

Overcoming Middleware Heterogeneity in Mobile Computing Applications

Paul Grace

B.Sc. Hons. University of York, 1999

M.Sc. Lancaster University, 2000

**Computing Department
Lancaster University**



**A thesis submitted for the degree of Doctor of
Philosophy**

March 2004

Overcoming Middleware Heterogeneity in Mobile Computing Applications

Paul Grace

B.Sc. Hons. University of York, 1999

M.Sc. Lancaster University, 2000

Computing Department

Lancaster University

A thesis submitted for the degree of Doctor of Philosophy

March 2004

Abstract

Recent technical advances have fuelled the popularity of mobile computing. Mobile devices such as smart phones and personal digital assistants are becoming more commonly used due to the reduction in their size and increase of computational power. In addition, wireless network hotspots (in airports, hotels and commercial outlets) are now beginning to populate the environment. With these advances, new types of mobile applications are becoming available to support users on the move. The mobile environment presents a number of challenges to application developers (including frequent network disconnection and variable bandwidth); therefore mobile middleware platforms have emerged to simplify the development process of distributed mobile applications. However, the range of platforms now available introduces the new problem of middleware heterogeneity, i.e., applications developed upon different types of middleware do not interoperate with one another. Hence, the next generation of mobile computing applications must be developed independently of specific middleware implementation to allow them to continue interoperating in new locations.

This thesis investigates the problem of middleware heterogeneity in the mobile computing environment. The approach taken to solve this problem involves the development of a component-based, higher-level middleware framework (named ReMMoC) that can dynamically adapt its underlying behaviour between different concrete middleware implementations e.g. in one location CORBA is utilised, whereas at the next location SOAP is used. Furthermore, this framework promotes a higher-level programming abstraction based upon the abstract services concepts of the

Web Services Architecture. The ReMMoC framework is evaluated to ensure that middleware transparency is achieved and that applications can be developed that will operate in unknown locations across unpredictable middleware implementation. Inevitably, the ability to overcome heterogeneity comes at the cost of an incurred performance overhead; hence, this thesis also evaluates the impact of this overhead in the domain of mobile computing.

Declaration

The work reported in this thesis has not been previously submitted for a degree, in this or any form.

The design and implementation of the ReMMoC framework was carried out by myself. The framework was implemented upon the OpenCOM component model, which was originally developed at Lancaster; I ported this platform to Windows CE, and enhanced it in order for it to be usable within the project. Finally, the implementation of the service discovery personalities, and the SOAP and IIOP personalities was performed by myself. For all but SOAP, this involved porting open source material to OpenCOM components. However, the implementation of the publish-subscribe binding personality was developed by Thirunavukkarasu Sivaharan.

Acknowledgements

The author wishes to thank the many people who have helped in the production of this thesis. Firstly, my supervisor Professor Gordon Blair, who first encouraged my interest in distributed systems research, and who has provided advice, encouragement and insightful criticism throughout my PhD and in the preparation of this thesis.

Secondly, the members of the Distributed Multimedia Research Group and in particular the Next Generation Middleware Group, who during my time at Lancaster have contributed in numerous ways, offering informed debate and a friendly work environment. Special thanks must go to Georgios Samartzidis, Thirunavukkarasu Sivaharan and Michael Clarke, for their input to the development of software for this thesis. And additional thanks to Geoff Coulson, Fabio Costa, Hector Duran-Limon, Jo Ueyama, Lee Johnstone, Akbar Joolia, Kevin Lee, Paul Okanda, Nikos Parlavantzas, Katia Saikoski, Ben Bappu, Wei Cai, Nelly Bencomo, Maomao Wu and Wai Kit Yeung for thought-provoking discussions and helpful advice.

Finally, I would like to offer my thanks to Bell Labs UK for sponsoring me for the duration of my Ph.D., and thanks in particular to Sam Samuel and Jerry Horton, who helped shape this research and made my time in Swindon a brief but enjoyable experience.

Table of Contents

CHAPTER 1	INTRODUCTION.....	13
1.1	OVERVIEW.....	13
1.2	MOBILE COMPUTING.....	14
1.2.1	<i>Overview</i>	14
1.2.2	<i>Mobile applications</i>	14
1.2.3	<i>Mobile Devices</i>	15
1.2.4	<i>Wireless networks</i>	17
1.2.5	<i>The Challenges of Mobile Computing</i>	18
1.3	MOBILE COMPUTING MIDDLEWARE	19
1.3.1	<i>The Importance of Middleware</i>	19
1.3.2	<i>Styles of Middleware</i>	20
1.3.3	<i>A New Problem</i>	21
1.4	ILLUSTRATING MIDDLEWARE HETEROGENEITY	22
1.5	AIMS	23
1.6	STRUCTURE OF THE THESIS	24
CHAPTER 2	MIDDLEWARE FOR MOBILE COMPUTING	26
2.1	INTRODUCTION	26
2.2	ESTABLISHED MIDDLEWARE	26
2.2.1	<i>Overview</i>	26
2.2.2	<i>Common Object Request Broker Architecture (CORBA)</i>	27
2.2.3	<i>Java RMI Solutions</i>	31
2.2.4	<i>Simple Object Access Protocol (SOAP)</i>	33
2.2.5	<i>Analysis of Enhancements to Established Middleware</i>	34
2.3	ASYNCHRONOUS MIDDLEWARE	35
2.3.1	<i>Overview</i>	35
2.3.2	<i>Asynchronous RPC – “An Early Solution”</i>	36
2.3.3	<i>Tuple Spaces</i>	37
2.3.4	<i>Publish-Subscribe Middleware</i>	40
2.3.5	<i>Analysis of Asynchronous Middleware</i>	45
2.4	DATA SHARING MIDDLEWARE	46
2.4.1	<i>Overview</i>	46
2.4.2	<i>Bayou</i>	46
2.4.3	<i>AdHocFS</i>	47
2.4.4	<i>XMIDDLE</i>	48
2.4.5	<i>Analysis of Data Sharing Middleware</i>	48
2.5	MOBILE AGENTS.....	49
2.5.1	<i>Overview</i>	49
2.5.2	<i>Java-based Mobile Agents</i>	49
2.5.3	<i>Tacoma and Tacoma Lite</i>	50
2.5.4	<i>Analysis of Mobile Agents</i>	51
2.6	SERVICE DISCOVERY	52
2.6.1	<i>Overview</i>	52
2.6.2	<i>Jini</i>	53
2.6.3	<i>Service Location Protocol (SLP)</i>	54
2.6.4	<i>Universal Plug and Play (UPnP)</i>	55
2.6.5	<i>Salutation</i>	56

2.6.6	<i>Service Discovery Protocol (SDP)</i>	56
2.6.7	<i>MARE</i>	57
2.6.8	<i>The Java Enhanced Service Architecture (JESA)</i>	58
2.6.9	<i>Centaurus</i>	58
2.6.10	<i>Analysis of Discovery Protocols</i>	59
2.7	ADAPTIVE MIDDLEWARE	60
2.7.1	<i>Overview</i>	60
2.7.2	<i>Reflective Middleware</i>	60
2.7.3	<i>Policy based Adaptive Middleware</i>	67
2.7.4	<i>Analysis of Adaptive Middleware</i>	71
2.8	CONCLUSIONS.....	72
CHAPTER 3 TACKLING MIDDLEWARE HETEROGENEITY.....		74
3.1	INTRODUCTION	74
3.2	WEB SERVICES ARCHITECTURE.....	74
3.2.1	<i>Overview</i>	74
3.2.2	<i>Analysis of Web Services</i>	77
3.3	WEB SERVICES INVOCATION FRAMEWORK (WSIF).....	78
3.3.1	<i>Overview</i>	78
3.3.2	<i>Analysis of WSIF</i>	80
3.4	MODEL DRIVEN ARCHITECTURE	82
3.4.1	<i>Overview</i>	82
3.4.2	<i>Analysis of Model Driven Architecture</i>	83
3.5	MIDDLEWARE BRIDGES.....	83
3.5.1	<i>Overview</i>	83
3.5.2	<i>Unified Component Meta Model Framework (UNIFrame)</i>	84
3.5.3	<i>Analysis of Middleware Bridges</i>	85
3.6	LOGICAL MOBILITY	86
3.6.1	<i>Overview</i>	86
3.6.2	<i>SATIN</i>	86
3.6.3	<i>Jini</i>	87
3.6.4	<i>Analysis of Logical Mobility</i>	87
3.7	UNIVERSAL INTEROPERABLE CORE	88
3.7.1	<i>Overview</i>	88
3.7.2	<i>Analysis of Universal Interoperable Core</i>	89
3.8	CONCLUSIONS.....	90
CHAPTER 4 TECHNOLOGIES FOR BUILDING A REFLECTIVE FRAMEWORK		91
4.1	INTRODUCTION	91
4.2	COMPONENTS IN REMMOC	93
4.2.1	<i>Overview of Components</i>	93
4.2.2	<i>Investigation of Available Component Models</i>	93
4.2.3	<i>Background on OpenCOM</i>	95
4.3	COMPONENT FRAMEWORKS	97
4.3.1	<i>Overview of Component Frameworks</i>	97
4.3.2	<i>Existing Component framework Models</i>	98
4.3.3	<i>ReMMoC's Component Framework Model</i>	99
4.4	REFLECTIVE MIDDLEWARE FOR MOBILE COMPUTING (REMMOC).....	105
4.4.1	<i>Requirements for the ReMMoC Middleware Framework</i>	105

4.4.2	<i>The Reflective Framework</i>	106
4.5	THE SERVICE DISCOVERY FRAMEWORK	108
4.5.1	<i>Overview</i>	108
4.5.2	<i>The “Cycle and See” Philosophy</i>	109
4.5.3	<i>The Architecture of the Service Discovery Framework</i>	110
4.5.4	<i>Service Lookup Personalities</i>	111
4.5.5	<i>Mirroring the Network Environment</i>	115
4.5.6	<i>New Discovery Protocols</i>	120
4.6	THE BINDING FRAMEWORK	121
4.6.1	<i>Overview</i>	121
4.6.2	<i>The Architecture of the Binding Framework</i>	123
4.6.3	<i>Binding Personalities</i>	124
4.6.4	<i>Integrity Maintenance</i>	129
4.6.5	<i>New binding types</i>	130
4.7	SUMMARY	130
CHAPTER 5 THE ABSTRACT SERVICE PROGRAMMING MODEL ...		132
5.1	INTRODUCTION	132
5.2	THE OVERALL REMMOC ABSTRACTION ARCHITECTURE	133
5.3	THE SERVICE DISCOVERY ABSTRACTION.....	134
5.3.1	<i>Overview</i>	134
5.3.2	<i>The Service Discovery Abstraction</i>	135
5.3.3	<i>Abstract to Concrete Mappings</i>	137
5.3.4	<i>Proof of Concept (Implementation of Mapping Components)</i>	138
5.4	THE ABSTRACT SERVICE BINDING MODEL	139
5.4.1	<i>Overview</i>	139
5.4.2	<i>Abstract Web Services</i>	139
5.4.3	<i>The Abstract Binding API</i>	142
5.5	MAPPING ABSTRACT OPERATIONS TO CONCRETE COMMUNICATION PARADIGMS.....	147
5.5.1	<i>Introduction</i>	147
5.5.2	<i>Mapping Abstract Operations to Remote Method Invocation</i>	148
5.5.3	<i>Mapping to Publish-Subscribe</i>	151
5.5.4	<i>Implementation of mapping components</i>	155
5.6	MANAGING ADAPTATION OF THE BINDING FRAMEWORK	159
5.6.1	<i>Overview</i>	159
5.6.2	<i>Rules for Configuration based upon Binding Information</i>	159
5.6.3	<i>Rules for Configuring Client and Server Side Bindings</i>	160
5.7	SUMMARY	161
CHAPTER 6 EVALUATION		162
6.1	INTRODUCTION	162
6.2	QUALITATIVE EVALUATION	163
6.2.1	<i>Overview</i>	163
6.2.2	<i>Mobile Scenario</i>	163
6.2.3	<i>Implementing the Scenario</i>	165
6.2.4	<i>Results of ReMMoC’s Operation within Case Studies</i>	174
6.2.5	<i>Analysis of Qualitative Evaluation</i>	179
6.3	QUANTITATIVE EVALUATION	180
6.3.1	<i>Overview</i>	180

6.3.2	<i>Abstract Operation Overhead in ReMMoC</i>	181
6.3.3	<i>Measurements of Coarse-Grained Reflective Operations</i>	185
6.3.4	<i>System Memory Costs incurred when using Reflection</i>	189
6.3.5	<i>Analysis of Quantitative Evaluation</i>	191
6.4	SUMMARY	193
CHAPTER 7 CONCLUSIONS		194
7.1	INTRODUCTION	194
7.2	THESIS OVERVIEW	194
7.3	MAJOR RESULTS	196
7.3.1	<i>Identification of Middleware Heterogeneity in Mobile Computing</i> ...	196
7.3.2	<i>The ReMMoC Approach</i>	197
7.3.3	<i>A Higher-level Middleware Abstraction</i>	198
7.4	OTHER SIGNIFICANT RESULTS	198
7.4.1	<i>The OpenCOM Component Framework Model</i>	198
7.4.2	<i>The “Cycle and See” Philosophy</i>	199
7.4.3	<i>The use of Reflection on Mobile Devices</i>	199
7.4.4	<i>Abstract-to-Concrete Mappings</i>	200
7.5	FUTURE WORK	200
7.5.1	<i>Additional Middleware Personalities</i>	200
7.5.2	<i>Security Component Framework</i>	201
7.5.3	<i>Resource Management Component Framework</i>	201
7.5.4	<i>Web Service Extensions</i>	201
7.5.5	<i>Semantic Service Matching</i>	202
7.5.6	<i>Dynamic Component Downloading</i>	202
7.5.7	<i>Ubiquitous Computing Environments</i>	203
7.6	CONCLUDING REMARKS	203
REFERENCES		204
APPENDIX A	COMPONENT FRAMEWORK META INTERFACES....	215
APPENDIX B	EXAMPLE XML COMPONENT CONFIGURATION ...	217
APPENDIX C	WSDL OF APPLICATION SERVICES	220

Table of Figures

Table 1.1 Example mobile computing applications	15
Figure 1.1 Heterogeneous mobile application services in two locations	22
Figure 2.1 The relational model specified in CORBA	27
Figure 2.2 The Object Request Broker	28
Figure 2.3 The Java RMI architecture	32
Table 2.1 Challenges of mobile computing met by established middleware	35
Figure 2.4 Filtering tuples from one tuple space to another [Wade99]	37
Figure 2.5 Transiently shared tuple spaces [Murphy01]	39
Figure 2.6 Event notification in CEA: a) direct and b) mediated [Bacon00]	41
Figure 2.7 Traffic light application demonstrating proximity group [Meier02]	43
Table 2.2 Challenges of mobile computing met by asynchronous middleware	45
Table 2.3 Challenges of mobile computing met by data-sharing middleware	49
Figure 2.8 Maintaining state in Tacoma using folders, briefcases and file cabinets	51
Table 2.4 Challenges of mobile computing met by agent-based middleware	52
Figure 2.9 Service discovery in SLP: (a) using a directory agent and (b) without using a directory agent	55
Figure 2.10 Number of messages to discover four services in different protocols	57
Table 2.5 Challenges of mobile computing met by service discovery middleware	59
Figure 2.11 The Meta-Space structure of OpenORB	62
Figure 2.12 The component frameworks of Open ORB	63
Figure 2.13 Reifying the dynamicTAO structure [Roman01]	64
Figure 2.14 dynamicTAO Components [Roman01]	65
Figure 2.15 Architecture to support adaptive applications	69
Figure 2.16 A CHARISMA application profile [Capra02]	70
Table 2.6 Challenges of mobile computing addressed by adaptive middleware	72
Figure 3.1. The elements of WSDL [Newcomer02]	75
Figure 3.2 Web Services Technologies [Booth03]	77
Figure 3.3 Example EJB binding in WSDL	79
Figure 3.4. The WSIF Client Framework	80
Figure 3.5 OMG's Model Driven Architecture [Miller01]	82
Figure 3.6 Architecture of the Unified Component Interoperability framework	85
Figure 3.7 Capabilities in a SATIN application	87
Figure 3.8 UIC Personalities [Roman01]	89
Figure 4.1 The OpenCOM architecture	96
Figure 4.2 An OpenCOM component framework.	100
Figure 4.3 Composition of Component Frameworks	101
Table 4.1 Operations for inspection of the internal CF structure	102
Table 4.2 Operations for dynamic reconfiguration	102
Figure 4.4 The IAccept Interface	103
Figure 4.5 Implementation of an OpenCOM component framework	104
Figure 4.6 XML description of a component configuration	105
Figure 4.7 The top level architecture of ReMMoC	107
Figure 4.8 The Service Discovery Component Framework Architecture	110
Table 4.3 Components of the SLP Lookup Personality	112
Figure 4.9 OpenCOM configuration for SLP lookup personality	113
Figure 4.10 The IUPnP Interface	114
Table 4.4 UPnP components	114
Figure 4.11 UPnP lookup component personality.	115
Figure 4.12 IDiscoveryDiscovery Interface	116

Figure 4.13 Discovery protocol tests	117
Figure 4.14 Part of the XML description for the SLP personality	118
Figure 4.15 Pseudo code for XML based configuration of personalities	119
Figure 4.16 IDL definition of IServiceDiscoveryCFAAdmin interface	121
Figure 4.17 The binding component framework architecture	124
Table 4.5 Component elements of the IIOP client personality	125
Figure 4.18 IIOP client binding personality	125
Table 4.6 Additional IIOP server components	126
Figure 4.19 IIOP Server side binding personality	126
Table 4.7 Components of SOAP RPC client personality	127
Figure 4.20 Component configuration for SOAP RPC client personality	127
Table 4.8 Component descriptions for subscriber personality	128
Figure 4.21 Component configuration of subscriber personality	128
Figure 4.22 Component configuration of publisher	129
Figure 5.1 The ReMMoC programming model	133
Figure 5.2 IDL definition of IServiceDiscovery interface	135
Figure 5.3 The ServiceReturnEvent data structure	135
Figure 5.4 Abstract to concrete service discovery architecture	138
Figure 5.5 ReMMoC's role in the Web Services Architecture	141
Figure 5.6 An abstract WSDL description for a sport news service	142
Figure 5.7 Invoking remote WSDL operations (RequestResponse and OneWay)	143
Figure 5.8. The WSDL data structure	144
Figure 5.10 Elements of a WSDL operation	148
Figure 5.11 Elements of a Remote Method Invocation	149
Figure 5.12 Mapping abstract Request-Response to RMI	149
Figure 5.13 Mapping abstract One-Way to RMI	150
Figure 5.14 Mapping abstract Solicit-Response to RMI	151
Figure 5.15 Mapping abstract Notification to RMI	151
Figure 5.16 General elements of a produced Publish-Subscribe event	152
Figure 5.17 Mapping Request-Response to Publish-Subscribe	153
Figure 5.18 Mapping One-Way to Publish-Subscribe	153
Figure 5.19 Mapping Solicit-Response to Publish-Subscribe	154
Figure 5.20 Mapping Notification to Publish-Subscribe	155
Figure 5.21 The IMap and IServiceCallback interfaces	155
Figure 5.22 The IIOP map component	156
Table 5.1 URL formats for binding types	160
Figure 6.1 The evaluation scenario	164
Figure 6.2 Implementation of the CORBA chat application	166
Figure 6.3 Implementation of chat application using publish-subscribe	167
Table 6.1 Discovery protocol advertisements of application services	168
Figure 6.4 Screen shots from the stock quote client application	169
Figure 6.5. Code extracts from the stock quote application	169
Figure 6.6 Screen shot from the jukebox client application	170
Figure 6.7 Code fragments of the jukebox client application	171
Figure 6.8 Screen shots from chat client application	172
Figure 6.9 Code fragments of the chat client application	172
Figure 6.10 Illustration of stock application behaviour across changing locations	176
Figure 6.11 Illustration of the jukebox application behaviour across changing locations	177

<i>Figure 6.12. Illustration of the chat application behaviour across changing locations</i>	179
<i>Figure 6.13 Comparison of service invocations</i>	182
<i>Figure 6.14 Abstract-to-concrete mapping costs during service invocation</i>	183
<i>Table 6.2 Cost of dynamic reconfiguration</i>	184
<i>Table 6.3 Component insertion measurements</i>	186
<i>Table 6.4 Binding framework configuration measurements</i>	186
<i>Table 6.5 Detailed binding framework configuration measurements</i>	187
<i>Table 6.6 Service Discovery framework configuration measurements</i>	188
<i>Figure 6.15 Performance of dynamic reconfigurations in discovery framework</i>	188
<i>Table 6.7 Memory footprint sizes of component configurations in ReMMoC</i>	190

1.1 Overview

Improving mobile device and wireless network technology has fuelled the popularity of *mobile computing* over the last decade. Handheld devices have become smaller, more computationally powerful and their usage is now commonplace. Wireless networks proliferate the environment we live in; high-speed wireless networks are available in particular hot spots such as hotels, coffee bars, university campuses and office buildings. Meanwhile, lower speed wireless networks cover wider geographical areas, ensuring mobile users can remain permanently connected. Consequently, new application types are being developed to exploit these technologies and provide novel methods of interaction between mobile device users and their environment.

Developing distributed mobile applications to operate across wireless networks is a complex task. The mobile environment is hampered by problems of weak connection, poor network *Quality of Service* (QoS) and changing context (e.g. device location). *Middleware* has proven a successful technology in supporting distributed computing across wired networks, overcoming the problems of platform heterogeneity and simplifying the development process. Hence, a large body of research has been carried out to examine how middleware should support distributed mobile applications and overcome the limitations of wireless networks [Mascolo02]. However, the heterogeneity that exists between different middleware solutions in turn generates a new problem. These solutions do not interoperate with one another; applications are unable to interact with different middleware implementations.

This thesis investigates the problem of *middleware heterogeneity* in mobile computing environments and examines how an adaptive middleware framework can overcome this problem. The remainder of this chapter is structured as follows. Section 1.2 provides an introduction to the facets of the mobile computing domain and section 1.3 describes the area of mobile computing middleware. Section 1.4 introduces the particular problem of middleware heterogeneity that this thesis aims to address. Finally, section 1.5 describes the main aims of the research, and section 1.6 documents the overall thesis structure.

1.2 Mobile Computing

1.2.1 Overview

Mobile computing is characterised by users carrying portable computational devices that interact with shared infrastructures independent of their physical location; this allows intercommunication between people and continuous access to networked services [Forman94]. This section first examines the applications that drive the research requirements of mobile computing. Then, in turn, the improving technological aspects of mobile devices and wireless networks are presented.

1.2.2 Mobile applications

Until the emergence of public-domain wireless networks, mobile computing was confined to performing desktop-style applications (e.g. word processors and spreadsheets) on the move. However, the services provided by next generation mobile applications are now explicitly linked to the mobility of the user. These applications seek to enhance user experience and productivity as they go about day-to-day tasks. Currently, distributed mobile applications fall into two categories:

- *Location-based Services*. In these application types, the service is moulded to the current location of the mobile device.
- *Communication Services*. Traditional distributed communication applications that operate across wireless networks; for example, e-mail, chat, mobile gaming, co-operative work and video messaging.

Example location-based mobile applications are illustrated in table 1.1; these serve to identify the activities that mobile users perform, rather than exhaustively document all types of mobile applications. The services provided include: entertainment, information, commerce and healthcare. This demonstrates the diversity of mobile applications, which will only extend further in the future as visions of how they can improve current environments are identified.

In all of the examples, users interact with mobile applications using a mobile device. Alternatively, *Ubiquitous Computing* [Weiser91] is a field of computer science that aims to make the computational device disappear and, as the user moves around, the environment responds to meet their requirements. For example, in Flump [Finney96]

personalised web content is exhibited on wall-based displays close to the user. Ubiquitous computing is closely related to mobile computing, and offers alternative application scenarios (i.e. the environment reacts to mobile users); however, it has different middleware requirements than mobile computing scenarios (i.e. there may be no mobile device for middleware to operate upon) and hence, these application scenarios are not addressed by this thesis.

Category	Description	Examples
Tourist Guide	Dynamically changes content to help tourists navigate through and be informed about nearby points of interest.	Guide [Davies99], CyberGuide [Long96]
Shopping Assistant	Guides shoppers through stores, helps locate items and informs them of special offers.	DealFinder [Chan01], Shopping Assistant [Asthana94]
Weather	Sends local weather updates to the mobile device.	[Jacobsen99]
Traffic Congestion	Monitors current traffic levels on local roads, warning the mobile user they are approaching congestion and suggests an alternate route.	Traffic congestion [Cole03]
Reminder	Reminds the user what to do at a particular time, when they reach a new location or they are co-located with another user.	ComMotion [Marmasse00], CybreMinder [Dey00]
Conference Assistant	Supports conference attendee by suggesting presentations to attend based upon preferences and provides extra information to users located at each talk.	Conference Assistant [Dey99]
Healthcare	Provides doctors with information such as medical records and changes in patient status, whatever their location.	[Mitchell00]

Table 1.1 Example mobile computing applications

1.2.3 Mobile Devices

A key factor in the popularity of mobile computing lies with the end system; this should be lightweight, conservative with power, and easy to use. Currently available mobile devices fall into categories based upon differences in physical size, screen size,

memory capacity and processor speed. These categories are: laptops, tablet PCs, handheld PCs, smart phones and wearable computers. They share the common characteristic of limited battery life, requiring the mobile device to be frequently recharged.

Laptop computers and *Tablet PCs* provide the closest in terms of performance to desktop machines, with similar processor and memory capacity including a hard disk drive providing large amounts of secondary storage. They have the largest screen sizes (ranging between 10” and 17”) and hence are physically the largest and heaviest mobiles (typically weighing between 1Kg and 2.5Kg). Laptops primarily use a keyboard and touchpad as input devices; however, these are difficult to use by “moving” users. Therefore, tablet PCs provide pen-based input through a touch screen.

Handheld PCs offer increased portability; the average screen size is between 3” and 4”. This means that they are significantly smaller and weigh much less than Laptops and Tablets (e.g. 150g to 250g). Although some models include a miniature keyboard, the primary input mechanism is by touch screen. Whilst still computationally powerful, they currently lag behind laptops in terms of processor performance and memory capacity. Typical devices have between 8 to 64Mb of RAM with no large secondary storage. A *Smartphone* is a mobile telephone that combines traditional cellular voice connectivity with handheld PC capabilities. Typically they have similar physical and performance characteristics to handheld PCs, however they provide user input through keys on the handset rather than a touch screen.

In contrast to the described mobile devices, wearable computers are being researched and developed. These aim to remove the handheld carrying of a portable device and support the vision of ubiquitous computing. For example, IBM has developed a wristwatch that runs the Linux operating system, has a touch screen display and Bluetooth wireless connectivity [Narayanaswami00]. Head-mounted sets (e.g. POMA from Cybernaut [POMA03]) demonstrate alternative output displays; for example, a 1” LCD screen that sits in front of the user’s eye.

1.2.4 Wireless networks

Wireless networks allow mobile devices to communicate with one another, connect to the Internet or access network services. Advances in wireless networking technologies have provided solutions for local area (LAN) and wide area (WAN) coverage. Wireless LANs cover small geographic areas (e.g. rooms, buildings, city centres etc) and operate in either *infrastructure* or *ad-hoc* fashion. In infrastructure mode, all network traffic passes through a fixed *access point*, whereas ad-hoc networking involves routing the traffic between the local devices in a peer-to-peer fashion. Alternatively, wireless WANs are infrastructure based (devices connect to and roam between fixed base stations) and cover larger areas ranging from whole cities to continents. This section introduces the key technologies currently in widespread use; for a more detailed survey of wireless networks see [Friday96], [Lin01], [Stallings02].

The IEEE 802.11 working group for local wireless networking [IEEE03] has proposed three separate standards: 802.11b, 802.11a and 802.11g. The most popular at present, 802.11b, is a radio frequency based technology that uses the 2.4 GHz microwave band designated for low-power unlicensed use and provides a theoretical maximum bandwidth of 11Mbps. Furthermore, both infrastructure and ad hoc operating modes are available. 802.11a uses the 5GHz band and provides up to 54 Mbps bandwidth; however, it is not interoperable with 802.11b. In contrast, 802.11g is backward compatible with 802.11b, uses the same frequency and offers an increased bandwidth (20+ Mbps).

Alternative ad-hoc local area network solutions use either infrared or radio frequency techniques. IrDA [IrDA01] is an example of an infrared LAN; its main use is the wireless connection of devices that would normally use cables. IrDA is a point-to-point, narrow-angle (30-deg), data-transmission standard designed to operate over a distance of 0 to 1 m and at speeds of 9.6 kb/s to 16 Mb/s. However, once an IrDA device is connected to the LAN, it must remain relatively stationary to maintain the connection. On the other hand, Bluetooth [Bluetooth99] is a radio frequency LAN that provides short-range, point-to-multipoint voice and data transfer and can transmit through solid, non-metal objects. The nominal range of a Bluetooth device is 10 cm to 10 m.

Examples of commonly used wide area networks are from the GSM family. GSM (Global System for Mobile Communication) [Rahnema93] is a circuit-switched technology that can transmit data at 9.6 kbps. However, using packet-based technologies has led to higher data throughput. GPRS (General Packet Radio Service) [Rysavy98] can transmit at a maximum of 171 Kbps and third generation networks e.g. UMTS (Universal Mobile Telecommunications System) [Muratore00] can send at up to 2 Mbps.

The collection of wireless technologies presented allows a user to be continuously connected; that is, they may *roam* from location to location using whichever network technology is available. For example, the user may be connected to the network while within a building covered by an 802.11b network, when they then move outside of the building the device communicates by using a GSM network. This concept is known as *overlay networks* and research has examined methods to make the transition seamless [Brewer98].

1.2.5 The Challenges of Mobile Computing

The following list describes some of the key challenges of mobile computing [Forman94][Satyanarayanan96b]; mobile middleware aims to address these problems to better support the development of distributed mobile applications.

1. **Disconnection.** Mobile devices are frequently disconnected from the network (*weak connection*); for example, when handing over to a new network, or when the device moves out of range of wireless coverage.
2. **Low Bandwidth.** The bandwidth of wireless networks can be low, particularly in wireless wide area networks, and when there are many users in a wireless cell.
3. **Variable Bandwidth.** The bandwidth available to a device can change dramatically, e.g. when more users use a wireless cell it will reduce. Furthermore, it can increase when the user moves from a low speed wide area network to high-speed local area network.
4. **Address Migration.** Existing applications send packets to a fixed network address. However, the “local” address of a mobile device changes as it moves between networks. Therefore, messages must be routed to and from this moving device.

5. **Low Power.** Mobile devices rely on a finite energy supply that eventually needs to be recharged.
6. **Small Storage Capacity.** The memory capacity of mobile devices is poor relative to fixed devices.

Additional properties of mobile environments that offer further challenges to mobile computing are: devices with low computational processing capabilities, high network latency and frequent loss of data packets transmitted across wireless networks.

1.3 Mobile Computing Middleware

1.3.1 The Importance of Middleware

Middleware is defined as “*a layer of software residing on every machine, sitting between the underlying operating system and the distributed applications, whose purpose is to mask the heterogeneity of the co-operating platforms and provide a simple, consistent and integrated distributed programming environment*” [Coulouris00]. Middleware has proved a successful technique in fixed networks for overcoming heterogeneity and integrating existing legacy systems. Typically, heterogeneity applies to networks, computer hardware, operating systems and programming languages. In addition, middleware offers a distributed programming environment that eases development and makes *transparent* particular aspects of distribution (e.g. *Access Transparency* makes local and remote operations identical). Finally, middleware upholds the concept of *Open Distributed Processing* (ODP) i.e. distributed systems can be extended and re-implemented, because key software interfaces are made public.

Well-established middleware standards for fixed networks are now in place, these include: Common Object Request Broker Architecture (CORBA) [OMG95], SOAP [Box00], Distributed COM (DCOM) [DCOM96], Enterprise Java Beans (EJB) [Monson-Haefel00] and Java Remote Method Invocation (Java RMI) [Sun97]. However, these are demonstrably inappropriate for the mobile domain. They have heavyweight implementations unsuited to memory-constrained devices [Roman01]. Their operation across unpredictable wireless networks is poor [Haahr00][Liljeberg97][Campadello00]; for example remote object invocations fail

during disconnection. Furthermore, their fixed black-box implementations whose underlying structure and behaviour is hidden from the programmer cannot be altered at run-time to cope with the changes that occur in the mobile environment e.g. fluctuating network QoS [Blair01][Seitz98]. Therefore, domain-specific middleware has emerged to meet the demands of mobile computing. These contrast greatly in style and implementation and are introduced in turn in the following section.

