's and laptops to embedded computers such as those in household appliances and cars. These devices are intended to be carried easily by the user and are therefore limited in resources compared to stationary computers. Constrains placed on mobile devices are battery power, data risks, user interface, storage capacity and processing power. These constraints are described below.

Battery Power

Mobile devices have limited battery power because they cannot be continuously connected to a power supply and also because of the limited size of the battery that can be carried inside the device. It is therefore in the best interest of applications running on the device to manage the available power as efficiently as possible. Wireless communication requires a significant amount of power and therefore any reduction in communication will increase the operating time of the device. Reduction in communication can be achieved through efficient data formats and compression. Voluntary disconnection, where a device disconnects itself from the network when it does not require a connection, can also be used to reduce power consumption.

Data Risks

The small size of mobile devices makes them vulnerable to being lost, stolen or damaged. This also leads to the data on the device being vulnerable. If the device is lost or stolen all the data stored on the device may be lost. The effect of lost data can be addressed by encryption, minimizing the amount of data stored on the device and the replication of data across distributed machines.

User Interface

Due to the size reduction of mobile devices the space available for a user interface is limited. This involves a reduced display screen and a reduced keyboard. To overcome these limitations most PDA's allow input through the use of pens rather than a mouse. Other solutions are speech recognition, gesture recognition and head mounted displays.

Storage Capacity

The limited physical size and battery power of mobile devices limits the amount of storage space available. Techniques used to overcome limited storage space are file compression, accessing remote storage, sharing code libraries and the use of scripting languages. The reduced storage space also has an effect on the applications running on the device. All applications must attempt to minimize storage requirements and the footprint needed to run the application.

Processing Power

Processing power of mobile devices is greatly reduced by the limitation in power supply. The effect this has on applications running on the device is that applications must attempt to be as lightweight in terms of processing power as possible.

2.2.2 Mobile Networking

Due to the nature of mobile environments wired network connectivity is not possible most of the time. Therefore to maintain continuous network connectivity mobile devices require a wireless connection to the network. To communicate successfully in a wireless network a number of obstacles have to be overcome that are not present in a wired network. The obstacles that have to be dealt with are network heterogeneity, disconnection, low bandwidth, bandwidth variability, security risks and usage cost. These issues are described below.

Network Heterogeneity

Mobile devices normally have the ability to move between networks. This movement may involve changing networking protocols or a change in transmission speeds. Reasons for changing networks could be moving out of the range of one transceiver and into the range of another or being within the range of two transceivers at the same time. In general a mobile device will have a number of options with which to connect to a network. For distributed applications that rely on the network for connectivity this heterogeneity can pose significant challenges

Disconnection

Disconnection can be caused by network failure or devices moving out of the range of connection points. Devices may also voluntarily disconnect from the network, possibly to cut down on power consumption. Mobile devices due to their wireless connectivity are much more susceptible to disconnection than fixed devices. Applications have to decide if it is better to spend more resources to avoid disconnection or to try and cope with disconnection gracefully. Disconnection can be partially dealt with by caching data and once connection is resumed using conflict detection and resolution approaches to update data.

Low Bandwidth

Mobile devices are constrained to bandwidth with approximately half the data rates of stationary devices. Due to the limited bandwidth, applications running on mobile devices must attempt to use as little bandwidth as possible. Techniques such as compression, buffering, delayed write back and pre-fetching data can be used to reduce communication and therefore limit the amount of bandwidth required.

Bandwidth Variability

Wireless devices have to deal with much greater bandwidth variability than do stationary devices. Bandwidth variability can be caused by number of factors such as devices moving out of the range of transceivers or interference caused by the local environment. Applications can make few assumptions about available bandwidth and may choose to deal with this in one of two ways. The first method is to assume low bandwidth variability and operate on this basis. The second option is to maintain awareness of bandwidth availability and to use what ever is available at any given point in time.

Security Risks

Due to the fact that it is relatively easy to gain a connection to a wireless network and to the public nature of the medium wireless networks use to communicate it is much more difficult to ensure the security of a wireless network. Security can be improved with the use of encryption and enforcing authentication using a public/private key encryption scheme. For applications to ensure security and to employ mechanisms such as encryption, an extra overhead is place on the resources of the device. Designers of applications have to decide if it's worth the cost and if it's necessary for the application being designed

Usage Cost

Wireless devices may be charged for connection to a network. Methods of charging are subscription based fees, billing per unit transmitted or a combination of both. Applications should take usage costs into consideration when deciding the level of communication it requires. Methods similar to those used to cut down on power consumption can be used to decrease usage costs.

2.2.3 Physical Mobility

In a mobile environment where devices can freely move from one connection point to another, the network address of roaming devices can also change frequently. This dynamic changing of network address introduces issues such as address migration, location-dependent information and migrating locality. These issues are outlined below.

Address Migration

The problem of address migration occurs when a mobile device moves between access points to a network and the network address for that device also changes. Routing protocols used by wired networks are generally not designed to deal with dynamically changing network addresses. When a device moves to a new location this usually means having to acquire a new network address which leads to the problem of keeping track of a mobile device's current address. There are a number of mechanisms for handling changing addresses, such as selective broadcast where a message is sent to all known devices asking the required device to respond, central location service where a well known central server maintains a list of current addresses, home bases where each device has its own static server that maintains its current address, and forwarding pointers where the mobile device leaves a pointer to its next address when it leaves one location and moves to another. Extra functionality has to be added to applications to handle dynamic addressing issues.

Location-Dependent Information

Location-dependent information is information that changes depending on the location of a mobile device such as the name of the nearest server, printer or gateway. This information needs to be determined dynamically by mobile devices as they move around. Applications running on mobile devices have to implement a discovery mechanism to overcome this challenge or work with middleware capable of dynamically discovering locality specific information.

Migrating Locality

Even if a mobile host is able to discover local information, that information might change due to the host moving around. This challenge can be addressed by dynamically changing connections to servers as they become closer. Like location-dependent information an application will either have to provide extra functionality or work with middleware to overcome this problem.

2.3 The Architecture for Location-Independent Computing Environments (ALICE)

ALICE is an architecture that allows 'mobility support to be added to any object-oriented middleware framework that supports a set of minimal requirements' [1]. ALICE is composed of a set of modular, reusable components that can be used to instantiate the architecture for different object-oriented middleware frameworks. The ALICE stack consists of five layers in total, four of which are part of the ALICE architecture and one which is considered to be part of the middleware framework for which ALICE is being instantiated. The four ALICE layers are summarized in the following sections but first the mobile environment for which ALICE was developed is outlined.

The model for communication for which ALICE is based on is the gateway model. In the gateway model the wireless network is situated at the fringes of the wired network. Mobile devices gain access to the network through gateways placed at the borders of the wired network. To communicate with other devices on the network a mobile device must be within range of a mobility gateway. The mobility gateway acts as a proxy for the mobile device and relays communication to the intended target from the mobile device. Mobile devices can move between gateways and usually a procedure called 'handoff' allows information about a device to be transferred from one gateway to another. Figure 1 illustrates the gateway mobility model.

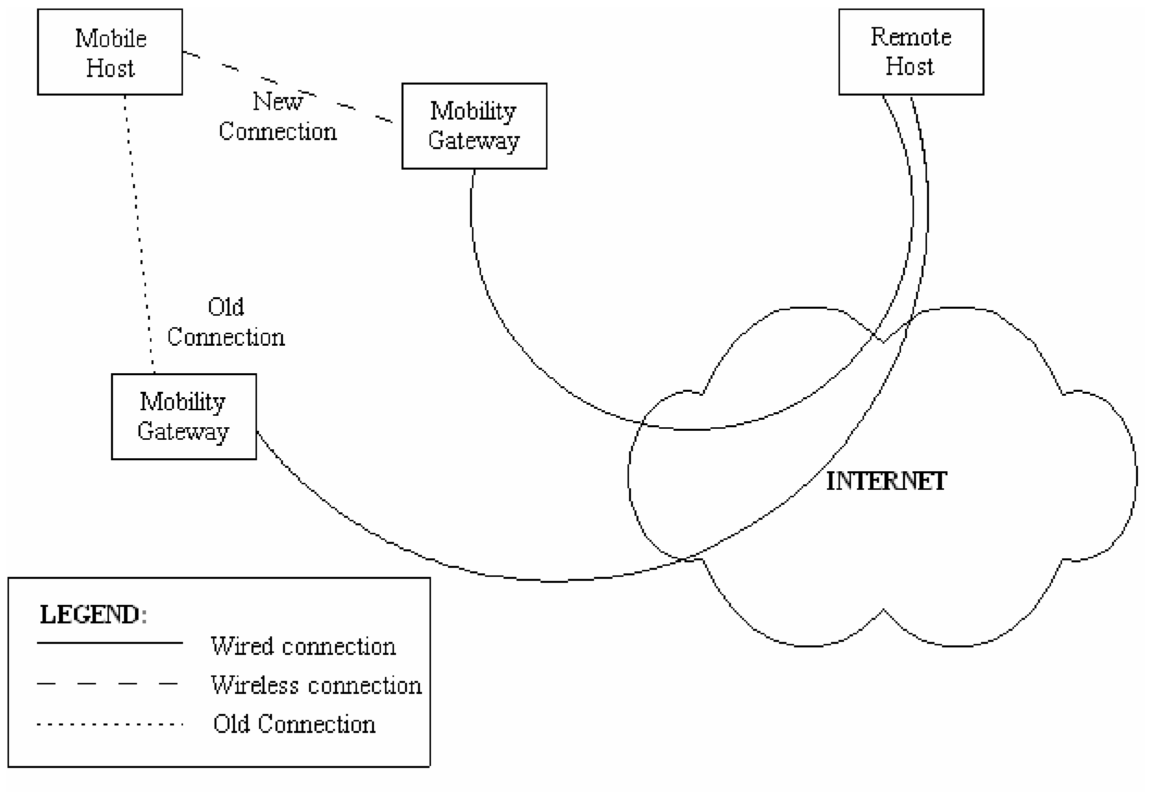


Figure 1: The Gateway Mobility Model

The terminology used in the above diagram is as follows:

Mobile Host (MH): is a mobile computing device that can connect to a mobility gateway either by wired or wireless means. A MH can have either a server or a client running on it. A MH can move between gateways.

Mobility Gateway (MG): is a computer that has continuous access to the wired network and to which a MH can connect. It is assumed that the network address for a MG changes very rarely.

Remote Host (RH): is a computer that communicates with a MH through a MG. A RH can have either a client or a server running on it and can be either a mobile or a fixed host.

2.3.1 Architecture Overview

As mentioned in the previous section the ALICE architecture consists of a number of layers each containing modular and reusable components. Figure 2 shows the entire ALICE architecture.

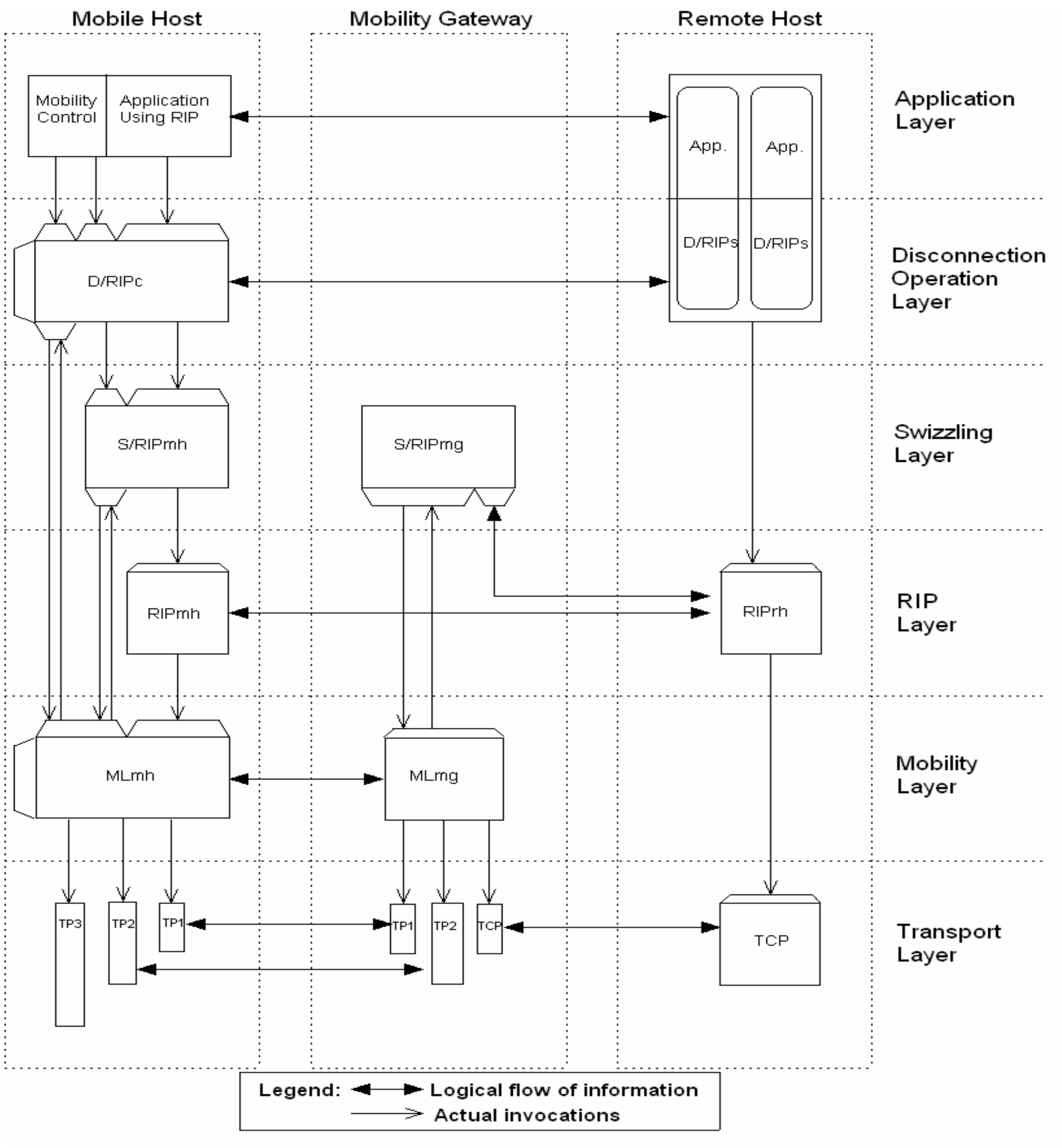


Figure 2: The ALICE Architecture

The purpose and functionality of each layer is as follows:

The Transport Layer

The Transport Layer is the lowest layer in the architecture and consists of a number of transport modules. Each module corresponds to a communication interface such as Bluetooth or InfaRed. The purpose of the transport modules is to encapsulate different communication interfaces and to provide a common programming interface to the Mobility Layer. The Mobility Layer manages the transport modules and depends on them for communication. At least one transport module is required in order for the mobility layer to be capable of working.