1.3.2 Styles of Middleware

The properties of wireless networks means that mobile devices may become disconnected involuntarily, or otherwise choose to become disconnected to save resources such as battery power. Furthermore, network QoS fluctuates, error rates are high and packets are lost. These characteristics have proven a driving factor in the initial development of middleware platforms for this domain. Different techniques and middleware paradigms have been developed to address these problems:

- *Asynchronous communication paradigms.* To resolve the problem of weak connection, communication mechanisms that do not rely on the sender and receiver being coupled in time and space have emerged. Examples are tuple spaces [Davies98][Murphy01] and publish-subscribe systems [Meier02][Cugola01]. Mobile clients transmit and receive information only whilst they are connected to the network.
- *Adaptive Middleware.* Fixed black-box middleware implementations cannot be altered at run-time to cope with changes that occur in the mobile environment. Therefore, adaptive middleware solutions exist that are *configurable* and dynamically *reconfigurable* to enable the platform to respond to changes in its environment and maintain the best level of operation under current conditions [Capra02][Blair01].
- *Established Middleware Enhancements.* Alternatively, other projects have extended traditional standards based solutions to make them effective over wireless networks. These include making their implementation smaller to fit on to resource constrained devices [Roman01][Klefstad03], or making their communication mechanism operate more effectively over wireless networks [Seitz98][Haahr00].

- *Mobile Agents.* The agent paradigm [Johansen95][Lange98], where executable code moves from host to host, is well suited to mobile computing. Client code can move to the server and perform all communication locally. With scarce bandwidth this limits network communication and reduces the possibility of failure due to partition or disconnection. Furthermore, moving the logic and communication processing to a more powerful server host reduces the load on resource-constrained devices.
- *Service Discovery Solutions.* An important element of mobile computing is the ability to discover what services are available at a particular location. A simple example of this is a room that has a service available to control the light switch; a person who then enters the room with a PDA can discover the service and switch the lights on and off using their handheld. There are a number of existing service discovery technologies currently available e.g. Jini [Arnold99], Service Location Protocol (SLP) [Veizades97], Universal Plug and Play (UPnP) [Microsoft00b] and Salutation [Salutation98].

1.3.3 A New Problem

We have seen above that the different solutions introduce the problem of middleware heterogeneity; they offer incompatible communication paradigms, including: remote method invocation, publish-subscribe, message-oriented, agents and tuple spaces. Furthermore, implementations of individual paradigms differ, e.g. SOAP and IIOP for remote method invocation. Similarly, different service discovery protocols do not interoperate, and with new technologies emerging to better support discovery in mobile environments (e.g. JESA [Preuss02] & Centaurus [Kagal01]) and across wireless ad-hoc networks (e.g. SDP in Bluetooth and Salutation Lite) this problem will become worse.

In reality, the primary goal of current mobile middleware is to support distributed programming. They only partially solve the heterogeneity problem, in scenarios where implementations of the middleware can be guaranteed to reside on every device. However, the scenario in the following section demonstrates applications where the user enters a new location with unknown middleware implementations.

1.4 Illustrating Middleware Heterogeneity

In this section, a mobile computing scenario illustrates how middleware heterogeneity exists in mobile environments. In this example, three application services are available to mobile users at two different locations. Instances of each service are implemented using different types of middleware and advertised using contrasting service discovery protocols. Application 1 is a *mobile sport news* application, whereby news stories of interest are presented to the user based on their current location. Application 2 is a *jukebox* application that allows users to select and play music on an audio output device at that location. Finally, application 3 is a *chat* application that allows two mobile users to interact with one another.

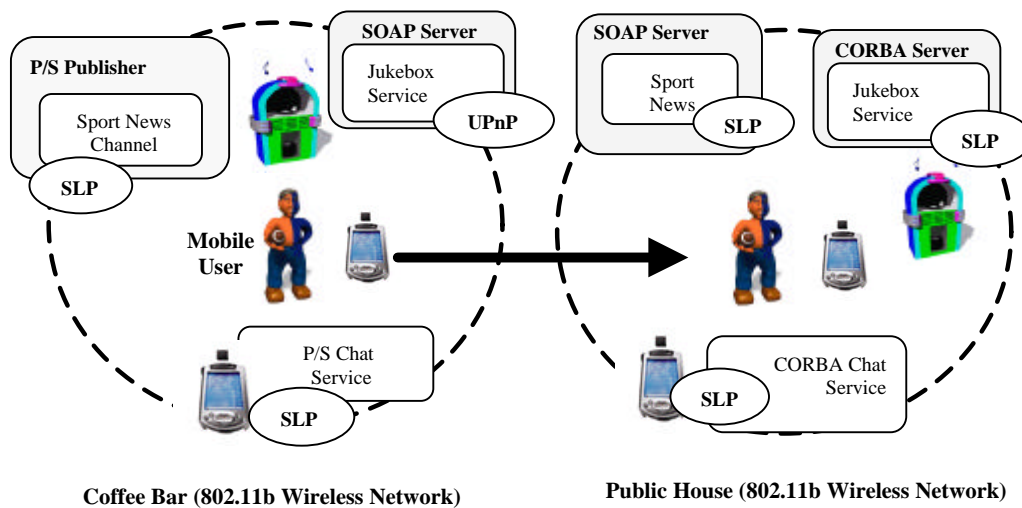


Figure 1.1 Heterogeneous mobile application services in two locations

Figure 1.1 illustrates the two locations (a coffee bar and a public house) in the session of a mobile user. At each location the same application services are available to the user, but their middleware implementations differ. For example, the Sport News service is implemented as a publish-subscribe channel at the coffee bar and as a SOAP service in the public house. Similarly, the chat application services and jukebox services are implemented using different middleware types. The service discovery protocols are also heterogeneous, i.e. the services available at the public house are discoverable using SLP and the services at the coffee bar can be found using both UPnP and SLP. For example, at the coffee bar the jukebox application must first find

its corresponding service using UPnP and then use SOAP to control functionality. When it moves to the public house, SLP and CORBA must be used. Given scenarios of this type a mobile middleware platform should be reconfigurable to interact with different middleware types and utilise different service discovery protocols. In turn, this will allow the development of mobile applications independently of fixed platform types whose properties are unknown to the application programmer at design time.

1.5 Aims

The main aim of this thesis is to investigate and overcome the problem of middleware heterogeneity in the mobile computing environment. That is, the goal is to develop a higher-level adaptive middleware framework, which allows mobile applications to be developed independently of concrete middleware implementations. Furthermore, the thesis investigates how to provide a suitable model for abstracting over the different communication paradigms presented by heterogeneous middleware implementations.

More specifically, the research takes the following approach:

- An investigation of the state of the art in middleware platforms that support mobile applications, including an analysis of how effective their techniques are to overcome the challenges of the mobile environment. Additionally, how these solutions add to the problem of middleware heterogeneity is examined.
- An evaluation of evolving standards-based approaches to the integration of established middleware, exploring the techniques employed to produce the required middleware transparency.
- The production of an adaptive middleware framework that resides on mobile client devices, which overcomes the problems of middleware heterogeneity in mobile environments.
- The production of a higher level distributed programming abstraction that addresses the differences in individual middleware communication paradigms, and provides middleware transparency.
- The implementation of a mobile scenario that evaluates the effectiveness and performance of the higher-level middleware framework in overcoming heterogeneity.

- An evaluation of the appropriateness of reflective middleware for mobile devices. The goal is to demonstrate that a technique often criticised for incurred overhead can operate effectively in the resource constrained domain.

This thesis does not address a number of related aspects, which the author considers important to the domain of mobile computing middleware. Firstly, the thesis offers no new solutions to the original challenges of mobile computing (e.g. problems of weak connection and poor QoS); rather this is left to the core middleware implementations that are encompassed by the framework. Secondly, resource management allows maximum utilisation of device resources (e.g. battery power), however this is left to future work. Finally, the research addresses traditional mobile computing scenarios, where mobile applications operate upon a mobile device; although the author believes the approach to be suitable for emerging application domains such as ubiquitous computing, smart home environments and the Grid, these are not evaluated and are left to future work.

1.6 Structure of the thesis

The following two chapters present a state of the art investigation in the related areas of research to this thesis. Chapter 2 surveys mobile and adaptive middleware platforms, examining closely the paradigms that each utilises and what challenges of mobile computing are addressed. Chapter 3 looks at some of the solutions to middleware integration and overcoming middleware heterogeneity.

Chapters 4 and 5 document the design and implementation of the adaptive middleware framework. Chapter 4 describes the underlying component model and reflective architecture, which provides the capabilities to interact with both heterogeneous middleware and service discovery technologies. Chapter 5 then describes the design of the platform's programming model that presents middleware transparency to the developer.

Chapter 6 follows with an evaluation of the proposed platform for overcoming heterogeneity. Firstly, investigating qualitatively if the platform meets the requirements of demonstrated mobile application scenarios, populated by

heterogeneous middleware. Furthermore, quantitative measures that evaluate the practicality of the solution in the mobile domain are documented. Finally, chapter 7 highlights the major results and contributions of the thesis, along with a discussion of future directions for this work.

2.1 Introduction

The primary goals of middleware are two-fold: 1) to mask heterogeneity of networks, end-systems, operating systems and programming languages, and 2) to provide a simple, integrated distributed programming model. Established middleware solutions, including CORBA [OMG95] and Java RMI [Sun97], have proved successful for business applications, supporting the integration of legacy systems. Current middleware research is now examining how middleware can benefit wider application areas e.g. mobile computing, multimedia, E-Science, real-time computing, programmable networking and peer-to-peer computing. This chapter focuses on the state of the art in the domain of mobile computing and adaptive middleware.

The previous chapter identified the domain specific problems faced by mobile application developers. A wide variety of middleware has been produced to overcome these issues and support the development of distributed mobile applications. These solutions can be separated into categories based upon the middleware functionality provided e.g. fixes to established middleware (promoting synchronous communication), asynchronous middleware, service-discovery, mobile code, data sharing and adaptive middleware. In the following sections, key examples of individual implementations of these paradigms are described and their effectiveness in overcoming the challenges of mobile computing, described in section 1.2.5, is analysed.

2.2 Established Middleware

2.2.1 Overview

CORBA [OMG95], .NET [Microsoft00], SOAP [Box00], DCOM [DCOM96], Enterprise Java Beans [Monson-Haefel00] and Java RMI [Sun97] are examples of established middleware in the fixed network domain. However, these have originally been identified as unsuitable for use in the field of mobile computing for two reasons: 1) the core communication mechanisms are synchronous, which are prone to failure due to disconnection in unpredictable wireless networks [Mascolo02], and 2) they

consist of heavyweight implementations, which exhaust mobile devices of their limited memory resources [Roman01] e.g. the static memory footprint of a CORBA ORB implementation (Orbacus 4.0.5) is approximately 8 Megabytes. Nonetheless, these platforms remain important in the mobile computing domain; they are well used and understood, and allow interoperation with a catalogue of pre-existing fixed network services. Therefore, research has examined methods to make these middleware technologies operate more effectively in wireless environments. The following sections describe enhancements to CORBA, Java RMI and SOAP for mobile computing.

2.2.2 Common Object Request Broker Architecture (CORBA)

Background

The Object Management Group (OMG) has defined a standard distributed open systems framework named the Common Object Request Broker Architecture (CORBA) [OMG95] to address the problems of developing portable distributed applications for heterogeneous systems. CORBA presents an *object model* and a *relational model* to specify how distributed objects interact. Within the object model, an object is an entity that provides one or more services that can be requested by a client through well-defined, strongly typed interfaces. Interfaces are defined in IDL (Interface Definition Language), which provides a language independent method to define the structured data types and operation signatures clients can communicate through.

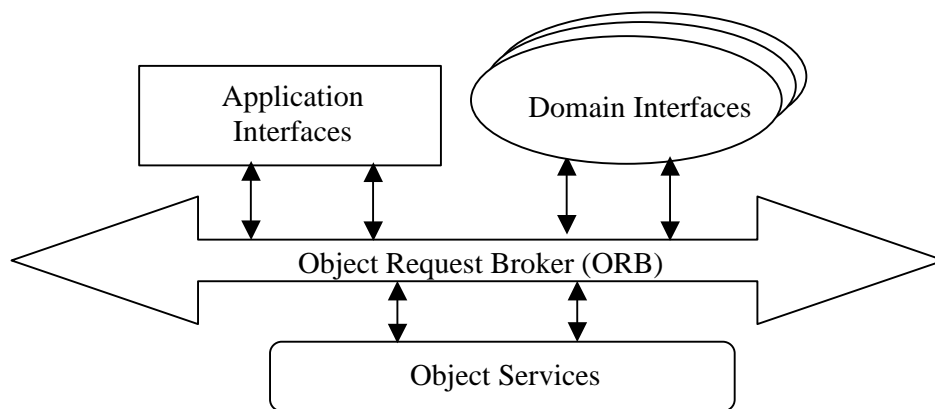


Figure 2.1 The relational model specified in CORBA

The relational model describes the categories of interfaces; figure 2.1 illustrates these categories that are conceptually linked by the Object Request Broker (ORB). *Object Services* are horizontally oriented interfaces used by many applications; for example, the OMG Naming Service provides references to objects that applications intend to use. *Domain interfaces* provide similar roles to object services, but are domain specific or vertically oriented e.g. healthcare applications. Finally, *application Interfaces* are developed specifically for newly created applications.

The fundamental component of CORBA is the ORB (illustrated in figure 2.2); this allows clients to transparently invoke the operations of objects hosted remotely. The low level mechanism is a synchronous Remote Procedure Call (RPC). The architecture allows both static and dynamic invocation of these requests. In the static approach, an interface description is translated into *stubs* and *skeletons* that are compiled into the application. A stub is a client side function that allows a remote invocation to be made via a local call. Similarly, the skeleton is a server side function that allows a request invocation to be received and dispatched to the appropriate object implementation. Dynamic invocation involves the construction of CORBA requests at run time. The dynamic skeleton interface accepts requests for which it has no skeletons, inspects its contents and invokes the object and method it is targeted for.

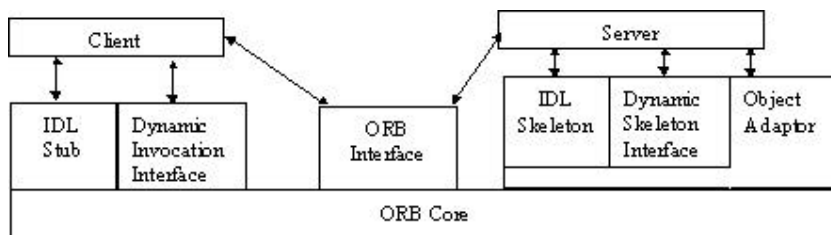


Figure 2.2 The Object Request Broker

The General Inter-ORB Protocol (GIOP) specification defines the Common Data Representation (CDR) for encoding method calls and the message formats transmitted during sessions. GIOP is a generic protocol that guarantees interoperability between ORB implementations from different vendors; GIOP is mapped onto different underlying transport protocols. For example, the Internet Inter-ORB Protocol (IIOP) is a specialised mapping of GIOP to TCP, the Internet transport layer.

Employing CORBA in a mobile environment is subject to problems. CORBA is a large standard and implementations will not fit on devices with limited resources; furthermore, they do not provide customisation tools that would select only the required functionality [Roman01]. In addition, CORBA specifies that a continuous connection be maintained throughout an invocation, which cannot be guaranteed given the properties of wireless networks. Initiatives to address these problems are now described in turn.

ALICE

The Architecture for Location Independent CORBA Environments (ALICE) [Haahr00] allows CORBA objects executing on mobile devices to interact transparently with objects residing upon fixed hosts. They have identified that using a full ORB on a mobile device is infeasible and hence propose that a subset, the IIOP protocol (the minimum protocol necessary to transfer invocations between ORBs), is suitable for mobile CORBA. The ALICE architecture is composed of Mobile Gateways, to which every mobile host connects; these gateways act as proxies, transferring invocations between mobile and fixed hosts. When a host handovers from one mobile gateway to another, ALICE ensures that all open client-server connections are transparently retained allowing invocations to complete. In particular, ALICE addresses the problems of disconnection, address migration and limited memory capacity. Although no changes need to be made to fixed ORBs, the solution relies on gateways being available in the network environment, which cannot be guaranteed across all wireless networks.

Notably, the ALICE framework has been extended to become general from CORBA; indeed ALICE is now short for Architecture for Location Independent Computing Environments [Biegel02], and as such can be applied to other RMI based middleware implementations including Java RMI and SOAP. This new framework extracts specific layers of mobile functionality that can be re-used per RMI implementation. A mobility layer handles connections between the mobile host and the local gateways. A swizzling layer translates server references to refer to mobile gateways (supporting server mobility), and a disconnection layer handles mobile host disconnection by caching server functionality locally.

DOLMEN

The DOLMEN project [Liljeberg97] seeks to overcome the wireless access and terminal mobility problems of CORBA. The solution uses CORBA bridging to connect the ORB on the mobile terminal to the fixed ORB in the core network domain. The bridge consists of two half bridges: the Mobile Domain Distributed Processing Environment Bridge (MDBR) and the Fixed Domain Distributed Processing Environment Bridge (FDBR). As the mobile host changes location it connects to different FDBRs; mobility functions built into the two bridges then enforce location transparency and invocations are mapped to the correct object. The two bridges form a closed interoperability domain. DOLMEN takes advantage of this by using a special Light-Weight Inter-ORB protocol (LW-IOP) for communication. The protocol is based upon efficient message formats and compressed data representation for object communication over a wireless link, ensuring minimum bandwidth is consumed. Like ALICE, this solution tackles address migration and disconnection problems using a proxy based approach. However, it also improves the operation of CORBA over low bandwidth networks.

Wireless Access and Terminal Mobility in CORBA

The OMG identified the need to support wireless access in CORBA and issued a Request For Information (RFI) in June 1998 [OMG98]. This concluded with a new adopted standard as of March 2003 [OMG03], which is heavily influenced by the features of DOLMEN. Its aim is to avoid modification to non-mobile nodes in order for them to interoperate with objects hosted by mobile terminals. The architecture consists of three domains. Firstly, the terminal domain is the mobile terminal that hosts an ORB and a terminal bridge. Secondly, the visited domain hosts one or more access bridges to which the terminal bridge can connect and communicate. Finally, the home domain hosts a home location agent that tracks which access bridges the terminal is currently associated with, ensuring location transparency is maintained in face of terminal mobility. GIOP messages are sent across the access bridge using a GIOP tunnelling protocol.

RAPP

The Reactive Adaptive Proxy Placement Architecture (RAPP) [Seitz98] seeks to improve the performance of CORBA across mobile hosts for mobile multimedia

applications. The approach utilises proxies acting on behalf of mobile hosts; these aim to reduce the communication requirements over the wireless link. There are two key components to the architecture, the Proxy Selection Process (PSP) and the Proxy Installation Process (PIP). The client application provides the PSP with MIME classifications of each data stream, the PSP then monitors the QoS of every communication stream and upon decreasing QoS selects proxies either automatically or based upon user preferences (e.g. the user states to use loss prone compression techniques). The PIP then installs the selected proxy in the specified location in the network; the application specifies preferences such as “on server machine” or “in server machine network domain” that are satisfied if the available hardware, operating systems and security settings match. A global proxy trading service manages a list of available proxy factories, which can create the new proxy in place. The final step of the PIP is to then connect the proxy into the client-server stream. Therefore, RAPP offers a solution to the problems of low and variable bandwidth. Like previous solutions RAPP utilises proxies, however it ensures these can be available in the network, through dynamic proxy installation based upon application preferences.

Embedded ORBs

Due to the limited resources of mobile devices, the large memory footprint size of a CORBA ORB is a fundamental obstacle. To overcome this, commercially available ORBs optimised for memory size and performance are available; examples include e*ORB (vertel.com) and orbix/e (iona.com). However, these provide static configurations targeted at embedded devices that cannot be changed at run-time. A mobile device’s memory resources fluctuate. Hence, customisable minimum ORBs e.g. UIC [Roman01] and Zen [Klefstad03], optimise the memory footprint and also allow this to be changed over time.

2.2.3 Java RMI Solutions

Background

Java RMI [Sun97] provides a Java distributed object model that integrates into the programming language and local object model. RMI is based on the separation of *definition of behaviour* (within a Java interface) and *implementation of that behaviour* (by a Java class). An advantage of RMI over CORBA is that RMI is based entirely on

Java; this means that there is no need to introduce a separate IDL. The overall implementation of RMI is shown in figure 2.3; it is a classical RPC style architecture, building on TCP as the transport protocol. Like CORBA, Java RMI offers synchronous RPC communication whose performance is poor across wireless networks. However, Java RMI is an important platform in the mobile domain, as Java mobile agent platform (such as Aglets [Lange98]) and Jini [Arnold99] utilise it as a communication framework. Two optimised solutions that address protocol performance and host mobility are examined in turn.

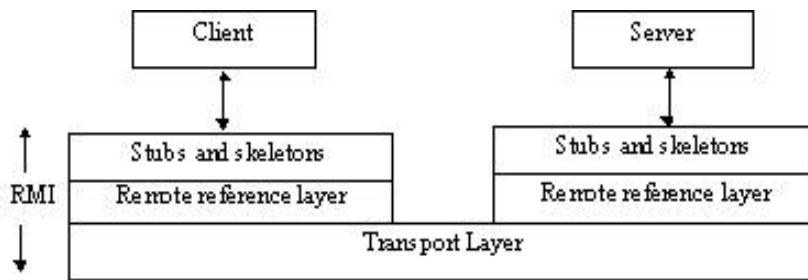


Figure 2.3 The Java RMI architecture

Wireless RMI

Wireless RMI [Campadello00] aims to improve RMI's poor performance across wireless networks, caused by high protocol overheads of data traffic and round-trip time (it does not consider terminal mobility). The invocation itself is only 5% of the total traffic transmitted and, due to an inefficient implementation of the specification, 6 round trips are required for a single invocation. Their solution is based upon mediators (performance enhancing proxies) to avoid changes to the RMI standard. An agent is placed upon the mobile terminal and a proxy executes at the wireless access point to the fixed network. All communication between client and server objects traverses this link. Five separate techniques are then used to improve performance: 1) data is compressed using the standard GZIP file format to reduce data traffic, 2) protocol acknowledgement is handled by the local mediator, avoiding sending of protocol headers, 3) dynamic stub downloading when not present at the client side is avoided; instead a generic stub is generated on the fly, 4) registry lookups are reduced by caching references locally, and 5) Distributed Garbage Collection invocations are optimised by decoupling the client and server; the local mediator renews leases on servers periodically until no more local references exist. Correspondingly, the

opposite mediator retains references of clients. The combination of these optimizations reports a 365% improvement in invocation time over the wireless link, demonstrating that the issue of low bandwidth can be solved. However, the other challenges of mobile computing are not directly tackled.

Mobile RMI

Mobile RMI [Wall01] directly addresses the problem of terminal mobility. Its design is based upon the generic concepts of the ALICE model [Biegel02] to manage the movement of mobile hosts. Hence, the same Mobile Gateways are a fundamental component of the architecture. However, the differences between RMI and CORBA require changes to the functionality offered. In RMI, once a client has received an object reference (a stub) it communicates directly with the server using this stub. If this object moves then the stub is outdated; therefore, Mobile RMI replicates the object stub at the local gateway, and clients receive a stub that references the gateway rather than the object. The gateway then redirects all incoming method invocations to the correct object. However, if the server object moves to another gateway, the client's stub cannot change and therefore, the previous gateway relays all requests to the new gateway and then onto the server object. Like ALICE, all method invocations are transparently completed in the face of network failure; reconnections are made transparently and lost data is resent. Hence, it is possible to overcome disconnection and address migration in Java RMI, although again the use of fixed mobile gateways is not an optimal solution.

2.2.4 Simple Object Access Protocol (SOAP)

SOAP [Box00] is a lightweight, XML-based protocol for the exchange of information in a distributed environment. It consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. Fundamentally, SOAP provides a standard way of serializing the information needed to invoke remote services into a format that can be transported across the wire. It also benefits from being able to traverse firewalls, as it is HTTP based.

SOAP is in some ways better suited to the mobile environment than CORBA and Java RMI, because it is simple and extensible; this means that several features of distributed object systems that cause problems in wireless networks need not be included e.g. distributed garbage collection and objects-by-reference. Furthermore, SOAP messages can be used for asynchronous and one-way message passing schemes, overriding the problems of weak connection. For example, SOAP can be transported over SMTP rather than HTTP [Cunnings01]. However, SOAP suffers from being a verbose text-based protocol that consumes bandwidth, compared to efficient protocols (e.g. IIOP). Lightweight memory footprint implementations that operate from mobile devices are available to demonstrate SOAP capabilities in this domain e.g. PocketSOAP [Fell03] and KSOAP (a Java version) [McHugh03]. However, these solutions are yet to address the problems of terminal mobility, and low and variable bandwidth.

2.2.5 Analysis of Enhancements to Established Middleware

A significant body of current mobile computing research dismisses established middleware as unsuitable for supporting mobile applications because of its use of synchronous communication paradigms and heavyweight implementations. Nonetheless, they offer advantages to the mobile developer. Distributed object programming is a well-understood and well-used programming model. A large body of existing services (within the fixed domain) can be accessed from mobile devices; hence, users can mirror functionality from their fixed terminal to their mobile. Given these benefits, it is inevitable that standard middleware (CORBA, RMI, SOAP etc.) will be used by mobile applications.

Furthermore, the enhancements described in this section demonstrate the feasibility of using established middleware across wireless networks. The majority of the challenges documented in section 1.2.5 have been addressed (illustrated in table 2.1). The CORBA enhancements have tackled more of the challenges than RMI and SOAP; however, this simply demonstrates that more work has been carried out on this particular platform. Notably, no enhancement has directly examined how to better manage battery consumption by these technologies, which is an important issue given

the resources required to transparently maintain synchronous communication in the face of disconnection.

Mobile Challenge	Challenge addressed	Examples
Disconnection	✓	Alice, DOLMEN
Low Bandwidth	✓	DOLMEN, Wireless RMI
Variable Bandwidth	✓	RAPP
Address Migration	✓	ALICE, Mobile RMI, DOLMEN
Low Power	✗	
Small Storage Capacity	✓	Embedded ORBs, Alice

Table 2.1 Challenges of mobile computing met by established middleware

Although these enhancements meet the individual challenges, it can be seen that these solutions are not optimally suited to the wireless domain. The majority of the described solutions concentrate on mobile devices roaming between network access points and therefore require a proxy or a gateway to be available in the environment. However, connecting to a wireless network with no gateway means that the clients do not retain the benefits of the enhancements. Notably, RAPP introduces a technique to dynamically install a proxy at different locations in the network.

2.3 Asynchronous Middleware

2.3.1 Overview

As a direct response to the problems posed by synchronous communication paradigms, which requires the client and server to be connected at the same time, asynchronous middleware has been proposed. This approach allows mobile hosts to communicate while not directly coupled in time and space; hence the problems of network partitioning and periods of disconnection (e.g. during network handover) are reduced. This section examines different types of asynchronous communication, from an early asynchronous RPC middleware to tuple space and publish-subscribe paradigms. Key platforms within each paradigm are discussed and their benefits to mobile computing are analysed.

2.3.2 Asynchronous RPC – “An Early Solution”

An early asynchronous middleware platform was the Rover toolkit [Joseph95]. This was notable because its core technical concepts (e.g. asynchronous communication, mobile code and data sharing) are utilised by future mobile middleware platforms documented in this chapter. Rover’s initial aim was to isolate the mobile application from the limitations of wireless networks, particularly disconnection and poor bandwidth.

Rover provides a distributed object model to the developer that consists of object downloading, caching and asynchronous object invocation. Remote objects, called Relocatable Dynamic Objects (RDOs), are dynamically downloaded into the local object cache on the mobile device; this reduces object interaction across the wireless link, an advantage offered by the mobile agent solutions described later. When an object method is invoked, the local cache is checked and if the object resides locally the invocation updates the cached object without contacting the server. Different strategies are then available to maintain consistency of objects in the cache with the server objects. Rover lazily updates the primary copy by sending the method call in a Queued Remote Procedure Call (QRPC) to the server. A failure in the delivery of the invocation, or link unavailability does not cause the RPC to fail; incomplete RPCs are written to a log that can be replayed when the connection returns.

Rover offers solutions to the majority of mobile computing problems. The asynchronous communication mechanism addresses both disconnection and address migration, and minimising network communication using mobile objects overcomes low bandwidth and reduces power consumption. However, Rover does not conform to any existing middleware standards, but rather proposes its own standard for mobile middleware to adhere to. This begins the trend of mobile middleware platforms that offer their own solution to mobility issues, but clouds the problem of platform and middleware heterogeneity.

2.3.3 Tuple Spaces

Background

The tuple space is a well-established asynchronous communication model [Gelernter85] that is effectively a shared distributed memory spread across all participating hosts. To communicate, hosts submit *tuples* and *anti-tuples* to the tuple space. Tuples are typed data structures and are comparable to objects in languages like C++; to be altered they must be removed from the space, changed and then re-inserted. However, anti-tuples capture requests seeking to remove or copy data from a space; they contain a template against which to match tuples. Anti-tuples can be regarded as questions and tuples the answers [Wade99]. Tuple spaces provide temporal and spatial decoupling; hosts communicate through the space without being online at the same time or attached by an explicit binding, an ideal approach for mobile computing. Example tuple space implementations designed specifically for mobile computing are described in turn.

L²imbo

The L²imbo platform [Davies98] is based upon the Linda tuple model architecture, but adds extensions for operation within a mobile environment. Linda features a single global tuple space. However, in an environment where communication links are unreliable and costly all operations being performed on a central space is infeasible, therefore L²imbo allows multiple tuple spaces to be created across hosts; tuples propagate between spaces using a bridging agent. Alternatively, an individual tuple space can span multiple hosts. The consistency between multiple tuple copies is maintained by a distributed tuple space protocol (that is implemented as a multicast group). The replication of tuple spaces across multiple hosts enables the tuple space to remain accessible during disconnection.

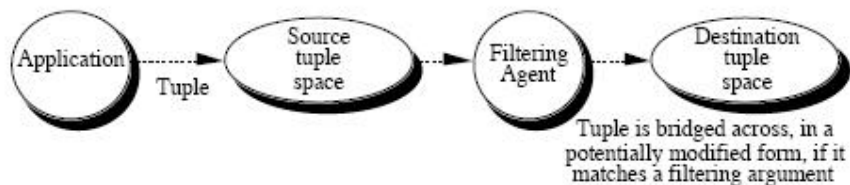


Figure 2.4 Filtering tuples from one tuple space to another [Wade99]

Furthermore, QoS attributes can be added to tuples, including delivery deadline, so that mobile multimedia applications can be supported; this also allows the system to adapt itself to make best use of network connectivity. System agents monitor QoS and the propagation of tuples between tuple spaces. The monitoring agents watch characteristics such as connectivity, communication cost and power and inject tuples (representing the current system state) into a management tuple space. These are globally accessible, allowing remote hosts to query current QoS conditions. Filtering agents, a special type of bridging agent (illustrated in figure 2.4), then allow L²imbo to adapt its behaviour by performing transformations on the tuples being distributed. For example, a filtering agent can act between two spaces dealing with MPEG video frames and only transmit I-frames, or by performing colour reduction on the I-frames if it detects a drop in bandwidth.

L²Imbo's implementation of the tuple space paradigm means that disconnection, address migration and low bandwidth problems are solved. Furthermore, its use of monitoring and adaptation agents help address variable network conditions, and could feasibly be used to manage power consumption.

Linda In a Mobile Environment (LIME)

LIME [Murphy01] utilises the concepts of the Linda co-ordination model and provides additional support to new types of distributed mobile applications. The underlying core is based upon a global virtual data structure (a tuple space whose content depends upon the connectivity of mobile hosts). This dynamically changing global context is accomplished by breaking up the Linda tuple space into many tuple spaces, each permanently associated to a mobile unit, and by introducing rules for transient sharing of the individual tuple spaces based on connectivity. The only way to access the global context from a mobile host is through an Interface Tuple Space (ITS); this contains the tuples that the host is willing to make available to other mobile units. The architecture of this model is shown in figure 2.5. Upon arrival of a new mobile unit, the tuples in its ITS are merged with those already within the global context and the result is accessible via the ITS. This abstraction provides the mobile application with the perception of a local tuple space contained within the federated space.

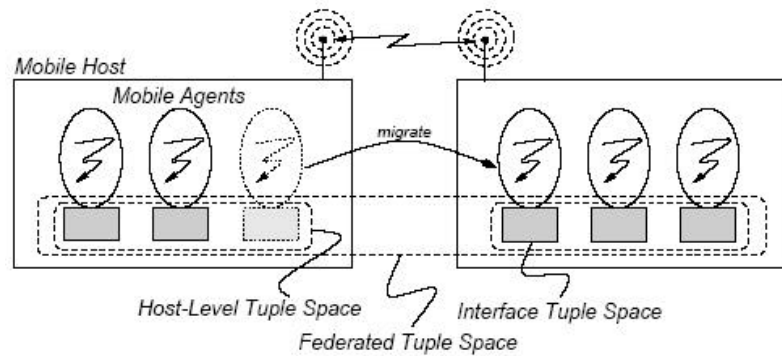


Figure 2.5 Transiently shared tuple spaces [Murphy01]

LIME also allows the ability to react to changes in context, an important factor in mobile application design (for example, when a new member node arrives). The Linda model is extended by the *Reaction* concept. A reaction $R(s, p)$ is defined by code fragment s , which specifies the behaviour when a tuple matching the pattern p is detected within the tuple space. Like L^2imbo , the tuple space implementation is particularly suited to addressing problems of terminal mobility, and the reaction concept can be utilised to improve operation in varying network conditions and resource consumption.