The Mobility Layer

The Mobility Layer (ML) is responsible for managing connectivity between the MG and the MH. The ML has three functions: transport management, proxy operation and handoff/tunneling. Transport management involves the management of the Transport Layer’s transport modules and using these to connect to MGs. Transport management also involves transparently trying to reconnect to MGs if connections are lost or broken. Proxy operations involve allowing the MG to act as proxy for any MHs that are connected to it. Any information sent to or from the MH will be relayed to the MG while the MG is acting as a proxy for the MH. Handoff and tunneling involves transferring information about the state of a MH from one MG to another when a MH moves between gateways.

The functions mentioned above are all transparent to the other layers in the architecture. The ML also provides additional information to other layers where knowledge of the state of connectivity is required. This information is delivered in the form of a callback from the ML to the other layers.

The Remote Invocation Protocol Layer

The Remote Invocation Protocol (RIP) Layer sits on top of the ML. The RIP Layer is an implementation of the remote invocation protocol used by the object oriented middleware for which ALICE is being instantiated. In the case of this project the remote invocation protocol is SOAP and the RIP Layer is known as the SOAP layer. The RIP Layer consists of two separate components, one on the MH and one on the RH. The RIP layer is not considered to be part of the ALICE architecture but rather a standalone layer.

The Swizzling Layer

The Swizzling Layer is responsible for the translation of server references at certain points in time and the redirection of clients to more recent server locations. Server translation and redirection is done transparently to the client and server. The Swizzling Layer needs to be aware of the state of connectivity at any point in time and depends on the ML for this information. The Swizzling Layer solves one instance of the address migration problem where the client holds a reference to a server but the server is no longer located at the address specified in that reference.

The Disconnected Operation Layer

The Disconnected Operation Layer allows server objects to be cached on the client-side. During periods of disconnection the Disconnected Operation Layer allows server-side functionality to be carried out on the client-side. By allowing objects to be replicated and cached on the client the Disconnection Operation Layer solves the problem of disconnection. Once a connection is re-established to the server objects can be sent back to the server where they are reconciled with the original objects. Like the Swizzling Layer the disconnected operation layer also needs to be aware of the state of connectivity at any point in time and depends on the ML for this information.

2.3.2 ALICE Requirements

To allow implementations of the same middleware architectures to interoperate each architecture specifies a protocol it uses to carry out remote invocations. Some of the ALICE layers need to interact with the remote invocation protocol and translate server references at certain points in time. In order for ALICE to be instantiated for a given middleware architecture, the remote invocation protocol and the format of server references used by that architecture must satisfy a set of requirements. There are six requirements in total and they are outlined here:

Requirement R1: Client/Server

The ALICE Mobility Layer works by providing an alternative to Berkeley Sockets. Berkley Sockets are probably the most popular programming interface for network programming and it is therefore preferable for ALICE to be interoperable with them.
The Berkley Sockets programming model is based that the client/server abstraction. In order for ALICE to retain compatibility with Berkley Sockets, the Mobility Layer requires the remote invocation protocol used by the middleware ALICE is being instantiated for to be client/server-oriented.

Requirement R2: TCP/IP

The ALICE Mobility Layer was implemented to consider only connection-based communication and protocols based on the Berkley Socket API. While it would be possible to implement support for connectionless protocols such as UDP it was avoided to simplify the design of the Mobility Layer. TCP offers a connection-based, reliable service and it is for this reason that the ALICE Mobility Layer was implemented to accept communication transmitted via TCP. Therefore it is required that the remote invocation protocol used by the middleware for which ALICE is being instantiated to be capable of transmitting data via TCP.

Requirement R3: Redirection

In order to support servers on mobile hosts the address migration problem must be solved. This can be done as mention in section 2.2.3 by four approaches, selective broadcast, central services, home bases and forwarding pointers. These four approaches are generally implemented using one of two approaches: explicit lookup or redirection. With explicit lookup the client holds an identifier for the server which does not specify the server's location. Using the identifier the client can query a central service for the actual reference to the server. With redirection the client holds a reference that contains a possible location for the server. The client tries to invoke the server using the reference and if the server is no longer at the location specified by the reference, the client is redirected to the server's new location.

To solve the address migration problem ALICE adopted an approach based on the four approaches mentioned above. The adopted approach is a hybrid scheme based on forwarding pointers and a home agent. To be capable of implementing the adopted approach it is required that the middleware architecture for which ALICE is being instantiated to support redirection of clients.

Requirement R4: Multi-Endpoint References

The hybrid scheme mentioned above that is used by ALICE requires the references held by clients to contain an address to a location from which the server may already have moved and an address to a fallback home agent. Clients try these addresses or endpoints in sequence. For example the client may try to invoke the server using the endpoint it holds that is a reference to the server. If for some reason the client cannot locate the server at that endpoint, the client will then try to locate the server using the endpoint that is a reference to the home agent. For this scheme to be implemented the middleware architecture must support server references that can contain multiple endpoints.

Requirement R5: Extra Information

The approach adopted by ALICE to solve the address migration problem requires redirection entities (MGs in the case of ALICE) to receive invocations on behalf of servers. If a server is currently attached to a MG when an invocation is received by the MG for the server, the MG will forward the invocation on to the server. If the server is no longer attached to the MG, the MG will return a redirection response with the new location of the server to the client. Considering that there may be more than one server attached to a MG, the MG must have sufficient information to understand what server a client is actually wishing to invoke. The question that arises is how will the MG acquire this extra information? The solution adopted by ALICE to require the client to supply the extra information during invocation. For this solution to be implemented ALICE requires that it be possible to embed extra information in a server reference and for this information to be passed from the client to the server during invocation.

Requirement R6: Object Mobility

ALICE supports long term disconnection between the client and the server. This is done with the use of a reconciliation step where the modified server state on the client side is transferred back to and reconciled with the server once a connection is re-established between the client and the server. This scheme requires that it be possible to replicate objects and for it also to be possible to transfer replicated objects between the client and the server. To support this scheme ALICE requires the middleware architecture for which ALICE is being instantiated to support object mobility.

2.4 Java 2 Micro Edition (J2ME)

The J2ME is a Java edition developed by Sun Microsystems [11] [12] to allow Java applications to be developed on resource constrained mobile devices, such a mobile phones and PDA's. J2ME is a cut down version of the Java standard edition capable of providing

Java functionality using limited resources. Some of the resource constraints which J2ME takes into consideration are:

- Limited memory
- Limited processing power
- Restricted User interface
- Low power availability
- Typically wireless network connectivity with low bandwidth.

Obviously the constraints on different devices will vary for example the user interface on a PDA would not be as constrained as the interface on a mobile phone. Realising that not all mobile devices have the same resource constraints, Sun introduced a number of configurations capable of running on devices with different constrains. One such configuration is the Connected Limited Device Configuration (CLDC).

2.4.1 Connected Limited Device Configuration (CLDC)

CLDC defines a set of application programming interfaces (APIs) and a virtual machine for resource-constrained devices like mobile phones, pagers, and PDA's. The APIs provided by the CLDC configuration are a cut down version of some of the APIs available in the J2SE edition. The virtual machine provided by CLDC is called the KVM. The KVM (also known as the k virtual machine) is a compact Java virtual machine intended for small, resource constrained devices.

The CLDC places a minimum set of requirements on devices on which it can be deployed on to. They are:

- 128 kilobytes of memory for running the JVM and the CLDC libraries.
- 32 kilobytes of memory available during application runtime for allocation of objects.

2.4.2 The Generic Connection Framework (GFC)

As the devices on which J2ME applications run on are very limited memory wise a lot of the functionality provided by the J2SE APIs had to be cut down for J2ME. This is the case with the java.io and java.net packages. There is a cut down version of the java.io package available but the java.net package was replaced by the Generic Connection Framework (GFC). The GFC defines a number of interfaces that provide the foundation for all network connections made by CLDC applications. The framework also consists of one class called the Connector that can create any type of connection e.g. file, http, socket etc. The Connector's open method is responsible for opening the connections and has the following form:

Connector.Open("protocol:address;parameters");

The protocol specified in the above statement is resolved at runtime. The Connector looks for the appropriate class that implements the requested protocol using a *Class.forName()* call.

The GFC provides no actual protocol implementations instead it is left up to vendors to provide their own implementations. This is very convenient for programmers as they can programme to the interfaces of the GFC and can be guaranteed that their applications will run on a range of different devices once the protocols are implemented by the different vendors.

2.5 The Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (SOAP) [6] [9] originates from the XML-RPC specification [13] which used XML encoding to make remote procedure calls. The W3C has standardized SOAP and released the SOAP 1.1 specification [6] in May 2000. The specification for SOAP 1.2 [7] [8] was released in June 2003. SOAP is XML, that is SOAP is an implementation of the XML specification and relies heavily on the XML Schema and Namespaces standards for its definition.

According to the W3C SOAP 1.2 specification ‘SOAP is a lightweight protocol intended for the exchange of structured information in a decentralized, distributed environment’ [7]. It is a lightweight communication protocol based on XML. SOAP also defines a binding to the Hypertext Transfer Protocol (HTTP) and the Simple Mail Transfer Protocol (SMTP). The bindings allow SOAP messages to be transported from one endpoint to another using the protocols to which SOAP is bound.

Even though SOAP messages are fundamentally one way communication between sender and receiver, they can also be used to implement the request/response message exchange pattern. A SOAP message consists of three parts, a mandatory SOAP Envelope, an optional SOAP header and a mandatory SOAP body.

SOAP Envelope

The SOAP envelope is specified in SOAP 1.1 as ‘The Envelope is the top element of the XML document representing the message’ [6]. The envelope is the topmost element in the SOAP message within which is the rest of the SOAP message. The envelope element must appear in the SOAP message for it to be valid and all other elements in the SOAP message must appear as sub-elements of the envelope element.

SOAP Header

The header element is an optional element that can appear as a direct sub-element of the envelope element. The header element is specified as ‘The Header is a generic mechanism for adding features to a SOAP message in a decentralized manner without prior agreement between the communicating parties’ [6]. The header element can be used to contain elements used for transaction, authentication or other information that should not be contained in the body element of the message. Attributes that might be contained in the header element are the actor attribute and the mustUnderstand attribute.

SOAP Body

The body element is a mandatory element that must be in a SOAP message. If there is a header element then the body element should directly follow it, otherwise it should be the first sub-element of the envelope element. The body element is specified as ‘The Body is a container for mandatory information intended for the ultimate recipient of the message’ [6]. The body contains the SOAP message request or response details. It could also contain one way message details or fault details.

2.5.1 SOAP Data Types

The data types supported by SOAP are the data types defined by the “XML Schema Part 2: Datatypes” [19]. Data types can be either simple types or compound types. Examples of simple types include int, float and string. Compound types are derived from simple types and include structs and arrays. The difference between a struct and an array is how the elements that they contain are differentiated.

2.5.2 SOAP Toolkits

A SOAP toolkit is a collection of software component designed specifically to provide SOAP functionality to a specific programming language. For example a Java SOAP toolkit is a collection of Java libraries designed to provide a mapping between SOAP and Java. Currently there is a SOAP toolkit available for nearly every programming language. These toolkits are implemented using what ever language they are intended to provide extra functionality for. The functionality the toolkits provide is normally some form of transparent mapping between the language and SOAP messages. The toolkits typically consist of all or a combination of a SOAP engine, a SOAP client and a SOAP server. This section describes the three possible components of a toolkit, the functionality they provide and how they interoperate.

SOAP Engine

A SOAP engine (also know as a SOAP parser or SOAP Proxy) is a software component that is designed to aid both the consumers and providers of SOAP messages to accomplish their task by hiding the intricacies of SOAP messagie handling or more specifically XML parsing. A SOAP engine is generally built on a generic XML parser with special type-mapping and text data-marshalling mechanisms specific to SOAP. A SOAP engine understands the data-type information in a SOAP messages and implicitly converts the SOAP message to data objects familiar to the application language the engine is implemented for e.g. JAVA or C++. The value of the engine to programmers is that it provides transparency between the programming language and a SOAP message. An application can pass objects to the engine which automatically parses them to a SOAP message. The application then sends the message and waits for the response. Once a response is received the application passes the response message to the engine which will return an application understandable object. Most programming languages have SOAP engines implemented for them that are capable of translating between SOAP messages and the language's native data types.

SOAP server

A SOAP server consists of three components: a listener which receives messages, a proxy (or SOAP engine) which takes a message and translates it into an action to be carried out and application code to carry out that action. The proxy component gets passed messages from the listener. Once the proxy receives a message it must carry out three functions:

1.) De-serialize the message from XML to some native format suitable for passing to the application code.

2.) Invoke the application code

3.) Serialize the response if there is one back into XML and pass it to the listener.

The listener will then return the response to the client if a response is required.

SOAP Client

A SOAP client is similar to a SOAP server except it does not have a listener waiting to receive messages. The client consists of proxy and application code. Like the Server the proxy component of the client is responsible for serializing and de-serializing SOAP messages. The functions of the client are to:

1.) Open up a connection to the SOAP server.
2.) Assemble the appropriate SOAP request using the proxy to invoke a remote method and send it to the server.
3.) If a response is expected the client uses the proxy component to de-serialize the SOAP response to the native format of the application.
4.) Close the connection to the server.

The interaction between a SOAP client and SOAP server is illustrated in the Figure 3. Figure 3 shows the interaction between a client and a server in a RPC messaging model. For a one way messaging model the client would not expect or receive a response.

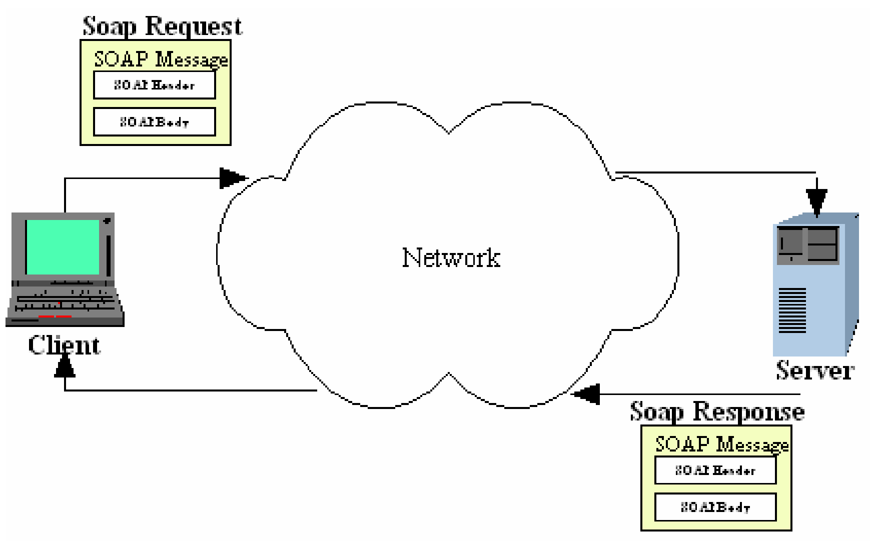


Figure 3: Interaction between a SOAP Client and Server

2.6 Summary

This chapter provided a background to the project by introducing the technologies that are involved in its development and by giving an overview of ALICE. The difficulties faced by computing in a mobile environment were discussed followed by a high level overview of the ALICE architecture as well as the requirements that ALICE places on the middleware for which it is being instantiated. The two main technologies that will be used for this instantiation of ALICE, SOAP and J2ME, were also reviewed. It is expected that any instantiation of ALICE would be reasonably lightweight and capable of running on small memory constrained mobile devices. It is for this reason that J2ME was chosen as the development environment.