Tuples On The Air (TOTA)

TOTA [Mamei03] is a tuple-space middleware designed to support adaptive context-aware applications in ad-hoc networks. Tuples are used to represent context information and enable uncoupled interaction between distributed application components. TOTA differs from previous implementations in that tuples are not specific to a mobile node; rather a tuple is injected into the network and then autonomously propagates according to a pattern defined by the application. The TOTA architecture consists of a peer-to-peer network of mobile nodes, each running a version of the TOTA middleware; these maintain references to neighbouring nodes. A TOTA tuple (T) is defined as $T = (C, P)$. The Content C represents the information carried by the tuple and the propagation rule P determines how the tuple will be transmitted. Tuples are injected into the system from a particular node and spread hop-by-hop according to this propagation rule. For example this may state the physical distance the tuple should travel (e.g. ten metres), or state how transmission is affected

by the presence of other tuples. In addition the rule can also state how the content changes while moving from host to host.

Other Tuple Spaces

Tuples spaces present powerful communication abstractions with simple programming interfaces. However, the application of the paradigm to real world systems is a complex task, requiring a thorough understanding of the process, and hence tuple spaces are not particularly well used compared to other paradigms. However, there is evidence that applications inherently matched to tuple spaces are simple to develop [Murphy01]. Furthermore, commercial tuple space implementations (designed for the fixed network) including JavaSpaces [Waldo98] and IBM's T-Spaces [Wyckoff98] are gaining prominence in the Java community; therefore, these will promote a greater understanding of the paradigm and better awareness of its benefits. However, only T-Spaces is currently suited to mobile computing, due to its reduced footprint size; the reduced Java virtual machines for mobile devices do not yet support the full JavaSpaces platform.

2.3.4 Publish-Subscribe Middleware

Background

In certain application scenarios, asynchronously occurring events need to trigger an immediate response. For example, a credit card cancellation operation by a banking service must invalidate a stolen card immediately and notify all affected services [Bacon00]. Frequent polling to learn whether events have occurred overloads communications and infrequent polling delays the response and users perceive the application as sluggish or insecure. Therefore, asynchronous *event notification* is an important communication paradigm for distributed applications. Publish-Subscribe is a particular implementation of this paradigm. It allows processes to exchange information based upon message content rather than direct message exchange between destination addresses. A component subscribes to the event types they are interested in, and then consumes the notifications when they are published. The decoupled nature of event-based communication is well suited to mobility; after a mobile host reconnects it can continue to retrieve the requested notifications.

A number of publish-subscribe middleware now exists, and we examine selected systems in turn. These consist of the pioneering event platforms that demonstrate the techniques of publish-subscribe, the enhancements to these to support mobile computing, and finally publish-subscribe services designed exclusively for wireless networks.

Cambridge Event Architecture (CEA)

The Cambridge Event Architecture (CEA) [Bacon00] demonstrates how existing synchronous middleware such as RMI, CORBA and DCOM can be extended to include asynchronous operation. Objects use an IDL to publish the event types that clients can subscribe to; hence CEA is language-independent. Each object has a register method in its interface to allow clients to subscribe for a particular class of event. Therefore, CEA integrates event functionality into the object interface, as opposed to alternative independent services, including the OMG Event Service [OMG98b]. This publish-register-notify paradigm allows a direct source from subscriber to publisher (seen in figure 2.6a); however, the architecture also consists of event mediators (event brokers), which are placed between publisher and subscriber (illustrated in figure 2.6b). These offer the advantage of moving filtering computation from resource deficient publishers to the broker. Furthermore, these mediators can register interest on behalf of mobile subscribers and then buffer all notifications (when the unit is disconnected). It also registers interest in the mobile client's location, and notification of an *attach* event (detecting the mobile user) triggers delivery of the accumulated events to the user at the new location.

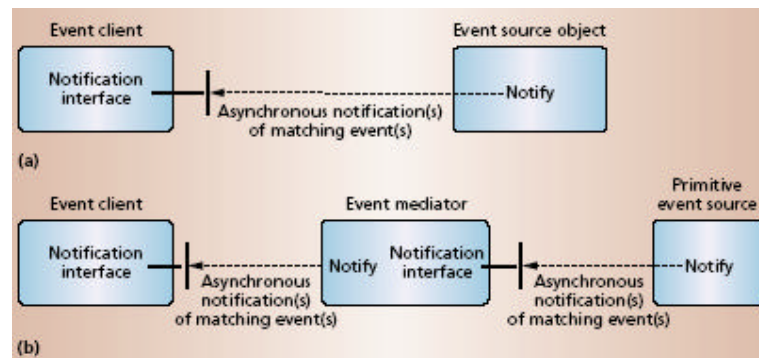


Figure 2.6 Event notification in CEA: a) direct and b) mediated [Bacon00]

Scalable Internet Event Notification Architecture (SIENA)

SIENA [Carzaniga01] is a publish/subscribe service implemented as a distributed network of servers (within a fixed network). The main aim is to achieve scalability for large numbers of communicating entities and high volumes of notifications. A SIENA server acts as both an access point, providing clients with an extended publish/subscribe interface, and as a store-and-forward network router. Publishers use the access points to advertise the information about the notifications they generate and to publish these notifications. Subscribers use the access points to subscribe for notifications of interest by supplying a predicate, called a filter, to be applied to the content of notifications. Underlying Siena's interface is a notification data model that governs the semantics of the service. A notification in the model is a set of typed attributes; each individual attribute has a type, a name, and a value. The attribute types belong to a predefined set of primitive types commonly found in programming languages and database query languages, and for which a fixed set of operators is defined. SIENA is an example of a publish-subscribe service not extended to operate across wireless networks; terminal mobility means that the subscriber will lose events of interest when the device becomes disconnected.

Scalable Events and Real Time Mobility (STEAM)

STEAM [Meier02] is a publish-subscribe middleware from Trinity College, Dublin specifically designed to operate in ad-hoc wireless networks. They argue that existing publish-subscribe services like (SIENA and CEA) use centralised components (to distribute events) located in the network, either co-located with producers or consumers or on separate remote machines. However, in many wireless networks the availability of these cannot be guaranteed. Therefore, STEAM presents an implicit event model that requires no separate middleware components to offer system wide services. Instead, group communication is used as a natural method to enforce this model. Communication groups create a one-to-many pattern that allows publishers to propagate events to a group of subscribers.

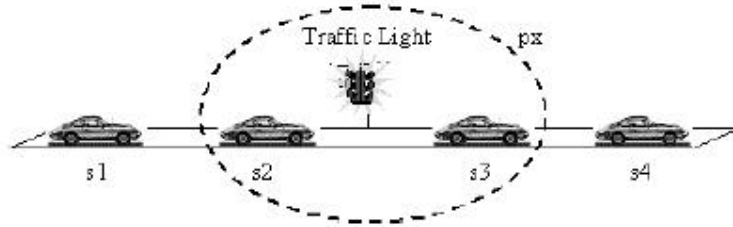


Figure 2.7 **Traffic light application demonstrating proximity group [Meier02]**

STEAM is influenced by the idea that in mobile application scenarios entities are likely to interact when they are in close proximity to one another i.e. the closer consumers are to producers the more likely they are to be interested in receiving their events. Therefore, *proximity groups* are the fundamental communication mechanism in STEAM. Mobile hosts must be in the same geographical area as the group and also express an interest in order to join and receive events. Figure 2.7 demonstrates an example application scenario that requires proximity groups; a traffic light publishes events describing its status and cars within a certain distance of the lights receive these and react appropriately.

The REBECA Event Based Electronic Commerce Architecture (REBECA)

REBECA [Muhl02] is a content-based notification service. Its architecture is an acyclic, connected graph of publishers, subscribers and event brokers. The edges are point-to-point communication links along which message are delivered in FIFO fashion. Three types of brokers form a broker network. *Local brokers* are co-located with the publisher or subscriber and serve as their entry point to the service. These are connected to at most one *border broker* that in-turn connects to *inner brokers* and other border brokers. Each broker maintains a routing table for notification forwarding. Elements in the table are a pair (F, L) where F describes the content filter and L the communication link to forward along if a match is made. This is a scheme similarly utilised by SIENA.

This architecture is unsuitable for wireless networks and hence REBECA's brokers have been extended to deal with the problems of terminal mobility [Fiege03]. Firstly, they maintain a buffer for all undelivered events over a period of time (to manage disconnections). When a subscriber reconnects its subscriptions are reissued

automatically, the broker network then reconfigures so that events can be routed to the new location (while the old route is removed), any messages sent to the old location are replayed before new ones are sent.

Java Event-Based Distributed Infrastructure (JEDI)

JEDI [Cugola01] is conceptually similar to REBECA. It is an event broker architecture that has added extensions to support client mobility; this is explicitly controlled by the subscribing application. Two methods are available to the subscriber: *moveIn* & *moveOut*. The client invokes *moveOUT* when it is about to disconnect from the network and this forces messages to be stored. When the client reconnects and invokes *moveIN* the old messages are routed to the new location. However, given the nature of wireless networks, connection loss is unpredictable and calling a method beforehand is infeasible.

Elvin

The original Elvin [Segall97] architecture proposed an individual server to act as a notification router between multiple connected clients. Clients can act as producers and/or consumers of events, and the server is responsible for routing notifications of interest to consumers. This has been extended to include federations of multiple servers, but the concept of routing notifications based on content to interested clients remains the same. Quenching is a unique feature of the Elvin service; producers receive information about what consumers are expecting of them so that they need only generate the events that are in demand. This is important for some classes of producers where the act of producing the event is expensive.

Elvin has been extended to operate in the wireless environment and cope with disconnections. However, rather than modifying (and possibly encumbering) the Elvin service, a prototype Elvin proxy has been developed, which can store notifications while clients are disconnected [Sutton01]. Hence, the proxy remains connected to the server and can act on the behalf of a mobile client whenever it reconnects.

2.3.5 Analysis of Asynchronous Middleware

Asynchronous middleware is naturally suited to mobile computing applications. Table 2.2 illustrates how effective each of the two asynchronous paradigms are in addressing the challenges of mobile computing. All solutions decouple the sender and receiver in time and space, thereby overcoming the problem of weak connection. It can be seen that tuple spaces are especially well suited to mobile computing; LIME, L²imbo and TOTA demonstrate that with context support the paradigm can solve the majority of mobile computing. Although these platforms have the potential to manage power consumption, none directly tackles this problem.

Mobile Challenge	Tuple Spaces	Examples	Publish-Subscribe	Examples
Disconnection	✓	L ² Imbo, LIME, TOTA	✓	All
Low Bandwidth	✓	L ² Imbo, LIME	✓	All
Variable Bandwidth	✓	L ² Imbo, LIME	✗	
Address Migration	✓	L ² Imbo, LIME	✓	REBECA, STEAM, JEDI, Elvin
Low Power	✗		✗	
Small Storage Capacity	✓	L ² Imbo, LIME, TOTA	✓	REBECA, STEAM, JEDI, Elvin

Table 2.2 Challenges of mobile computing met by asynchronous middleware

However, the tuple space it is not a well-understood paradigm. Therefore, event-based publish-subscribe services offer a different, more accessible programming model. Again, the described platforms offer good solutions to the problems of disconnection, terminal mobility, low bandwidth and low memory capacity. However, like enhancements to synchronous middleware, mediator solutions to terminal mobility require middleware components to be available in the network (this cannot be guaranteed), although STEAM offers a notable solution to this. Furthermore, these solutions do not directly address power consumption or variable bandwidth that would certainly be an issue for mobile multimedia applications.

2.4 Data Sharing Middleware

2.4.1 Overview

Data sharing is an alternative communication abstraction for distributed systems. Examples of early data sharing systems for fixed networks are: CODA [Satyanarayanan90] and Globe [Bakker00]. Typically, these employ a strong distinction between servers that store central copies of data (as data files, or objects), and clients that hold personal caches. A client requests a replica of the data, which can then be updated locally; replication algorithms that enforce strong data consistency across hosts are then generally employed. However, mobile middleware researchers have identified that the data sharing abstraction has a number of advantages in environments where disconnection may be the normal state and network bandwidth is scarce. This section examines three key data sharing middleware platforms for mobile applications; the first two, Bayou and Ad-hocFS, share data held in files, whereas XMIDDLE shares meta-data.

2.4.2 Bayou

Bayou [Demers94] is a data-sharing platform designed to explicitly support mobile users who wish to share information like appointment calendars, databases, and meeting notes. The middleware seeks to address the problem of data sharing in environments with communication outages. Therefore, the architecture focuses on server machines (laptops, or PDAs) that hold copies of one or more databases; mobile clients then access data residing on the servers within communication range. This method has the advantage that devices with limited storage capacity may still communicate, as they do not need to store large local caches of data. Furthermore, Bayou employs a read-any/write-any replication strategy to ensure consistency between all copies of the data; a client can read and write to any copy of the database, although timeliness in propagation of the updates between replicas cannot be guaranteed and hence, only weak consistency is achieved. Bayou uses reconciliation techniques (anti-entropy) to ensure that all copies of a database converge to the same state and will eventually be identical if there are no new updates (i.e. servers receive all writes to the replica and order them consistently). Finally, Bayou uses “fluid” replication strategies that dynamically create new replicas based upon network

characteristics. Therefore, low bandwidth problems are addressed by replicating the data closer to the client. Notably, Bayou addresses the problems of disconnection and address migration through database replication. However, a single client out of range cannot continue operating unless a local server also resides on the device.

2.4.3 AdHocFS

AdHocFs [Boulkenafed03] is a novel middleware platform in that it specifically uses a data-sharing paradigm to target co-operative shared work applications for mobile hosts connected across ad-hoc wireless networks. To form a collaborative group, the peers in wireless range are discovered using the SLP protocol (without directory agents); members then authenticate themselves using digital certificates obtained from a trusted third party (not required to be on-line). The group members are then free to collaborate on the shared data space using an encrypted protocol; members can leave and new trusted members can join at any time. To maintain data consistency a unique token is associated with each newly created shared data; each member must obtain the token in order to modify the data. Update propagation of the modified data is only executed when a member tries to access the information. This reduces data communication and hence saves power resources. Furthermore, an adaptive replication strategy is utilized; storing all shared data upon each host will quickly consume both energy and storage spaces. However, when a member leaves the data may be lost. Therefore, an adaptive replication strategy is utilized. A profile for each host is used by the replication protocol; this states available storage space, whether it can store replicas and estimated time in the group. This information can then be used to distribute replicas.

The replication of data at local clients ensures that disconnection problems are avoided (changes can be made locally until reconnection). Furthermore, AdhocFS provides good solutions to variable bandwidth, memory capacity and power constraints. Data is intelligently replicated based upon context information about the devices in the environment and network conditions.

2.4.4 XMIDDLE

XMIDDLE [Mascolo02b] is a data sharing middleware designed to support mobile applications that use both replication and reconciliation of data over wireless ad-hoc networks. Hence, XMIDDLE is well suited to collaborative applications whereby users exchange or work upon shared information; for example, a collaborative e-shopping application where multiple users add to a shared shopping list, which is reconciled before items are purchased. Each mobile device stores its data as an XML tree; this allows complex data structures to be created using hierarchies of data nodes. On each device a set of access points for the tree is defined that can be read and modified by peers. Therefore, a host explicitly links to the access point of the tree to download the required section. This host can then modify the data and when the two become connected in the future this information is reconciled. The reconciliation of XML trees is implemented using tree-differencing techniques.

XMIDDLE offers a powerful technique to share both data and its meaning. For example, it is well suited to the sharing of context information between mobile users. However, the tags associated with XML data are an extra overhead compared to standard file sharing. Therefore, transmitting large XML trees may be expensive over low bandwidth wireless networks (although it is possible to share sub-trees only).

2.4.5 Analysis of Data Sharing Middleware

Table 2.3 illustrates the effectiveness of data sharing middleware solutions in overcoming the challenges of mobile computing. It can be seen that each of the challenges have been addressed; in particular, weak connection and address migration are addressed by the uncoupled nature of the paradigm. In addition, changing data replication strategies solves the problems of poor bandwidth and scarce memory resources. Finally, the adaptive replication strategies proposed in AdhocFS overcomes both changing network QoS and low power. However, this paradigm is applicable to only the subset of application classes for which it fits well, e.g. collaborative and information sharing applications. The alternative paradigms already discussed propose more natural solutions for different application classes e.g. information dissemination. Therefore, it is likely that data-sharing systems will be confined to use by specialised applications.

Mobile Challenge	Challenge Addressed	Examples
Disconnection	✓	Bayou, AdhocFS, XMIDDLE
Low Bandwidth	✓	Bayou, AdhocFS
Variable Bandwidth	✓	Bayou, AdhocFS
Address Migration	✓	Bayou, AdhocFS, XMIDDLE
Low Power	✓	AdHocFS
Small Storage Capacity	✓	Bayou, AdhocFS

Table 2.3 Challenges of mobile computing met by data-sharing middleware

2.5 Mobile Agents

2.5.1 Overview

Agents can be described as “software entities”, which can be either stationary or mobile. A Stationery agent resides on the same host throughout its lifetime, working on behalf of the user, e.g. connecting to ftp sites or browsing through URLs. Conversely, a mobile agent transfers executable code (behaviour) and state from machine to machine. The mobile agent paradigm is well suited to supporting distributed applications within the mobile environment. Firstly, the agent acting on behalf of the client can move to the server and perform all communication locally before returning; this limits the interaction over the wireless link, saving bandwidth consumption and hence power. Furthermore, this reduction in communication limits the possibility of failure due to partition or disconnection; the agent may monitor the network status and control its mobility between hosts. Finally, moving the logic processing to a more powerful server host overcomes the problems of limited resources on the client. In this section, we examine typical Java-based agent solutions, and also a system designed specifically for mobile hosts.

2.5.2 Java-based Mobile Agents

Mobile agents are now a popular technique to create distributed applications across the Internet and a large number of systems are now available to do this. Examples are

AgentSpace [Silva97], Aglets [Lange98], Concordia [Wong97] and Jumping Beans (www.jumpingbeans.com). Agent systems are normally implemented in Java because of the portability benefits offered by the Java Virtual Machine. Therefore, these systems have very similar capabilities. For example, Aglets [Lange98] are Java-based autonomous software agents that extend the model of network-mobile code, as used by Java applets. Like an applet, the class files can migrate across a network. But unlike applets, when an aglet migrates it also carries its state. An aglet is therefore a running Java program (code and state) that can move from one host to another on a network. These solutions are well suited to overcoming some of the challenges of mobile computing e.g. disconnection, variable bandwidth and limited power. However, their reliance on Java Virtual Machines and heavyweight implementations makes these platforms unsuitable for the majority of mobile devices. This had led to the emergence of agent platforms, with similar capabilities that execute on lightweight Java Virtual Machines e.g. the Grasshopper platform [IKV99].

Furthermore, agents do not have to be Java-based, and other solutions are now emerging; the .NET framework [Microsoft00] contains a Common Language Runtime (CLR) that agents are able to execute upon. For example, μ Code is a Java agent solution that has been ported to .Net [Delamaro02].

2.5.3 Tacoma and Tacoma Lite

The Tacoma model [Johansen95] focuses on how agents can be used to solve problems traditionally addressed by other distributed computing paradigms. The agent is a process (code and state), with mobility as its primary characteristic. In order to maintain state, the agents must manipulate data, i.e. leave data at a site and carry data when it moves. For example, an agent visiting multiple sites, with each site completing part of an overall computation needs to carry the sub results along with it when it leaves. To support this, Tacoma introduces the concept of *folders*, *briefcases* and *cabinets*, shown in figure 2.8. Folders contain data and code (including the source code of the agent) relevant to different computations. A collection of folders associated with an agent is known as a briefcase. Furthermore, stationary folders are needed for permanent data repository purposes; therefore, file cabinets hold folders at individual nodes. The fundamental property of the architecture is the *meet* abstraction;

agents do not communicate by exchanging messages they simply meet at the same location and exchange briefcases. For example, a client may request a service from a system agent by passing it a briefcase containing a service specification; the system agent then returns the service result in another folder.

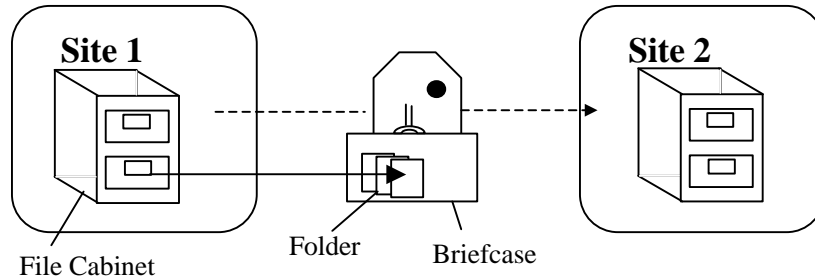


Figure 2.8 **Maintaining state in Tacoma using folders, briefcases and file cabinets**

To explore the benefits of mobile agents within the domain of mobile computing, the Tacoma Lite system [Jacobsen99] was developed. Tacoma Lite adds an extra layer to the initial TACOMA architecture, which consists of an entity called the hostel that acts as a network proxy for the mobile device. This is required because applications assume the presence of a host that mobile agents cannot reach if the mobile device is disconnected. To illustrate its capabilities a number of applications were developed, including a weather alarm (to alert users when certain conditions arose) and a stock ticker, which periodically checks stock prices for the users. Tacoma Lite offers two benefits. Firstly, the system can overcome periods of disconnection without loss of information. Secondly, the amount of data transferred to and from device is reduced through the intelligent use of agents (rather than downloading all stock prices, the agent finds only those required).

2.5.4 Analysis of Mobile Agents

Mobile agents are well suited to overcoming the challenges of mobility, as illustrated in table 2.4. They overcome weak connection by moving from host to host when the connection becomes available again. Communicating with the agent to inform it of the new network address solves address migration. Furthermore, mobile code improves power consumption and limited processing capabilities by moving the computation away from the mobile device and onto more powerful machines. Agents can also monitor QoS to effectively overcome varying levels of bandwidth. However, agent

solutions remain resource heavy and rely upon virtual machines to allow portable execution. Furthermore, agent traversal across the network is not suited to low bandwidth networks (as the code must also be transmitted); in such environments the agent can utilise alternative communication styles e.g. Remote Procedure Call and asynchronous messaging.

Mobile Challenge	Challenge Addressed?	Examples
Disconnection	✓	Java agents, Tacoma Lite
Low Bandwidth	✗	
Variable Bandwidth	✓	Tacoma Lite
Address Migration	✓	Tacoma Lite
Low Power	✓	Java agents, Tacoma Lite
Small Storage Capacity	✓	Tacoma Lite, Grasshopper

Table 2.4 Challenges of mobile computing met by agent-based middleware

2.6 Service Discovery

2.6.1 Overview

A key characteristic of mobile computing is the mobile host's interaction with location-based services (as described in section 1.2.2). Hence, the ability to discover what services are available at a particular location is especially important. An illustration of this is a smart room, which contains a music player that can be controlled by a mobile device. A user enters the room, discovers the available service and then controls the service using their portable device. Only the music player at the current location is of interest to the user. Therefore, technologies known as service discovery protocols have emerged, which provide functionality to support dynamic scenarios of this type.

At present there are four key service discovery protocols in the commercial realm: Jini, Service Location Protocol, Universal Plug and Play, and Salutation, which are

described in turn in the following sections. However, many aspects of their implementations are not ideally suited to the domains of mobile computing. Particular concerns are that their implementations are often heavyweight and contain centralised network components, and their descriptive protocols are overly verbose. In the mobile computing domain these are unacceptable. Hence, this section also describes current research solutions that attempt to answer these criticisms.

2.6.2 Jini

Jini [Arnold99] is a Java based service discovery platform that provides an infrastructure for delivering services and creating spontaneous interactions between clients and services regardless of their hardware or software implementation. New services can be added to the network, old services removed and clients can discover available services all without external network administration.

The Jini architecture centres on central federated lookup services that physically exist on remote machines in the network domain; clients and services first discover lookup services in their vicinity before utilising them. The lookup service consists of a directory of service items, which are made up of three elements: 1) its service interface (defined as a Java Interface), 2) a Java object (service proxy) on which calls to use the service can be made, and 3) a set of service attributes that describe the service. In order to be discovered, new services register this information to one or more lookup services. Furthermore, Jini employs the concept of leasing; a service registers itself for a given time period, called a *lease*. When the lease expires the service is no longer advertised.

When an application discovers the required service, the service proxy is downloaded to their virtual machine so that it can then use this service. A proxy may take a number of forms:

- The proxy object may encapsulate the entire service. This strategy is useful for software services requiring no external resources.
- The downloaded object is a Java RMI stub, for invoking methods on the remote service.

- The proxy uses a private communication protocol to interact with the service's functionality.

Therefore, the Jini architecture allows applications to use services in the network without knowing anything about the wire protocol that the service uses or how the service is implemented; one implementation of a service might be RMI-based, and another CORBA-based. This offers one particular solution to the problem of middleware heterogeneity through the use of mobile code to manage interactions. This property will be discussed further in chapter 3.

The Jini architecture is in general not well suited to the mobile computing domain. Jini depends upon centralised middleware elements e.g. lookup directories, which cannot be guaranteed to be available in all wireless networks. Furthermore, Java virtual machines that include a full Jini implementation consume a large memory footprint.

2.6.3 Service Location Protocol (SLP)

The Service Location Protocol [Veizades97] has been specified by the Internet Engineering Task Force (IETF) and aims to provide a vendor independent standard for service discovery. Three key parts compose the core architecture: 1) *user agents* that perform discovery on behalf of the user or application, 2) *service agents* that advertise the location and characteristics of a service, and 3) *directory agents* that collect service addresses from service agents and responds to user agents. The operation of these agents is illustrated in figure 2.9.

The Directory Agent is a fixed centralised element that operates in a similar manner to a Jini Lookup Service. Hence, user agents and service agents first attempt to find the local directory agent in their domain. However, unlike Jini, if none are found then user agents and service agents can interact directly. A service agent can receive a lookup request; if its service matches it can send a response. Therefore, SLP has a flexible and scalable architecture that is suitable for different network types i.e. it can be used in both wireless and fixed enterprise networks, as illustrated by figure 2.9.

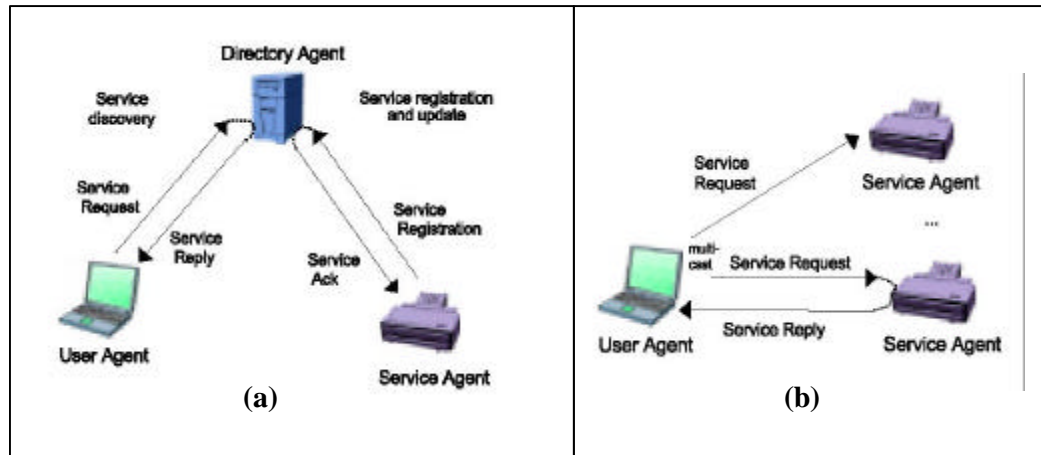


Figure 2.9 Service discovery in SLP: (a) using a directory agent and (b) without using a directory agent

2.6.4 Universal Plug and Play (UPnP)

Universal Plug and Play [Microsoft00b] is a platform and language independent discovery architecture designed to connect devices in unmanaged or ad-hoc networks. Internet technologies including: IP, TCP, UDP, HTTP and XML form the core of the architecture. When a new UPnP device is added to the network, the UPnP discovery protocol allows a device to advertise its service to UPnP control points (users of devices or services) on the network, using a multicast protocol i.e. all control points listen to the same group address. Similarly, control points multicast discovery messages searching for services that match their requirements. All UPnP devices listen on the standard multicast address (239.255.255.250:1900) and respond to requests that match their service. These service discovery messages are broadcast using the Simple Service Discovery Protocol (SSDP) [Goland99]. Another key feature of UPnP is device description. After discovering a device, the control point downloads an XML description of that device; this holds information including: manufacturer, model and serial number, as well as a list of the embedded services that provide its functionality. Finally, the SOAP protocol [Box00] is used to control (invoke) services once they have been discovered.

The discovery architecture of UPnP is well suited to mobile computing, as it is largely multicast based and requires no central components. However, the use of XML, HTTP

and SOAP to manage description and control are less effective as these are synchronous and verbose protocols whose performances suffer over bandwidth limited wireless networks. The graph in figure 2.10 illustrates that the significant number of messages of this type used for UPnP discovery consumes more bandwidth than alternative technologies e.g. SLP and MARE.

2.6.5 Salutation

An open industry consortium (www.salutation.org) is developing the *Salutation* architecture [Salutation98]; this consists of Salutation Managers (SLMs) and Transport Managers (TMs) that together perform the role of service brokers. An SLM provides a transport independent interface where services can register their capabilities and clients can query the SLM to lookup services. Transport Managers are transport dependent elements that discovers other Salutation Managers and form the network of brokers over which discovery takes place. They use different discovery techniques dependent on the underlying transport, including: checking a static table of known Salutation Managers, contacting a central element or broadcasting a query. Like Jini and UPnP, Salutation includes capabilities to use a service after discovery; for this it utilises the Sun Microsystems' Open Networking Computing Remote Procedure Call protocol [Srinivasan95].

The developers of Salutation have identified its weaknesses in supporting mobile and ubiquitous computing and have produced Salutation Lite [Salutation00] to overcome these. The protocol operates over IRDA [IrDA01] and the amount of data exchanged during service discovery has been reduced. Furthermore, the implementation provides only discovery capabilities not session management; hence the footprint is reduced to better suit mobile devices.

2.6.6 Service Discovery Protocol (SDP)

The Bluetooth protocol stack contains the *Service Discovery Protocol* (SDP) [Bluetooth99b], which is used to locate services provided by or available from a Bluetooth device. It has been modified to suit the dynamic nature of ad-hoc communications and addresses service discovery over the Bluetooth protocol and

hence it is network dependent. The following inquiries are available for service discovery: search for services by service type (e.g. a printer service), search for services by service attributes (e.g. a printer service with colour printing) and service browsing without a priori knowledge of the service characteristics. SDP does not include functionality for accessing services. Once services are discovered with SDP, they can be selected, accessed, and used by mechanisms outside of the scope of SDP.

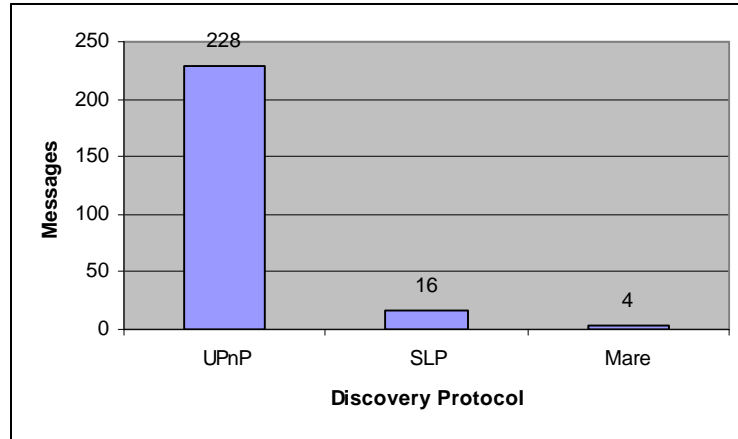


Figure 2.10 Number of messages to discover four services in different protocols

2.6.7 MARE

MARE [Storey02] is a research based resource discovery platform for operation across wireless ad-hoc networks (it was designed to support applications for teams of mountain rescue workers). The main aim is to minimise the network traffic created for discovery compared to the high protocol overhead of the previously described discovery protocols. The two key aspects of the implementation are tuple spaces and mobile code. Tuple spaces are the communication mechanism to transmit resource information, messages and importantly mobile agents. MARE uses the L²imbo implementation [Davies98], hence participating nodes in the distributed tuple space form a multicast group; each machine transmits a beacon to this group containing all resources present in its MARE instance. MARE then uses mobile agents to move operations closer to the data source and hence reduce bandwidth use. The graph in figure 2.10 demonstrates that MARE uses fewer messages than both SLP and UPnP to support service discovery.

2.6.8 The Java Enhanced Service Architecture (JESA)

JESA [Preuss02] is a lightweight, Java-based service platform for devices with limited resources that communicate across ad-hoc networks. At the core of this architecture is the JESA Service Discovery Protocol (JSDP), which is a lightweight protocol offering similar functionality to Jini: locating services, retrieving service proxies and querying service attributes. The major goal of JSDP is to work transparently if there is a central broker available or not. When a broker is available, providers register their service here and stop responding to requests. A client requests service information and a provider announces its presence and capabilities using messages delivered atop UDP multicast (responses are sent by UDP unicast). Therefore, this approach utilises elements of both Jini and SLP to produce a Java protocol well suited for discovery in ad-hoc networks.