3 State of the Art

3.1 Introduction

This chapter reviews three toolkits that use SOAP as their communication protocol and are also designed to operate on mobile devices. The systems are reviewed in terms of the fourteen mobility challenges discussed in the section 2.2. The three toolkits reviewed are IBM's Web Services Toolkit for Mobile Devices, embedded SOAP and Wingfoot's Mobile Web Services Platform. Once reviewed the three toolkits are compared in terms of the mobility challenges addressed.

3.2 Web Services Toolkit for Mobile Devices (WSTKMD)

The Web Services Toolkit for Mobile Devices (WSTKMD) [15] is a toolkit developed by IBM that provides tools and run-time environments that allows the development of applications that use Web Services. The applications developed using the WSTKMD are intended to run on small mobile devices. It consists of a Web Service stack and set of plugins for IBM's WebSphere Studio Device Developer (WSDD). The toolkit provides a runtime environment for both Java and C/C++ based Web Services on memory constrained devices

The WSTKMD uses IBM's own implementation of JSR 172 to support Java-based web services. The JSR 172 run-time environment has been optimized to run on small mobile devices. Previous versions of the toolkit used kSOAP to support Java-based services. However due to the introduction of JSR 172, support for kSOAP is not provided in the latest version. An implementation of JSR 172 can be used to replace the XML parsing capabilities of kSOAP. Java based web services are supported on Blackberry, PocketPC and Palm OS4 platforms. gSOAP is used to support C/C++ based web services. C/C++ based Web Services can be deployed on Palm and Symbian devices.

The Toolkit consists of three components: the Device Web Services Framework, the Pluggable Discovery Framework and the Web Services Security Mechanism.

Device Web Services Framework

The Device Web Services Framework is comprised of a device process and a gateway process. Like ALICE it is based on the gateway model for communication. The gateway process has similar functionality to the MG in ALICE and the device process is similar to the MHs outlined in section 2.3. The gateway process contains a list of devices and web services that can be accessed by clients. The location information for the devices and a description of the services they offer are stored in a database maintained by the gateway.
The gateway accepts both SOAP and SOAP over HTTP requests from clients. The devices accept either SOAP over HTTP or push requests, such as SMS messages, from the gateway, and can make HTTP requests to the gateway. The architecture for the Device Web Services Framework can be seen in Figure 4 which is taken from the documentation that is packaged with the WSTKMD toolkit and is available for download from the IBM website.

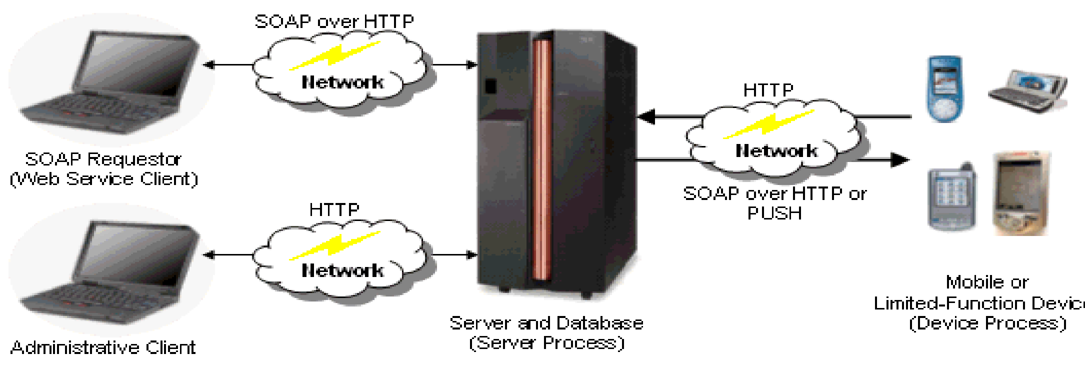

SOAP over HTTP
Network
SOAP Requestor (Web Service Client)
HTTP
Network
Administrative Client
Server and Database (Server Process)
HTTP
Network
SOAP over HTTP or PUSH
Mobile or Limited-Function Devices (Device Process)

Figure 4: Device Web Services Framework

The device process is capable of performing three functions:

- Device and service registration with the gateway.
- Message communication with the gateway.
- SOAP message processing.

The gateway process provides the following functions:

- Gateway administrator. Which involves:
    - Device and service registration and un-registration.
    - Gateway configuration and logging. Configuration would mainly involve database administration.
- SOAP Gateway where it acts as a proxy for SOAP messages intended for mobile devices.

The main function of the gateway is to route SOAP requests to the devices and return responses back to the requestor. The gateway can be implemented as a servlet running on a web server, or as a stand-alone server. The gateway provides a facility for devices and services running on those devices to be registered. Once a device is registered with a gateway a client can invoke a service that is running on the device. Devices can move between gateways or register with several gateways at the same time. No tracking is performed by the gateway and it is the responsibility of the device to un-register itself from a gateway if it will no longer be available through that gateway.

A device may have an IBM Mobile SOAP Server running on it. The Mobile SOAP Server is a lightweight SOAP server that is J2ME-compliant. SOAP requests are forwarded directly by the gateway to the device. Once the server receives a request it parses the SOAP request using the XML parser provided by the WSTKMD. The device processes the request and builds the SOAP response. The device then forwards the response on to the gateway, which returns the response to the requesting client. If a client does not have a SOAP server running or is incapable of receiving requests from the gateway, then the device will have to query the gateway for requests that are intended for it. The device would have to do this regularly to avoid a client timeout occurring.

Pluggable Discovery Framework

As mentioned above the toolkit has a pluggable discovery component. The Discovery Framework allows applications to discover resources and services necessary for their operation, rather than requiring a static administrative step in which they are manually configured with references to those resources. The discovery framework does not replace other lower level discovery mechanisms or protocols such as the Universal Description, Discovery and Integration (UDDI) or the Web Services Inspection Language (WSIL), instead, it is designed to operate in an overall discovery environment in which a number of technologies interplay. An overall discovery environment is one where different discovery protocols can plug into a single component and present a single discovery interface to clients. The framework is designed to run on both a client and a server in a J2ME environment.

Each instance of the discovery framework is called a discovery agent and every enterprise or network may have its own agent. A discovery agent represents a service point for coordinating one or more underlying discovery mechanisms and enforcing local access control policies. The discovery agent basically acts as an internal module to a single network which deals with multiple discovery protocols and presents a simpler interface to visiting applications. The operations of a discovery agent can be seen in the Figure 5 also taken from the WSTKMD documentation.

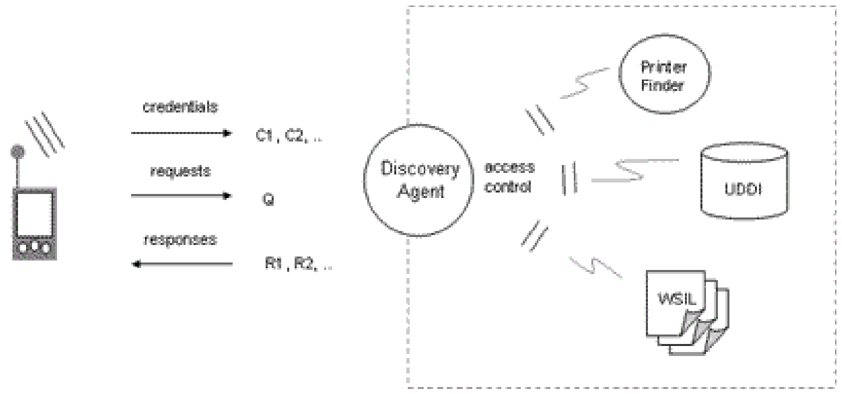


Figure 5: Illustrating the Operation of a Discovery Agent

Web Services Security Mechanism

The WSTKMD toolkit uses Web-Services Security (WS-security) to secure the exchange of SOAP messages between devices. WS-security is a specification that proposes a standard set of SOAP extensions that can be used to implement integrity and confidentiality. It is being defined by IBM, Microsoft and VergiSign. XML digital signatures are used to ensure the integrity of SOAP messages. XML encryption is used to ensure the confidentiality of SOAP messages.

Of the fourteen Mobility challenges the WSTKMD addresses eight of them. The toolkits ability to run on the J2ME environment allows it to use the features of J2ME to address the user interface, storage capacity, and processing power challenges. The use of lightweight SOAP toolkits such as gSOAP and JSR 172 helps to partially address the problem of battery power. The WS-security component of the toolkit addresses the security risks associated with mobile computing while the discovery component addresses the location-dependent information challenge. Also the use of compression techniques in gSOAP and JSR 172 address the low bandwidth challenge.

3.3 eSOAP

Embedded SOAP (eSOAP) is a commercial SOAP framework jointly developed by ConnectTel Inc and Exor International and is available from embedding.net [16]. It is aimed at the development of web services for embedded systems. The eSOAP toolkit is implemented in both C++ and JAVA and is intended as a SOAP engine that will allow embedded systems to communicate using SOAP. The implementation of eSOAP is based on the SOAP 1.1 specification and provides an engine that is:

- Compact. The memory footprint of the eSOAP engine is less than 150KB.
- Fast and efficient.
- Portable. The eSOAP engine is written C++ which makes it capable of running on a wide range of platforms that have a standard C++ compiler.

The toolkit provides a compiler for the C++ mapping to/from WSDL/SOAP. The compiler generates C++ stub and proxy for a given web service interface. The toolkit also provides a Java library of interface classes for client development. The Java interfaces are intended to be used in developing applications that aim to communicate with other web services. The toolkit also comes with an optional module mod-eSOAP that can be used to provide SOAP capabilities for Apache web-servers.

eSOAP is specifically designed to be embedded and run on small, resource constrained devices. Therefore it addresses three of the mobility challenges introduced in section 2.3. By having a small footprint eSOAP addresses the storage capacity, battery power and processing power constraints.

3.4 Wingfoot Mobile Web Services Platform

Wingfoot Software [17] is an organization that provides infrastructure to integrate memory constrained devices with enterprise applications. Their Mobile Web Services Platform allows mobile devices to integrate with J2EE, .NET and legacy applications. The Mobile

Web Services Platform comprises of two components, the Wingfoot Parvus SOAP engine and the Wingfoot SOAP client.

Wingfoot Parvus 1.0

The Wingfoot Parvus 1.0 is a lightweight SOAP engine that can be embedded in memory constrained devices or deployed on a servlet container. It is implemented in Java and provides both SOAP to JAVA and WSDL to JAVA mappings. It was designed with memory constrained, mobile devices in mind so has a small footprint and runs on the KVM. Parvus comes packaged with a range of tools and utilities that allow the exposure of existing applications as web services.

Wingfoot SOAP

Wingfoot SOAP is a lightweight SOAP client aimed at the MIDP/CLDC platform but can also be used on the Personal Java/CDC, J2SE and J2EE platforms. It uses kXML as its XML parser. Wingfoot SOAP 1.03 is SOAP 1.1 compliant and comes packaged with two binaries:

- kvmwsoap_1.03.jar targeted at the CLDC/MIDP platforms. It includes a XML parser and is 37k in size.
- j2sewsoap_1.03 targeted at the CDC/Personal Java, J2SE, and J2EE platforms. It also includes a XML parser and is 34.5k in size.

The API for both binaries is identical except for the transport implementation. kvmwsoap_1.03.jar uses J2ME HTTPTransport which provides a means for MIDP applications to send a SOAP payload over HTTP, while j2sewsoap_1.03 uses J2SE HTTPTransport which provides PersonalJava/CDC, J2SE and J2EE applications with a means to send a SOAP payload over HTTP.

Like the Parvus SOAP engine the Wingfoot SOAP client was designed with mobile devices in mind so has a small footprint and runs on the KVM. The Wingfoot SOAP client uses the Parvus SOAP engine for serialization and de-serialization of Java objects.

The Wingfoot Mobile Web Services Platform addresses four of the mobility challenges. It is lightweight so addresses storage capacity, battery power and processing power. Wingfoot SOAP also provides its own mechanism that allows MIDP applications to send a SOAP package over HTTP and thus addresses the low bandwidth challenge.

3.5 Summary

This section presented the state of the art in SOAP toolkits that are intended for operation in a mobile environment by reviewing three different toolkits. Table 1 summarises what mobility challenges that are addressed by the different toolkits. A bullet (●) indicates good support for dealing with the challenge, a circle (○) indicates partial support and a blank indicates no support.

General Category	Specific Challenge	WSTKMD	eSOAP	Wingfoot
Mobile Devices	Battery Power	○	●	●
	Data Risks			
	User Interface	●		
	Storage Capacity	●	●	●
	Processing Power	●	●	●
Mobile Networking	Network Heterogeneity			
	Disconnection			
	Low Bandwidth	●		●
	Bandwidth Variability			
	Security Risks	●		
	Usage Costs			
Physical Mobility	Address Migration			
	Location-Dependent Information	●		
	Migrating Locality			

Table 1: Summary of Approaches to Mobility Challenges

As can be seen from Table 1 IBM's WSTKMD is by far the most complete toolkit. This is to be expected considering the WSTKMD is intended to be a fully functionally standalone system while both eSOAP and Wingfoot are intended to be used in conjunction with already existing systems. The Wingfoot toolkit is intended to provide integration from mobile devices with already existing enterprise systems by using SOAP as the communication protocol. eSOAP is a basic SOAP engine that is designed to be specifically lightweight and allow embedded systems to communicate.

4 Design

4.1 Introduction

This chapter describes the design of an implementation of the Swizzling and Disconnected Operation Layers using Java and SOAP. The design was performed with the objective of producing a design as similar as possible to the design outlined in [1] for the ALICE Swizzling and Disconnection Operation Layers. This was to reinforce the idea of ALICE being a generic architecture. The design of the integration to the Mobility Layer is also presented. First it is outlined how SOAP satisfies the ALICE requirements and a number of initiatives that allows SOAP to be combined with an application programming language on mobile devices are reviewed.

4.2 SOAP and the ALICE requirements

In order for ALICE to be instantiated for an object oriented middleware framework, the framework must first satisfy a minimal set of requirements. The ALICE requirements were outlined in section 2.3.3. Here it is outlined how SOAP satisfies those requirements.

Requirement R1: Client/Server

The SOAP specification outlines conventions for using SOAP for remote procedure calls based on the client/server abstraction. Combined with section 2.5 which outlined the functionality of a SOAP server and client when implementing the client/server abstraction, it can be concluded that SOAP easily satisfies R1.