2.6.9 Centaurus

Centaurus [Kagal01] is an example of an intelligent service platform designed specifically for environments such as *SmartHomes* and *SmartOffices*. For this, it introduces its own service discovery platform. A Centaurus System is a fixed architecture localised to a particular space e.g. a room; clients can then access the services of a Centaurus system by connecting to it. A room is equipped with a Centaurus Communication Manager, whose client can be downloaded onto mobile devices to allow users to interact with it directly. When a user enters the room for the first time they have the option to install this software on their portable device; once installed, it continuously reads the updated list of registered services. A user is then able to choose a service and execute selected functions. This is an example of user driven service discovery and interaction using a generic proxy; however, this has the disadvantage that automated software agents are unable to perform discovery.

Centaurus is an example of a growing trend of service platforms for smart spaces and ubiquitous computing scenarios that generally include their own proprietary discovery mechanisms. Another example is the Gaia middleware infrastructure [Roman02], which supports similar smart spaces.

2.6.10 Analysis of Discovery Protocols

The four established discovery protocols (SLP, Jini, UPnP and Salutation) offer varying levels of suitability for wireless networks. Jini and Salutation are large implementations not suited to limited memory capacity (although Salutation Lite attempts to address this). Furthermore, Jini relies upon centralised network components to be available to connect to. SLP, UPnP, and MARE address this problem by using multicast communication to remove reliance on centralised elements. Hence, elements may become disconnected, and services can be found as the user changes location. However, SLP and UPnP utilise verbose text-based protocols that in turn consume limited bandwidth and power.







Mobile Challenge	Challenge Addressed?	Examples
Disconnection		UPnP, SLP, MARE, JESA
Low Bandwidth		MARE
Variable Bandwidth		
Address Migration		UPnP, SLP, MARE, JESA
Low Power		MARE
Small Storage Capacity		Salutation Lite, MARE, JESA,

Table 2.5 Challenges of mobile computing met by service discovery middleware

Given these constraints, new discovery protocols especially designed for wireless networks have been developed. For example, JESA enhances the Jini architecture to operate across wireless networks without centralised entities and using a lightweight implementation. Furthermore, the MARE platform promotes an efficient discovery protocol that does not waste network and device resources. Table 2.5 illustrates that service discovery protocols have emerged to meet the challenges of mobile computing (variable bandwidth has not been addressed, as this is not a significant issue in service discovery).

2.7 Adaptive Middleware

2.7.1 Overview

The previous middleware implementations in general offer fixed platforms; that is, they do not support the level of reconfiguration required to accommodate mobile computing in the face of wide ranging context changes. The mobile environment is dynamic in nature; the environmental context changes (e.g. network bandwidth) and the end system resources fluctuate. Therefore, solutions have emerged that adapt middleware behaviour to ensure that an application maintains the best level of service in the face of these changes. Many techniques have been used to dynamically adapt software; in this section we examine two complementary mechanisms that have been applied in the middleware domain, namely reflection (which offers open access to system inspection and reconfiguration) and policies (which support mechanisms to control dynamic changes). Other adaptive middleware, e.g. reconfigurable protocol stacks, are not considered in order to limit the scope of this section.

2.7.2 Reflective Middleware

Overview

A reflective system maintains a representation (the *meta-space*) of its own behaviour that is available for introspection and adaptation. Fundamentally, this meta representation is *causally connected* to the underlying behaviour it describes. This causal connection ensures that changes made to the self-representation are mirrored in the underlying system's state and behaviour, and vice-versa. This technique has been used in language design, for example the Java Core Reflection API [Sun02], operating system design [Yokote92] and concurrent languages [Watanabe87]. However, in this section we focus on the use of reflection in middleware.

The reflective middleware community identified that well-established middleware like CORBA, EJB and DCOM maintain a black-box philosophy, whereby a fixed service is available to users, and it is typically impossible to view or alter this implementation. Hence, [Blair01] propose that the next generation of middleware platforms should be configurable, to meet the needs of a given application domain, dynamically reconfigurable to enable the platforms to respond to changes in their environment, and

evolvable to meet the needs of changing platform design. This openness is achieved by applying reflection to middleware.

The key to the reflective approach is to offer a meta-interface, or *meta-object protocol (MOP)*, supporting access to the engineering of the underlying platform. This MOP provides operations to inspect the internal details of a middleware (*introspection*), and by exposing the underlying implementation it is also possible to insert behaviour, e.g. quality of service monitors. In addition, the MOP typically provides operations to alter the underlying middleware (*adaptation*), e.g. changing the implementation of the underlying transport protocol to operate efficiently over a wireless link or inserting a filter to reduce the bandwidth requirements of a media stream. Reflective middleware platforms typically offer two styles of reflection:

- *Structural reflection* supports introspection of the underlying system structure, often in terms of the set of interfaces supported. Also supported is the adapting of system behaviour; the MOP provides access to make changes to the architecture of the system, e.g. in terms of components and connectors.
- *Behavioural reflection* is concerned with introspection and adaptation of activity in the underlying system, e.g. in terms of the arrival of invocations. Typical mechanisms include interceptors and dynamic proxies.

A significant number of reflective middleware platforms have emerged, including: OpenORB [Blair01], Dynamic TAO [Kon00], Flexinet [Hayton98], K-ORB [TCD00] and MULTE-ORB [Kristensen00], which are described in turn. Notably, UIC [Roman01] is a reflective middleware designed to tackle middleware heterogeneity in mobile environments, and is hence investigated in chapter 3 of this thesis.

OpenORB

The OpenORB [Blair01] design philosophy promotes a marriage of reflection, component technologies and component frameworks, to create families of reflective middleware. Components are the building blocks of the middleware, where a component is “a unit of composition with contractually specified interfaces, which can be deployed independently and is subject to third party creation” [Szyperski98]. This technique promotes configurability, re-configurability and re-use at the middleware level. Reflection is used to provide a principled mechanism to inspect and

dynamically adapt the component structure. Finally, component frameworks constrain the design space and the scope for evolution, where a component framework (CF) is defined as a collection of rules and contracts that govern the interaction of a set of components [Szyperski98].

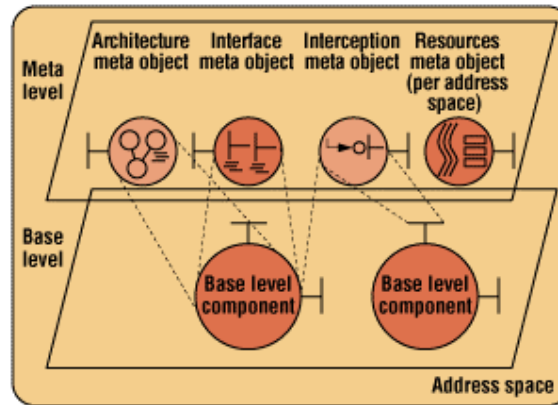


Figure 2.11 The Meta-Space structure of OpenORB

Figure 2.11 illustrates the meta-space model that forms the basis of the OpenORB design. Every component offers a *meta-interface* allowing access to an underlying *meta-space*. Four distinct meta-models represent the meta-space: the *interface*, *architecture*, *interception* and *resource* meta-models. Where the interface and architecture meta-models support structural reflection, and the interception and resource meta-models support behavioural reflection.

The *interface meta-model* provides access to the external representation of a component in terms of the set of provided and required interfaces. Furthermore, the *architecture meta-model* accesses the software architecture of a component represented by two elements: a *component graph* and a set of *architectural constraints*. The component graph is represented by a set of connected components, where a connection maps between a required and provided interface in the same address space. Hence, the architecture meta-model can be used to both discover and make changes to this structure at run-time.

The *interception meta-model* enables the dynamic insertion of *interceptors*, which enable the insertion of pre- and post- behaviour onto interfaces. In contrast, the

resources meta-model offers access to underlying resources and resource management [Duran00], and is based upon the abstractions of *resources* and *tasks*. Resources can be either primitive (e.g. raw memory or OS threads) or complex (e.g. buffers or user-level threads multiplexed on to kernel-level threads). Tasks are the logical unit of activity in the system, which have a pool of resources that support their execution. As with other meta-models, it is then possible to either inspect or adapt activity associated with resources.

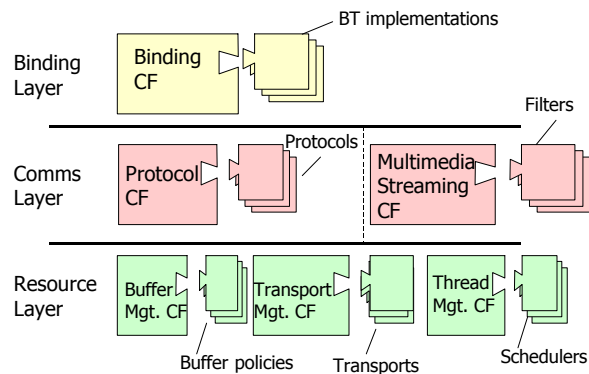


Figure 2.12 **The component frameworks of Open ORB**

Figure 2.12 illustrates the architecture of the OpenORB middleware decomposed into an extensible set of component frameworks, such as buffer management and binding establishment. Hence, OpenORB is structured as a set of configurable and reconfigurable component frameworks, and reflection is then used to discover the current structure and behaviour, and to enable selected changes at run-time. The end result is a flexible middleware technology that has been specialised to domains including multimedia and real-time systems.

DynamicTAO

DynamicTAO [Kon00] is a reflective CORBA ORB built as an extension of TAO [Schmidt99]. Where TAO is a portable, flexible, extensible, and configurable ORB that conforms to the CORBA standard and utilises the Strategy design pattern [Gamma95] to encapsulate different aspects of the ORB internal engine. TAO contains a configuration file that specifies the strategies the ORB uses to implement aspects like concurrency, request de-multiplexing, scheduling, and connection

management. When the ORB is initiated, the configuration file is parsed and the selected strategies are loaded. TAO is used in static real-time applications.

DynamicTAO extends TAO to support on-the-fly reconfiguration; this is achieved by keeping an explicit representation of the ORB internal components, and of the dynamic interactions among them. The ORB is then able to change specific strategies without having to restart its execution; this process is managed by component configurators [Kon00b]. A component configurator maintains the dependencies between a component and other system components. The process running a *dynamicTAO* ORB contains a configurator called the *DomainConfigurator*, which is responsible for maintaining references to instances of the ORB and to servants running in that process. In addition, each instance of the ORB contains a customized configurator called the *TAOConfigurator* that contains hooks to which implementations of *dynamicTAO* strategies are attached. Figure 2.13 illustrates these features within a process containing a single instance of the ORB.

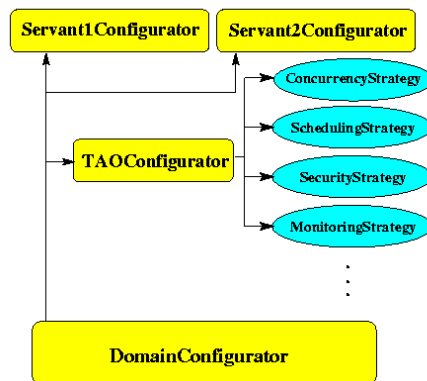


Figure 2.13 Reifying the *dynamicTAO* structure [Roman01]

This reflective mechanism supports inspection and reconfiguration of the ORB. This is achieved by exporting an interface for (1) transferring components across the distributed system, (2) loading and unloading modules into the ORB runtime, and (3) inspecting and modifying the ORB configuration state.

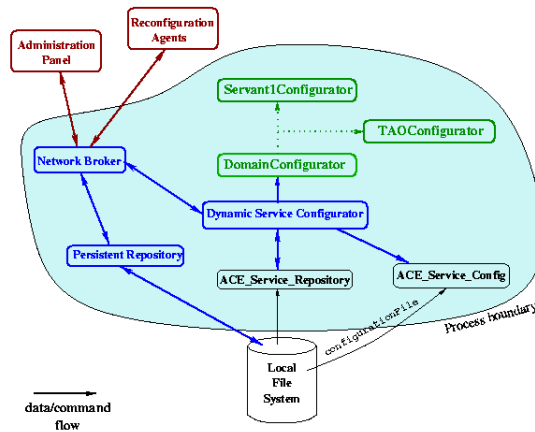


Figure 2.14 **dynamicTAO Components [Roman01]**

The *dynamicTAO* architectural framework is depicted in Figure 2.14. The *Persistent Repository* stores category implementations in the local file system. Once a component implementation is stored in the local repository, it can be dynamically loaded into the process runtime. A *Network Broker* receives reconfiguration requests from the network and forwards them to the *Dynamic Service Configurator*. The latter contains the *DomainConfigurator* (shown in Figure 2.13) and supplies common operations for dynamic configuration of components at runtime.

Flexinet

The FlexiNet platform [Hayton98] is a component-based Java based middleware platform that emphasises the use of reflection within the protocol stack. There are four key elements of the FlexiNet architecture: software components, transparent component binding, policy definition, and automated deployment.

The component model is based upon bindings between components, so that a programmer may locate one from another. Components may pass references between each other in a transparent way; in these circumstances, FlexiNet associates the implicit binding request with the relevant policies and ensures that the constructed binding respects these policies. Notably, a reflective protocol stack is related to the binding to carry out the call process. FlexiNet provides a layered protocol stack, in which the layers can be viewed as reflective meta-objects that manipulate an invocation using Java Core Reflection [Sun02]. Reflection allows the component to

have an open implementation; depending on what is required, the component can adapt itself by adding or removing sub-components that provide a degree of functionality. This means that rather than altering a stack of micro-protocols, the more complex layers of the FlexiNet architecture adapt themselves to changes in the environment.

K-ORB

K-ORBs [TCD00] are instantiations of the minimumCORBA framework (a subset of OMG's CORBA 2.2 specification that is targeted at resource constrained environments) that allows developers to build ORBs for domains such as embedded systems, PDAs, intelligent devices and real-time systems. The K-ORB framework is an extension of the Mobile IIOP Engine developed in the Alice Project [Haahr00].

The K-ORB framework allows developers to build ORBs where the environment and set of resources available to the ORB are subject to change at runtime. Mobile devices, for example, use the dynamic reconfiguration of the network protocol to select the most appropriate underlying network transport at runtime. For example, when a PDA with a GSM modem disconnects from the fixed network (an Ethernet or Wireless Ethernet connection), its transport protocol is dynamically reconfigured to Mobile IIOP (TCP/IP over GSM) to enable CORBA clients and servers on the PDA to recommence communication with hosts on the fixed network. Similarly, when a PDA with a GSM modem connects from the fixed network, its transport protocol is dynamically reconfigured from Mobile IIOP to IIOP to provide higher bandwidth connections for CORBA clients and servers on the PDA. Hence, the K-ORB framework uses dynamic reconfiguration to tackle many of the challenges of mobile computing e.g. disconnection, address migration, low bandwidth, small memory capacity and variable bandwidth.

MULTE-ORB

MULTE-ORB [Kristensen00] is a reflective multimedia object request broker. The main programming models behind MULTE-ORB are explicit stream bindings, stream interfaces and flows. A binding type identifies the type of stream interfaces that participate in the binding, where a stream interface consists of source and sink media

flows. Reflection is provided through reification of the composition of the binding, making it available for inspection and adaptation through meta-object protocols. Furthermore, Quality of Service management is supported by a set of components monitoring the behaviour of the system and binding controllers that encapsulate user policies for reconfiguration and adaptation.

2.7.3 Policy based Adaptive Middleware

Overview

Rather than perform transparent adaptation of middleware behaviour (e.g. as performed by Ad-hocFS [Boulkenafed03] and CODA [Satyanarayanan90]), the application is in a better position to determine how to adapt to context changes. Hence, systems promoting application-aware adaptation have emerged [Satyanarayanan96]. These generally allow the application to state its rules for adaptation as a policy that can be interpreted by the underlying middleware. For each particular condition the matching rule is applied to change the middleware behaviour. Four example systems that promote this technique are described in turn.

Odyssey

Odyssey [Satyanarayanan96] is an extension to the file sharing system CODA, designed to support access to shared information from mobile hosts. Information is stored on remote, reliable and centralised servers and Odyssey supports the access to this by mobile clients. The application specifies the policies to adapt the behaviour of the platform in terms of utilisation of system resources. Interest is registered in particular resources, and for each resource an application resource handler is also created. A Viceroy component then monitors the resources being utilised by the applications; when the resource availability drops below a set value, the resource handler of the application is invoked. This notifies the application that it needs to adapt its behaviour to cope with the change.

Puppeteer

Puppeteer [Flinn01] is an adaptive component based middleware system to explicitly manage the energy consumption of mobile devices. It concentrates on the distribution and presentation of media and documents, for example Microsoft Powerpoint

presentations, and reduces their energy resource use. By utilising the exported APIs of each application and the structured nature of the documents Puppeteer modifies the behaviour of the application without access to source code.

The Puppeteer architecture consists of four tiers: the applications to be adapted, Puppeteer local proxies, Puppeteer remote proxies and data servers. The applications and data servers send all communication through the Puppeteer local and remote proxies, who are responsible for performing adaptation. The remote proxy parses requested documents, exposes their component structure (as a tree) and associates the data with each node. This skeletal structure is then returned to the local proxy, which based upon the application specific policy, fetches sets of elements from within the skeleton at a specified fidelity. The application is then updated with this newly fetched data. Hence, Puppeteer supports two forms of policy-based adaptation: 1) sub setting adaptation, where only parts of the document are presented to the application, and 2) versioning adaptation where a different version of a document is presented e.g. a low-resolution image.

The Lancaster Context Architecture

Applications need to adapt to multiple contexts; however, adaptive behaviour triggered by one attribute can cause side effects for other attributes. These can in turn create conflicting actions e.g. a request to reduce power consumption enforces applications using the network to postpone their activities, as a consequence the network bandwidth increases and this could trigger a request to applications to utilise the spare bandwidth (i.e. the two are in conflict). Therefore, as well as supporting multiple context types co-ordination between adapting mechanisms is also needed [Efstratiou02].

Researchers at Lancaster University have presented an architecture that provides a common space for co-ordinated system wide interaction between adaptive applications and a complete set of context attributes. The system they present decouples adaptation policies and mechanisms (illustrated in figure 2.15). The *context space* acts as a repository for context information, storing information from the device monitors, applications and middleware for use by adaptation strategies. The *adaptation control module* is a key component of the architecture driven by a set of

adaptation policies; it is responsible for co-ordinating adaptations and resolving potential conflicts. Furthermore, it is identified that the decoupling property of the architecture allows it to be integrated with a range of existing platforms such as the event-based, tuple spaces and object-based middleware described earlier.

It is clear that multiple context information must be supported for mobile applications and that the resolution of conflicts is an important research issue. However, the proposed architecture presents additional overhead that may not be suitable for all mobile devices. Devices that only execute a single application (e.g. Palm OS) do not need to resolve conflict; therefore, the architecture may waste valuable resources.

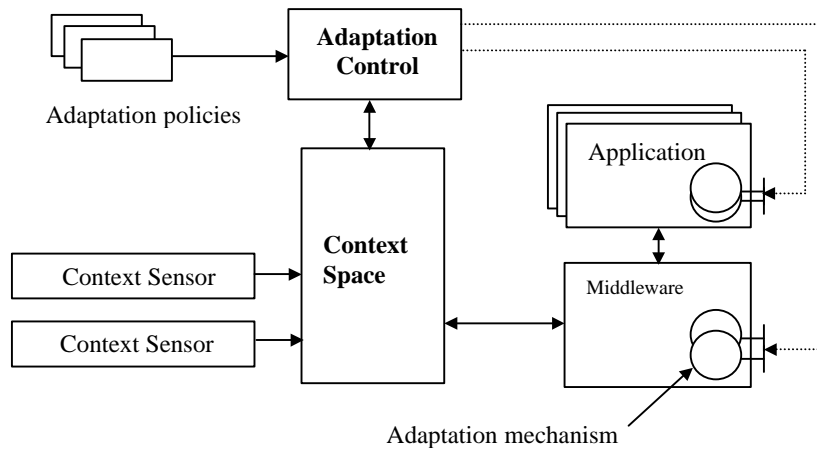


Figure 2.15 Architecture to support adaptive applications

CHARISMA

The CHARISMA platform [Capra01] developed at University College London is a reflective policy-based framework for adapting the behaviour and operation of an underlying middleware platform. In their case they utilise the XMIDDLE data-sharing platform (see section 2.4.4), although the generic features of the framework make it applicable to other middleware solutions [Capra02]. The work concentrates specifically on the important issue of how context information (e.g. device context e.g. power, memory, etc and external context e.g. network connection, bandwidth, location etc.) affects the performance of a mobile application and how middleware adaptation

can be performed to maintain the best level of performance in the face of these changes.

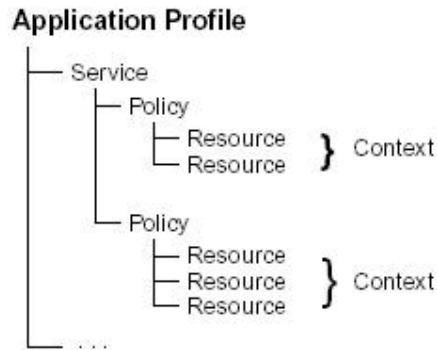


Figure 2.16 A CHARISMA application profile [Capra02]

In a specific context, an application may require the middleware to behave in a particular way e.g. an image processing application may ask to display pictures in black and white rather than colour when the battery power is low. Each application describes their adaptation requirements in an application profile. This contains associations between the services that the middleware delivers, the policies that can be applied to deliver the services, and the context configurations that must hold in order for a policy to be applied. An example policy is illustrated in figure 2.16. Hence, for the previously described example the middleware service 'DisplayPicture' may have two policies: the 'Black&White' policy with a context of 'Battery Power low', and a 'Colour' policy with context 'battery power high'. Each time the application invokes DisplayPicture the middleware consults the required profile and then selects the appropriate policy based upon the current context.

Every application submits its policy to the middleware upon initialization, however, given the dynamic nature of mobile applications it is expected that the policies themselves need to be changed dynamically. Therefore CHARISMA provides a reflective API that allows introspection and dynamic reconfiguration of this policy. CHARISMA also manages the end-system resources of the mobile device being utilised by competing mobile applications. Different policies have different non-functional requirements e.g. the present Quality of Service is different, and they also utilise different amounts of resources. The resolution of these conflicts is resolved by an auction protocol [Capra02b]. Each application submits a bid for resource use citing

non-functional concerns e.g. security, performance, availability etc. The resource goes to the highest bidder. In a similar fashion, reflection allows the application to dynamically change the non-functional properties of its bid if its requirements dynamically change.

This is a loosely coupled framework supporting behavioural reflection; the meta-level provides a description of the middleware's operation under certain conditions, and for individual applications, rather than the actual platform structure. The policies act as the middleware meta-level and any changes to these documents create an implicit change in the middleware's behaviour rather than the platform structure. CHARISMA cannot be singularly classified as either a reflective middleware or a policy based adaptive middleware; rather it bridges the two and provides a generic higher-level framework.

2.7.4 Analysis of Adaptive Middleware

The previous sections of this chapter demonstrated that fixed middleware overcomes the individual problems of mobile computing e.g. weak connection, address migration and poor bandwidth. However, these fixed solutions do not support well the challenges of changing user and environmental context, fluctuating network conditions and inconsistent device resources. Hence, adaptive middleware has been developed to best support individual mobile applications to react to changes in the environment, and continue to provide the best operation. Table 2.6 describes how adaptive middleware addresses the challenges of mobility. This shows that these solutions aim to solve the problems of variable network QoS and variable end-system resources. The frameworks generally do not examine disconnection and address migration, which are the responsibility of core middleware implementation (K-ORB is the notable exception).

With the exception of K-ORB and UIC (see section 3.7), the reflective middleware platforms were not designed explicitly for mobility and therefore suffer from the problems of exhausting resources through their implementation. Hence, this poses the question as to whether the resource heavy technique of reflection is suited to the mobile domain.

Mobile Challenge	Challenge addressed	Example
Disconnection	✓	K-ORB
Low Bandwidth	✓	OpenORB, DynamicTAO, K-ORB, Puppeteer, CHARISMA, Odyssey
Variable Bandwidth	✓	OpenORB, DynamicTAO, K-ORB, Puppeteer, CHARISMA, Odyssey
Address Migration	✓	K-ORB
Low Power	✓	Puppeteer, CHARISMA, Odyssey
Small Storage Capacity	✓	K-ORB, CHARISMA, Puppeteer, Odyssey

Table 2.6 Challenges of mobile computing addressed by adaptive middleware

Policy driven mechanisms can be seen as a higher-level framework for adaptation i.e. they sit above reflective middleware, which concentrates on low-level middleware change. They offer the key benefit of supporting reaction to given context changes and events, ensuring the correct policy is executed. Therefore, a policy framework when applied in the domain of mobile computing would better serve the changes made by reflective middleware e.g. OpenORB and DynamicTAO.

2.8 Conclusions

This chapter has illustrated the many middleware solutions that are now available to mobile application developers. These offer a spectrum of middleware styles e.g. remote method invocation, publish-subscribe, tuple spaces, data sharing, agents, service discovery and adaptive middleware. There are many challenges within the mobile domain, and advancements have been made to meet these problems, for example publish-subscribe paradigms to solve weak connection. In addition, combinations of these technologies (hybrid architectures) can improve system operation e.g. a CORBA enhancement combined with an adaptive framework to support changing QoS.

However, in general most of these solutions are not standards based. Each presents its own standard for application developers to adopt. This creates a domain populated

with heterogeneous middleware platforms that are inoperable with one another. There is heterogeneity between middleware styles, e.g. publish-subscribe, RMI, agents and data-sharing, and these cannot interoperate with one another. Furthermore, there are heterogeneous implementations of each paradigm e.g. SOAP and CORBA for remote method invocation, SLP, UPnP and Jini for service discovery, and CEA and Siena for publish-subscribe; again, the different implementations cannot interoperate because they do not conform to a common standard within that paradigm.

This middleware heterogeneity problem is likely to get significantly worse with the emergence of proprietary middleware solutions for the domain of ubiquitous computing (cf. Gaia and Centaurus). No single middleware solution will “win” because: 1) different styles of middleware are better suited to different classes of application, 2) one middleware is better than another at dealing with a specific mobility problem, 3) these middleware are already well used and understood by developers, and 4) legacy applications and implementation that use them are already in place.

The issue of tackling middleware heterogeneity is investigated in the following chapter. In particular, the author believes that mobile computing requires higher-level middleware frameworks that are able to deal in an integrated manner with the multitude of service implementations that a client may encounter as it moves from location to location. Hence, adaptive middleware has the potential to support such behaviour, however it has yet to be fully investigated as a solution to this problem (The UIC middleware platform [**Roman01**], which presents an initial solution is investigated in detail in section 3.7).

Chapter 3 Tackling Middleware Heterogeneity

3.1 Introduction

As described in the previous chapter, a range of middleware solutions have been developed to tackle the challenges of mobile computing. However, these solutions add to the escalating problem of middleware heterogeneity. The different styles of middleware (e.g. RMI, tuple space, publish-subscribe etc.) do not interoperate with one another. In addition, individual implementations of middleware paradigms e.g. CORBA and SOAP (for RMI), and LIME and L²imbo (for tuple space) cannot interact. Chapter 2 concluded that this is an important problem in the mobile domain. The next generation of mobile applications will operate across multiple locations consisting of unknown and heterogeneous middleware implementation. Hence, it is not sufficient to develop client applications upon a single platform type. The middleware heterogeneity problem must be tackled to provide the support to interact with newly discovered services.

This chapter examines and evaluates the state of the art in tackling middleware heterogeneity. The problem of middleware heterogeneity in the fixed network is now well documented, and a number of contrasting solutions have emerged. These range from: new higher-level interoperation architectures, solutions based upon software bridges between middleware domains, and the exchange of mobile code. Notably, one platform has examined the problem in the mobile domain. The Universal Interoperable Core is a reflective middleware that uses dynamic adaptation to address the problem. In the following sections, each of these will be analysed in turn.

3.2 Web Services Architecture

3.2.1 Overview

The Web Services Architecture (WSA) [Booth03] is an evolving open standard whose goal is to ensure interoperability between software applications running on a variety of platforms and/or frameworks by utilising the technologies of the World Wide Web. WSA is a *Service Oriented Architecture*, where a service is a software agent that provides functionality on behalf of its owner through well-defined operations.

Requesters (client applications or other Web Services) make use of these services through the exchange of XML messages. The Web Services standard is rapidly expanding, and this description captures the essence of its core features; future extensions, including semantic based service agreement, can be followed in the work of the Web Services Architecture Working Group (www.w3.org/2002/ws/arch/).

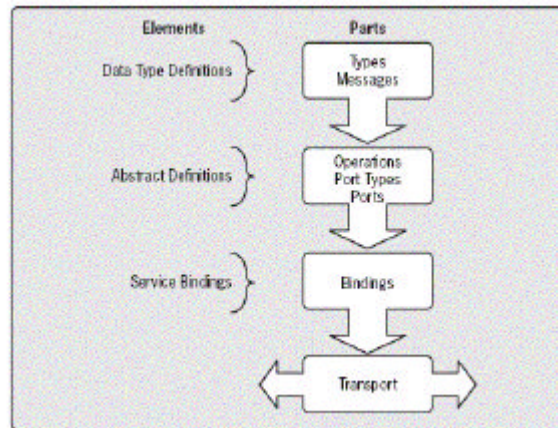


Figure 3.1. **The elements of WSDL [Newcomer02]**

A Web Service is an abstract entity, whose service description (interface) is documented using the XML based Web Service Description Language (WSDL) [Chinnici03]. WSDL is made up of three parts, as illustrated in figure 3.1. Firstly, a standard method for describing the *data types* passed in messages; using XML provides a standard, flexible and extensible data format, which overcomes the difficulties of different platform's type systems. Secondly, the abstract definitions of the *service's operations*; these are described in terms of a loosely coupled message exchange between requestor and provider. Four styles of abstract operation are available; hence, Web Services can abstractly describe RPC, publish-subscribe and asynchronous messaging:

- *Request-Response (input message followed by an output message)*, a service receives a request of its functionality and responds to it.
- *Solicit-Response (an output message followed by an input message)*, a service provider acts as a service requestor.
- *One-Way (an input message)*, a service receives a notification message.
- *Notification (an output message)*, a service outputs a notification message.

The WSDL specification is a work in progress, and its development can be followed on the Web Services activity page (<http://www.w3.org/2002/ws/>).

Thirdly, a *service binding* describes the network transport protocol that will carry messages between interacting agents. A concrete agent that physically sends and receives these messages then implements the WSDL interface. Therefore, the power to overcome middleware heterogeneity with Web Services emerges from the separation of an abstract definition of a communication endpoint from its concrete implementation (or data format binding). A particular Web Service may be implemented by a SOAP based agent one day, and by a CORBA based agent the following day; the service or client application using the operations of this service continues interacting transparently.

WSDL is a primitive language for describing distributed systems; it does not include the description of non-functional characteristics e.g. QoS, cost, sequencing of operations and security properties. Extensions of the specifications to encompass these are likely to appear in the future; the Web Services Endpoint Language (WSEL) [Hung02] is an example of a language for describing non-functional properties. Furthermore, WSDL provides only simple interactions; for example, choreographing a collection of web services to achieve a particular application goal is not possible. Hence existing languages, e.g. Web Services Flow Language (WSFL) [Leyman01], XLANG [Thatte01], DAML-S [Ankolekar01], RDF [W3C99] propose complex interaction patterns and further extensions to Web Services description.

Figure 3.2 illustrates the technologies that underlie the Web Services Architecture. The abstract messages, described in WSDL, are encapsulated into SOAP messages (although the concept of Web Services does not discount other message formats), which may then bound to different transport protocols (e.g. HTTP, FTP, IIOP, JMS); specifications for SOAP to HTTP [Box00] and SMTP [Cunnings01] bindings have been defined. SOAP provides a protocol neutral format for secure, reliable, multi-party messaging; the information needed to invoke remote services can be serialised and transported across the wire and interpreted by the remote service regardless of its platform.

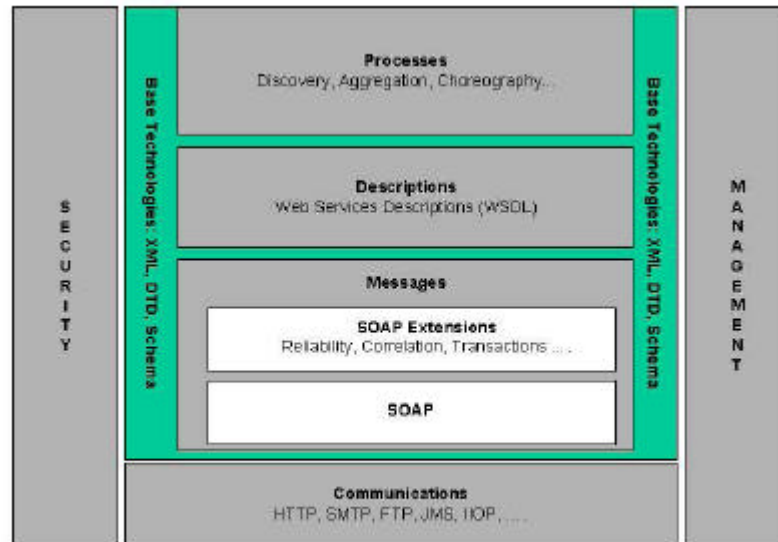


Figure 3.2 **Web Services Technologies** [Booth03]

An important role in Web Services is discovery; through open publication, software processes are available to use by a wide audience. Before a service requestor and provider can interact, the correspondents must agree on the service description and semantics of the interaction. Discovery can be performed with or without human intervention; a user can use a suitable discovery tool (c.f. Jini browser), or an autonomous agent can select a suitable service. The Web Services architecture does not specify how the discovery process is to be carried out; it may be a search engine process or a discovery protocol like Jini. However, in practice only the Universal Description, Discovery and Integration (UDDI) mechanism [Oasis02], a centralised registry architecture for WSDL interfaces, has been applied.