Requirement R2: TCP/IP

SOAP messages can be transfer using HTTP or can be streamed from one location to another using sockets. The decision regarding whether to use sockets or HTTP is an application design decision and depends on the requirements of the application. Either way SOAP messages can be transferred using TCP/IP so R2 is easily satisfied.

Requirement R3: Redirection

As mentioned in section 1.3, SOAP does not specify any format for server references or any redirection scheme. Also mentioned was the fact that SOAP can be combined with a programming language and through this combination extra functionality can be added to an application using SOAP as the RIP. If an application was to use sockets as the means for transferring SOAP messages a protocol that would allow two systems to communicate would have to be defined. Within the defined protocol a redirection scheme could be defined that would be supported and understood by distributed systems once they implemented the protocol. If an application choose to use HTTP as the means to transfer SOAP messages the support for redirection built into HTTP could be used to implement a redirection scheme. Either way it is possible for SOAP to satisfy R3.

Requirement R4: Multi-Endpoint References

As said in the previous section SOAP does not specify any format for server references. It is when SOAP is combined with a transfer protocol such a HTTP that the server reference formats are defined. If sockets are the chosen means for transferring SOAP messages, the application within which SOAP is being used would have complete control over what format a server reference should take. This is the approach taken for this instantiation of ALICE so the reference format was designed to satisfy R4.

Requirement R5: Extra Information

Like R4 the reference format was designed to satisfy R5.

Requirement R6: Object Mobility

As outlined in section 2.5 SOAP parsers can be used to serialize objects into a SOAP message. The SOAP message is sent to the required destination where it is de-serialized back into its original format. It is only the state of the object that is serialized and transferred to a remote system. Through the use of serialization it is possible for SOAP to satisfy R6.

4.3 SOAP Mobile Initiatives

A discussed in section 1.3, SOAP becomes more powerful when combined with an application programming language. As this instantiation of ALICE will require SOAP to be combined with a programming language this section reviews three toolkits that could possible be used in this project. Also as the devices for which ALICE is designed are expected to be mobile and therefore constrained memory wise the toolkits reviewed were chosen as they were designed with this constraint in mind.

4.3.1 kSOAP

kSOAP [18] is an open source project available from ObjectWeb for SOAP parsing on J2ME/MIDP platforms. ObjectWeb is an open-source software community created at the end of 1999. kSOAP is based on the generic open source XML parser kXML also available from ObjectWeb. When combined with kXML, kSOAP only has a footprint of 42k which makes it very suitable to operation on memory constrained devices. It consists of a SOAP API that is used as a tool for composing and extracting Java data objects to and from SOAP messages.

KSOAP is designed to operate in the J2ME/CLDC platform which as mentioned in section 2.4 is a Java runtime environment whose purpose is to accommodate the limited footprint of the KVM. Due to the limited footprint the KVM does not provide support for the (de)serialization of Java objects and it is due to this that kSOAP becomes useful as it provides a mechanism for (de)serializing objects so that they can be sent across the network.

4.3.2 gSOAP

The gSOAP [19] is a Web Services development toolkit that offers a XML to C/C++ language binding. It was written by Robert A. van Engelen at the Florida State University and is currently available from sourceforge.net. Its aim is to ease the development of SOAP/XML Web services in C and C++ by providing a compiler that generates SOAP data types for native and user-defined C and C++ data types. The compiler achieves SOAP/XML interoperability with a simple API which relieves developers from the burden of WSDL and SOAP details.

gSOAP is platform independent and also has a small footprint which makes it ideal for deployment on small mobile memory constraint devices. Clients and server applications can be created that are under 100K with a total memory footprint under 150K. gSOAP supports SSL for security and selective input and output buffering to increase the efficiency of message size. gSOAP also includes a WSDL generator that automates Web Service publishing.

4.3.3 Java Specification Request 172 (JSR-172)

The Java Specification Request 172 (JSR-172) [20] is a specification defined by the Java Community Process (JCP) that defines a set of API's for Web Services running in the J2ME environment. The final draft of the JSR-172 was released in October 2003. The main goals of the specification were to provide two new capabilities to the J2ME environment. Those capabilities are:

- To be able to access remote SOAP/XML based web services.
- To be capable of parsing XML data.

The JSR-172 delivered two new optional packages for the J2ME platform.

1.) A package for adding XML parsing to the platform - subset of javax.xml.rpc

2.) A package to facilitate access to XML based web services for CDC and CDLC based configurations - javax.microedition.xml.rpc.

Sun provided an implementation of JSR-172 in the J2ME Web Services API (WSA) package which is available for download from the sun web-site.

4.3.4 Summary

It was decided that Java would be the programming language that would be combined with SOAP for this instantiation of ALICE due to the author's familiarity with the language. gSOAP is designed for the purpose of binding SOAP to C++. This meant that if gSOAP was to be used in combination with Java, a wrapper which would allow Java to access gSOAP's C/C++ code would have to be added to the gSOAP package. This would be extra work and was determined to be unnecessary. kSOAP was chosen as the toolkit to be used in this project as it provided more functionality for SOAP parsing that the WSA.

4.4 The Swizzling Layer

As outlined in section 2.3.2 the swizzling layer is responsible for client redirection and server reference translations. Also as can be seen in Figure 2 the Swizzling layer consists of two components: one residing on the MH and one residing on the MG. The component that resides on the MH is responsible for reference translation and as this project is an instantiation of ALICE for SOAP it is known as S/SOAP$_{mh.}$ The component that resides on the MG is responsible for client redirection and is known as S/SOAP$_{mg.}$ This section describes the design of both of these components. It should be noted that the design of these two components closely resembles the Swizzling Layer design as outlined in [1]. This is to reinforce the concept of ALICE being a generic architecture.

4.4.1 The ALICE Approach to Address Migration

As mentioned in section 2.3.3, ALICE adopted a hybrid scheme to solve the address migration problem. This scheme combines forwarding pointers with a home agent type approach. The idea is that mobile servers leave forwarding pointers on MGs as they move from one gateway to another. They also keep a home agent updated so that a client can fall back on it if for some reason the chain of forwarding pointers left by a server gets broken. The scheme is depicted in Figure 6 taken from [1, p144].

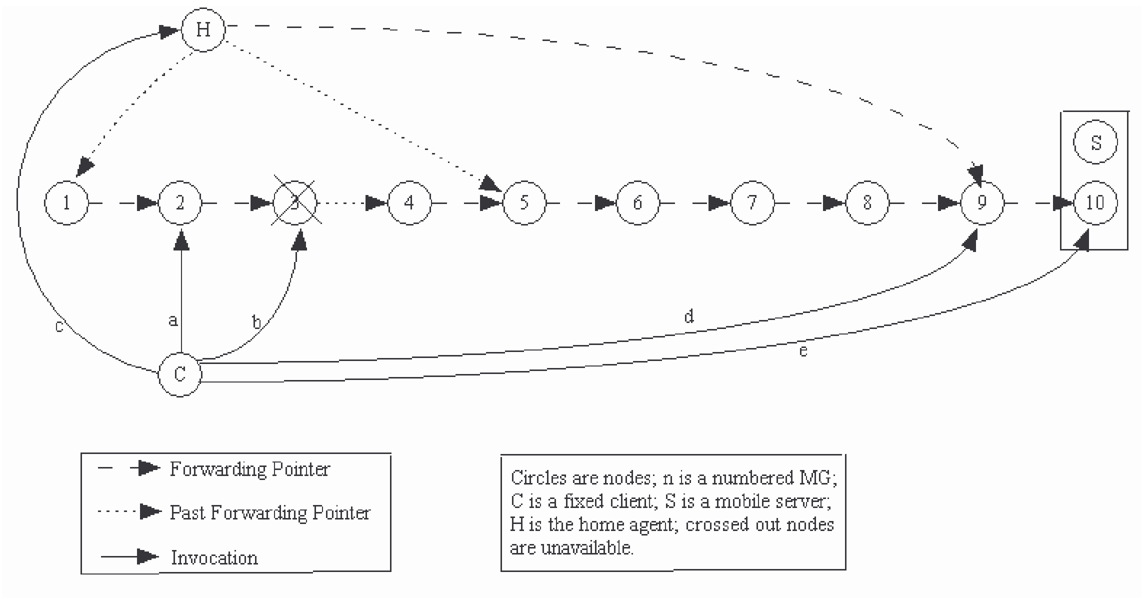

H
1
2
3
4
5
6
7
8
9
10
S
C
a
b
c
d
e
Forwarding Pointer
Past Forwarding Pointer
Invocation
Circles are nodes; n is a numbered MG; C is a fixed client; S is a mobile server; H is the home agent; crossed out nodes are unavailable.

Figure 6: The ALICE Approach to Address Migration

S is a mobile server that has moved from MGs 1 through to 10. Each MG maintains a forwarding pointer to the next known location of the server. The chain of forwarding pointers is broken at MG 3. HA is the home agent for the server which also has a forwarding pointer to the server. The forwarding pointer kept by the home agent is updated at regular intervals by the server. C is a client trying to invoke the server. The client holds a reference to MG 2. When the client try's to invoke MG 2 (arrow a) it receives a redirection to MG 3 (arrow b) which is unavailable. Realizing that MG 3 is unavailable the client fallbacks on the home agent and try's to invoke it (arrow c). The home agent redirects the client to MG 9 (arrow d). MG 9 in turn redirects the client to MG 10 (arrow e) which is the current location of the server.

4.4.2 Dissemination of Server References

The scheme outlined above assumes that the client has a reference to the server before the first invocation attempt. As SOAP does not specify any feature that allows clients to look up server references, a scheme that will allow clients to obtain server references will have to be implemented using the programming language that is being used in combination with SOAP. A common scheme used by middleware is a directory service where the server is responsible for creating its own reference and registering it with the directory. Clients can then query the directory to obtain a reference to the server. The references are usually mapped to a unique name or id within the directory and it is this name that the client uses to query the directory service for the reference.

4.4.3 Server Reference Management

This section describes the S/SOAP$_{mh}$ component. It is responsible for translating server references such that they refer to the MG and the home agent so that scheme outlined in section 4.4.1 is implemented. It is assumed that each server will have a copy of its own reference stored on the server. The S/SOAP$_{mh}$ works by keeping this copy updated so that it refers to the current MG that the server is connected to. This updating is done transparently to the server. The effect of this is that whenever a server decides to register its reference with a directory it is always an up to-date reference that is registered with the directory.

The S/SOAP$_{mh}$ is a component that is attached to each mobile server. It transparently intercepts all calls a server makes that involves the creation and registration of references. Normally the server would create its own reference and register is it with the directory. However if ALICE is being using the reference is transparently translated before it is registered with the directory. This allows the client to obtain a translated reference from the directory.

Server Reference Format

Most object-oriented middleware architectures have server reference formats that are specific to their own architecture. This is not the case for SOAP as the format of server references depends on the transport method (e.g. HTTP or sockets) that is being used. The transport method used in this project was sockets. The advantage of this is that it is a lightweight approach and it also gives us complete control over what format a server reference should take. However the advantage of using sockets comes at the cost of having to design a protocol that will specify how two systems will communicate. The server reference format was designed so that it would satisfy the ALICE requirements. A server references format that fulfils all the ALICE requirements is outlined in Figure 7.

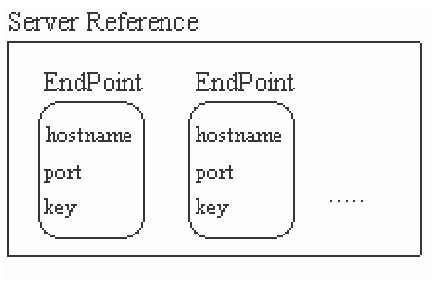


Figure 7: Server Reference Structure

As can be seen from Figure 8 a server reference can contain one or more endpoints. An endpoint specifies a location on the network and it contains three fields. The hostname is the name or IP address of a physical machine on which the server is running. The port is the port number on which the server is listening and the key field is used to transfer extra information between client and server.

The reference translation done by the S/SOAP$_{mh}$ component involves modifying the server's endpoint so that it refers to the current MG and also adds an endpoint that refers to the home agent to the reference. The process of reference translation is called swizzling and a server reference can be either in a swizzled or unswizzled state at any point in time. A swizzled reference is one that has been updated to contain the extra endpoints and an unswizzled reference is a reference that is in its original unmodified state. There are three operations that transform the reference between the two states: swizzling, unswizzling and reswizzling. The three operations are outlined in [1, section 4.7.1] and are summarized here.

Swizzling

Swizzling involves adding the MG and home agent endpoints to a server reference. Swizzling is carried out on both server references and individual endpoints. Swizzling occurs when a server reference is created while the MH is currently connected to a MG. Only endpoints which refer to the local interface are swizzled. Swizzling an endpoint involves changing it so that it no longer refers to the MH but to the S/SOAP$_{mg}$ component to which the MH is connected.

Unswizzling

Unswizzling involves removing the added endpoints from a server reference and returning it to its original state.

Reswizzling

Reswizzling involves updating an already swizzled reference. This occurs when a MH moves from one MG to another. Like swizzling, reswizzling is applied to both server references and individual endpoints. Reswizzling an endpoint involves updating its hostname component so that it refers to the new MG to which the server is currently attached.

4.4.4 Client Redirection

This section describes the S/SOAP$_{mg}$ component. The S/SOAP$_{mg}$ component is responsible for receiving and processing invocations intended for mobile servers. If the server for which the invocation is intended is connected to the MG then the invocation is tunneled on to the server and its response returned to the client. If the server is not connected to the MG then a redirection response that contains a new server reference is returned to the client.

The S/SOAP$_{mg}$ uses callbacks from the Mobility Layer to track what MHs are currently connected to it. These callbacks are received continuously from the Mobility Layer regardless of incoming client invocations. To keep track of mobile servers the S/SOAP$_{mg}$ maintains two lists: a local list and a remote list. The local list contains entries for mobile servers that are being currently being proxied by the MG. The remote list contains entries for mobile servers that are currently being proxied by other MGs.

If a client holds a swizzled reference that identifies a mobile server, the client will try to invoke the server using the first endpoint in the reference. When the client attempts to invoke the swizzled endpoint the invocation will be received by a S/SOAP$_{mg}$ component. During an invocation the client will pass the along the reference it holds to the server. This is to allow a server to retrieve the key field of the endpoint being invoked and thus information about the intended invocation. Using the key field the S/SOAP$_{mg}$ component can determine what server the invocation is intended for and identify whether the server is local, remote or unknown. If the server is local then the invocation is relayed to the mobile server via the ML and the result is returned to the client. If the server is remote a new reference is returned to the client based on the forwarding pointer contained in the remote list. If the server is unknown the S/SOAP$_{mg}$ component constructs a new reference containing just one endpoint which identifies the S/SOAP$_{mg}$ running on the server's home agent.

4.5 The Disconnected Operation Layer

The Disconnected Operation Layer is designed to address the problem of long term disconnection. This is achieved by allowing server functionality to be cached on the client side during periods when the MH is disconnected from the MG. Server objects can be replicated and cached on the client side before disconnection. The replicas are then flushed back to the server and reconciled with the original objects when connectivity is regained.

As can be seen from Figure 2 the Disconnected Operation Layer consists of two components: one residing on the MH and one on the RH. Like the swizzling layer an implementation of the Disconnected Operation Layer is dependant on the RIP with which it is being instantiated. Therefore for this instantiation of ALICE, the disconnected operation layer component that resides on the client is known as the D/SOAP$_c$ and the component that resides on the server is known as the D/SOAP$_s$. Figure 2 also shows the server residing on the RH and the client residing on the MH, but the reverse configuration is also possible. Also as can be seen from Figure 2 the Disconnection Operation Layer needs to be supplemented with application specific code to perform replication and reconciliation. The Disconnection Operation Layer needs to be aware of the state of connectivity at any point in time and relies on ML for this information. Like the Swizzling Layer this information is obtained by callbacks from the mobility layer.

4.5.1 Cache Management and Redirection

This section describes the D/SOAP$_c$ component. This component manages a cache of replicated server objects that are controlled by the application and redirects invocations to the replicas during times of disconnection.

Cache Management

The D/SOAP$_c$ component manages a cache of replicated server objects. The D/SOAP$_c$ component retrieves the server objects from the server when requested. The objects are also returned or flushed to the server by the D/SOAP$_c$ component when requested to do so. These functions are made available through the D/SOAP$_c$ component's API. The API is:

Cache (ServerReference) – causes the D/SOAP$_c$ to retrieve the server object specified by the server reference from the server. Once retrieved the server object is cached and can be invoked locally.

IsCached (ServerReference) → Boolean – checks if the object specified by the server references is cached and returns true if the object is cached and false if it is not.

Flush (ServerReference) – causes the D/SOAP$_c$ to remove the replicated object specified by the server reference from the cache and return it the D/SOAP$_s$ component from which it came. One the replica is received by the D/SOAP$_s$ it carries out conflict detection and resolutions functions on it.

Redirection

When the D/SOAP$_c$ component is present, it will intercept all calls made by the client that involve performing an invocation on an object. Once a call is intercepted the D/SOAP$_c$ component will either redirect that call to its own cache of objects or forward it on to the intended server. If the object for which the invocation is being called is replicated and present in the D/SOAP$_c$ cache then the call will be redirected to the cache during times of disconnection, otherwise the call will be forwarded on to the intended server.

4.5.2 Replication and Reconciliation

This section describes the D/SOAP$_s$ component. The D/SOAP$_s$ component handles request for replication and reconciliation of server objects. It should be noted that the D/SOAP$_s$ component does not take the form of a standalone component rather a component that must be integrated into an application that wants its objects replicated and cached.

Replication of Server Objects

The D/SOAP$_c$ component caches replicates of remote objects by communicating with the D/SOAP$_s$ component. The API for this communication is as follows:

IsCacheable (ServerReference) → Boolean returns true if the object specified by the server reference is cacheable and false if it is not.

Replicate (ServerReference) → DSOAPReplica - returns a replicate of the object specified by the server reference.

Reconciliation of Replicas

The D/SOAP$_s$ component allows the client to return a replica and reconciles it with the authoritative object that is stored on the server. This is done using the following invocation on the D/SOAP$_s$ component:

Reconcile (ServerReference, DSOAPReplica) – instructs the D/SOAP$_s$ component to accept the replica and reconcile it with the original object as specified by the server reference. This will cause application-specific conflict detection and resolution to be carried out.

4.6 The Mobility Layer

As mentioned in section 2.3.2, the Mobility Layer is responsible for managing connectivity between the MH and the MG. The Mobility Layer offers mobility support to mobile applications by implementing a superset of the standard Berkley Sockets API.

An implementation of the mobility is currently available in C and it is a goal of this project to integrate with that layer. This is keeping with the idea of ALICE being a generic architecture and also makes use of the functionality provided by this mobility layer.

The standard approach to integrating Java code with legacy code such a code written in C or C++ would be to write a JNI interface that would act as a wrapper to the legacy code. The Java Native Interface (JNI) is a native programming interface that allows Java to interface with native C/C++ code. However, J2ME is the chosen runtime environment for this project and as mentioned in section 2.4, J2ME has its own implementation of the JVM called the KVM. The KVM is a cut-down, compact version of the JVM designed to run on memory constrained mobile devices. To cut-down on the memory requirement of the KVM it was decided that JNI would not be supported. Instead an interface that is a logical subset of JNI was designed called the K Native Interface (KNI). While KNI is a logical subset of JNI it cannot be used as a complete alternative to JNI as it cannot be used as a means of linking to native code at runtime. To call native code at runtime, the native code would have to be linked directly to the virtual machine at compile time. A KNI wrapper can be used to allow Java code access to whatever native code it needs to call.

One possible solution is to integrate the Mobility Layer into the KVM via the GCF. As mentioned in section 2.4.2 the GCF does not provide any protocol implementations instead it is up to the vendors to provide actual implementations. Integrating the Mobility Layer by plugging it in to the GCF would involve recompiling the KVM and altering some make files so that the Mobility Layer code is integrated into the KVM. A KNI wrapper would have to be written that would allow access to the required methods of the actual C code.

For the GCF to be able to access this wrapper it has to be called ‘Protocol.java’ and placed in a folder specified by the GCF.

4.7 Summary

This chapter outlined the design of an instantiation of ALICE for SOAP. It was outlined how SOAP satisfies the ALICE requirements. The design of the Swizzling Layer and the Disconnected Operation Layer were presented in detail and it was outlined how integration with the Mobility Layer could be achieved. It was also shown that SOAP can satisfy the six ALICE requirements.

5 Implementation

5.1 Introduction

This chapter describes how the Swizzling and Disconnected Operation Layers were implemented. As mentioned previously SOAP needs to be combined with a programming language for extra functionality. The programming language used for this implementation was Java with the J2ME edition. The project was developed on the Linux Fedora Core 2 platform to aid integration with existing ALICE components.

5.2 The SOAP Swizzling Layer

This section describes the S/SOAP layer, an instantiation of the abstract Swizzling Layer for SOAP combined with Java. The Swizzling layer consists of two components: the S/SOAP$_{mh}$ component that performs server reference translation and the S/SOAP$_{mg}$ component that performs client redirection. Together the two components solve the address migration challenge.

5.2.1 SOAP Server

The SOAP servers are implemented using Java to carry out the functionality of the server. The server listens on a port and receives communication in the form of a stream of bits. kSOAP is used to parse the received stream into a server understandable SOAP message. The server will then create the SOAP message that is to be returned to the client as the response. kSOAP is used again to write the SOAP response to a stream of bits that can be returned to the client.

5.2.2 SOAP Client

The SOAP client is also implemented using Java to provide the functionality. When a client needs to make a request to a server it formats the request into a SOAP message and uses kSOAP to write the message to a stream of bits that can be sent to the server. When the client sends a request to a server it waits for a response. When a response is received in the form of bits, kSOAP is used to parse it into an understandable SOAP message.

5.2.3 Server Reference Management

As mentioned in section 4.2, SOAP does not specify any format for server references. Instead it is left up the transport protocol that is being used to transfer SOAP to specify a reference format. This project used sockets to transport SOAP messages from one endpoint to another. Sockets are lightweight which suited this project but come at the cost of having to implement a protocol that two systems have to use to communicate. Sockets also gave us the freedom of being able to specify what format server references would take. The server reference format implemented was designed to satisfy all the ALICE requirements that referred to server references.

Server Reference Format

A server reference consists of one or more endpoints. An endpoint consists of three fields:

1.) hostName

2.) port

3.) key

The hostName field is the name or the address of the physical machine, the port field is the number of the port on which the server is listening and the key field allows extra information to be sent from the client to the server whenever it wishes to carry out an invocation. The relationship between a server reference and endpoint is illustrated in Figure 8.

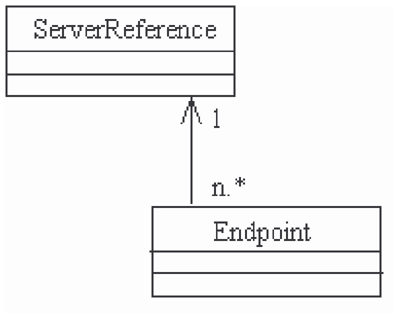


Figure 8: Relationship between Server References and Endpoints

Reference Swizzling

As outlined in section 4.4.3 both endpoints and server references are swizzled. It is the endpoints that are swizzled first and when all endpoints that refer to the local interface are swizzled the reference is then swizzled. The local interface is the endpoint for the MH on which the mobile server is running. Endpoint swizzling, reswizzling and unswizzling can be seen in Figure 9.

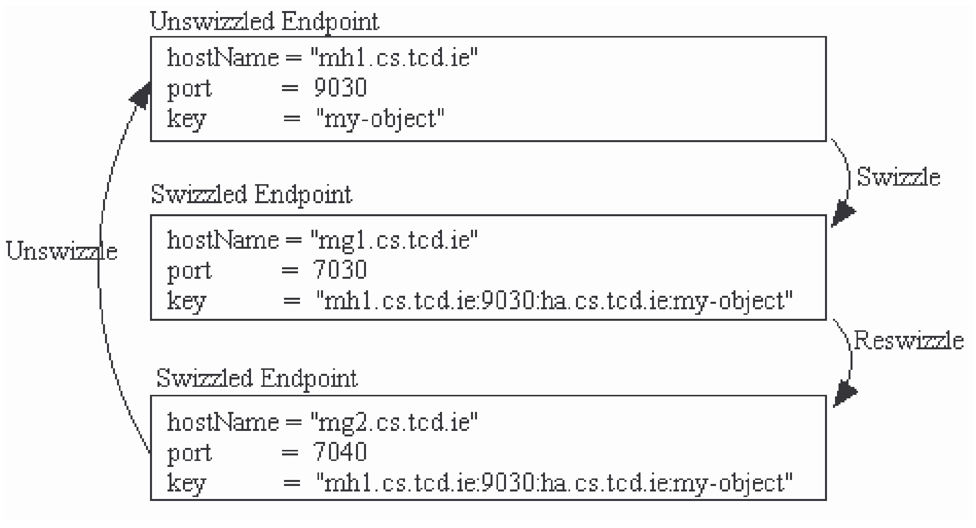

Unswizzled Endpoint
hostName = "mh1.cs.tcd.ie"
port = 9030
key = "my-object"
Swizzle
Swizzled Endpoint
hostName = "mg1.cs.tcd.ie"
port = 7030
key = "mh1.cs.tcd.ie:9030:ha.cs.tcd.ie:my-object"
Unswizzle
Reswizzle
Swizzled Endpoint
hostName = "mg2.cs.tcd.ie"
port = 7040
key = "mh1.cs.tcd.ie:9030:ha.cs.tcd.ie:my-object"

Figure 9: Swizzling, Reswizzling and Unswizzling Endpoints

The swizzling, unswizzling and reswizzling processes for endpoints follow the designed outlined in section 4.4.3. A server reference is considered to be swizzled if all the endpoints it contains that refer to the local interface are swizzled and an endpoint for the home agent has been added to it. Server reference swizzling and unswizzling is illustrated in Figure 10. The swizzling, reswizzling and unswizzling process for server references follows the designed outlined in section 4.4.3.

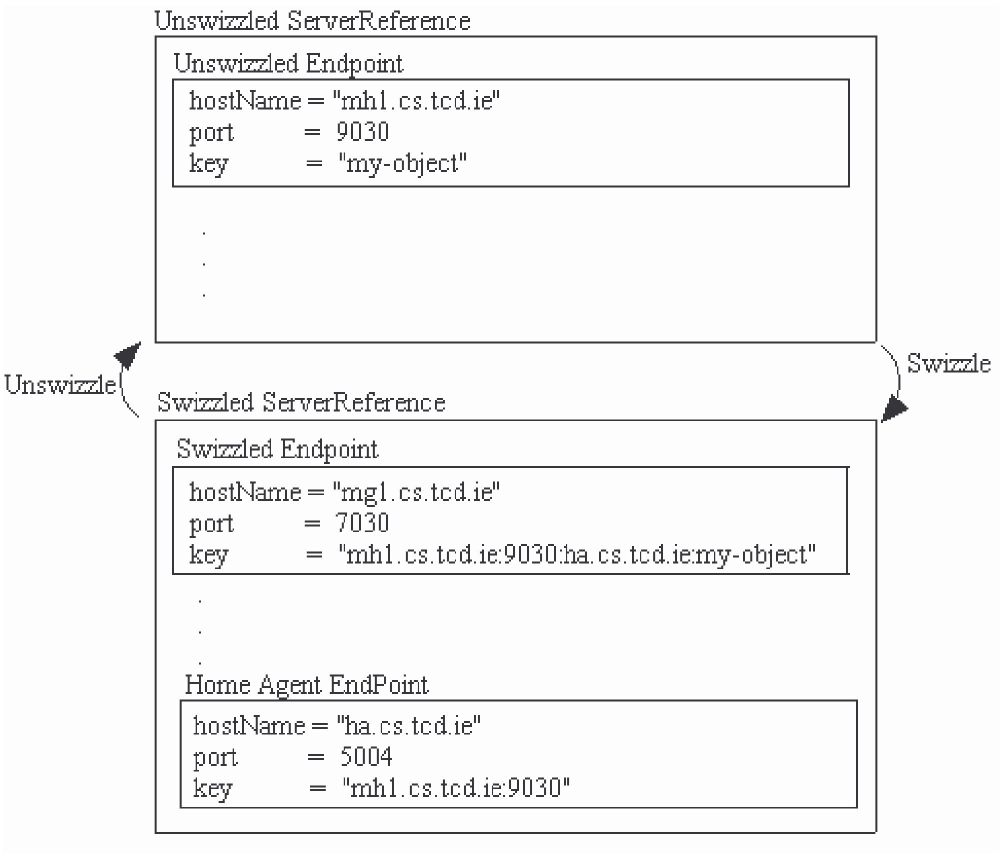

Unswizzled ServerReference
Unswizzled Endpoint
hostName = "mh1.cs.tcd.ie"
port = 9030
key = "my-object"
Unswizzle
Swizzle
Swizzled ServerReference
Swizzled Endpoint
hostName = "mg1.cs.tcd.ie"
port = 7030
key = "mh1.cs.tcd.ie:9030:ha.cs.tcd.ie:my-object"
Home Agent EndPoint
hostName = "ha.cs.tcd.ie"
port = 5004
key = "mh1.cs.tcd.ie:9030"

Figure 10: Swizzling and Unswizzling of a Server Reference

Reference dissemination

As outlined in section 4.4.2 SOAP does not specify how server references should be disseminated. A method for reference dissemination can be implemented using the programming language with which SOAP is being combined. This is the approach taken in this project. An approach similar to that taken in Java RMI [3] was implemented. A directory service called the Registry is running on a well known host and port number. The difference between this approach and RMI is that there is only one central registry, where RMI has a registry running on every host.