3.2.2 Analysis of Web Services

Web Services have been used as a distributed systems solution to expose new services across the Web, and hence to a wider audience; many technologies including: Microsoft's .NET platform [Microsoft00], IBM's Web Services Toolkit [IBM00], and Apache Axis [Apache03] are available for this. Furthermore, it is becoming well used as a tool for integrating existing middleware solutions, due to its loosely coupled

nature and reliance on XML messaging [Vinoski02]. For example, EJB and CORBA components can interact after being wrapped as Web Services.

The separation of abstract services from the concrete middleware implementation is potentially the key to overcoming heterogeneity. At present, Web Services technologies rely solely upon SOAP and do not consider different message bindings; new specifications as extensions to WSDL would be needed for each. However, using SOAP messaging and bindings means that existing service implementations must be re-implemented (or wrapped) as Web Services. Given the diversity of middleware available to mobile application developers this is unlikely to occur. Furthermore, the choice of an XML message format in both discovery and message exchange is more expensive than alternative protocols. Several studies have shown that SOAP and XML incur a substantial overhead compared to binary protocols [Bustamante00] [Davis02] [Govindaraju02]. These results show that SOAP is up to ten orders of magnitude slower than Java RMI when transmitting large data arrays, and that XML marshalling, un-marshalling and communication costs are between two and four orders of magnitude slower than IIOP. In addition XML message size is typically between six and eight times larger than the corresponding binary representation [Bustamante00].

The independence from specific discovery mechanisms (e.g. CORBA Naming Service, or Jini) overcomes the problems associated with heterogeneous discovery protocols. The architecture's discovery process requires that both interacting parties obtain the WSDL service description; however the majority of service discovery protocols provide their own service description, and/or have no mechanism to distribute an XML file (e.g. SLP and Salutation). Therefore, only certain discovery solutions are suitable for Web Services; this is why only UDDI has been applied in practice.

3.3 Web Services Invocation Framework (WSIF)

3.3.1 Overview

The Web Service Invocation Framework (WSIF) [Duftler01] is a Java API, originating at IBM and now an Apache release, for invoking Web Services irrespective of how

and where these services are provided. Its fundamental goal is to achieve a solution to better client and Web Service interoperability by freeing the Web Services Architecture from the restrictions of the SOAP messaging format. WSIF utilises the benefits of discovery and description of services in WSDL, but applied to a wider domain of middleware, not just SOAP and XML messages.

The structure of WSDL allows the same abstract interface to be implemented by multiple message binding formats, e.g. IIOP and SOAP; to support this, the WSDL schema needs to be extended to understand each format. Figure 3.3 illustrates an example of a WSDL binding statement for executing an operation upon an EJB component. Similar extensions are available for JMS and local Java classes. Hence, the same WSIF client code can, in theory, interact across any available binding. WSIF is a client side framework, none of its implementation resides at the service side, and therefore existing middleware solutions can be used in place. For example, a CORBA service can be exposed as a Web Service by creating and then advertising a WSDL description of the service.

```
<binding name="EJBBinding" type="tns:AddressBook">
  <ejb:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="typens:address"
      formatType="addressbook.wsiftypes.Address" />
    <format:typeMap typeName="xsd:string" formatType="java.lang.String" />
  </format:typeMapping>
  <operation name="addEntry">
    <ejb:operation
      methodName="addEntry" parameterOrder="name address"
      interface="remote" />
    <input name="AddEntryWholeNameRequest" />
  </operation>
</binding>
```

Figure 3.3 Example EJB binding in WSDL

The core of the framework is a pluggable architecture into which providers can be placed. A provider is a piece of code that supports each specific binding extension to the WSDL description, i.e. the provider uses the specification to map an invoked abstract operation to the correct message format for the underlying middleware. Figure 3.4 illustrates the operation of WSIF. A remote service is represented by its WSDL description. The client does not care how this is implemented; it simply needs to obtain the description dynamically, using a discovery process (typically UDDI). The client then loads and parses this to create its representation of the service, which

is responsible for generating the abstract operations for the client to invoke. When such an abstract operation is invoked, the WSIF provider takes this information and produces messages through serialisation; these correspond to the described binding mechanism, interact with the remote service and respond with the abstract results.

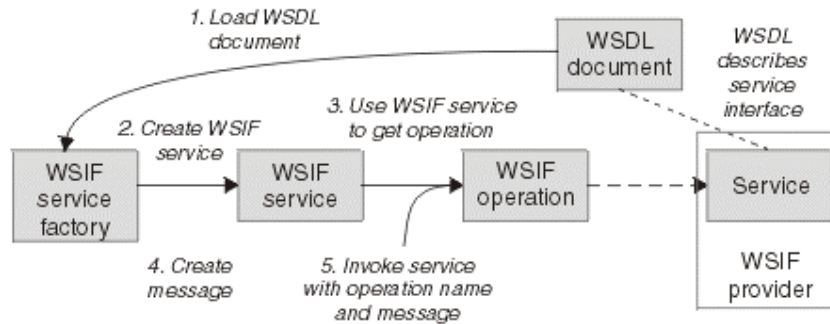


Figure 3.4. The WSIF Client Framework

Like Web Services, and CORBA before it, WSIF allows for both static and dynamic invocation. For static invocation, the stub is generated from the WSDL description and operations are invoked upon that stub. The Dynamic Invocation Interface follows the WSDL schema closely; the abstract input and output messages are constructed dynamically and then used to execute the operation. Furthermore, the API allows abstract operations to be invoked synchronously (*executeRequestResponse*) or asynchronously (*executeAsyncRequestResponse*) depending upon the developer's preference in how they want to receive results.

3.3.2 Analysis of WSIF

WSIF relies upon service developers exposing implementation as Web Services. The method of wrapping heterogeneous middleware services as web services has been criticised because the choreography of individual middleware platforms are not the same as the choreography of Web Services [Vinoski03]. For example, CORBA is both Service Oriented and Session Oriented. Exposing a session oriented CORBA object would cause specific CORBA implementation details, like remote object references, to appear in the abstract description; this is against the Web Services philosophy of separation. However, WSIF provides specified extensions to follow for each binding; hence, it disallows such details, enforcing the abstract Service Oriented Architecture

over that particular binding type. Hence, service providers must sensibly expose existing implementation, wrapping any binding specific interfaces to hide their details (without using a SOAP endpoint).

WSIF provides a remote method invocation programming style; the developer invokes abstract operations and receives their results (although the user can choose for this to be asynchronous). However, this does not take into account the different programming models of the various underlying middleware it abstracts from. The performance of *executeRequestResponse* over IIOP, SOAP, EJB and local Java classes will be predictable (a result or fault will be returned), as these follow the RMI paradigm. However, with event based middleware a request may be unanswered for some time, although this does not indicate an error has occurred. Therefore, developing in this style would lead to varied performance of the application depending upon the computational model of the current underlying paradigm.

WSIF follows the discovery model of web services, and requires new and existing services to be available through advertising of the WSDL file (e.g. in a UDDI registry). Like Web Services, the performance of the WSIF platform will suffer due to its reliance on XML in discovery. This doesn't account for heterogeneous discovery mechanisms and downloading the service description consumes bandwidth; for example, simple WSIF description files offering only one or two abstract operations are between 2Kbytes and 4Kbytes. Furthermore, services will be implemented and advertised without exposing a WSDL file; these cannot be interacted with, as the message exchange format cannot be determined. Hence, the technique requires that all providers follow this solution, which cannot be guaranteed.

To add a new provider type in WSIF (i.e. new binding format), a new binding extension to WSDL must be defined, and the serialisers and deserialisers for these elements must be created. These are then registered with a central registry; when an unknown binding type is encountered the information and implementation can be obtained from here. However, the reliance on a centralised architecture does not map well to mobile computing; it would require an accessible registry in every wireless network. Furthermore, new WSDL extensions are not open standards; therefore,

providers may implement multiple specifications of the same binding type introducing new interoperability problems.

3.4 Model Driven Architecture

3.4.1 Overview

The Model Driven Architecture (MDA) [Miller01] is an OMG specification that aims to support interoperability and integration throughout the systems lifecycle. The MDA defines how to specify an IT system in terms of system functionality, separated from its implementation on a particular platform. To perform this, the MDA is separated into key models, which are shown in figure 3.5. For creating MDA-based applications, the first step is to create a Platform Independent Model (PIM), which is expressed in UML. The PIM provides a formal specification of both the structure and function of the system, which is abstract from any technical details. Similarly, the Platform Specific Model (PSM) defines in UML how a PIM is realised on a particular platform e.g. EJB/CORBA, as shown in figure 3.5. This mapping of PIM to PSM UML descriptions can be automated for standard mappings (Each platform specific model is then physically implemented). Finally, the integration between alternative PSM implementations can be overcome by the automated insertion of a suitable bridging solution.

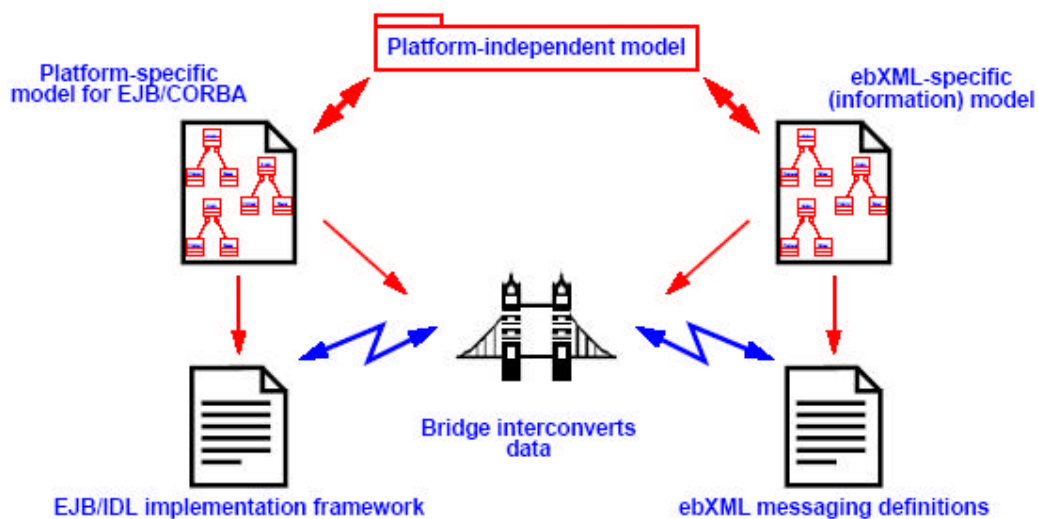


Figure 3.5 OMG's Model Driven Architecture [Miller01]

3.4.2 Analysis of Model Driven Architecture

The MDA is a powerful tool for specifying systems that may be composed of heterogeneous elements. The complete architecture can be designed at the abstract level and this viewpoint does not consider heterogeneity problems. Rather the automation of platform specific implementation (e.g. integrated through bridges) carries this out. As with web services, the solution to overcome heterogeneity is to provide a higher-level abstraction. However, the model is suited to system design and initial configuration; it does not deal with unforeseen changes in heterogeneity during the lifecycle. Therefore, while good for integrating systems in fixed networks, further research into how to specify and cope for dynamic change must be executed to support interoperability in the mobile computing domain.

3.5 Middleware Bridges

3.5.1 Overview

A software bridge is a process that enables communication between different middleware environments. Hence, clients in one middleware domain can interoperate with servers in another middleware domain. The bridge will take messages from a client in one format and then marshal this to the format of the server middleware; the response is then mapped to the original message format. Bridges can be static or dynamic. A static bridge requires a stub implementation to perform marshalling between endpoints, but must be recompiled if the interface of the service changes. A dynamic bridge provides a generic proxy that can be placed between endpoints and doesn't require recompilation if service interface changes.

Many Bridging solutions have been produced between established commercial platforms e.g. DCOM/CORBA and CORBA/SOAP; they are also used to connect proprietary middleware. However, this section seeks to illustrate the technique rather than exhaustively survey the state of the art, hence some examples are illustrated. For example, the OMG has created the DCOM/CORBA Inter-working specification [OMG97] that defines the bi-directional mapping between DCOM and CORBA and the locations of the bridge in the process. OrbixCOMet [IONA99] are implementations of the DCOM-CORBA bridge. SOAP2CORBA (<http://soap2corba.sourceforge.net>) is

an open source implementation of a fully functional bi-directional SOAP to CORBA bridge.

Bridging offers a solution to connect heterogeneous middleware, however it is a low level mechanism that must be supported by a higher-level abstraction (cf. Web Services and MDA) to fully support the integration of multiple platform types. The following section describes such a framework, with software bridges at the core of the architecture.

3.5.2 Unified Component Meta Model Framework (UNIFrame)

The UNIFrame approach [Shah03] attempts to unify distributed component models under a common meta-model to allow discovery, interoperability and collaboration between components using generative programming techniques. The key parts of the framework are: the Unified Meta Model (UMM), the Unified Component Interoperability framework (UCI) and automated system generation. The UCI framework is the technology involved in overcoming heterogeneity, so is described in further detail.

The UCI allows for the static and dynamic assembly of heterogeneous components. The architecture, described in figure 3.6, consists of platform independent formal specifications and a heterogeneous component integrator. The formal specification contains both the functionality and QoS contracts of the component. The component integrator is made up of a translator, an internal representation and the Middleware Bridge Generation Engine (MBGE). The translator takes the platform specific component specification and creates a platform independent specification. Then as seen in figure 3.6, the abstract representations of two components can be supplied to the MBGE to automatically produce a bridge between them to allow them to interoperate.

UNIFrame is similar to the MDA. However it differs in that the independent model is generated from the specific model (rather than the other way round). Furthermore, the architecture allows for the dynamic creation and insertion of bridges to overcome heterogeneity. This is a more suitable method for mobile computing and has been

applied in the domain. However, generating a bridge for each interoperation between components is an expensive operation that must be executed for each change in heterogeneity context. Given the dynamic nature of mobile environments it is likely that a new bridge would have to be generated frequently.

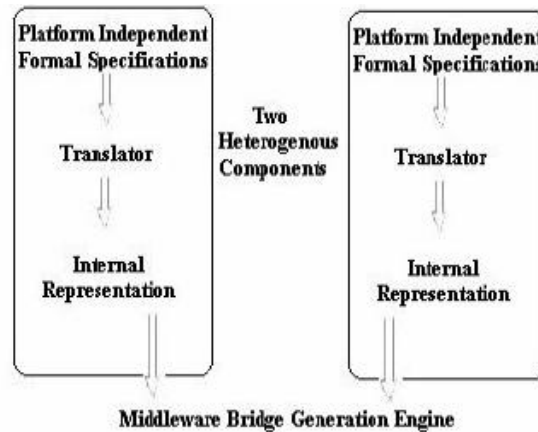


Figure 3.6 Architecture of the Unified Component Interoperability framework

3.5.3 Analysis of Middleware Bridges

There are two types of middleware bridge: static and dynamic. The static bridge is used in fixed networks to connect two fixed domains of middleware implementation, and consists of a complete mapping between two middleware implementations. Static bridging is not well suited to interoperation with multiple middleware types. Furthermore, static bridges are not suited to the mobile environment because they are a fixed component, which must reside in the network and clearly cannot be maintained in dynamically changing wireless networks. Dynamic bridges offer a specific mapping between two service implementations, and hence the technique can be used to support higher-level middleware abstractions (c.f. UNIFrame). This insertion of dynamic bridges is suited to the domain of mobile computing; however, the generation and insertion of bridges is an expensive operation that will occur frequently as the user moves.

3.6 Logical Mobility

3.6.1 Overview

The properties of logical mobility (mobile code) offer potential solutions to the problem of middleware heterogeneity. Service discovery and service interaction can be combined into a process whereby the client obtains both information about the service and the code directly to interact with the service. We examine two platforms in turn, one designed specifically to overcome heterogeneity (SATIN), and one whose properties offer a potential solution (Jini).

3.6.2 SATIN

SATIN [Zachariadis03] is a low footprint component based middleware, which aims to address the problem of heterogeneous service implementations in dynamically changing mobile environments. It argues that the use of logical mobility (code mobility) is limited within current mobile middleware platforms, but offers genuine benefits for interoperability.

In a scenario where a mobile host is able to access the local services of an ad-hoc network, the peer should be able to obtain code to discover its required services using the discovery mechanism in place and then use it. To do this, the SATIN architecture composes applications and the middleware itself into a set of capabilities (a unit of functionality), for example, a discovery mechanism or compression algorithm. Capabilities are registered with the host's core, which can be statically or dynamically configured. At the heart of SATIN is the ability to advertise and discover service implementations that may be advertised using different techniques; each discovery mechanism is represented by a different capability that can be added to the host when needed in the environment. SATIN then utilises its own "higher level" XML based discovery mechanism for initialisation; that is, the advertising mechanisms currently in use can be discovered. For example, a host uses SATIN to find the discovery capabilities being used and then downloads these. The required application services are looked up and their interaction capabilities are downloaded to complete the cycle.



Figure 3.7 **Capabilities in a SATIN application**

Figure 3.7 illustrates example SATIN capabilities in an application scenario. A conference offers a media stream that the mobile phone wishes to play on its media player. The phone discovers that MULTICASTADV is the discovery technology (using the abstract discovery protocol) and so downloads this capability. The remaining capabilities (a codec) to allow interoperation with the service can then be discovered and downloaded.

3.6.3 Jini

As described in section 2.5, applications download a Jini proxy as part of service discovery. This proxy interacts directly with the remote service, and although generally implemented as RMI, any middleware implementation could be used. Hence, the solution to heterogeneity is to wrap all code to access the service and the middleware into an agent that can then be downloaded and used by any device. Although a natural and elegant solution to the problem, it does not fully address the problems of heterogeneity. Firstly, Jini acts as a single discovery mechanism and any competing discovery technologies would be unusable. Furthermore, proxies are implemented in different styles; in some cases a complete application with a user interface will be available, in others a remote interface must be invoked by the discovering application. Therefore, developing applications to react to these differences would be a complex process. Finally, Jini relies on all parties understanding its architecture; however, as illustrated in chapter two different middleware implementations co-exist.

3.6.4 Analysis of Logical Mobility

The use of logical mobility provides an elegant solution to the problem of heterogeneity; applications do not need to know in advance the implementation details

of the services they will interoperate with, rather they simply use code that is dynamically available to them at run-time. SATIN offers an improved solution over Jini in that the problem of heterogeneous discovery mechanisms is addressed and resolved by a higher-level, albeit non-standardised, abstraction. Furthermore, it concentrates on a dynamic client-side architecture; hence services implemented independently of SATIN, can in theory still be utilised. However, both techniques are limited in fully addressing heterogeneity. Both Jini and SATIN rely on participants conforming to a least part of their architecture i.e. servers and clients both understand a Jini proxy, or the SATIN abstract discovery mechanism. Therefore, the solutions do not scale to include application services not implemented with knowledge of these techniques.

3.7 Universal Interoperable Core

3.7.1 Overview

The Universally Interoperable Core (UIC) [Roman01] is a reflective middleware, whose design is based upon the reflective architecture of DynamicTAO. The goal of the middleware is to support interactions with multiple service platforms from a mobile device in ubiquitous environments. UIC provides the capability to interact with a service implemented in CORBA, and also with the same service type implemented in Java RMI and SOAP.

UIC, like other reflective platforms, is implemented as a collection of components. Fundamentally, it provides a skeleton of abstract components that form the base architecture. To enable the system to have the properties of particular middleware platforms (e.g. CORBA), components are dynamically added to specialise the abstract components. A UIC *personality* is a particular instance of the UIC obtained after the specialization, as illustrated in figure 3.8. Personalities can be classified as client-side, server-side or both. UIC can also be classified as single-personality or multi-personality. A single-personality interacts with a single middleware platform, while a multi-personality UIC can interact with more than one platform at the same time.

The design of the platform is driven by the principle of *What You Need Is What You Get*. UIC identifies that existing middleware platforms contain all possible functionality, even if the application only uses a subset; this is not suitable for devices with limited resources. Therefore, UIC provides only the minimum required functionality to guarantee interoperability with existing middleware platforms.

UIC personalities can be configured either statically or dynamically. In static configurations, personalities are built at compile time by statically assembling all the components together. The result is a single personality that cannot be dynamically reconfigured, although it has a smaller memory footprint. In dynamic configurations, personalities are a collection of dynamically loadable libraries that can be fully reconfigured at run time. The main benefit of the dynamic configuration is the ability to modify the architecture of the personalities dynamically without affecting the applications (hence overcoming heterogeneity); however the size of the core increases, because tools for loading and unloading components are required.

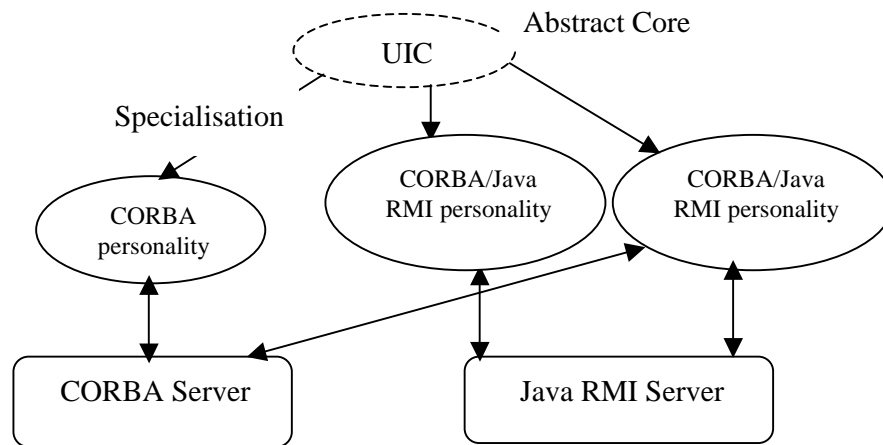


Figure 3.8 UIC Personalities [Roman01]

3.7.2 Analysis of Universal Interoperable Core

UIC uses dynamic adaptation to directly tackle the problem of heterogeneous middleware in mobile environments. This technique has the potential to address the changing middleware heterogeneity as the user moves location. However, the design of the platform defines a standard skeleton structure targeted to only object-oriented request brokers (CORBA, Java RMI, and DCOM); it offers no solution to the different

paradigms of mobile middleware (e.g. publish-subscribe, data-sharing etc.). In addition, UIC offers no higher-level abstraction to invoke heterogeneous services. The platform will operate for all RMI based implementations, but it cannot be extended to include contrasting communication paradigms. Furthermore, UIC does not address heterogeneous service discovery. It is utilised within a framework that offers a single discovery mechanism.

3.8 Conclusions

The solutions presented in this section demonstrate that the following key conclusions can be drawn about tackling middleware heterogeneity in the mobile domain.

- A higher-level abstraction as proposed by Web Services, MDA and UNIFrame is required to develop systems in which heterogeneous middleware components may interact. Requiring developers to conform to a higher-level abstraction rather than an individual discovery mechanism (e.g. SATIN, Jini) or middleware binding increases the chances that heterogeneity will be addressed.
- All of the described solutions depend upon a single style of discovery mechanism (fixed point for discovery). However, as seen in chapter 2 many discovery protocols are used in wireless networks and a single protocol cannot always be guaranteed to be available.
- Current implementations of the Web Services Architecture, WSIF, MDA and middleware bridges are designed for fixed networks. Hence, they consider interoperation between fixed endpoints and can solve the static heterogeneity in these scenarios. However, as mobile users change locations they will encounter changing heterogeneity. The stated platforms offer no support to detect or react to this changing context.
- To support mobile client interoperability, the implementation must not assume capabilities between communicating endpoints. Rather it is natural for one endpoint to discover the capabilities of the other and then dynamically adapt itself to mirror the implementation. Hence, adaptive middleware service binding (as seen in UIC) and adaptive service discovery offers an interesting approach to overcoming middleware heterogeneity.

4.1 Introduction

The previous chapters of this thesis have identified the problem of middleware heterogeneity, and the particular difficulties it poses to new classes of mobile applications. Current mobile middleware solutions escalate the problem, promoting their own standard and offering no support to address heterogeneity at the middleware level. Hence, mobile application services (for example, a tourist guide service) can be designed and implemented on a single middleware implementation, but these are then not open for use by applications and devices utilising a contrasting middleware implementation. The next generation of mobile applications envisage users being able to enter a new location and re-use their existing applications in this setting, making use of the available application services. In order to support this the client side middleware must maintain continuous interoperation with application services at new locations, which have been implemented upon heterogeneous middleware implementations.

The author argues that such a middleware must be an adaptive, abstract framework. With such an approach, no particular concrete middleware paradigm or standard is promoted; rather, the middleware adapts its underlying implementation to mirror the current environmental settings. For example, a tourist guide client application implemented upon the STEAM publish/subscribe middleware only operates in locations where the tourist service is implemented upon STEAM; however the same client developed upon an abstract middleware may interoperate with tourist services implemented on any middleware type (e.g. SOAP, CORBA, JEDI, STEAM...) and discovery protocol (SLP, UPnP); the abstract middleware then adapts to select the right protocols to match the environment. As an analogy, different screwdrivers are used for screws of different sizes; rather than attempt to create a screwdriver that works for every screw, you select the correct individual screwdriver for the task.

Therefore, there are two key requirements of an abstract middleware for mobile computing client applications:

- 1) An application can find the required application service functionality irrespective of the discovery protocol or discovery mechanism advertising it.
- 2) An application can interact with the found service irrespective of the middleware it is implemented upon and the messaging format it uses.

The author argues that reflection is the most suitable method for developing such a configurable and dynamically reconfigurable middleware; it provides a principled, as opposed to ad-hoc technique to make changes to the middleware implementation on the fly. Through inspection and adaptation the platform is dynamically evolvable. Therefore, there is scope to perform fine-grained changes to behaviour, e.g. change the current protocols, and also longer-lasting evolutions can be made, e.g. adding new functionality to the middleware framework.

The remainder of this thesis focuses on the design, implementation and evaluation of ReMMoC (Reflective Middleware for Mobile Computing), the middleware framework that meets these required characteristics. The approach taken in designing ReMMoC follows the OpenORB [Blair01] philosophy of using components, reflection and component frameworks (see section 2.7.2) to create a configurable and dynamically reconfigurable middleware. This is a scalable approach across application domains, hence it is suited to mobile computing and also allows the work to be applied to different domains in the future. Furthermore, the use of components and component frameworks provides platform extensibility. New functionality to improve the platform performance, or alter it per application domain can be added at a later date.

The chosen component model is described and compared against alternatives in section 4.2. Available component framework solutions are described in section 4.3, including the design of a new component framework model especially for ReMMoC. The remaining sections of this chapter then concentrate on the design and implementation of the core reflective architecture of ReMMoC. In particular, the following are described in detail: component configurations, algorithms for reconfiguration, and integrity maintenance mechanisms.

4.2 Components in ReMMoC

4.2.1 Overview of Components

A component is defined as: “*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*” [Szyperski98].

Interfaces are used to connect components, where each interface lists the operations that can be invoked by a client. Hence, the interface provides a contract between the client and provider. Component programming at the middleware level has the advantage of enhancing configuration, reconfiguration and re-use. Therefore, components become the middleware building blocks, acting as both units of composition and reconfiguration.

The ReMMoC middleware framework operates on mobile devices, which typically have limited end-system resources. That is, they are restricted by available memory and processor performance. The component model must address these concerns and operate in such an environment. Rather than implement a new component model suitable for this task, a set of available platforms were investigated (see below) as potential development environments, using the following requirements:

1. A Lightweight implementation. The static memory footprint must be suitable for limited end systems, e.g. PDAs, Smartphones and wearable computers.
2. Offer underlying support for openness and adaptation. Hence, a reflective middleware can be built upon these foundations.
3. Efficient performance. The extra cost incurred from using components is reduced to a minimum.

4.2.2 Investigation of Available Component Models

A number of both commercial and research based component models are now available; these offer potential solutions for a reflective middleware to be developed upon. Enterprise architectures including Enterprise Java Beans [Monson-Haefel00] and the CORBA Component Model [OMG02] are discounted, as they are heavyweight implementations that focus on service side implementation and do not meet the requirements of ReMMoC. The following is a description of component solutions that

were evaluated for this thesis (it does not aim to be an thorough overview of the state of the art in component technology):

- **Component Object Model (Microsoft COM)** [COM95] is built upon three concepts: unique interface specifications, unique components that implement multiple interfaces and dynamically discoverable interfaces (through the base IUnknown interface). Also, the COM standard defines the way components interoperate at the binary level in terms of a vtable (a table of function pointers based on C++ call conventions). COM can be used on mobile devices that run Windows CE 3.0 and above, and it performs efficiently, however there is no provision of reflective capabilities.
- **Java Beans** [Sun97b] is the component architecture of the Java language. Java classes are made into components (re-usable and composable elements) by implementing the serializable interface. Beans are linked together through events; each bean declares the events that it generates and the component users register for these. Attributes can be assigned to Beans, which can then be introspected along with the available events. Java Beans offer an interesting approach with provided support for reflection. However, its performance is less efficient compared to COM.
- **.NET** is an alternative component model from Microsoft [Microsoft00] that aims to simplify the software development process by reducing the complexity offered by COM. .NET components are built on a platform independent runtime called the Common Language Runtime (CLR). Assemblies are used to support sharing and reuse of code, where an Assembly contain the classes that implement the component functionality and metadata to describe the component. Reflective capabilities are available to fully introspect the metadata and component capabilities. CE.NET [Microsoft01] the version for Windows CE devices is only just emerging and was therefore unsuitable as a design choice.
- **THINK** [Fassino02] is one of many research based component models. THINK is designed for use in the domain of operating system kernels, and hence offers a highly efficient component solution. Furthermore, it is implemented as a Java component model and can operate on mobile and resource limited devices. A component is a run-time structure that encapsulates data and behaviour. An interface is the named interaction point, which can be of client type (operations

invoked from it) or server type (operations invoked on it). A component interacts with its environment only through interfaces. Notably, THINK also supports the binding between components in both a local and distributed fashion. However, THINK does not offer reflective support at the base component level.

- **OpenCOM [Clarke01]** is a lightweight, efficient and reflective component model that uses the core features of Microsoft COM to underpin its implementation; these include the binary level interoperability standard, Microsoft's IDL, COM's globally unique identifiers and the IUnknown interface. The higher-level features of COM, including distribution, persistence, transactions and security are not used. Notably, it was designed specifically for the implementation of an efficient version of the OpenORB reflective middleware [Blair01], hence offers support for building reflection functionality. The addition of reflective capabilities to the prior benefits of COM mean OpenCOM is ideally suited for reflective middleware development on mobile devices.

OpenCOM meets the three requirements of the component model; therefore it was chosen as the development platform for ReMMoC. The following section describes the OpenCOM architecture in more detail.

4.2.3 Background on OpenCOM

The key concepts of OpenCOM [Clarke01] are *interfaces*, *receptacles* and *connections*. Each component implements a set of custom interfaces and receptacles, as shown in figure 4.1. An interface expresses a unit of service provision, a receptacle describes a unit of service requirement and a connection is the binding between an interface and a receptacle of the same type. OpenCOM deploys a standard runtime substrate per address space (illustrated in figure 4.1) that manages the creation and deletion of components, acts upon requests to connect/disconnect components and provides service interfaces for reflective operations. The runtime substrate dynamically maintains a system graph of the components currently in use. The explicit maintenance of dynamic dependencies between components provides the support for introspection and reconfiguration of component configurations. The reflective interfaces of OpenCOM follow the meta-models proposed by OpenORB i.e.

the Interface meta-model (*IMetaInterface*), the Architecture meta-model (*IMetaArchitecture*) and the Behaviour meta-model (*IMetaInterception*).