A server registers its reference with the registry when it is first created. Each reference is bound to a unique name in the registry that is supplied by the server upon registration. The registry ensures that the name is unique before allowing a reference to be registered. When a client wishes to look up a server's reference it does so using the unique name of the server. The registry specifies a protocol that must be implemented by a client or server wishing to communicated with it.

The protocol for interfacing with the registry is:

SUCCESS – Returned from the registry to a client or server if a REGISTER, LOOKUP, or REMOVE attempt was successful.

FAILURE - Returned from the registry to a client or server if a REGISTER, LOOKUP, or REMOVE attempt was unsuccessful. This will be returned with an explanation of the failure. An example would if a server attempted to register using a name that was already bound to another server's reference.

REGISTER (ServerReference, Name) – Sent from a client to the registry. This is used to register a ServerReference to a unique Name. On receipt of a REGISTER the registry will return either a SUCCESS or a FAILURE to the client.

LOOKUP (Name) – Used by a client to obtain a server's reference from the registry. The required reference is specified by the Name parameter. The registry will either return a SUCCESS along with the requested reference or a FAILURE along with an explanation for the failure. Normally a failure would occur only if the required reference was not registered with the registry.

REMOVE (Name) - This is used by a client to remove a reference from the registry. The reference to be removed is specified by the Name parameter. The registry will respond with either a SUCCESS or a FAILURE.

5.2.4 Implementation of the S/SOAP$_{mh}$ Component

The S/SOAP$_{mh}$ component is implemented as a Java object that is attached to every mobile server. A SOAP server is either mobile aware or unaware. If a server is mobile aware it will create an SSoapMH object that will carry out server reference translation on its behalf. Whether a server is mobile aware or not it will register its reference with the registry. When a server is mobile aware all calls that involve the server's reference are directed to the S/SOAP$_{mh}$ component. Figure 11 shows the sequence of events that occurs when a mobile aware server is created.

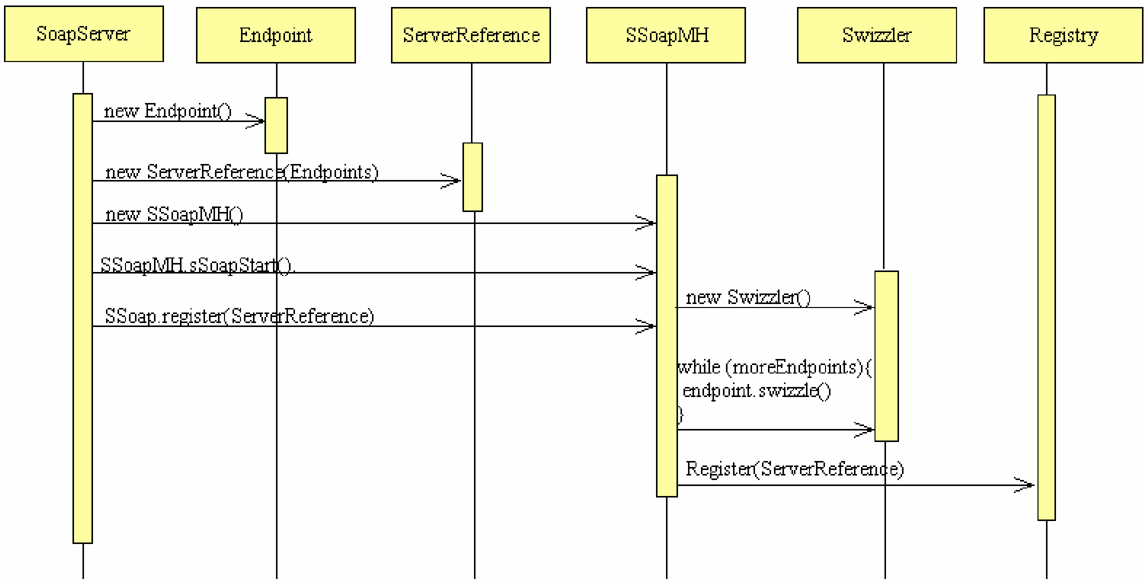


Figure 11: S/SOAPmh Interaction Diagram

As can be seen from Figure 11 when a SOAP server is created it will create its own server reference. This is done by creating the necessary Endpoints and supplying them to a

ServerReference object. Once the reference has been created the server creates a SSoapMH object and calls *sSoapStart()* on it. This causes the SSoapMH component to initialize itself and create a Swizzler object which it will use for swizzling endpoints. When the SOAP server wishes to register its server reference the call to register it is redirected to the SSoapMH component. The SSoapMH component swizzles the server reference before registering it with the Registry.

5.2.5 Client Redirection

The section describes how SOAP clients are redirected to implement the scheme outlined in section 4.4.1. The implementation of the S/SOAP$_{mg}$ component is also described.

As mentioned in section 4.4 when sockets are used as the means for transferring requests/responses from one endpoint to another a protocol that allows two distributed systems to communicate is required. The protocol implemented for SOAP client and server communication for the purposes of this project is a follows:

REQUEST – Sent from a client to a server to signal that the client wishes to carry out a request on the server.

RESPONSE – Sent from the server to a client in response to a REQUEST.

REDIRECTION – Sent from a sever to a client signaling that the client should send its request to another server whose reference will be embedded in the REDIRECTION response.

It should be noted that whenever one of the above is sent from a client to a server or vice versa the reference for the server involved in the communication will always to be sent along with the message. This feature was added to avoid complications with the reswizzling operations of the S/SOAP$_{mg}$ component.

Using the above protocol the scheme outlined in section 4.4.1 can be implemented. An invocation is sent from a client to a server in the form of a request. If the server can carry out the request then a response will be returned to the client. If for some reason the server cannot carry out the request but is aware of a server which can then a redirection is sent from the sever to the client containing the reference for the server which the client should try. The process of redirection is carried out transparently to the client. What this means is that if a client receives a redirection no input is required from the client, instead a request is sent automatically to the server to which the client is being redirected.

Management of Forwarding Pointers

The S/SOAP$_{mg}$ component allows forwarding pointers to be updated by means of a SETLOCATION call to it. A mobile sever can send a SETLOCATION to a S/SOAP$_{mg}$ to update its forwarding pointer if it moves from one MG to another.

5.2.6 Implementation of the S/SOAP$_{mg}$ Component

The S/SOAP$_{mg}$ component is implemented as a daemon that runs on all MGs. The S/SOAP$_{mg}$ component is effectively a server in its own right that runs on the MG.

Mobile Server Tracking

The S/SOAP$_{mg}$ component keeps track of mobile servers by maintaining two lists of server pointers. A local list which is the list of servers currently being proxied by the MG and a remote list which is a list of forwarding pointers. A ServerPointer is a Java object that has four fields, mhName, mhPort, mgName, mgPort. The mhName and mhPort refer to the MH on which a server is running. The mgName is the name of the MG which currently is acting as a proxy for the mobile server specified by mhName and mhPort. The mgPort refers to the port number on the ML that is being used as a proxy for the mobile server.

When the S/SOAP$_{mg}$ receives a listen callback from the ML its *listenStart()* method is called and an entry is added to the local list. When the S/SOAP$_{mg}$ receives a close callback its *listenStop()* method is called and an entry is removed from the local list. If the S/SOAP$_{mg}$ receives a handoff callback from the ML it will remove the entries it has for the MH that's moving from its local list to the remote list.

Operation during Invocation

When the S/SOAP$_{mg}$ component receives a request it will extract the server reference from the request message to determine what server the request is intended for. The S/SOAP$_{mg}$ is able to determine what server the request is intended for by unswizzling the server reference. If the server is local to the MG then a tunnel is set up to the server and the request is forwarded to the server. The server returns the response to the MG which returns it to the client.

Reswizzling during Redirection

When on receiving an invocation for a server that resides on an MH, which is no longer hosted by the MG, the S/SOAP$_{mg}$ will have to construct a new server reference for the server. The new reference is constructed based on the forwarding pointer held by the S/SOAP$_{mg}$ and returned to the client as a redirection response.

Implementation of the Home Agent

The home agent component is implemented as an S/SOAP$_{mg}$ component. Any S/SOAP$_{mg}$ component can act as a home agent for a mobile server. The forwarding pointers stored on a home agent will be stored in the remote list in the exact same way as an S/SOAP$_{mg}$ component would store a forwarding pointer to a mobile server. A mobile server can use a SETLOCATION call to an MG to give the MG a forwarding pointer to itself and to keep that pointer updated.

5.3 The SOAP Disconnected Operation Layer

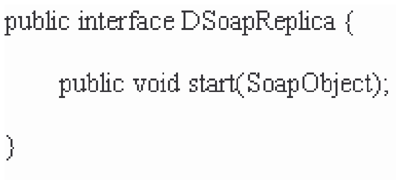
This section describes the SOAP Disconnected Operation Layer. The D/SOAP layer follows closely the Disconnected Operation Layer outlined in section 2.3.2. First we outline how SOAP can be used to transport Java objects and then the actual implementation of the Disconnected Operation Layer is described.

5.3.1 Object Mobility in SOAP

As stated in section 1.3 SOAP does not understand the concept of an object. However by serializing an object into XML form, SOAP can be used to send objects from one system to another. For Java objects it is the state of the object that is serialized into XML form not the behavior. It is assumed that the receiving system understands the semantics of the data it will be receiving. On receipt of a SOAP message, the receiving system will create a new object of the type it is expecting to receive and initialize it with the values it has received in the SOAP message.

5.3.2 D/SOAP Approach

The Disconnected Operation Layer consists of two components, the D/SOAP$_s$ component which resides on the server and the D/SOAP$_c$ component which resides on the client side. The components pass replicas of objects over and back between then. The type of the object being replicated is application specific but it must implement the DSoapReplica interface that is given in Figure 12. The DSoapReplica has just one method that that initializes the replica after it has been sent across network.

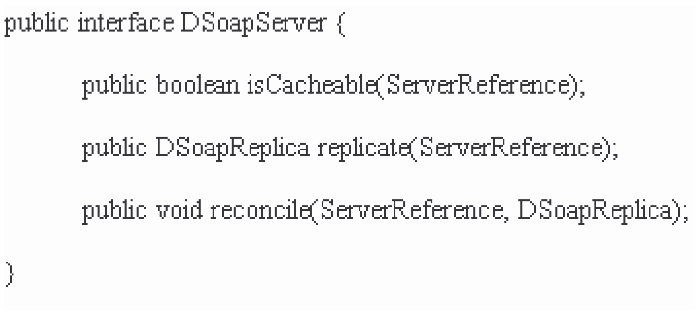


```
public interface DSoapReplica {

       public void start(SoapObject);

}
```

Figure 12: DSoapReplica Interface

On receipt of a SOAP message that contains a Java object the receiving component will create a DSoapReplica object and pass it the SOAP message by calling its start method. The *start()* method will initialize the replica with the values contained in the SOAP message.

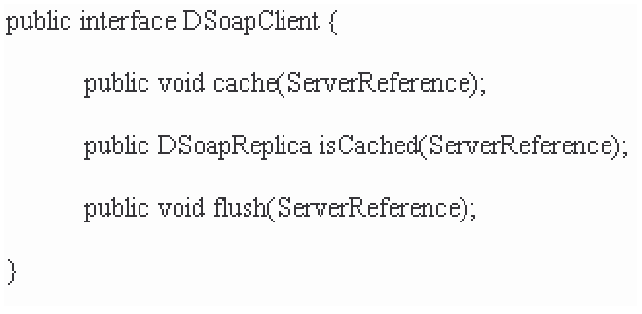
The D/Soap$_s$ component relies on the Application Layer for a complete instantiation due to the fact that it is the application that determines the actual type of the objects that will be replicated. The D/SOAP$_s$ component is a Java interface that must be implemented by the application server. The D/SOAP$_s$ interface is giving in the Figure 13.



```
public interface DSoapServer {

        public boolean isCacheable(ServerReference);

        public DSoapReplica replicate(ServerReference);

        public void reconcile(ServerReference, DSoapReplica);

}
```

Figure 13: DSoapServer Interface

The D/SOAP$_c$ component is a server that runs on the client and is responsible for managing the cache of replicas. The client application interfaces with the D/SOAP$_c$ component to instruct it to replicate server objects or to flush replicated objects back to the server. Figure 14 illustrates the DSoapClient interface.



```
public interface DSoapClient {

        public void cache(ServerReference);

        public DSoapReplica isCached(ServerReference);

        public void flush(ServerReference);

}
```

Figure 14: DSoapClient Interface

5.3.3 Replication

Replicas are Java objects that are stored on the client side. They are transferred to the D/SOAP$_c$ component from the D/SOAP$_s$ component as the result of a replication request from the D/SOAP$_c$ component. As stated in the previous section the D/SOAP$_c$ component runs on the client. The client controls the D/SOAP$_c$ by invoking its API. A client can request the D/SOAP$_c$ to replicate an object that is specified by a server reference. The D/SOAP$_c$ checks if the object it intends to request is cacheable before requesting it from the server. It does this by invoking the D/SOAP$_s$ components *isCacheable()* method. If the object is cacheable the D/SOAP$_c$ will invoke the *replicate()* method on the D/SOAP$_s$. When the D/SOAP$_c$ component receives a replica it adds it to its cache and informs the client that it has successfully added the requested object to its cache.

5.3.4 Cache Management

Upon receiving a replica the D/SOAP$_c$ component creates an instance of the replica and initializes it by calling its start method and passing it the marshaled state it received from the D/SOAP$_s$ component. The replica un-marshals its state and prepares to start receiving requests. Once a replica has been successfully initialized, the D/SOAP$_c$ component adds it to the cache. The cache is a simple table where the D/SOAP$_c$ component stores the replicas.

The replicas are identified by a unique id and can be retrieved from the table at any time. The replica is kept in the table until the client calls the D/SOAP$_c$ components *flush()* method, at which time the replica is removed from the cache and returned to the D/SOAP$_s$ component.

5.3.5 Redirection

The D/SOAP$_c$ component sits between the client and the D/SOAP$_s$ component and intercepts all out going requests from the client. When the D/SOAP$_c$ component is operating in connected mode all invocations are passed directly to the D/SOAP$_s$ component. When the D/SOAP$_c$ is operating in disconnected mode all requests are examined to see if a replica for the requested object exists in the cache. If there is a replica in the cache the request is redirected to the replica. If there is no replica in the cache and the D/SOAP$_s$ component cannot be reached the client is informed that the request could not be carried out.

5.3.6 Reconciliation

When the client calls the *flush()* method on the D/SOAP$_c$ the specified object is removed from the cache, its state is marshaled into a SOAP message and is returned to the D/SOAP$_s$ component. The D/SOAP$_s$ component receives the replica, un-marshals its state from the SOAP message and updates the authoritative object.

5.4 Summary

This chapter described the implementation of this project. A fully functional implementation of the Swizzling Layer and Disconnected Operation Layer was presented. The implementation of the S/SOAP$_{mh}$ and S/SOAP$_{mg}$ components was outlined. Also the implementation of the SOAP server, client and registry were described. Finally the implementation of the two components of the Disconnected Operation Layer was presented. All implementation was done using Java and SOAP as the communication protocol.