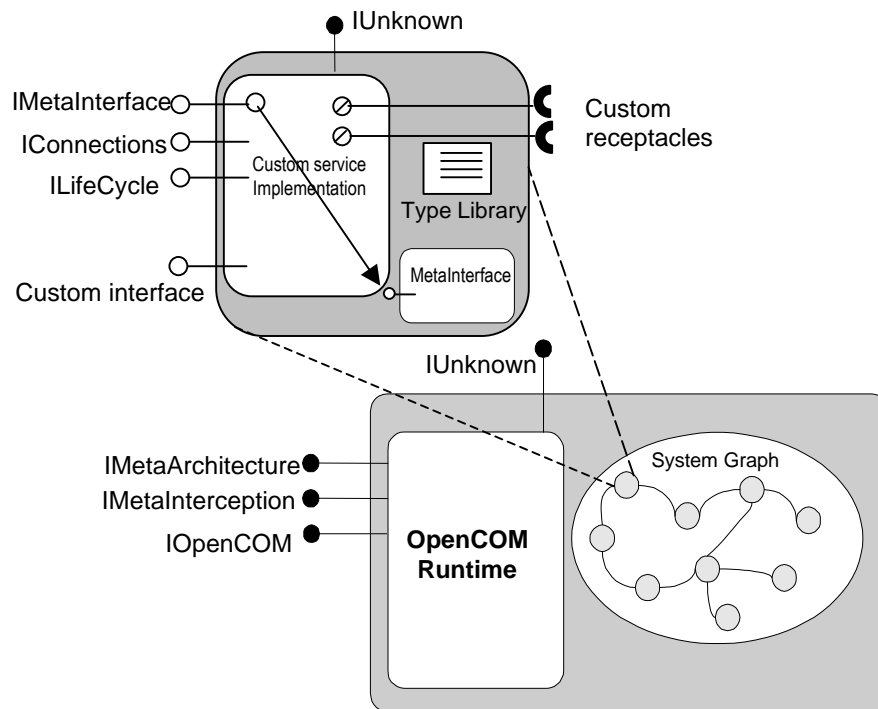


Figure 4.1 The OpenCOM architecture

In the first version of OpenCOM [Clarke01], every component implemented five standard interfaces: two component management interfaces (*ILifeCycle* and *IReceptacle*) and three meta-interfaces (*IMetaInterception*, *IMetaArchitecture* and *IMetaInterface*). However, the implementation of each interface increases the memory footprint size of a component, and in many cases this functionality is unused. Therefore, the newest version of OpenCOM (version 2) now requires only three interfaces to be implemented by each component, with the remaining reflective operations available from the runtime interfaces. The three base interfaces of a component are as follows:

- *ILifeCycle* provides operations called *startup* and *shutdown* that are called when a component is created or destroyed.
- *IConnections* (previously *IReceptacles*) offers methods to modify the interfaces connected to a component's receptacles. These are only called by the OpenCOM runtime component.

- *IMetaInterface* supports inspection of the types of interfaces and receptacles declared by the component. The meta information to support these operations is stored in the type library of each component.

In addition, the OpenCOM runtime provides a meta-interception (*IMetaInterception*) and a meta-architecture (*IMetaArchitecture*) interface. Interception enables pre and post methods to be associated with a given interface on a component; these are then invoked before or after every method invocation on that interface. The meta-architecture interface allows the programmer to obtain information about the underlying component architecture i.e. information about connections made to other components.

OpenCOM aims to be platform independent, and at present has been implemented on the Windows 32 bit, Windows Compact Edition (CE) and Linux operating systems. Notably, the minimal memory footprint of the Windows CE version (27.5 Kilobytes for devices with StrongARM processors) is ideally suited to ReMMoC's requirements.

4.3 Component Frameworks

4.3.1 Overview of Component Frameworks

Component frameworks are defined by Szyperski as “*a collection of rules and contracts that govern the interaction of a set of components*” [Szyperski98]. Therefore, a component framework enforces architectural principles on the components it supports; this is especially important in reflective architectures that dynamically change, and whose changes must be verified. The motivation behind component frameworks is to constrain the design space and the scope for evolution. Moreover, they simplify component assembly and increase the understanding and maintainability of the system. A component framework maintains an architecture consisting of a component graph and its constraints. Users interact with CFs for services through interfaces that encompass the operations of the CF's constituent components.

The component framework model used by ReMMoC is constrained by the choice of OpenCOM as the component model. However, the following section examines alternative existing component frameworks (to illustrate their functionality), as well as

a component framework model available to OpenCOM developers. The latter is investigated and identified as insufficient to meet the needs of ReMMoC. Therefore, section 4.3.3 documents the design and implementation of a generic component framework model that was created as part of the ReMMoC design.

4.3.2 Existing Component framework Models

The number of component framework models that are commercially available is limited, and often products that claim to be component frameworks do not match the requirements described above (e.g. JavaBeans and OLE/ActiveX) [Szyperki98]. This section therefore describes three component framework models in turn: OpenDOC, BlackBox and the OpenORB method.

- **OpenDOC** [Apple94] was a multi-platform, document-centric component framework developed initially by Apple in the mid-1990s. The core concept of OpenDOC is that of a document part. Every part can contain other parts and itself belongs to a compound document. Furthermore, every part has an associated part editor that can be used to edit that document part. Communication between the frameworks and components used SystemObjectModel (SOM), or a CORBA ORB. OpenDOC offers a flexible, generic component framework but does not promote reflective operations on contained document parts.
- The **BlackBox** [Oberon97] Component Framework (BCF) focuses on visual components (i.e. these concentrate on visual appearance and interaction with contained and containing components). BCF defines a general abstraction for containers, designed in a way that user interface details are hidden (the blackbox abstraction). Blackbox is not a generic architecture that is flexible to other domains, and like OpenDOC does not promote inspection and dynamic adaptation of contained elements.
- **OpenORB** [Blair01] is implemented using OpenCOM, and therefore promotes its own component framework model atop OpenCOM. A component framework is represented by a single component instance, known as a Component Framework Representative (CFR). This CFR defines a set of receptacles that define the components that can be plugged into the framework. The CFR is then implemented as a management element, ensuring that adaptation of its plug-ins

occurs on appropriate events. Hence, a framework of CFRs is used to impose the architecture of OpenORB, as illustrated in figure 2.12.

Due to the choice of OpenCOM as the component platform, the OpenORB solution is the only viable alternative for ReMMoC; however it has a number of limitations. A CFR implements a fixed set of interfaces that cannot be dynamically changed. However, in dynamic scenarios, the change of a component framework configuration may present new functionality that can be accessed through a new or different interface. Furthermore, the CFR only has knowledge of direct connections (plug-ins), allowing only architectures that conform to strict hierarchical trees to be maintained. This does not allow for other types of software architectures or patterns to be dynamically changed and maintained. Finally, a limited model of reflection is implemented to inspect and change CFR architectures, making reconfiguration code difficult to program and overly repetitive. Therefore, a new component model for OpenCOM is described in the following section that aims to address these issues and provide a generic model of component frameworks for OpenCOM.

4.3.3 ReMMoC's Component Framework Model

Overview

OpenCOM provides no base support for creating composite components (the key constituent of a component framework); only ad-hoc “architectures” of connected, primitive components can be created per OpenCOM runtime instance. Rather than a disadvantage, this allows specialised component frameworks per application domain to be developed atop. Furthermore, OpenCOM's reflective capabilities are limited to simple operations; therefore code to dynamically view and make elaborate changes to the system graph is repetitive and difficult to program. Therefore, this section documents the author's design and implementation of a component framework model suitable for both ReMMoC and other application domains, which aims to simplify the process of developing and maintaining architectures of components.

The design of this component framework model is based upon the concept of composite components proposed by OpenORB [Blair01] and promotes the following key properties:

1. A component framework in OpenCOM is a composite OpenCOM component.
2. A component framework provides an additional meta-interface for inspection and dynamic adaptation of the local architecture of the composite component.
3. The integrity of each component framework is maintained in the face of dynamic change, using developer specified architectural rules plugged into the component framework.

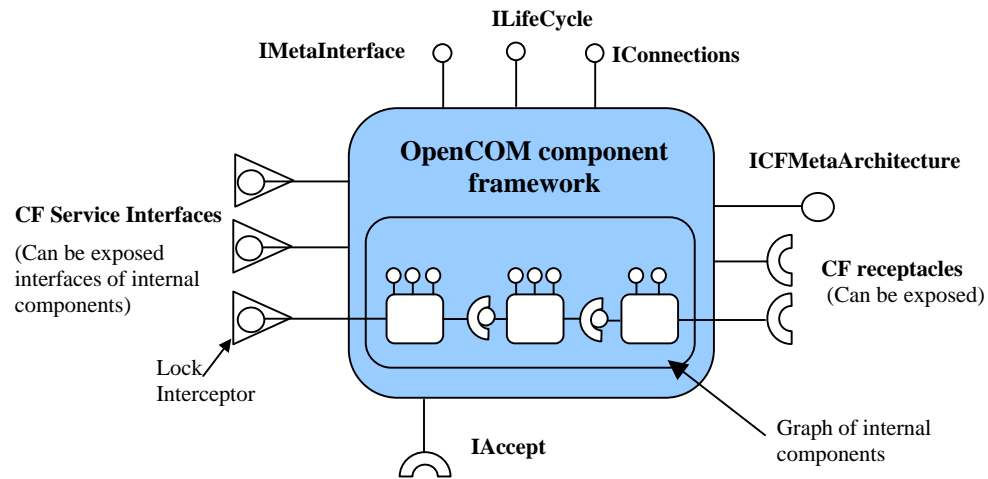


Figure 4.2 An OpenCOM component framework.

Therefore, component frameworks in OpenCOM are implemented as OpenCOM components; that is, they implement the same base interfaces (IMetaInterface, ILifeCycle and IConnections), custom service interfaces and receptacles. However, a CF contains its own internal structure (a configuration of components) that implements its functional capabilities, together with an additional meta-interface (ICFMetaArchitecture) to manipulate it. The diagram in figure 4.2 demonstrates the general architecture of a CF. The benefit of this design is that an OpenCOM CF can be treated as an OpenCOM component, simplifying the composition of hierarchical architectures and promoting re-use.

The Component Framework as a composite component

To be subject to introspection and dynamic reconfiguration, each component framework maintains a local graph of its internal structure. To reduce data duplication, this is simply a view of the information held in the OpenCOM system graph. Therefore, each CF maintains a list of Component Identifiers that point to their corresponding position in the system graph.

This local graph can be used for integrity checking of the framework after each reconfiguration, by ensuring that it meets the criteria for the particular domain of concern. Given that integrity checks apply per component framework the following rule for component composition must be followed: *“No individual component instance may exist in more than one component framework instance”*; if a component were to be changed in one framework it would also be changed in the other. The second framework has no knowledge of the change; hence its integrity would be compromised. However, the component framework model allows composition of component frameworks; therefore, hierarchical component structures can be created to resolve dependencies of this type. This is illustrated in figure 4.3. Component frameworks B and C both require component A; rather than place A into both B and C, the higher-level component framework D manages the two dependencies to the single instance.

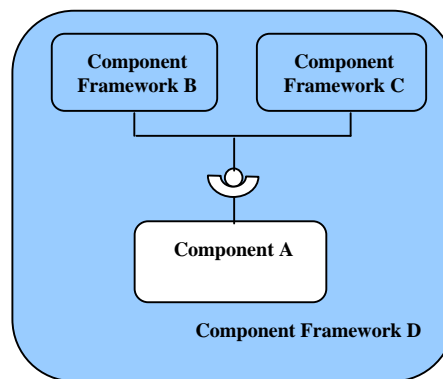


Figure 4.3 **Composition of Component Frameworks**

Reflection Operations on Component Frameworks

Every component framework is subject to the same reflective operations as a standard OpenCOM component (because a CF is an extended component). For example, the runtime `IMetaArchitecture` interface provides dynamic inspection of the external structure of a CF (i.e. what it is connected to), and the local `IMetaInterface` interface lists the interfaces and receptacles available from the CF. However, no OpenCOM interface supports inspection or dynamic adaptation of the internal structure of a component framework. Therefore, every CF implements its own additional meta-interface, named the **ICFMetaArchitecture** interface that consists of methods for introspection (table 4.1) and reconfiguration (table 4.2); the complete syntax of these

operations is listed in Appendix A. The implementation of these operations then relies upon the local graph meta-representation.

Operation Name	Description
get_internal_components	Returns a list of the components that make up the current component framework configuration.
get_Bound_Components	Returns a list of all components bound to a particular component.
get_internal_bindings	Returns a list of all connections within the current component framework configuration.

Table 4.1 **Operations for inspection of the internal CF structure**

Given that the component model offers a hierarchy of encapsulated (possibly unconnected) graph structures, the corresponding meta-model must allow recursive unfolding of these structures by introspection to allow reconfiguration to be applied at the correct level. Therefore, meta space is unfolded using the introspection methods of the ICFMetaArchitecture to find the components and configurations within a component, and IMetaInterface to view the interfaces and receptacles. Notably, primitive components only implement IMetaInterface, therefore the base of recursion is the querying of the ICFMetaArchitecture interface.

Operation Name	Description
insert_component	Create and insert a new component into this CF configuration.
remove_component	Delete a component from the configuration.
Replace_component	Replace an instance of one component with another, ensuring connections reconnected.
local_bind	Establish a local binding between two components from interface to receptacle.
break_local_bind	Break a local binding between two Components in the framework.
Expose_interface	Map the interface of an internal component as a new external interface of the CF.
unexpose_interface	Remove an exposed external interface.
Expose_receptacle	Map the receptacle of an internal component as a new receptacle of
unexpose_receptacle	Remove an exposed Receptacle.
Replace_configuration	Replace the current graph of components with a new component configuration.
init_arch_transaction	Initiate a transaction for architecture reconfiguration.
commit_arch_transaction	Completes the reconfiguration.
Rollback_arch_transaction	Rolls back changes made during a transaction.

Table 4.2 **Operations for dynamic reconfiguration**

An additional feature of the meta object protocol for component frameworks is the *expose_interface* and *expose_receptacle* operations; these allow inner component functionality to be dynamically exported to create the component framework's service provision and requirements. Therefore, a CF becomes a dynamic entity, unlike the fixed primitive components; this is especially important for frameworks that can cover different styles of functionality, which may change over time (see the binding framework in section 4.6). However, the ability to change both the component configuration and the functionality offered by a component framework means that integrity maintenance of frameworks is an important issue.

Integrity Maintenance of Component Frameworks

A component framework must constrain the configuration of components to a valid implementation within its domain. Therefore, after a CF is configured or reconfigured it must be checked to ensure that it provides the correct functionality. To do this, each component framework provides a receptacle named IAccept into which developers can plug their own checking implementation. Figure 4.4 illustrates the interface of this receptacle; this consists of a single operation that takes as parameters the local graph of the component framework and the list of interfaces that expose the structure's functionality. When executed it returns a Boolean value to indicate if the structure is valid; if true, the component framework continues its operation. Otherwise, if false, the component framework rolls back to the previous known good configuration (stored prior to the change) and generates a message to indicate a failed reconfiguration. The complexity of checking depends upon the implementation of the Accept component, which can be dynamically changed by plugging in a new component.

```

Interface IAccept : lunknown {
    ////////////////////////////////////////////////////////////////////
    // Method: isValid
    // Parameters:  [in] lunknown* list[] – Local graph
    //              [in] IID intf[] – List of exposed interfaces
    //              [in] int cComps – Number of components in graph
    //              [in] int cIntfs – Number of Interfaces
    // Return: Boolean – Yes/No if graph is valid
    ////////////////////////////////////////////////////////////////////
    Boolean isValid(lunknown* list[], IID intf[], int cComps, int cIntfs);
}

```

Figure 4.4 The IAccept Interface

The Component Framework Lock

The previous section demonstrates how the structure of a component framework is checked for validity. However, it does not ensure that reconfigurations are made at an appropriate time. If a change to the configuration was made while one or more service calls of the component framework were executing, then the results of these invocations could be compromised or lost. Therefore, each component framework provides a readers/writers lock to access the local CF graph. Each service call through any of the interfaces other than ICFMetaInterface accesses the lock as a reader (there can be n readers using the lock at any time). Any call to change the configuration of the CF, accesses the lock as a writer (a single writer can access the lock when there are no readers). The algorithm to implement this property is a standard readers/writers solution with priority for readers.

Interceptors are used to ensure that all exposed configurations access the lock as a reader before a service call is executed. Each interface exposed by a CF automatically has an interceptor attached with pre and post method behaviour to implement the reader role of a readers/writers solution. That is, the pre method accesses the lock and increments the readers count, while the post method decrements the count and if it is the last reader the lock is released for writers.

Implementing a Component Framework

The implementation of the component framework model ensures that development of composite components is similar to primitive components. The ICFMetaArchitecture interface is implemented as a C++ class that can be re-used in every new component framework, i.e. an object to implement the interface can be created through a factory when the CF is initiated. This technique is illustrated in figure 4.5. and it is identical to the technique employed in OpenCOM to implement the IMetaInterface interface.

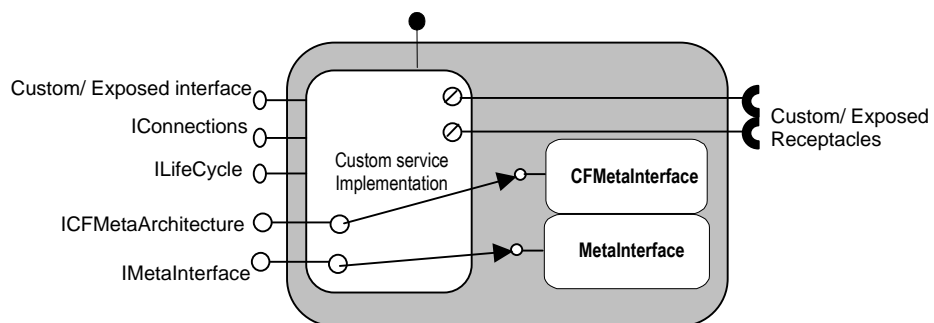


Figure 4.5 **Implementation of an OpenCOM component framework**

The functionality of the component that implements the IAccept interface to be plugged into the framework is chosen by the framework developer, who is then not constrained by architectural rules enforced by the designers of the component framework model. Rather they are free to define their own integrity checks. Example implementations include: no checking (no component connected), a simple topology check matching the graph against an XML description of legal configurations (an example configuration is described by figure 4.6 with the full configuration in appendix B), or alternatively incorporate the architectural style rules proposed by [Moreira01].

```
...
<Component>
  <Name>ReMMoC_GIOP</Name>
  <ID>{14C7E7CF-5750-46de-9924-D219DED7CB2A}</ID>
  <Connections>
    <Interface>{D892611A-F14B-4f27-9646-07A6E7EC013A}</Interface>
    <Interface>{ABFC5317-BF1D-4644-A19C-1A6766AA8349}</Interface>
  </Connections>
</Component>
...
```

Figure 4.6 XML description of a component configuration

4.4 Reflective Middleware for Mobile Computing (ReMMoC)

4.4.1 Requirements for the ReMMoC Middleware Framework

The goal of the ReMMoC middleware framework is to provide the following capability to mobile client application developers. A single client application can be developed independently of concrete middleware implementations and discovery mechanisms to allow it to continue operating across environments consisting of heterogeneous middleware implementation. To provide this property, the following three requirements have been identified:

- The middleware must provide a reconfigurable service discovery mechanism. Application developers can then find matching service types irrespective of advertisement implementations. The mechanism must mirror the environment and perform lookup using the discovery protocols in use.
- The middleware must provide a reconfigurable service binding mechanism. The framework can then bind to a discovered application service using the type of middleware the service is implemented upon.

- The programming model of the middleware must be abstract from concrete implementation details. The application developer can then perform lookup, binding and invocation of application services independently of individual middleware. The dynamic nature of the binding and discovery mechanism makes concrete middleware programming models infeasible.

Furthermore, the platform must operate in an environment with two fundamental characteristics: 1) the mobile device has limited end system resources (e.g. memory and battery power), and 2) the wireless network provides poor quality of service characteristics (e.g. throughput). Given these properties the design of ReMMoC must address the following additional requirements:

- The reflective framework, into which the currently required functionality is dynamically plugged, must be of minimum footprint size.
- All middleware functionality (components) need not reside locally on the device and can therefore be downloaded across the wireless network on demand.
- All components and component frameworks must be lightweight implementations, reducing memory consumption and allowing transmission across the network.

4.4.2 The Reflective Framework

The overall architecture of ReMMoC consists of a reflective framework that can reside upon a mobile device, and into which a concrete middleware implementation is configured and dynamically reconfigured. This architecture is designed as a collection of OpenCOM component frameworks that can be extended at a later date to add new functionality. Using many component frameworks in a middleware design (cf. OpenORB [Blair01], and demonstrated in figure 2.12) increases the size of the implementation in terms of memory footprint; the extra management functionality exhausts the constrained resources of a mobile device. Similarly, the additional overhead of indirection will reduce platform performance on devices with limited computational power. Therefore, the architecture of ReMMoC (illustrated in figure 4.7) provides a minimal two-tier architecture consisting of a top-level component

framework into which a set of components and component frameworks are plugged. There are three sections of the top-level component framework:

1. The **concrete middleware section**, which is composed of two key component frameworks: (1) a **binding framework** for interoperation with mobile services implemented upon different middleware types, and (2) a **service discovery framework** for discovering services advertised by a range of service discovery protocols. The binding framework is configured by plugging in different binding type implementations e.g. IIOP Client, Publisher, SOAP client etc. and the service discovery framework is similarly configured by plugging in different service discovery protocols. A detailed description of the services provided by the two frameworks and their properties for reconfiguration are discussed in the following sections.
2. The **Abstract middleware-programming model**, which implements an API for performing service discovery and service interaction independent of middleware implementation.
3. The **abstract to concrete mapping** section, which consists of components to map abstract service requests to the current binding and discovery implementations in place.

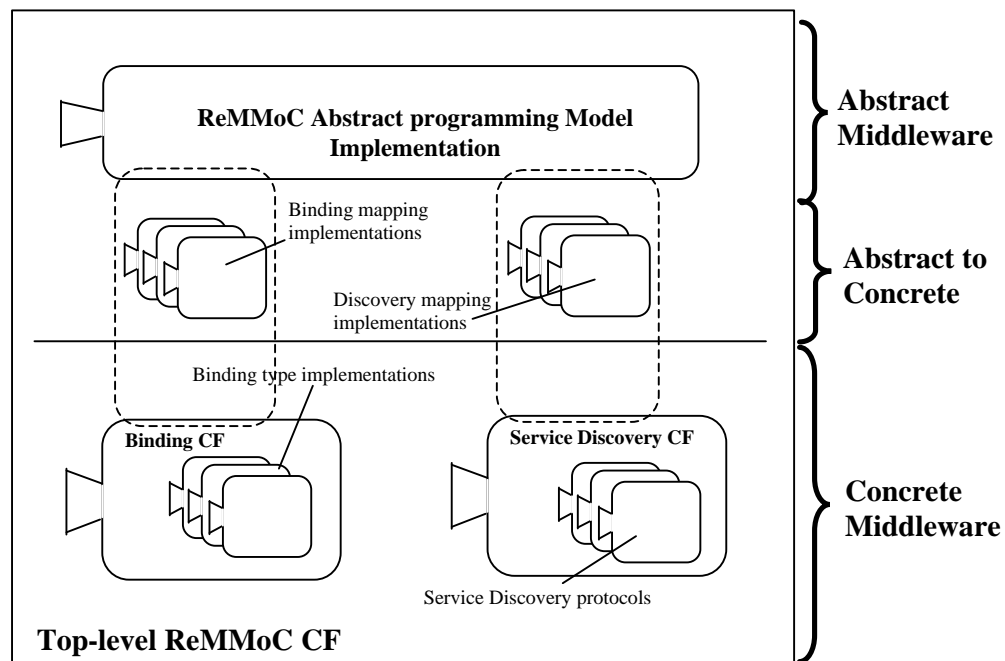


Figure 4.7 The top level architecture of ReMMoC

The framework itself is configurable to meet the application developer's requirements. For example, the platform can be configured to just the concrete section, or indeed one of the two component frameworks. This may be required for applications using fixed types of middleware (needing no abstraction) on low resource embedded devices (e.g. wearable computers); memory footprint size is significantly less and the indirection and extra processing overhead is avoided. Similarly, the platform is extensible to allow more component frameworks for other non-functional properties such as security and resource management to be added.

This remainder of this chapter concentrates on the concrete middleware section of this framework, with the services provided by the two frameworks discussed in detail. The abstract programming model and mapping components are discussed further in chapter 5.

4.5 The Service Discovery Framework

4.5.1 Overview

The principal function of the service discovery framework is to provide a reconfigurable service discovery mechanism that can perform lookup operations across a set of different discovery protocols. Hence, the service discovery framework provides the base of the implementation-independent discovery service that forms the core of the ReMMoC platform. An application developer can discover the application service that matches their requirements, based upon matching service type and attributes, irrespective of the discovery mechanism that is advertising it. Hence, in one location a tourist guide service advertised using SLP is found and in the next location the same service type is found advertised using UPnP. To meet this goal, the service discovery framework has the following key characteristics.

- The framework automatically initialises itself to a configuration of components to mirror the current environmental conditions, i.e. depending on what type of discovery technologies are currently used in the environment.
- The framework dynamically reconfigures itself when the environmental context changes e.g. when the discovery mechanisms used in the environment change. This is most likely to occur when the mobile user changes location.

- When a single discovery protocol is used in the network environment, e.g. SLP is in use at a particular location, the framework takes the role of a single lookup personality (e.g. an SLP lookup configuration is created).
- When multiple protocols are in use in the local network, the framework takes the role of a multiple lookup personality. For example, if SLP and UPnP are both being utilised at a location then the framework configures itself to contain lookup implementations for both types. A single lookup request can then be simultaneously executed over each discovery type.

4.5.2 The “Cycle and See” Philosophy

To mirror the current environment, the framework must discover discovery protocols in use. However, the author believes that no solution will completely solve the problem of heterogeneous discovery mechanisms, i.e. how you discover all the discovery protocols. In order to discover a service you must have knowledge of the discovery mechanism used to advertise it; if the discoverer does not know that mechanism it cannot find the services. Solutions promoting a fixed point of agreement, e.g. an agreed higher-level discovery mechanism for finding discovery protocols, are infeasible because: 1) not all elements can be guaranteed to use this technology, and 2) the higher-level mechanism itself may change (this simply moves the problem to a higher level). Therefore, the design of the service discovery framework follows a “Cycle and See” philosophy. This entails that the framework execute discovery of discovery protocols by cycling through a set of tests for each individual discovery protocol it is aware of (see section 4.5.5). The probability of all services being found increases as the number of tests to cycle through increases. “Cycle and See” does not rely on agreement between participating elements, and is evolvable to include future discovery mechanisms. Therefore, the author argues that “Cycle and See” is a natural approach to solving discovery protocol heterogeneity.

However, the “Cycle and See” approach is limited in two respects: 1) cycling through discovery protocol tests is both time and resource consuming, and 2) as the number of tests increase the performance of the platform degrades. However, tests can be performed in parallel to reduce time, and the author believes that the use of knowledge based context information will dramatically reduce this particular resource use. For

example, if you know the types of discovery protocol used in an environment (from a previous visit, or through shared knowledge) you can test for only these. Furthermore, depending upon the mobile device, the developer may select to reduce the number of tests carried out when resource consumption is critical.

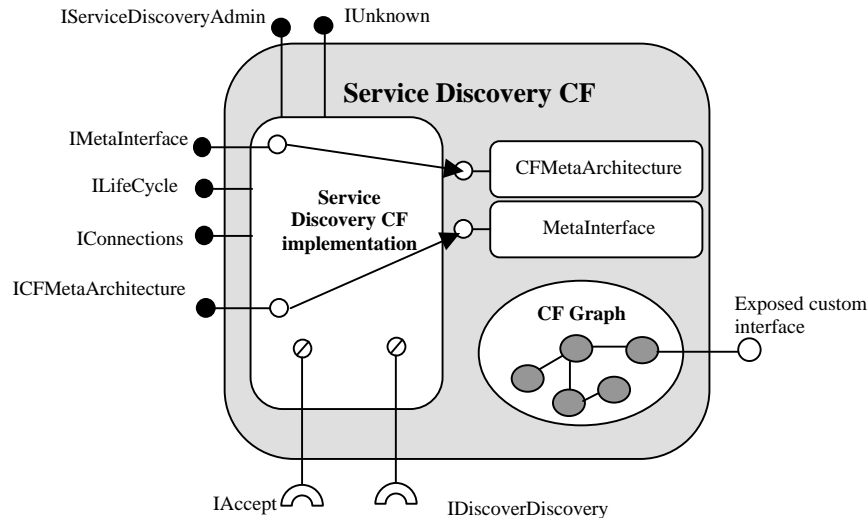


Figure 4.8 The Service Discovery Component Framework Architecture

4.5.3 The Architecture of the Service Discovery Framework

The architecture of the service discovery framework follows the concepts of component frameworks described in section 4.3. There are five key parts to this architecture (which is illustrated in figure 4.8):

- The core functionality of the discovery framework is maintained in the local graph of the framework as a configuration of OpenCOM components. For example, this may be a single lookup personality, e.g. SLP lookup (figure 4.9) or UPnP lookup (figure 4.11), or a multiple lookup personality, e.g. SLP & UPnP lookup (figure 4.12).
- The custom interfaces of each personality are exposed as interfaces of the component framework to be used directly by ReMMoC's mapping components. When an SLP lookup personality is configured, the service functionality of the framework is accessed through an ISLPServiceFind interface, whereas IUUPnP is exposed when an UPnP configuration is in place. Furthermore, both interfaces would be exposed for the corresponding multi-personality configuration.

- The ICFMetaArchitecture interface provides reflective operations to allow the programmer to make dynamic fine-grained or coarse-grained changes to the internal composition of the discovery of the personality at any time.
- The IAccept receptacle offers a plug-in to maintain the integrity of the discovery framework.
- The IDiscoverDiscovery receptacle provides for a component to be plugged into the framework, which monitors the environment and automatically reconfigures the framework based upon its findings. This component provides parts of the functionality to implement the “Cycle and See” philosophy.

4.5.4 Service Lookup Personalities

Overview

Component based implementations of individual service discovery protocols (service lookup personalities) form the core functionality of the discovery framework. These ensure that the physically communicated network messages for service lookup can interoperate with the discovery protocols used by services in the environment. A service lookup personality is either a single or multiple personality. A single lookup personality executes service lookup using a single discovery protocol e.g. Service Location Protocol messages are exchanged across the network. The multiple service lookup personality simultaneously executes service discovery over two or more discovery protocols. For example, a discovery of service A, with attributes B and C can be performed across both Universal Plug and Play and SLP. The following sections discuss the design and OpenCOM based implementation of two discovery protocols (UPnP & SLP) that can be plugged into the framework.

Design & Implementation of Lookup Personalities

Each individual lookup personality is designed as a reconfigurable configuration of OpenCOM components that implements the functionality of an individual service discovery protocol. This allows for future research into fine-grained changes in ReMMoC’s operation e.g. the multicast protocol can be changed when the device moves from an infrastructure based network to an ad-hoc network. Personalities also exhibit the capability to be utilised as stand-alone protocols. Furthermore, each personality implements its own lookup interface, rather than mapping to a fixed,

common lookup interface; additional functionality specific to each personality is then directly available to the developer and discovery framework. Mapping to a fixed, overarching abstract discovery interface is left to the abstract to concrete section of the ReMMoC Architecture. This technique ensures that the discovery framework is evolvable over time; additional functionality may be added through extension of the component configuration. For example, service advertisement or security features could be added to service lookup. The implementation of SLP and UPnP personalities are now examined in turn.

Service Location Protocol (SLP)

The operation of SLP was described in detail in section 2.6.3. This SLP personality implements the client side portion of the protocol. Hence, when a central directory agent is available lookup messages are directly sent to it, otherwise lookup requests are multicast across the network for service agents to respond to (the implementation specifically concentrates on the second part because it is well suited to wireless networks). Table 4.3 illustrates the six components that compose the Service Location Protocol lookup personality.

Component Name	Description
Socket	Wraps the socket API, to provide an operating system independent interface for network programming.
SLPMessage	Creates and reads SLP messages that conform to SLP standard [Veizades97]. Operations to multicast SLP messages to service agents.
DADiscovery	Operations to discover and communicate with Directory Agents when these are available in the network.
SLPServiceFind	Programmer operations to perform SLP lookup.

Table 4.3 Components of the SLP Lookup Personality

Figure 4.9 shows the complete configuration of the four components for the SLP personality. However, the configuration can be minimised by removing the directory agent component (DADiscovery) when only service agent interaction is required. Notably, the lookup operations provided by the SLPServiceFind interface return their results asynchronously. Therefore, the user of the personality must pass handler functions to manage the returned results of both service lookup and attribute lookup. The four components were implemented based upon the Open Source OpenSLP (www.openslp.org) C++ implementation of the SLP standard. Therefore, the

implementation consisted of separating the functionality into individual, replaceable OpenCOM components, recreating the appropriate network communication implementations, and porting the message formats to the Windows CE operating system.

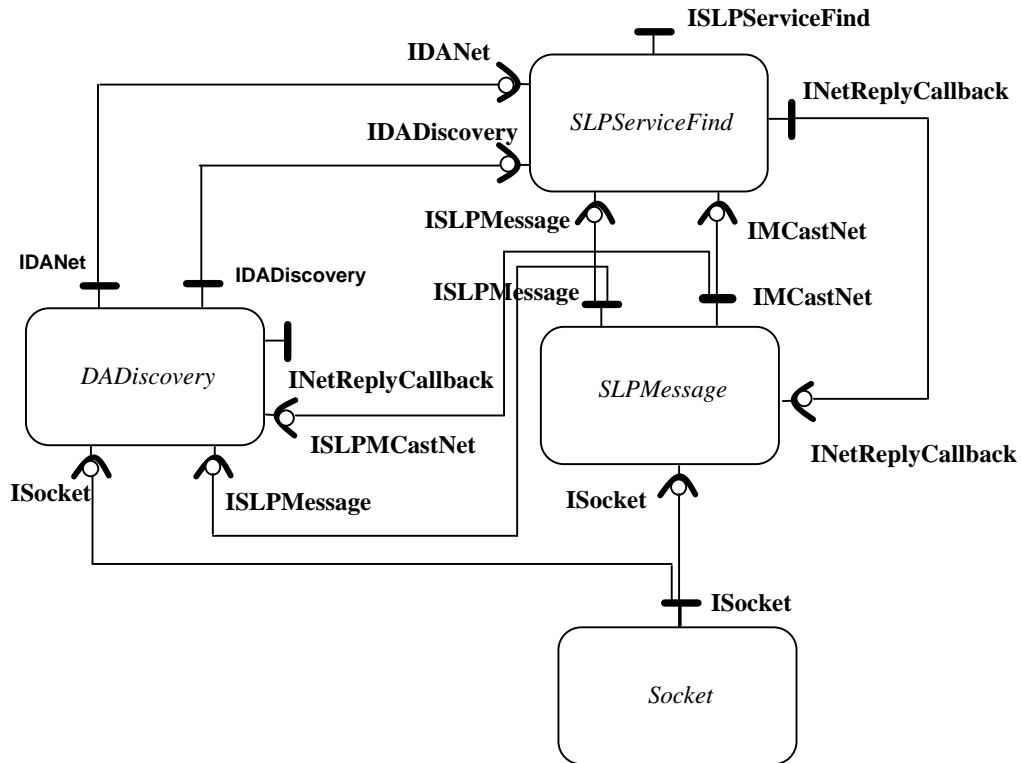


Figure 4.9 OpenCOM configuration for SLP lookup personality

Universal Plug and Play (UPnP)

The UPnP standard is based upon UPnP devices whose functionality is offered through a set of services. The description of these services is advertised by XML documents, which can be downloaded by the client. Hence, the UPnP personality is implemented to lookup services, download the corresponding XML description, and then parse this file to find service attributes. The operations of the personality are described in the IDL definition of the IUPnP interface, illustrated in figure 4.10. This illustrates a minimum UPnP personality that concentrates solely on the service lookup features of UPnP, it does not take into account UPnP device lifetime management that uses the General Event Notification Architecture to signal state changes between services. Although, it is feasible for this capability to be added later using additional components.

In UPnP, lookup is performed by the Simple Service Discovery Protocol (SSDP), which operates by multicasting HTTP messages using the User Datagram Protocol. A Client (Control point) multicasts SSDP messages to the SSDP multicast address (239.255.255.250:1900) and receives one or more unicast response messages for each matching service. Whatever is requested, e.g. all root devices, a service or a specific device, only the identifier of the UPnP device is retrieved (a URL). The client then uses this URL to obtain XML descriptions of the device, service and attributes using a standard HTTP over TCP approach. Therefore, the implementation of the UPnP personality is made up of the five components described in table 4.4; these can be connected as shown in figure 4.11 to create a full UPnP personality. Like SLP, a Client function is registered to be called back when matching services are found. The SSDP component creates HTTP messages using functions from the HTTP component but doesn't use the HTTP transport methods, instead it sends and receives the UDP messages (unicast and multicast) using the Socket component. The personality is also dynamically configurable; separate personalities for individual service lookup and XML downloading can be configured.

```
int UpnpSearch (UpnpClient_Handle Hnd, int MaxRetry, const char * ServiceType, const void *Cookie);
IXMLDoc* UpnpDownloadXml (const char *url_const);
ServiceList* UpnpListServices(char* xmlDoc);
ActionList* UpnpListActions(char* xmlDoc);
```

Figure 4.10 **The IUPnP Interface**

Component Name	Description
Socket	Wraps the socket API, to provide an operating system independent interface for network programming.
TCP	Wraps TCP socket functionality
HTTP	Creates HTTP headers.
SSDP	Implements the SSDP protocol. Lookup commands are wrapped in HTTP messages and then multicast over UDP. Unicast responses are received, which generates a service found event.
UPnP	UPnP functionality. Wraps SSDP lookup and offers extra services of XML downloading and attribute discovery

Table 4.4 **UPnP components**

Three of the components: HTTP, TCP and Socket were developed from scratch to match their protocol specification. However, the UPnP and SSDP components were ported from the UPnP Linux development kit (upnp.sourceforge.net) from Intel to

create two separate OpenCOM C++ components. The majority of this task involved porting the code to the windows CE operating system and then integrating it with the three existing communication components.

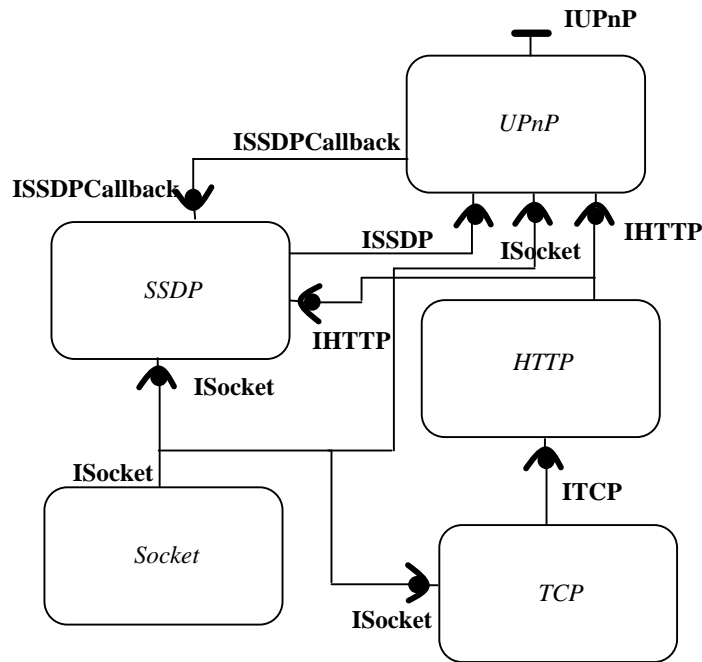


Figure 4.11 UPNP lookup component personality.

UPnP & SLP Multi-personality

In order to implement a multiple service lookup personality the two individual SLP and UPnP personalities are combined. Hence, the diagrams in figure 4.9 and 4.11 are joined, whereby they share the common component Socket; however, it is feasible for separated personalities (there is no direct connection between the personalities) to implement a multi-personality. A multi-personality simply exports one interface per individual lookup personality e.g. ISLPServiceFind and IUPnP; this allows service lookup operations to be executed simultaneously across each protocol.

4.5.5 Mirroring the Network Environment

Overview

The initial configuration and further dynamic reconfiguration of the service discovery framework is driven by the current context of the mobile device. The functionality provided by the framework must mirror the current environmental conditions. Hence,

if N discovery mechanisms are being utilised across the current wireless network then the service discovery personality should simultaneously implement each of the N lookup mechanisms. When the framework is initiated it must obtain context information about the discovery mechanisms currently in use and create a corresponding configuration. Furthermore, as the framework continues operation it must monitor context information about discovery protocols in order to dynamically respond to any changes. In this section, the algorithms for initial configuration and dynamic reconfiguration are described. The required context information about discovery protocols in the environment is obtained using the component plugged into the DiscoverDiscovery receptacle of the framework. A description of the implementation of this component is provided in this section.

```
interface IDiscoverDiscovery: IUnknown {
    HRESULT AsynchronousDiscoveryProtocolSearch([in] ServiceDiscoveryType list[],
        [in] int TimeToSearch, [in] ReMMoCServiceFindHandler cback);
    HRESULT SynchronousDiscoveryProtocolSearch([in] ServiceDiscoveryType sdt);
}
```

Figure 4.12 **IDiscoverDiscovery Interface**

The DiscoverDiscovery Component

In order to perform configuration and reconfiguration, the framework must be aware of environmental context information i.e. the set of protocols currently used to advertise services. It is the task of the DiscoverDiscovery component to perform individual tests for each known protocol (the framework maintains a list of discovery protocols that it is aware of). The service discovery framework then manages the execution of individual tests for each of the protocols. The interface IDiscoverDiscovery illustrated in figure 4.12, documents how these operations are called. There are two styles of operation: synchronous and asynchronous. The synchronous operation (**SynchronousDiscoveryProtocolSearch**) takes a protocol type as parameter (e.g. SLP or UPnP) and performs a single test for this, the result is a synchronous Boolean response indicating if that protocol is in use. The asynchronous operation takes a list of protocol types to test for (e.g. SLP and UPnP) and the time to search as parameters. Environmental monitors are then initiated for each; the test continuously polls the environment. If a detection is made an event is generated through the callback method passed by the framework, which can then reconfigure on

this trigger. Notably, continuous monitoring is an expensive operation that quickly consumes resources (e.g. battery power and bandwidth); therefore, this operation is utilised sparingly by ReMMoC.

Implementation of the DiscoverDiscovery Component

The service discovery framework currently knows of two protocol types, namely SLP and UPnP (these were the implemented lookup personalities). Therefore, the DiscoverDiscovery component was implemented to perform synchronous and asynchronous tests for both of these. The philosophy behind the implementation was to create a single, lightweight component with no dependencies on other components. The diagram in figure 4.13 illustrates how the individual tests were implemented. For SLP you can test the environment for either a directory agent or service agents, although in a mobile setting service agents are more likely; if neither exist it is not possible to advertise SLP services. Therefore, the DiscoverDiscovery component creates an SLP header containing the lookup request “service:service-agents”, which is then multicast to the SLP multicast address 239.255.255.253:427. Any service agents return a response directly on the requesting socket. Therefore, any response is an indication SLP is in use. Similarly, for UPnP a HTTP/SSDP header as shown in the diagram is created and multicast to 239.255.255.250:1900; if a response is returned from a UPnP device then UPnP is used in the environment.

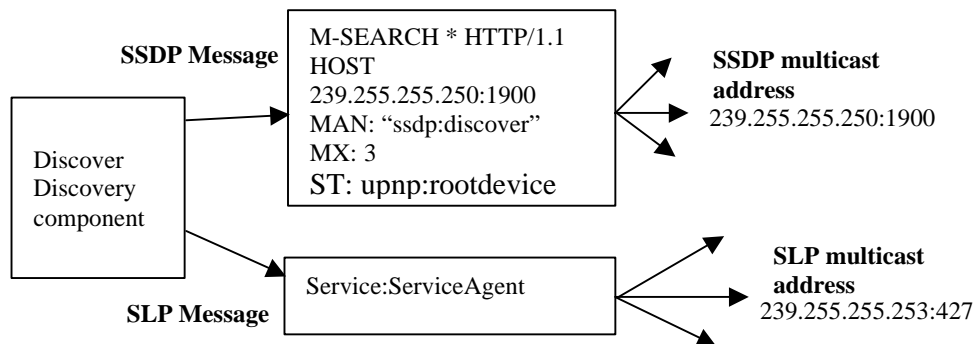


Figure 4.13 Discovery protocol tests

XML configuration

The discovery framework automatically creates component personalities based upon the context information it receives (i.e. the results of DiscoverDiscovery). Along with the list of discovery protocols that are known, an XML description of the component architecture is maintained; XML is utilised to simplify the definition of component

framework personalities by middleware developers. This XML document lists the components in a particular personality and how they are connected together. An example XML architecture description (part of the SLP personality) is illustrated in figure 4.14; this consists of a set of components and interfaces. The list of interfaces identify the personality e.g. SLP, UPnP, SLP&UPnP. Through reflection this can be used to identify if parts of the personality are already in use before reconfiguration is attempted. The list of components documents all components in the personality; each component describes its name and unique identifier (this information is then used to create a new instance of the component in the framework using the insert_component method), along with a set of connections describing the interfaces it is connected to.

```

<ReMMoC_Configuration>
  <Interfaces>
    <Interface>{BC906B4C-9902-48ed-8449-8C82C85EBB11}</Interface>
  </Interfaces>
  <Components>
    <Component>
      <Name> UPnP</Name>
      <ID>{32DBE3A3-23CB-408e-BB9F-DDCCBE9F0DAD}</ID>
      <Connections>
        <Interface>{50B10B7D-10CD-4465-B830-DA91BEC2530B}</Interface>
        <Interface>{1A0E8B36-8857-11d3-9448-00A024B801B7}</Interface>
        <Interface>{D993631C-FD4C-4f27-9646-07E6E7EC098A}</Interface>
      </Connections>
    </Component>
    <Component>
      <Name> SSDP</Name>
      <ID>{90572423-3E65-42a6-8C86-97A6517A5B83}</ID>
      <Connections>
        <Interface>{70E545C4-FF5A-4851-8646-E301EB22654A}</Interface>
        <Interface>{1A0E8B36-8857-11d3-9448-00A024B801B7}</Interface>
        <Interface>{D993631C-FD4C-4f27-9646-07E6E7EC098A}</Interface>
      </Connections>
    </Component>
    ...
  </Components>
</ReMMoC_Configuration>

```

Figure 4.14 Part of the XML description for the SLP personality

The algorithm for reconfiguration is illustrated by pseudocode in figure 4.15. This is implemented as a combination of XML parsing and reflective operations on the component framework. For example, the list of interfaces are parsed and then checked against those in place using enumIntfs of the CF. If different, reconfiguration is started; each component description is parsed to obtain the information required to insert a new instance of the component into the graph (insert_component). Each of the

connection statements is then parsed and the corresponding components are bound together (local_bind).

Initial configuration

The service discovery framework is initiated through its ILifecycle interface; hence, the startup method is invoked. This method implements the initiation algorithm of the framework; this involves reading the current known discovery protocols and for each create a thread that invokes the discoverdiscovery's synchronous discovery method e.g. SynchronousDiscoveryProtocolSearch(SLP) will run the test for SLP in the environment. The result of these tests returns a Boolean value; if true is the response, the XMLConfigure operation is invoked passing the XML description of the lookup personality.

```
Find corresponding XML personality description;
LOAD XML description into parser;
For index = 1 to number of <component> tags
    Read componentID and componentName;
    meta operation: Insert_component(componentID, componentName);
Endfor;
For index = 0 to number of <component> tags
    Read CompID

    For index 2 = 0 to number of <connection> tags
        Read Source Interface Identifier IntfID;
        Find Sink Component in local graph: Iunk1 = GetPIUknown (CompID);
        Find Source Component:
            meta operation: get_internal_components(Clist, Num);
            For index 3 = 0 to Num
                EnumIntfs(Clist[index3], IntfList, Num2);
                For index 4 = 0 to Num2
                    If IntfList[Index4]== IntfID;
                    Iunk2= GetPIUknown (Clist[index3]);
                    meta operation: Local_bind (Iunk1, Iunk2, IntfID);
                Endfor;
            Endfor;
        Endfor;
    Endfor;
Endfor;
```

Figure 4.15 Pseudo code for XML based configuration of personalities

Environment Monitoring

The service discovery framework also implements a method for continuous monitoring of discovery protocols in the environment; this is named ConMonitor.

When this method is invoked, each known discovery protocol is read and a thread that calls asynchronousDiscovery for each is spawned. The discoverdiscovery tests will then monitor the environment; when a protocol is discovered it calls back a handler that contains the code to invoke the XMLConigure method for the corresponding protocol, and hence change the configuration based on an event trigger.

Integrity maintenance

The final task of the discovery framework is to maintain integrity in the face of dynamic changes. Two requirements are placed on integrity checking 1) only valid, complete lookup protocols are allowed to compose the framework's functionality, and 2) changes to the framework cannot be made until all existing lookup requests have completed. These are enforced by the framework implementation and IAccept component that were described in section 4.3.3. The readers/writers lock of the framework is accessed as a reader for the lookup operations of the exposed lookup interfaces e.g. ISLPServiceFind and IUPnP, and as a writer for ICFMetaArchitecture change operations. Therefore, changes to the component configuration are blocked until invocations of the exposed interfaces are complete. The Accept component stores XML descriptions of single and multi-personalities in the format illustrated in figure 4.14. Therefore, for the implementation of the framework three descriptions are maintained (SLP, UPnP and UPnP&SLP). When a change is made, the framework invokes *isValid* passing the local graph as a parameter. The Accept component then checks this graph against each description it currently stores. Only when there is a complete match between components, connections and exposed interfaces is a Boolean true response returned.

4.5.6 New Discovery Protocols

A key aim of the discovery framework is to be extensible to dynamically incorporate new discovery protocols as they become available. We have implemented personalities for SLP and UPnP; however, in the future it must be possible to extend the framework to allow it to discover services using new discovery mechanisms. This is especially important in the domain of mobile computing, where much work on creating new discovery solutions for ad-hoc wireless networks and ubiquitous applications is being carried out.

To add a new discovery protocol to the framework, three tasks must be carried out:

- Make the framework aware of the new protocol type, and the component personality required to perform service lookup.
- Add synchronous and asynchronous tests to the DiscoverdDiscovery component that will detect if the new protocol is in use in the local environment.
- Add the XML description of component personality to the Accept component in order for the component configuration to be verified correctly when it is created.

```
interface IServiceDiscoveryCFAdmin: IUnknown {  
    HRESULT AddNewProtocol(char * ServiceDiscoveryType, char* XML);  
}
```

Figure 4.16 **IDL definition of IServiceDiscoveryCFAdmin interface**

To add a new protocol, an administrator or application obtains a reference to the interface *IServiceDiscoveryCFAdmin* (described in figure 4.16) available from the framework. They may then invoke the *AddNewProtocol* operation passing two strings 1) the name of the type of discovery protocol, and 2) the XML description of the component configuration as exemplified in figure 4.14. They must then implement a new version of the DiscoverDiscovery component, which will add a synchronous and asynchronous test for the newly created type. The old version of the DiscoverDiscovery component can then be disconnected and dynamically replaced by the newer version. Note that the discovery framework must be shutdown before this process is initiated and then re-started at completion otherwise the monitoring operations, which detect when a discovery protocol begins to be used in the environment, will fail.

4.6 The Binding Framework

4.6.1 Overview

The principal function of the binding framework is to provide a configurable and dynamically reconfigurable binding mechanism that allows mobile clients to bind and

interoperate with application services implemented upon particular implementations of middleware paradigms (e.g. Remote Method Invocation, Publish-Subscribe, Asynchronous Messaging). Furthermore, the binding framework allows two-way interoperation with services, i.e. as well as client binding, the service itself is able to bind back to the client and communicate across its own style of binding. Hence, the binding framework provides the base for the implementation-independent binding mechanism that forms part of the core of the ReMMoC platform. To interoperate with a discovered service, the binding framework dynamically reconfigures itself to an identical binding mechanism e.g. if a CORBA service is found the framework becomes a CORBA client side personality; similarly if a STEAM publisher is found the framework configures to a STEAM subscriber. This allows service interoperation to become independent of heterogeneous binding mechanisms, as is described in more detail in chapter 5.

To meet this goal, the binding framework has the following key characteristics:

- A configurable and dynamically reconfigurable client side binding personality, i.e. this configuration performs client style binding operations (e.g. service requests, message sends). This is a single personality that can execute a single operation and then reconfigure to a new personality. For example, a mobile jukebox player application can send a SOAP request to play a song in one location, while at another the list of available songs can be read using a publish-subscribe implementation.
- A configurable and dynamically reconfigurable service side binding personality, i.e. the framework is able to host service implementations that respond to requests of this service and messages sent to the service. This allows for the same application service to be hosted upon heterogeneous implementations. For example, a chat service can be hosted upon IIOP to interoperate with a CORBA client, and later when a SOAP client attempts to interoperate with the hosted service the personality can change its base binding to SOAP.
- Configuration and dynamic reconfiguration of the framework is controlled by higher-level elements. In ReMMoC's case the top level ReMMoC CF receives information from the service discovery framework to drive the correct configuration i.e. it finds a SOAP service therefore reconfigures to SOAP. However, the framework is also an independent element that can be used

individually by clients that can use the meta-interfaces to make their required changes.

4.6.2 The Architecture of the Binding Framework

The architecture of the binding framework follows the concepts of component frameworks described in section 4.3. There are five key parts to this architecture (which is illustrated in figure 4.17):

- The core functionality of the framework is maintained within the local graph. This multi-personality maintains component configurations for just a client personality, just a server personality or a client and server personality. For example, the local graph could contain an IIOP client, an IIOP server or both an IIOP client and server.
- Individual custom interfaces of the client and server are exposed as interfaces of the component framework. It is feasible that the binding framework can contain multiple clients and server personalities operating in parallel, however, this capability is not considered (due to the complexity of implementation involved) in the implementation described by this thesis.
- The ICFMetaArchitecture interface provides reflective operations to allow the programmer to make dynamic fine-grained or coarse-grained changes to the local graph of components. Coarse-grained changes include changing a complete personality. Fine-grained changes can be made in face of changing network conditions e.g. when the device encounters frequent disconnection a SOAP personality can switch its transport to SMTP rather than HTTP.
- The IAccept receptacle offers a plug-in to maintain the integrity of the discovery framework. Like the service discovery framework, this performs topology checks of the local graph against known XML component configurations.

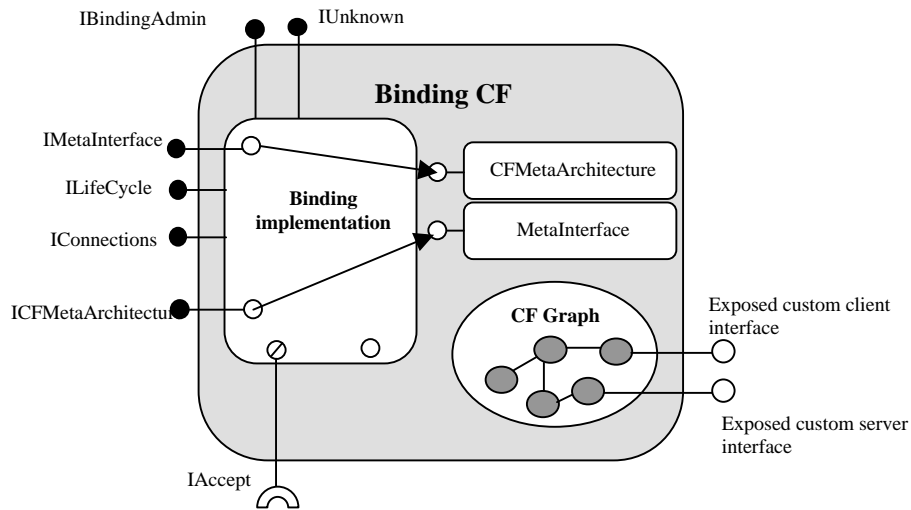


Figure 4.17 The binding component framework architecture

4.6.3 Binding Personalities

Overview

The requirement of binding personalities is to interoperate directly with application services over the matching underlying binding implementation. Generally, these personalities specifically fit one of the roles of a client or server personality. For example, an IIOP client performs a complete remote method invocation or sends a one-way message to a server; whereas the IIOP server responds to RMI invocations and receives one-way messages. Similarly, the client side subscribe personality receives published events; whereas the server side publisher only forwards these events. This section examines in turn the design and implementation of client and server binding personalities that have been developed for the ReMMoC project, namely an IIOP client, an IIOP server, a SOAP client, a publish-subscribe subscriber and a publish-subscribe publisher.

IIOP Client

Like the ALICE project [Haahr00], IIOP is used as the minimum implementation of a CORBA ORB to address the memory restrictions of the mobile device. The implementation is based upon the IIOP personality from the Universal Interoperable Core implementation [Roman01], hence only the Dynamic Invocation Interface is implemented to invoke remote operations. The goal of the personality is to

interoperate with established CORBA ORB implementations (e.g. ORBACUS) and the IIOp server personality described later. The components that make up this configuration are described in table 4.5.

Component Name	Description
Socket	Wraps the socket API, to provide an operating system independent interface for network programming.
TCP	Wraps TCP socket functionality
GIOP	GIOP operations (send/receive GIOP messages)
IIOp	Map GIOP to TCP/IP. Implementation of the IIOp programming interface e.g. IORs and objects.
CORBAMarshalling	Marshalling and demarshaling of primitive CORBA type as defined in GIOP CDR.

Table 4.5 Component elements of the IIOp client personality

The individual components are designed for particular dynamic changes e.g. the transport protocol can be replaced e.g. TCP with UDP. The marshalling component can be replaced by a version for marshalling and demarshaling of more elaborate types (the current version manages primitive CORBA types and arrays, while structs could be a future extension). The configuration of components is illustrated in figure 4.18.

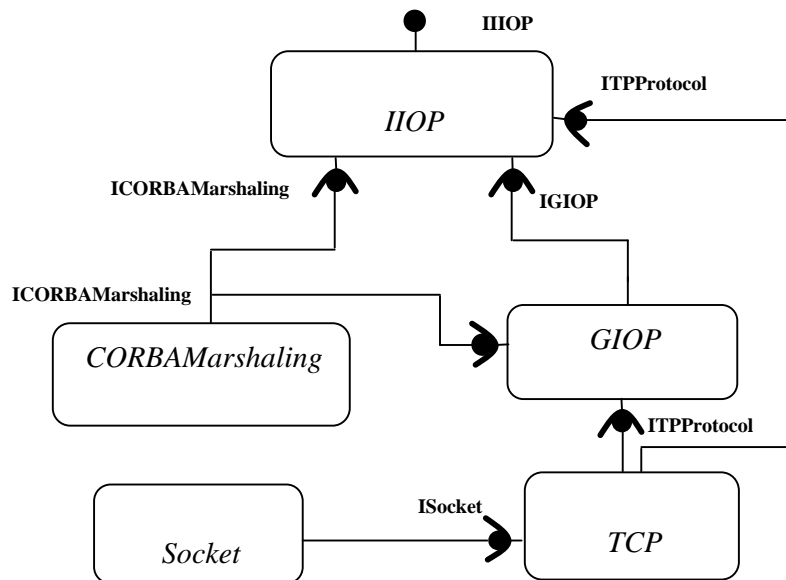


Figure 4.18 IIOp client binding personality

The implementation of this personality utilises the socket and TCP components that have already been described; the remaining three components were implemented using the Universal Interoperable Core open source implementation for Windows CE devices. This code was first converted to OpenCOM components, and then the network programming was replaced by the existing TCP and Socket components.

IIOB Server

The role of the IIOB server personality is to host objects whose operations can be invoked remotely. Like the client side personality, the implementation is based upon the UIC model of a minimum CORBA ORB implementing only the dynamic invocation interface (to reduce memory consumption). The configuration of components for the server side personality is illustrated in figure 4.19. It can be seen, that three components of the client side personality can be re-used (Socket, TCP and Marshalling), and two new components described in table 4.6 are introduced.

Component Name	Description
GIOPServer	GIOP operations (send/receive GIOP messages)
IIOBServer	Map GIOP to TCP/IP. Implementation of the IIOB programming interface e.g. IORs and objects.

Table 4.6 Additional IIOB server components

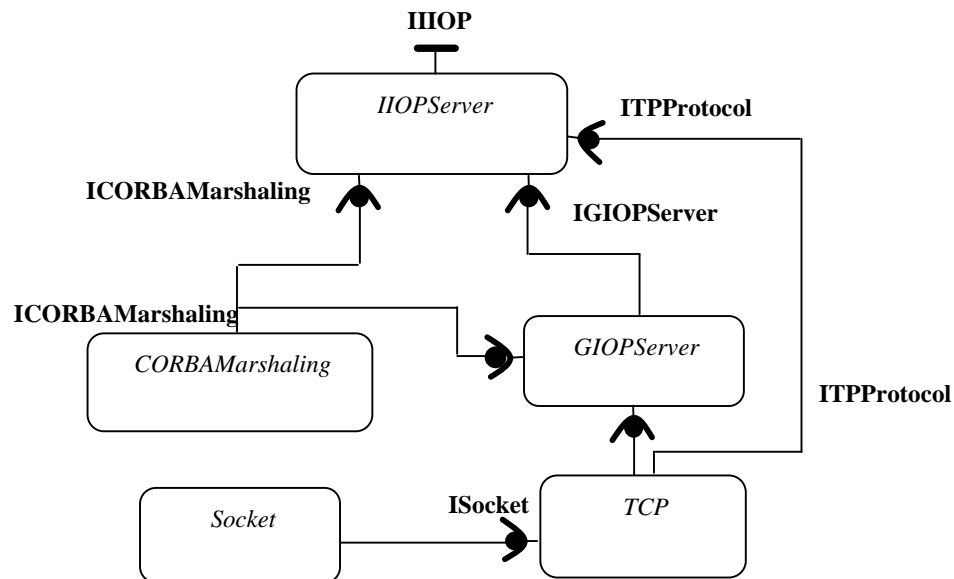


Figure 4.19 IIOB Server side binding personality

SOAP (RPC) Client

In SOAP, two styles of communication are possible: 1) a synchronous request response exchange of XML messages between a SOAP client and server, typically used to invoke a remote procedure, and 2) asynchronous messaging passing, whereby an XML message is sent and its content must be parsed by the receiver. This personality concentrates on the first role (although the second style is used by the subscriber implementation later). The design of the SOAP RPC client concentrates on a minimum implementation of the SOAP specification, and unlike the previous personalities was implemented from scratch. The components that make up the personality are described in table 4.7, and the configuration is illustrated in figure 4.20. The SOAP marshalling component, like IIOP concentrates on primitive SOAP types and arrays, and in the future could be extended to include struct definitions, base64 encoding and MIME types.

Component Name	Description
SOAP	Provides SOAP programming interface to send SOAP RPC request and receive response
SOAPtoHTTP	Maps the SOAP operations onto the current transport – HTTP.
HTTP	Implements HTTP 1.1 specification. Methods to create and read HTTP headers, and transmit HTTP data.
SOAPMarshall	Marshalls and demarshalls
TCP	Wraps TCP sockets for HTTP to be layered over. Replaceable by UDP and other transports.
Socket	Wraps platform dependent network socket implementation.

Table 4.7 Components of SOAP RPC client personality

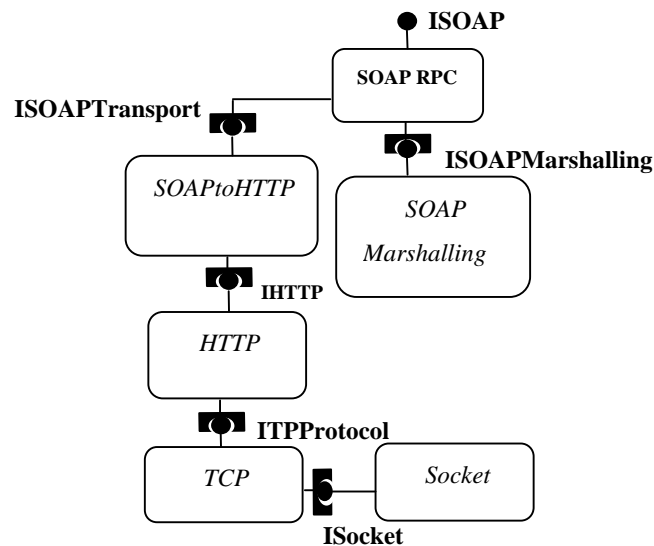


Figure 4.20 Component configuration for SOAP RPC client personality

Publish-Subscribe Subscriber

A publish-subscribe personality was designed and implemented to demonstrate that fundamentally different communication paradigms to remote procedure call or remote method invocation can operate within the binding framework. The STEAM model for publish-subscribe (section 2.3.4) across wireless networks using group communication was followed. An IP multicast address is used to create a particular channel to disseminate events upon e.g. one publisher uses one address, while another may have a different channel. XML messages contain the physical content of the message and are transmitted as asynchronous SOAP messages. Finally, a filter language is used to describe the event types the client wishes to subscribe to (i.e. the filtering takes place at the client); in this case they can filter by the subject of the message or by the content of the message. Further information about the implementation of the filter language can be found in [Sivaharan02]. The list of components that compose a subscriber personality are shown in table 4.8.

Component Name	Description
Subscribe	Provides API to subscribe to events of particular types.
SOAPMessaging	Creates and transmits asynchronous SOAP messages.
SOAPtoMulticast	Maps SOAP messages onto a multicast transport interface.
Filter	Creates content and subject event filters.
Multicast	Implements IP multicast operations for Windows CE platform.

Table 4.8 Component descriptions for subscriber personality

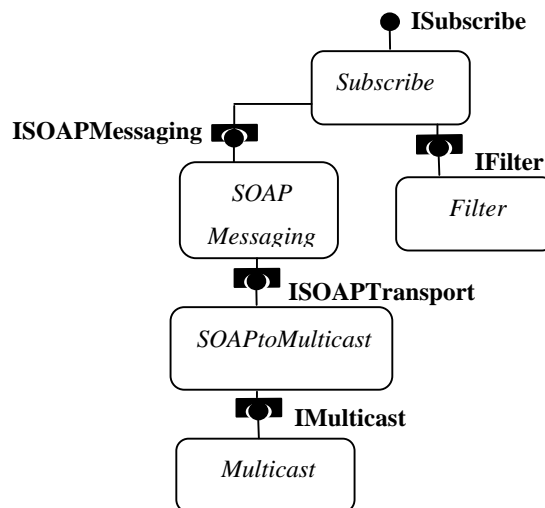


Figure 4.21 Component configuration of subscriber personality

The configuration of components for the subscriber personality is shown in figure 4.21. The personality can be dynamically changed to deal with both environmental change and evolutionary requirements. Upon a change in network type (for example, a change from IEEE 802.11b in infrastructure mode to ad-hoc mode) the IP multicast component can be replaced by an implementation of Application Level Multicast (ALM) [Sivaharan04]. Furthermore, the filter component can be replaced with more elaborate mechanisms such as filtering by context information, e.g. only receive events that come from publishers within ten metres (see [Sivaharan02]).