6 Evaluation

6.1 Introduction

This chapter evaluates the success of the components developed for this instantiation of ALICE. The Swizzling and the Disconnected Operation Layers are evaluated separately. First the Swizzling Layer is evaluated in terms of performance, code size and transparency. Then the Disconnected Operation Layer is evaluated in terms of performance. The different configurations used to evaluate the two layers are also outlined.

6.2 The Swizzling Layer

The Swizzling Layer is responsible for server reference translation at certain key points in time and client redirection. This section evaluates the Swizzling Layer by first describing a number of configurations that were used to evaluate the layer and then evaluating the success of the layer under the headings of performance, code size and transparency.

6.2.1 Experimental Configuration

This section outlines four different configurations that were used to evaluate the Swizzling Layer.

Configuration 1: Client and Server without ALICE

This configuration was used to set a baseline for the cost of 1000 invocations between a SOAP client and a SOAP server. The configuration is illustrated in Figure 15.

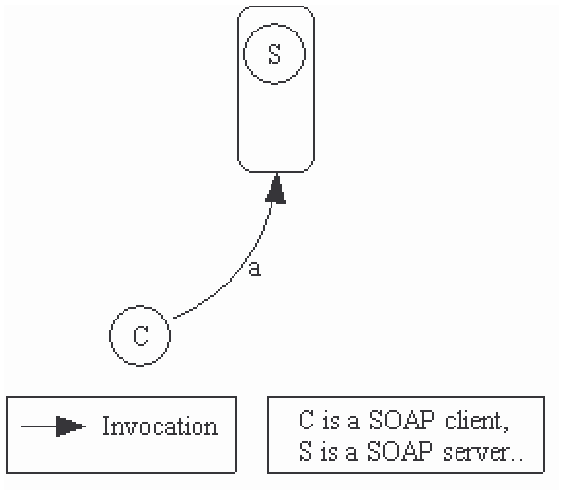


Figure 15: SOAP Client and Server without ALICE

As can be seen from the above diagram, this configuration involves sending invocations from a client to a server with no ALICE components involved. The client performs a simple invocation (arrow a) that is sent directly to the server.

Configuration 2: Server on the MH with ALICE and One MG

As can be seen from Figure 16 this configuration involves ALICE with one level of redirection. The client retrieves a reference to the server it wishes to invoke as it would before the introduction of ALICE. However due to the introduction of ALICE the server reference no longer refers directly to the server but to the mobility gateway that the server is attached to.

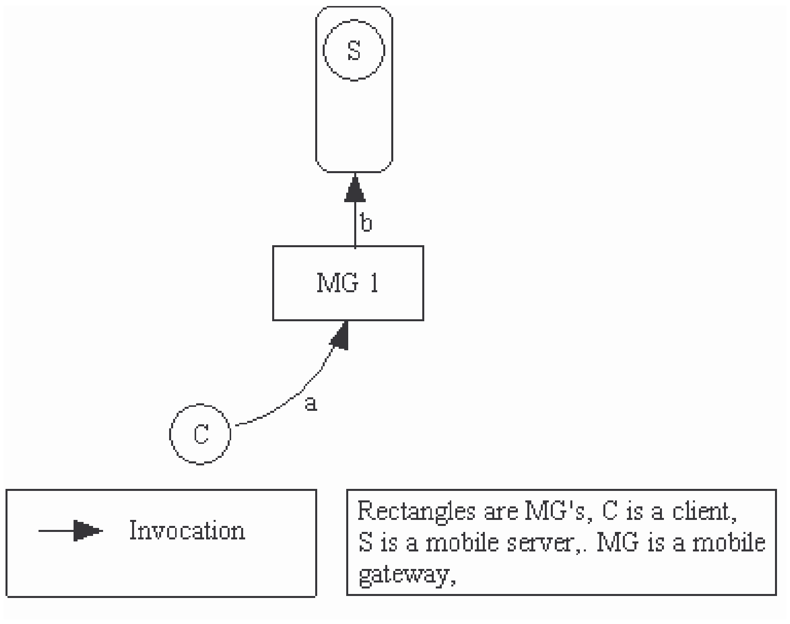

S
b
MG 1
a
C
Invocation
Rectangles are MG's, C is a client, S is a mobile server,. MG is a mobile gateway,

Figure 16: Server on the MH with ALICE and One MG

The client tries to invoke the server (arrow a) but the invocation request is transparently sent to MG 1. MG 1 checks if the requested server is currently attached to it and, realizing that the server is currently attached to it, forwards the request on to the server (arrow b).

Configuration 3: Server on the MH with ALICE and Two MGs

This configuration introduces another level of redirection using ALICE. As can be seen from Figure 17 the server has moved form MG 1 to MG 2 but has left a forwarding pointer at MG 1 which can be used to redirect clients to its new location.

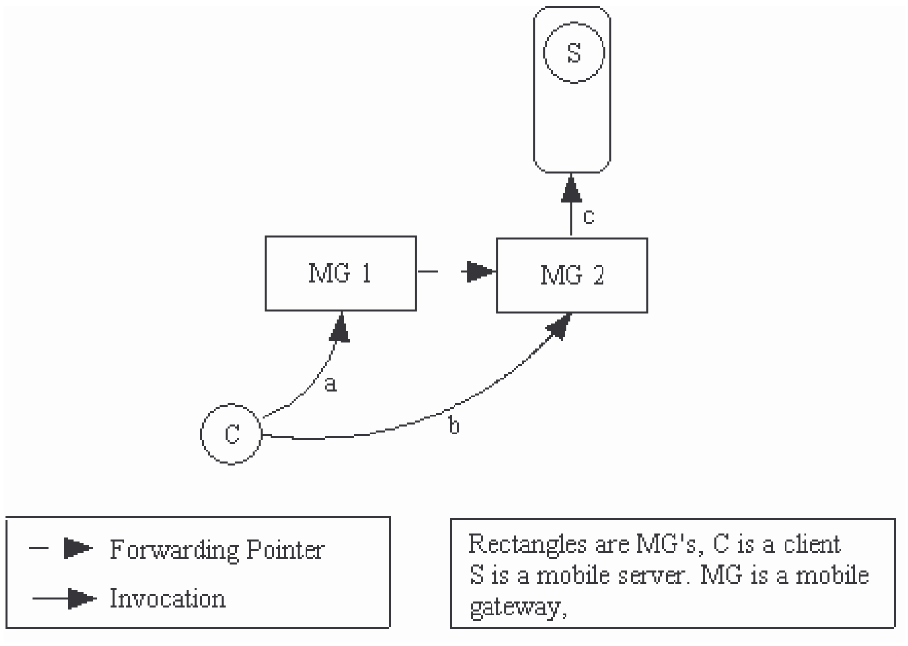

S
c
MG 1
MG 2
a
C
b
Forwarding Pointer
Invocation
Rectangles are MG's, C is a client
S is a mobile server. MG is a mobile
gateway,

Figure 17: Server on the MH with ALICE and Two MGs

The client tries to invoke the server through MG 1 (arrow a) but the server is no longer attached to MG 1. On realizing the server is no longer attached, MG 1 constructs a new server reference and returns it to the client in a redirection response. The client gets redirected (arrow b) to MG 2. The required server is attached to MG 2 and the clients request is forwarded (arrow c) on to the server by MG 2.

Configuration 4: Client on the MH with ALICE, Two MGs and One Home Agent

This configuration introduces two new levels of redirection. The chain of forwarding pointers that the server has left as it moves from gateway to gateway has become broken because MG 2 does not have a forwarding pointer to the mobile server. Configuration 4 is illustrated in Figure 18.

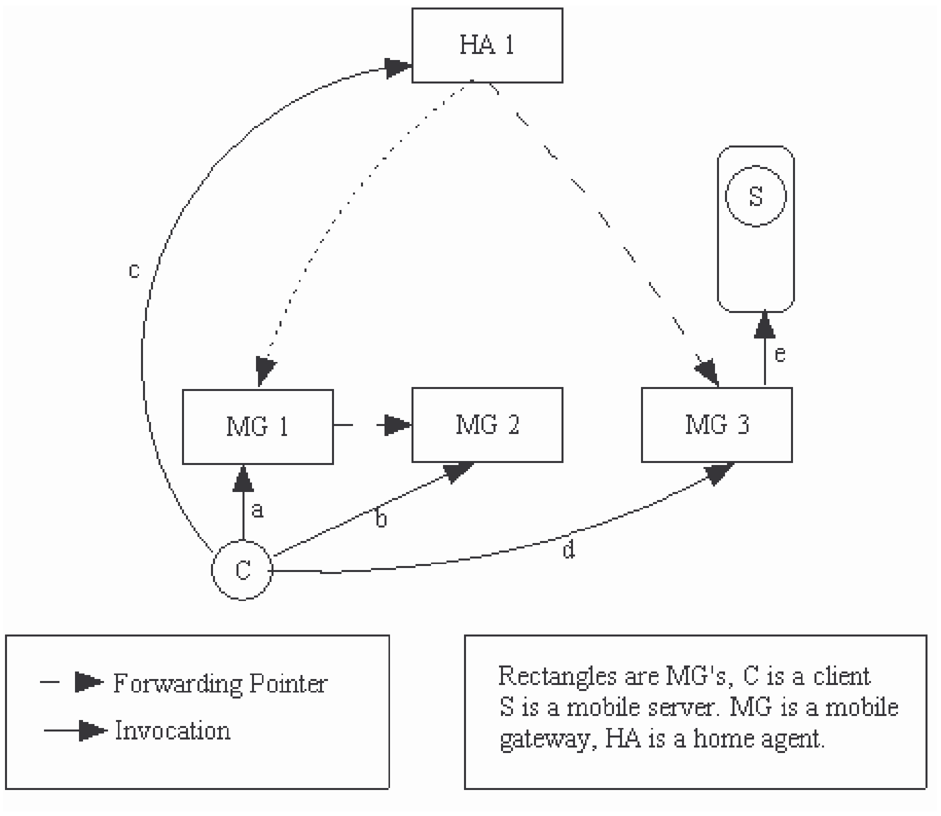

HA 1
S
c
e
MG 1
MG 2
MG 3
a
b
d
C
Forwarding Pointer
Invocation
Rectangles are MG's, C is a client S is a mobile server. MG is a mobile gateway, HA is a home agent.

Figure 18: Server on the MH with ALICE, Two MGs and One HA

The client acquires a server reference that points to MG 1 as the server's location. The client attempts to invoke the server (arrow a) through MG 1. MG 1 returns a redirection to the client redirecting it to MG 2 (arrow b) but the server has left MG 2 without leaving a forwarding pointer. MG 2 on realizing it does not have a pointer to the required server redirects the client (arrow c) to the server's home agent (HA 1). The HA 1 contains a forwarding pointer for the server and redirects the client to MG 3 (arrow d). The server is attached to MG 3 so the clients request is forwarded on to the server (arrow e).

6.2.2 Performance

The performance of the Swizzling Layer is measured through a comparison of the time taken for 1000 invocations using the different configurations presented above. The total invocation times are illustrated in Figure 19.

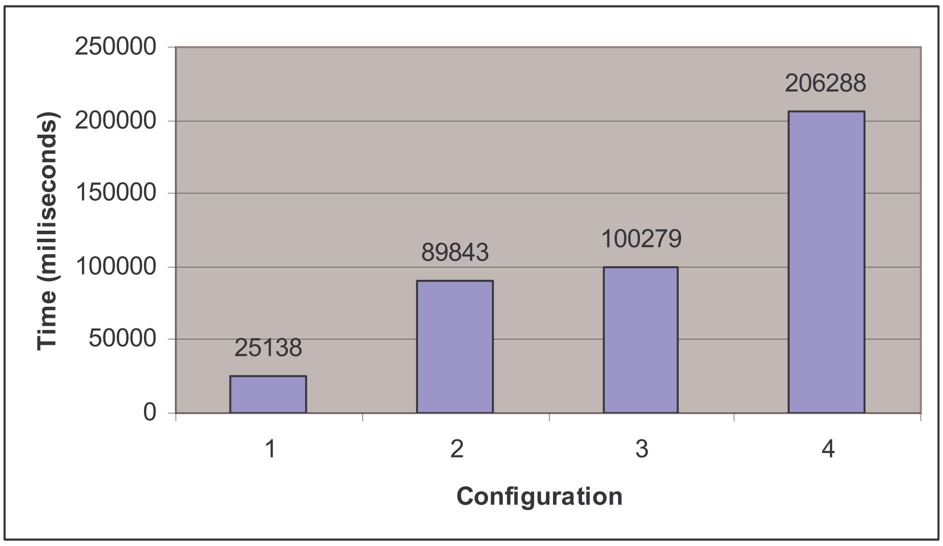


Figure 19: Invocation times for 1000 invocations using the different configurations

Configuration 1:

The total time taken for 1000 invocation for configuration 1 is 25,138 milliseconds. This is an average of 25.138 milliseconds for a single invocation. This is a low invocation time and is due to the fact that the invocation is a straight forward request-response with no computation taking place in between the client and the server.

Configuration 2:

The total time taken for 1000 invocations for configuration 2 is 89,843 milliseconds. This is an average of 89.843 milliseconds for a single invocation. This represents a 257% increase from configuration 1 which is a significant increase. This increase is due to the introduction of the MG and the time taken for it to complete the necessary computation. This leads us to conclude that the introduction of ALICE leads to a significant increase in the time taken for a single invocation but this is to be expect because due to the introduction of the MG.

Configuration 3:

Configuration 3 includes two levels of redirection. The totally time taken for 1000 invocations with configuration 3 is 100,279 milliseconds. This is an average of 100.279 milliseconds per invocation. This is an 11.61% increase on configuration 2. This increase is to the fact that the client has to be redirected to another gateway. From this we can conclude that time for a redirection from one MG to another leads to an expected but modest increase in the time taken for an invocation.

Configuration 4:

Configuration 4 includes three different redirects. Client redirection from MG 1 to MG 2, redirection from MG 2 to HA 1 and redirection from HA 1 to MG 3. The total time taken for the 1000 invocations was 206,288 milliseconds. This is an average of 206.288 milliseconds per invocation. This is a 105% increase on configuration 3 which is a very significant increase. The increase can be explained by the fact the invocation has to go through twice as may gateways as configuration 3 (four compared to two). However as can be deduced from the performance of configuration 3, the introduction of one new gateway only leads to an 11.61% increase in invocation times, so we have to conclude that the majority of the 105% increase is due to the introduction of the home agent. This would lead us to conclude that it is more beneficial to put more effort into maintaining the chain of forwarding pointers than rely on the home agent as a fall back if the chain is broken.

6.2.3 Code Size

It is a requirement that any instantiation of ALICE be capable of running on a mobile device with limited memory. Therefore it is important that any code developed have a small foot print. The memory requirements of the code developed is summarized is Table 2 and 3.

Software Component	**Code Size (KB)**
CLDC	128
kSOAP	40
SOAP server	8.5
S/SOAP$_{mh}$ component	9.8
Total	186.3

Table 2: The code requirement for the Swizzling Layer on the MH

Table 2 illustrates the code requirements for the Swizzling Layer on the MH. The CLDC is required to run the SOAP server as well as S/SOAP$_{mh}$ component. kSOAP is required for SOAP parsing. As can be seen the actual S/SOAP$_{mh}$ component only requires a total of 9.8 KB of memory and when combined with the other components totals to a memory requirement of 186.6 KB which is an acceptable overhead for mobile devices.

Table 3 illustrates the code requirements for running the Swizzling Layer on the MG. The S/SOAP$_{mg}$ is a standalone component so does not require a SOAP server to run. The total memory requirement for the Swizzling Layer on the MG is 183.5 KB which is also an acceptable overhead for mobile devices.

Software Component	**Code Size (KB)**
CLDC	128
kSOAP	40
S/SOAP$_{mg}$ component	15.5
Total	183.5

Table 3: The code requirement for the Swizzling Layer on the MG

6.2.4 Transparency

One of the overall aims of ALICE is that the mobility support offered by ALICE be transparent to the architecture that ALICE is being instantiated for. What this means is that components of a system should not need to be aware of ALICE. They should be able to operate the same as they were before ALICE support was added to the system. How successfully this objective was achieved for the Swizzling Layer is evaluated here.

Client Side

ALICE support is completely transparent to the SOAP client. The SOAP client is not aware of any of the ALICE components. The client retrieves a server reference and tries to invoke the server using the reference in exactly the same way it would if ALICE components were not involved. The fact the client requests goes through an MG before reaching the server is completely transparent to the client.

Server Side

ALICE support is not as transparent to the SOAP server as it is to the client. On creation the server needs to create the S/SOAP$_{mh}$ component to handle the swizzling of its reference. Once up and running, ALICE is completely transparent to the server. The server is completely unaware of the involvement of the MG component.

6.3 The Disconnected Operation Layer

This section evaluates the Disconnected Operation Layer that was implemented for this project. Components of the Disconnected Operation Layer are application specific. That is they are reliant on an application for a complete instantiation which meant that in order to evaluate the layer an application had to be implemented. A simple application was implemented that was capable of replicating objects and caching them on the client side to perform this evaluation.

6.3.1 Experimental Configuration

This section outlines the two different configurations of the Disconnected Operation Layer that were used to evaluate performance.

Configuration 1: The Disconnected Operation Layer in Connected mode

Configuration 1 was used to evaluate the Disconnected Operation Layer operating in connected mode. Connected mode is when the D/SOAP$_c$ component is connected to the D/SOAP$_s$ component. When in connected mode all invocations are passed over the network to the server on which the server object reside. This is illustrated in Figure 20.

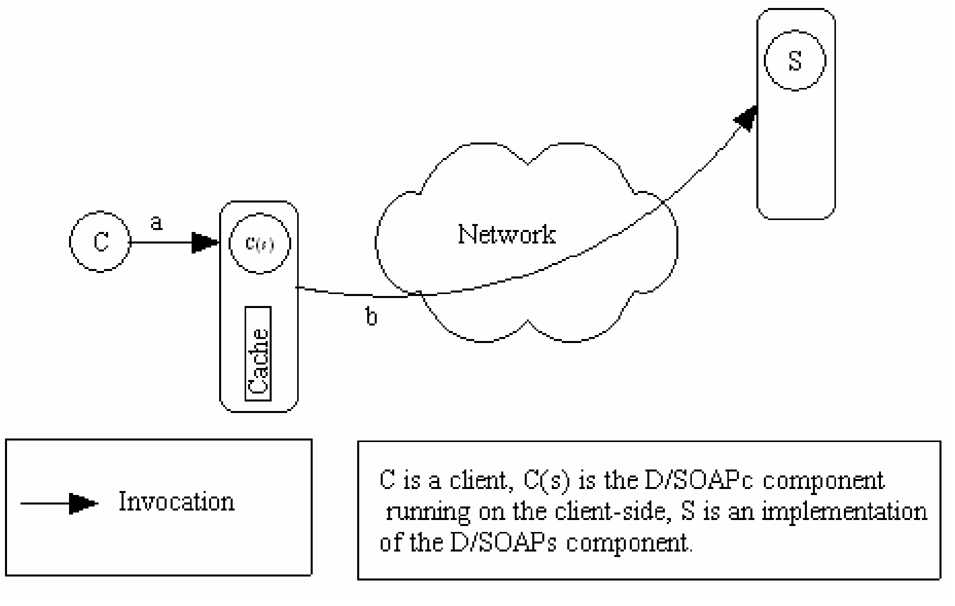

S
C
a
C(s)
Cache
Network
b
Invocation
C is a client, C(s) is the D/SOAPc component running on the client-side, S is an implementation of the D/SOAPs component.

Figure 20: Disconnected Operation Layer in connected mode

As can be seen from Figure 20 the Disconnected Operation Layer consists of a client application that controls the cache on the D/SOAP$_c$ component and a D/SOAP$_s$ component which is the authoritative server on which the original objects are stored. When in connected mode the client's requests are redirected to the D/SOAP$_c$ component (arrow a). The D/SOAP$_c$ component realizing that it has a connection to the server forwards the request on to the D/SOAP$_s$ component (arrow b).

Configuration 2: The Disconnected Operation Layer in Disconnected mode

This configuration was used to evaluate the performance of the Disconnected Operation Layer operating in disconnected mode. The Disconnected Operation Layer operates in disconnected mode when a previous connection to the server is broken. The connection can be broken either voluntarily by the client or by the client moving out of range of a gateway. Configuration 2 is illustrated in Figure 21.

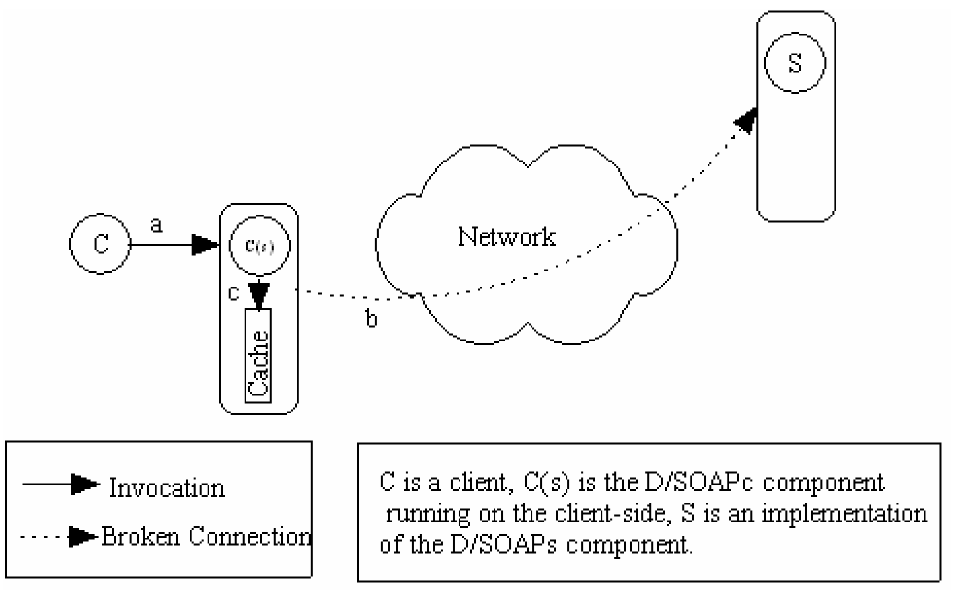

S
a
C
C(s)
c
Cache
Network
b
Invocation
Broken Connection
C is a client, C(s) is the D/SOAPc component running on the client-side, S is an implementation of the D/SOAPs component.

Figure 21: Disconnected Operation Layer in disconnected mode

As in connected mode the client's requests are redirected to the D/SOAP$_c$ component (arrow a). On realizing that the connection to the server is no longer available (arrow b), the D/SOAP$_c$ component redirects all invocations for server objects to its cache (arrow c).

6.3.2 Performance

The performance of the Disconnected Operation Layer was evaluated for 1000 invocations using the two configurations presented above. The results for 1000 invocations in both modes are illustrated in Figure 22.

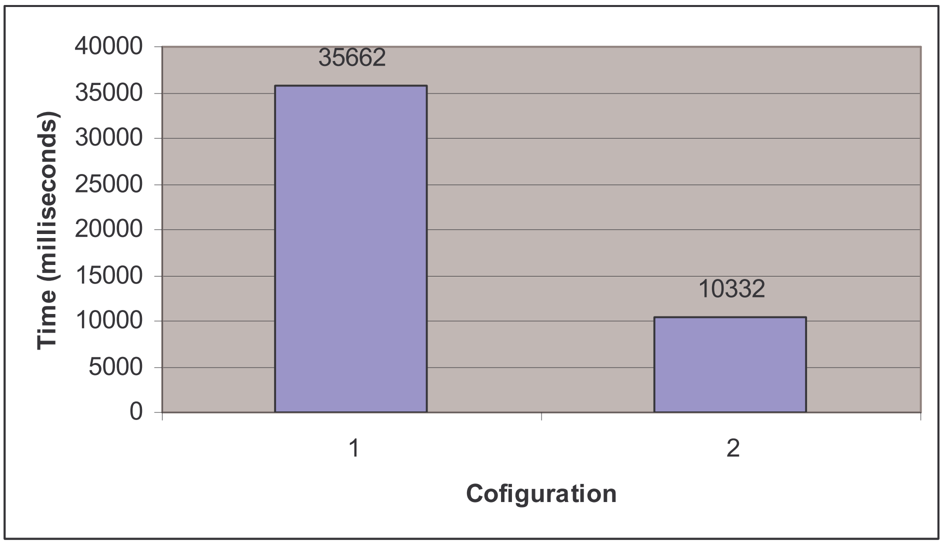


Figure 22: Invocation times for connected and disconnected modes

Configuration 1:

As can be seen from Figure 22 the time taken for 1000 invocations in connected mode is a totally of 35,662 milliseconds. This works out at an average of 35.662 milliseconds per invocation.

Configuration 2:

The total time taken for 1000 invocations in disconnected mode was 10,332 milliseconds, which is an average 10.332 milliseconds per invocation. This is a significant 71.02% reduction on the time taken for an invocation in connected mode and is due to the fact that the requests do not have to be sent over the network to the server. This reduction leads us to conclude that a significant amount of time can be saved by the client operating in disconnected mode.

6.4 Summary

This chapter evaluated the Swizzling Layer in terms of performance, code size and transparency. The Disconnected Operation Layer was evaluated in terms of performance. It was concluded that the introduction of ALICE added a significant increase to the average time of an invocation but each redirection performed by ALICE only caused a modest 11.61% increase in the average time for an invocation. The effect of breaking the chain of forwarding pointers was also examined and it was concluded that having to fall back to the home agent resulted in a very significant increase in the average for invocation times and it was best to try and avoid this scenario. It was also demonstrated that the Swizzling Layer has a reasonable footprint size and level of transparency.

The Disconnected Operation Layer was evaluated operating in both connected and disconnected mode. It was shown that a very significant amount of time could be saved by operating in disconnected mode.

7 Conclusion

7.1 Introduction

This chapter gives a summary of the work completed during this project and the remaining work that needs to be done. Possible future work is also outlined.

7.2 Work Completed

A SOAP client, server and registry architecture has been designed and implemented. This architecture is lightweight and capable of running on J2ME enabled devices. The architecture was used as the basis for this instantiation of ALICE and was designed to satisfy all of the ALICE requirements.

The SOAP Swizzling Layer has been designed and implemented completely. A complete instantiation of the Disconnected Operation Layer has also been completed and a simple application was developed for the purposes of evaluating it. The design for integration into the existing Mobility Layer has also been outlined.

7.3 Remaining Work

The design for integration with the existing Mobility Layer has been completed and the implementation of this design has begun. The implementation of the design involves writing a KNI wrapper interface for some of the Mobility Layer classes and also recompiling the KVM so that the Mobility Layer's C code is linked to it. KNI is a new technology and the process of writing of KNI wrappers is not well documented. This could lead to a steep learning curve for anyone wishing to interface with legacy C code using KNI. Recompiling the KVM is a straight forward process and is well documented.

7.4 Future Work

Future work on this project might include introducing transporting SOAP over HTTP for the SOAP instantiation of ALICE. This would involve investigating the running of a HTTP server on J2ME enabled devices and possibly the implementation of such a server.

7.5 Summary

This chapter concludes this dissertation. In this chapter the work completed and the work still for the project was summarized. Possible future work was also outlined.

8 Bibliography

[1] Mads Haahr. ‘Supporting Mobile Computing in Object-Oriented Middleware Architectures’. PhD Thesis. University of Dublin, Trinity College, Ireland, October 2003.

[2] Object Management Group. The Common Object Request Broker: Architecture and Specification, July 2002.

[3] Sun Microsystems. Java Remote Method Invocation: ‘Distributed Computing for Java’. White Paper.

[4] Raymond Cunningham. ‘Architecture for Location Independent CORBA Environments’. Masters’s thesis, University of Dublin, Trinity College, Ireland, September 1998.

[5] Paul MacSweeney. ‘RMI in a Mobile Environment’. Masters’s thesis, University of Dublin, Trinity College, Ireland, September 2004.

[6] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystky Nielsen, Satish Thatte and Dave Winer. Simple Object Access Protocol (SOAP) 1.1 Technical Report, W3C, May 2000.

[7] W3C. SOAP Version 1.2 Part 1: Messaging Framework, July 2002.

[8] W3C SOAP Version 1.2 Part 2: Adjuncts, July 2002.

[9] James Snell, Doug Tidwell and Pavel Kulchenko. Programming Web Services with SOAP. O'Reilly 2002.

[10] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, Francois Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation 04 February 2004.

[11] Sun Microsystems. Java 2 Micro Edition (J2ME). http://java.sun.com/j2me

[12] John W. Muchow. Core J2ME Technology & MIDP. Prentice Hall. 2002

[13] Dave Winer. XML-RPC Specification. June 1999.

[14] Paul V. Biron, Ashok Malhotra. XML Schema Part 2: Datatypes. W3C Recommendation May 2001.

[15] IBM. Web Services Toolkit for Mobile Devices (WSTKMD). Downloadable documentation. http://www.alphaworks.ibm.com/tech/wstkmd

[16] embedding.net. Online documentation. http://www.embedding.net/eSOAP/

[17] Wingfoot Software. Online documentation. http://www.wingfoot.com/index.jsp

[18] kobjects.org. Online documentation. http://ksoap.objectweb.org/index.html

[19] Robert A. van Engelen. Florida State University. Online Documentation for gSOAP. http://www.cs.fsu.edu/~engelen/soap.html

[20] Jon Ellis, Mark Young. Java Specification Request 172: J2ME web Services Specification. Java Community Process. March 2004