Publish-Subscribe Publisher

The role of the Publisher is to implement the publishing of events based upon the STEAM model, in order for the previous subscriber to interoperate correctly. The implementation of the component configuration for this task (illustrated in figure 4.22) closely follows the model used by the subscriber. The only change is that the publish component, which provides the API to create new event channels and send messages of particular types upon, replaces the subscribe component.

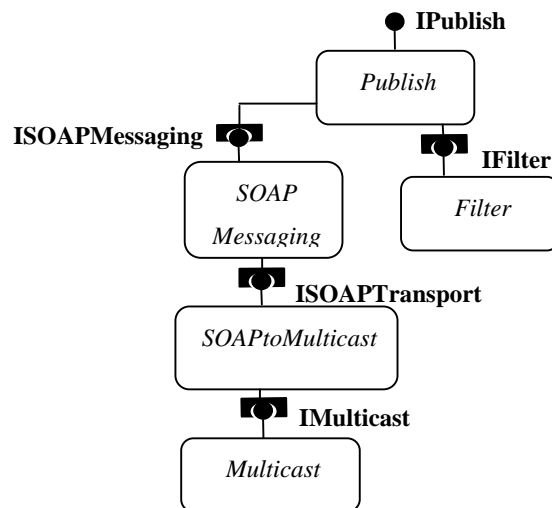


Figure 4.22 Component configuration of publisher

4.6.4 Integrity Maintenance

The integrity of the binding framework is managed in identical fashion to the discovery framework. The accept component stores a list of possible client and server configurations that may correctly compose the framework. These can consist of a single client type, a single server type or a combined server and client. For example,

IIOp client, IIOp server or IIOp client and server; similarly, contrasting styles e.g. publisher and IIOp client are still valid configurations. Rules allowing these combinations are defined within the binding framework implementation. These state that two types of exposed interfaces are allowed, one of a server type and the other of a client type. These must then be implemented by the corresponding component configurations.

4.6.5 New binding types

The binding framework only addresses two communication paradigms, and only a few individual implementations of these at present. Given the heterogeneous nature of the mobile environment, as documented in chapter 2, it is likely that new binding implementations will be required across different locations. Therefore, these must be dynamically added to the framework. Adding a new binding protocol is a much simpler task than adding a new discovery protocol and requires only the following steps from the ReMMoC administrator:

- The binding type (personality) must be implemented as a set of OpenCOM components.
- The component configuration must be described in XML and added to the implementation of the Accept component.
- The higher-level mechanism controlling the binding framework must be made aware of this new binding type (its type name, and the XML description of how to configure it). Therefore, when it discovers a service implemented upon this new binding type it is able to take appropriate action.

4.7 Summary

This chapter has presented the ReMMoC architecture, a middleware framework to support interaction with both heterogeneous discovery protocols and core middleware binding implementations. The core underlying elements of this framework are:

- OpenCOM components are used as the building block of the frameworks; these act as the units of configuration and composition.
- A new component framework model for OpenCOM has been designed to specifically support ReMMoC. The model promotes the use of composite

components to build particular functionality, along with an enhanced meta-object protocol to simplify reflective programming in OpenCOM.

- ReMMoC is based upon a concrete middleware section composed of a service discovery framework and binding framework
- The service discovery framework tackles two specific problems. Firstly, the problem of discovering what discovery mechanism is in use at a particular location, and then performing lookup requests over the one or more discovery protocols that have been found.
- The binding framework supports interoperation with contrasting middleware implementations e.g. different RMI (SOAP and IIOP) and different publish-subscribe implementations.

The next chapter of this thesis examines how the core elements are put together to create an adaptive middleware framework that solves the problem of heterogeneous middleware implementation. Furthermore, it elaborates on the higher-level abstraction and abstract to concrete mapping mechanisms that have been introduced here.

5.1 Introduction

The primary focus of this thesis is tackling middleware heterogeneity in the mobile computing environment. The previous chapter promoted the concept of matching the correct middleware behaviour to individual tasks to solve this problem, e.g. using discovery protocols currently in use in the environment, and matching the client binding protocol to the type used by the found service. Dynamic reconfiguration of middleware behaviour demonstrated how the interoperation problems between heterogeneous middleware implementations are then overcome. However, this alone does not provide a complete solution. A programmer using this technology would need to explicitly program each dynamic change, e.g. when the discovered service is of type SOAP, a series of reflective operations to configure the SOAP components must be programmed before the service is invoked. Similarly, at a new location the found service is of type CORBA, hence this time a series of reflective operations for reconfiguration must be programmed. Program code of this nature is inevitably repetitive, overly long (unnecessarily consuming memory resources) and detracts from the application logic. Furthermore, it is impossible to predict in advance the course of a mobile user; they are unlikely to encounter predictable middleware implementation, especially in newly entered locations.

This thesis argues that to address middleware heterogeneity in the mobile environment the following proposed approach is required. First, the choice of a higher-level middleware abstraction that is independent from both concrete service discovery protocols and middleware bindings, as described previously (section 4.4.2). Second, the definition of mappings of abstract operations (service invocation and service lookup) to concrete operations across the underlying protocols. Mapping is a well-identified solution to the problem of middleware heterogeneity [Vinoski03]; normally direct mappings are made between contrasting middleware types. However, this chapter demonstrates how the technique of mapping from the abstract level to the concrete level provides a flexible and dynamic programming environment in the face of changing middleware heterogeneity. With this approach there is the danger that yet another middleware is produced and the heterogeneity problem moves up a level. In this chapter we analyse the likelihood of this happening with ReMMoC.

5.2 The Overall ReMMoC Abstraction Architecture

There are two fundamental requirements of the abstract programming model:

- 1) The application developer must be able to perform generic service lookup, stating the service type with attributes that they wish to discover. Hence, matching services advertised by different discovery mechanisms can be found.
- 2) The application developer must interoperate with services using abstract operations. The developer is then unaware of the individual communication paradigm (e.g. RMI, Send/Receive) or middleware implementation (e.g. SOAP, CORBA) that the operation is executed across. Hence, the API must provide *middleware transparency*. The application developer need not concern themselves with dynamic reconfigurations between different middleware behaviours (however, ReMMoC is an open platform, therefore the developer can manage adaptation if they wish).

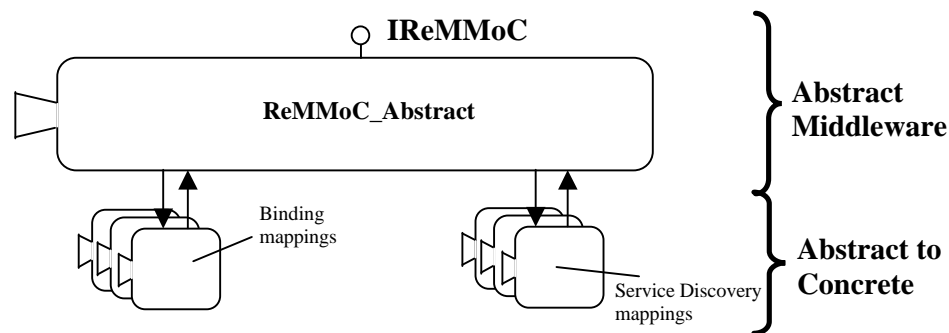


Figure 5.1 The ReMMoC programming model

ReMMoC's abstract programming model is separated into two distinct parts: 1) an abstract service invocation model (which can be mapped to the binding framework), and 2) an abstract service discovery model (that is mapped to the service discovery framework). The combination of these two complementary models provides an overriding abstraction for service use in mobile environments. ReMMoC's role in this abstraction is implemented by the architectural elements illustrated in figure 5.1; a single component (**ReMMoC_Abstract**) provides the abstraction API through the **IREMMoC** interface, and furthermore manages the underlying adaptation of concrete middleware to provide middleware transparency. Two separate mapping components are plugged into the **ReMMoC_Abstract** component, i.e. a mapping to the binding framework and a mapping to the service discovery framework. These form the abstract to concrete section of the ReMMoC architecture.

ReMMoC's abstraction, unlike other higher-level abstractions, considers abstract service discovery (rather than a single discovery mechanism). Section 5.3 describes the design of this new abstract service discovery model, which relies on the common features of individual discovery protocols. Furthermore, the service invocation abstraction is based upon core elements of the Web Services Architecture (it does not utilise the full specification). Section 5.4 describes the choice of Web Services as opposed to alternative abstractions. In addition, the techniques used to map from Web Services to the dynamically changing concrete binding implementations are specified.

5.3 The Service Discovery Abstraction

5.3.1 Overview

The key property of the abstract service discovery model is to provide a generic service lookup interface that hides the details of heterogeneous service discovery protocols from the application programmer. For example, when the user wishes to find services offering share service functionality, the generic lookup operation returns matches of all share services irrespective of the advertising technique (e.g. Jini, SLP, UPnP and Salutation).

The solution employed by ReMMoC is to provide a higher-level abstraction of discovery. This takes the form of a custom API, which is based upon the generic features of the majority of service discovery protocols. This API is then mapped by individual mapping components onto the implemented interfaces exported from the service discovery framework. This thesis concentrates on service lookup as part of a generic service discovery framework; other common features including leasing and service events are not considered because they are not available in all protocol implementations.

This section first examines the common features of discovery protocols, which leads to a specification of the service discovery abstraction. The architecture for abstract to concrete mapping is then defined, along with example implementations of mapping components.

5.3.2 The Service Discovery Abstraction

The IReMMoC interface provides the developer with a generic lookup API, as described by the interface in figure 5.2. This consists of two methods: *ServiceLookup* and *GetAttributes*. The required service type and list of attributes are passed to the ServiceLookup operation together with a handler to receive a returned event and an integer stating the time to search for. The information returned in this event is described by the data structure in figure 5.3. The key items of information returned are the ServiceType, the URL (used to identify the service location), and the Attribute list. ReMMoC uses this information to map subsequent abstract invocations to a particular service; hence, the data type is ReMMoC's service identifier. The GetAttributes operation returns all attributes for the identified service.

```
interface IServiceLookup : IUnknown {
    HRESULT ServicesLookup([char* ServiceType, Attributes[] attrs,
                          int TimeToSearch, ReMMoCServiceFindHandler cback,);
    HRESULT GetAttributes(ServiceReturnEvent ServiceID, AttributeList* list);
}
```

Figure 5.2 IDL definition of IServiceDiscovery interface

```
typedef struct _Attribute{
    char* Name;
    char* XMLValue;
}Attribute;

typedef Attribute AttributeList[MAX_ATTRIBUTES];

typedef struct _ServiceReturnEvent{
    char* ServiceURL;
    char* ServiceType;
    AttributeList List;
}ServiceReturnEvent;
```

Figure 5.3 The ServiceReturnEvent data structure

For example, finding weather services using the ServiceLookup operation across two discovery configurations, e.g. UPnP and SLP, returns a list of matched services from both types, i.e. multiple ServiceDiscoveryEvents. However, the developer does not know which protocol returned the event.

The design of the generic lookup API is based upon the similarities in implementations of each individual discovery mechanism. The abstraction relies on each protocol advertising a service by a single string element describing the service

type (e.g. Printer Service, Stock Quote Service). Furthermore, this technique relies upon the assumption that all services of the same service type provide the same service functionality. The abstract service binding (described later) utilises WSDL abstract service descriptions; hence, services with the same description (service type) offer the same type. In addition, the discovery abstraction relies on each discovery protocol describing service attributes (properties of the service) as a name value pair. The following sections now describe how the four major discovery protocols meet these requirements.

Service Location Protocol

The Service Location Protocol standard [Veizades97] advertises services through URLs. The URL has the form: “service:<abstract-type>:<concrete-type>” followed by a list of attributes, where an attribute is a name-value pair. Services are found using the service request message (SrvRqst). The service type string "service:<abstract-type>" matches all services of that abstract type. If the concrete type is also included only the specific service is found. For example: a SrvRqst that specifies "service:printer" as the Service Type will match the URL service:printer:lpr://hostname and service:printer:http://hostname. If the requests specified "service:printer:http" they would match only the latter URL.

Universal Plug and Play

Two complementary techniques combine in UPnP to provide the service type and attribute details of a service. The Simple Service Discovery Protocol (SSDP) is used to find services of a particular type. In UPnP devices host other devices and individual services. The service request operation allows devices and services to be searched for by device type or service type (of the format “service:serVICETYPE”). The resulting device location is returned, from which the URL of the service (similar to SLP) is obtained to determine the service location. Attribute information (available operations and/or device information) is stored in XML documents that must be downloaded from the device after a service match. The XML base of these descriptions provides attribute information in the required name-value pair format.

Jini

In Jini, services are discovered through matching Java class definitions. Normal operation requires that the complete class name and implemented methods match. However, this would not match the “ServiceType” as a string requirement; instead it is also possible to search by class name alone. Therefore, generation of an empty class from the ServiceType string is required to achieve this. Furthermore, attributes are attached to each service in the form of an attribute bean and it is possible to match information based upon these. The beans store information in name-value format; therefore it matches with the generic attribute properties.

Salutation

Salutation advertises services based upon a function ID that matches its behaviour (e.g. printer) along with a set of attribute values. Hence, the function ID matches directly to service type and the Salutation attributes map directly to the generic attributes. Notably, Salutation is the closest discovery protocol to the proposed generic abstraction.

5.3.3 Abstract to Concrete Mappings

The role of the abstract to concrete mapping section of ReMMoC’s generic service discovery architecture is to take the abstract lookup information passed by the application and map it onto the lookup APIs of each and every protocol currently configured in the service discovery framework. The architecture for doing this is illustrated in figure 5.4. A multi-receptacle is implemented by the ReMMoC_Abstract component; a multi-receptacle allows more than one component to be connected to the same receptacle at the same time. Therefore, multiple mapping components can exist between the abstraction and the concrete discovery components whose behaviour is exported by the discovery framework. A mapping component implements the ILookup interface that contains methods to pass information (including the event return handler) from service lookup operations through to the component. Each component then implements a receptacle to connect it to the discovery protocol (e.g. ISLP for Service Location Protocol). When the ServiceLookup operation is called by the application the multi-receptacle lookup method is invoked; this forces all of the connected mapping components to be called simultaneously (i.e. if three are

connected all three will be invoked). It is this design that allows discovery across heterogeneous protocols to be implemented in parallel.

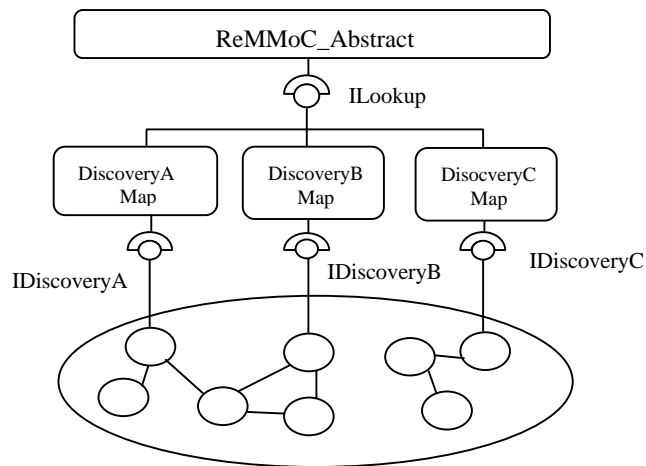


Figure 5.4 Abstract to concrete service discovery architecture

5.3.4 Proof of Concept (Implementation of Mapping Components)

Two mapping components were implemented in order for the two implemented discovery configurations (SLP and UPnP) to be used within the ReMMoC framework. As described previously, the properties of the SLP and UPnP APIs are fundamentally similar to the proposed generic API. Therefore, the lookup mappings are direct in nature.

For SLP, the generic service type is mapped to an SLP abstract type and the service request is made. For additional attribute matches, each passed attribute is mapped to the SLP attribute format. SLP returns its results in the format of a URL and a separate attribute list. Therefore, this information is placed directly into the data structure ServiceReturnEvent (seen in figure 5.3).

For UPnP, the generic service type is mapped to an SSDP service request in the format “service:ServiceType” (device lookup is not utilised). However, in UPnP the URL of the device hosting the service is returned (not the service) for matching services. Therefore, the mapping component downloads the list of services on that device and extracts the URL of the matching service, which can then be passed back as the result. The mapping component also performs additional functionality for

attribute matching. When a service is matched it downloads the attribute list and checks that the requested attributes match the XML defined attributes, before returning the result to the application.

5.4 The Abstract Service Binding Model

5.4.1 Overview

The key requirement of the abstract service binding model is to hide binding heterogeneity from the developer. Therefore, the invocation of a particular abstract service will be executed irrespective of the middleware implementation. For example, a user wishes to find out about latest share prices. Different share services may be implemented on heterogeneous middleware; for example a SOAP shares service, a CORBA shares service, a Java RMI shares service, a shares event publishing service, and so on. However, when the abstract operation *getQuote* is invoked the same information is returned whichever of the previous implementations actually executed the service.

In order to successfully tackle heterogeneity of this type another level of indirection is required (namely a service abstraction layer). Abstraction is a well-used solution for different types of heterogeneity (e.g. middleware originally addressed platform and operating system heterogeneity). In this case, a well-established, extensible open standard is required for interoperating parties to agree upon. This section first documents the reasons behind the choice of Web Services as the model for the higher-level binding abstraction. The remainder of the section then focuses on the mapping of abstract Web Service operations to individual communication paradigms, i.e. remote method invocations and publish-subscribe.

5.4.2 Abstract Web Services

Why Web Services?

Chapter 3 discussed current solutions to middleware heterogeneity, where only three described a higher-level, open interoperability standard. Web Services are described by abstract XML descriptions. The Model Driven Architecture models systems in

terms of Platform Independent Models. Similarly, UniFrame promotes its own language to model services abstractly.

The Web Service abstraction was chosen for the abstract binding model of ReMMoC for the following reasons:

- Web Services are already being heavily utilised as the key technology in integrating existing heterogeneous middleware platforms [Vinoski02].
- Web Services are simple, compared to complex modelling tools and languages (e.g. MDA and UNIFrame). The simplicity of the technique has driven the current interest in Web Services. Furthermore, it is interesting to compare Web Services with the World Wide Web; the Web is not the most sophisticated hypertext system but it is the largest and most used.

Therefore, the potential benefit of Web Services is that they will be the most frequently used technology for middleware interoperability, which is the most important factor when attempting to tackle heterogeneity. However, there remains the possibility that Web Services will become one of many competing open standards (this section has already discussed two competitors) to follow the predictable trends of previous middleware standards. Andrew S. Tanenbaum said, “*The nice thing about standards is that there are so many to choose from*”, which is especially true of the middleware domain. Hence, middleware hasn’t solved the interoperation problem, rather it has been moved up a level. However, with Web Services there is not the company driven competing standards (there is already worldwide agreement on technologies like XML), rather these companies are collaborating on these meta-standards. Hence, by complying with Web Service standards ReMMoC is less likely to become simply another middleware.

However, in the event of new meta-standards ReMMoC is extensible to incorporate a new higher-level abstraction. For example, the technologies of the reflective architecture described in chapter 4 can be applied within the Model Driven Architecture. For a new abstraction, the abstract and abstract-to-concrete sections of the ReMMoC architecture would need to be designed and implemented.

The Web Services Architecture

The intended goal of Web Services [W3C99] is to allow different service providers to implement centrally defined service interfaces using their chosen concrete middleware binding. For example, a news service may be implemented using SOAP by one vendor while another may use publish-subscribe. Client applications can then be developed to interoperate with either service upon dynamic discovery. The key to this technique is the concept of abstract Web Services; the Web Services Description Language (WSDL) [Chinnici03] separates abstract service definitions from definitions of concrete middleware binding messages. Therefore, WSDL, through abstract service descriptions, offers a higher-level abstraction for interoperation in a service-oriented architecture. An example of a WSDL described abstract service (in this case a Sport News service) is illustrated in figure 5.6. One or more port types describe each service; these are equivalent to interfaces as they describe units of service provision. Like an interface, each port type contains one or more operations (there are four types of operation, as described in section 3.2). Operations are defined by Input and Output messages, which are composed by a list of types defined in XML; the example shows an input message containing a single input parameter of type string.

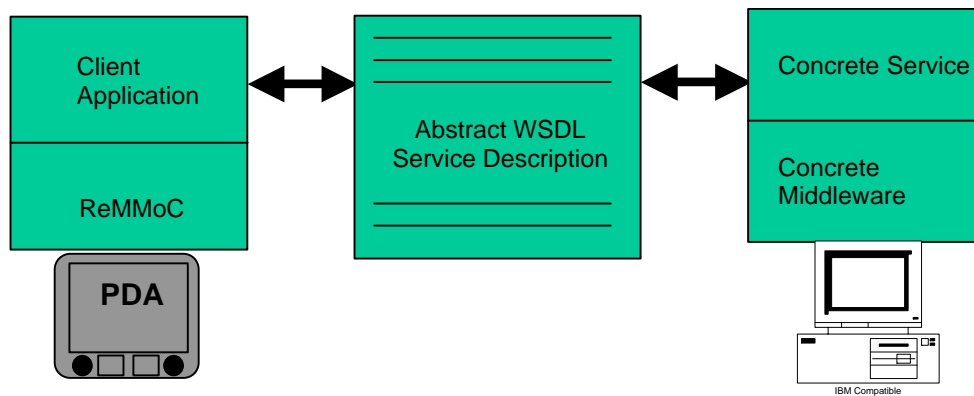


Figure 5.5 ReMMoC's role in the Web Services Architecture

The Web Services Architecture (see section 3.2) consists of three key roles: a service provider, a service requestor and the discovery agency, which the requestor uses to find the service description. Figure 5.5 illustrates ReMMoC's role in this architecture. Web Services are implemented upon any chosen concrete middleware, and advertised using any discovery protocol. The mobile client application (implemented atop

ReMMoC) is programmed against the abstract service description portion of the WSDL file. Firstly, the client application performs service lookup for the service type described in the abstract description. Secondly, the client invokes abstract operations described by the abstract description. The abstract operations are mapped to the corresponding messages of the underlying middleware binding (not just the SOAP protocol).

```

<?xml version="1.0"?>
<definitions name="SportNews">
  <types>
    <element name="LatestStoryRequest">
      <complexType>
        <all><element name="topic" type="string"/></all>
      </complexType>
    </element>
    <element name="LatestStory">
      <complexType>
        <all><element name="story" type="string"/></all>
      </complexType>
    </element>
  </types>
  <message name="GetLastestStoryInput">
    <part name="body" element=" LatestStoryRequest "/>
  </message>
  <message name=" GetLastestStoryOutput ">
    <part name="body" element=" LatestStory "/>
  </message>
  <portType name=" SportNewsPort ">
    <operation name=" GetLastestStory ">
      <input message=" GetLastestStoryInput "/>
      <output message=" GetLastestStoryOutput "/>
    </operation>
  </portType>
  <service name="SportNewsService">
    <port name="SportNewsPort" ></port>
  </service>
</definitions>

```

Figure 5.6 An abstract WSDL description for a sport news service

5.4.3 The Abstract Binding API

The abstraction section of the ReMMoC architecture (the ReMMoC_Abstract component) implements the IReMMoC interface that application developers use to find and invoke services. The operations from this API are now discussed in turn. The syntax of these methods is stated and their behaviour is specified. The first three methods parse and manipulate WSDL service descriptions, and the remainder invoke abstract service operations. Notably, the API is event-based. This is because different middleware types provide different models of computation e.g. synchronous styles are opposite to asynchronous styles, and therefore they differ in how information flows to

and from the application. Therefore, using an event-based programming model for every abstract operation ensures information flow is consistent to the application.

1. HRESULT WSDLGet(WSDLService* servDesc, char* XML);

WSDLGet parses an XML WSDL description passed to the method as a string; a data structure of type WSDLService (illustrated in figure 5.8) is then created to hold this information. Figure 5.7 demonstrates the processes involved in invoking abstract services. An event handler is registered to receive the result of the operation. Therefore, when an abstract operation is invoked its corresponding abstract data structure is mapped to the current binding interface (IIIOP in the diagram), the concrete operation is called, and when the result is returned it is mapped into the data structure and the event handler is up called.

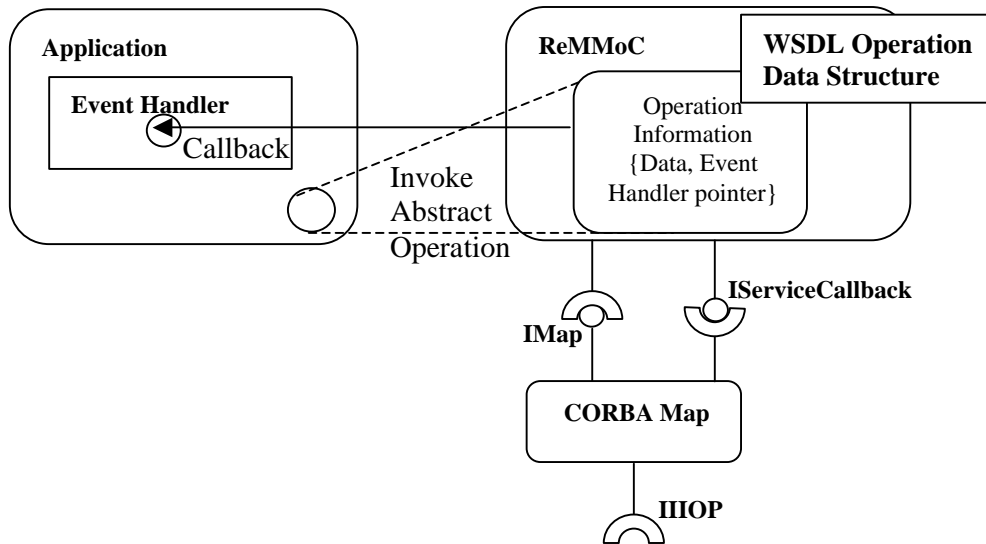


Figure 5.7 Invoking remote WSDL operations (RequestResponse and OneWay)

At the heart of this process is the WSDL Operation data structure; Figure 5.8 documents the layout of this important data type. WSDL elements including: Operation name, messages and type elements are stored. Furthermore, two additional pieces of information are maintained per operation; firstly, the event handler that will be called when the result of an operation returns (this is also the remote operation invoked by other services, described later by CreateOperation), and secondly it stores the number of times the operation must be executed.

2. HRESULT AddMessageValue(WSDLOperation *Operation, char* ElementName, VARIANT value, WSDL_TYPE type);

AddMessageValue allows the programmer to set values for the input message e.g. setting a ticker symbol to “IBM” before invoking a getQuote operation. The operation data structure, element name and then the type and value to be set (to create the parameter) are passed. The operation data structure is then updated with this information.

```
typedef enum {
    RequestResponse, OneWay, Notification, SolicitResponse
} WSDLTransmissionType;

typedef struct _WSDLMessageElement{
    char* Name;
    Parameter Param;
} WSDLMessageElement;

typedef struct _WSDLMessage{
    char* Name;
    int ElementCount;
    WSDLMessageElement Body;
} WSDLMessage;

typedef struct _WSDLOperation{
    char* OperationName;
    WSDLMessage Output;
    WSDLMessage Input;
    WSDLMessage Fault;
    void* Handler;
    int Evts;
    WSDLTransmissionType Type;
} WSDLOperation;

typedef struct _WSDLPort{
    char* PortType;
    char* Binding;
    WSDLOperation* OperationList;
} WSDLPort;

typedef struct _WSDLService{
    char* ServiceType;
    WSDLPort* PortList;
} WSDLService;
```

Figure 5.8. The WSDL data structure

3. HRESULT GetMessageValue(WSDLOperation *operation, char* ElementName, char* MessageName, VARIANT *value);

GetMessageValue allows the programmer to retrieve values from the abstract output message e.g. retrieving the returned float value of a stock request. The element name is passed to find the position in the WSDLOperation data structure, and the value is returned to the caller as a VARIANT type.

4. HRESULT KnownOperationCall(ServiceReturnEvent* LookupEvent, WSDLOperation* ServiceDescription, int Iterations, OperationHandler* Handler);

KnownOperationCall performs the invocation of abstract operations (of the type Request-Response and One-Way). The location of the service must be known; therefore the ServiceReturnEvent generated from the ServiceLookup method is passed to the operation so the invocation is directed to the concrete service. The operation data structure, the number of times the operation should be executed (if the application requires multiple results) and finally, the event handler that will receive the operation results is passed.

5. OperationCall(char* ServiceType, WSDLOperation* ServiceDescription, int Iterations, OperationHandler* Handler);

OperationCall provides the same operation as KnownOperationCall, however it is not directed at a specific service endpoint. Instead, service lookup is performed first and the abstract operation is performed on the first instance of a found service.

6. HRESULT CreateOperation(ServiceReturnEvent* LookupEvent, WSDLOperation* ServiceDescription, int Iterations, CreateOperationHandler* Handler);

CreateOperation allows the programmer to specify a local operation that can be invoked remotely by other services. Other services describe their service requirements in SolicitResponse and Notification operations. Hence, the application programmer uses CreateOperation to create services to match these requirements. Again, the service description and operation name to create is passed. In addition, the handler this time is the service behaviour, rather than a result handler; therefore, a C method that will be invoked remotely is passed. The iterations parameter specifies the number of times the operation is expected to be invoked, typically once or infinitely (-1). Note that the lookup event is also passed to determine the binding to host the operation on.

ReMMoC is a client side framework that mirrors the binding of the current service it is interacting with. Therefore, when interacting with a CORBA service, abstract operations are created as CORBA services, and when interacting with a publish-subscribe service, the abstract operations are hosted upon a publisher. Figure 5.9 shows the layout of a CreateOperation call. The binding will receive incoming requests and these will then be mapped by the mapping component to the corresponding handler for the abstract operation.

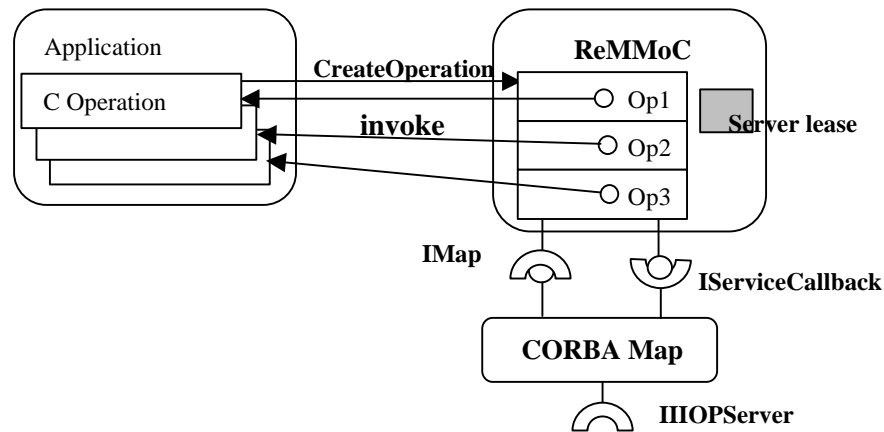


Figure 5.9 Creating operations (Solicit-Response and Notification)

7. HRESULT Receive();

After initialising one or more operations through the CreateOperation method, calling Receive() configures ReMMoC to begin receiving remote invocations. That is, the operations previously registered become available to use by remote services.

8. HRESULT EndReceive();

EndReceive() is an important operation that stops incoming messages and also releases the lease (seen in figure 5.9) on the current service side implementation. ReMMoC currently only allows services to be hosted upon one binding type at a time e.g. all operations hosted as IIOP. This is due to the implementation of the binding framework. Future work could allow services to be hosted over multiple binding types in the style of multi-personality ORBs such as UIC [Roman01]. Therefore, after the lease has been released a new binding can be configured onto which the hosted operations can be mapped. When the device changes location or interacts with a new service of a different binding type (i.e. after it has performed a new lookup)

EndReceive flushes the previous underlying service binding and allows the operations to be hosted over a new binding through re-inocations of the CreateOperation method. In addition, the lease is automatically released when the iteration count for every hosted operation is zero.

9. Char* GetID();

Certain applications require knowledge of the identifier of the service e.g. passing the ID of a service to an interacting element in order for it to communicate back. The GetID() operation returns a string reference of the hosted service (set of abstract operations). The string ID depends upon the underlying binding; for example this is the IOR for CORBA and the URL for SOAP and publish-subscribe.

5.5 Mapping Abstract Operations to Concrete Communication Paradigms

5.5.1 Introduction

In this section we demonstrate how the abstract operations of WSDL can be mapped to the two contrasting binding paradigms that are implemented by the concrete section of ReMMoC, namely Remote Method Invocation (SOAP and IIOP) and Publish-Subscribe. There are four abstract operations in WSDL that must be mapped to the corresponding operations in the concrete paradigms; these abstract operations are formatted as follows:

- 1) *Request-Response (input message, output message)*. The service provider sends a response to a request of its service. The information to request a service is detailed in the input message, while the output message contains the response.
- 2) *Solicit-Response (output message, input message)*. The service provider acts as a service requestor. The information about the request is held in the output message and the input message contains the response.
- 3) *One-Way (input message)*. The service provider receives a notification message.
- 4) *Notification (output message)*. The service provider outputs a notification message.

For these mappings to be effective, the following assumptions are made about the current scenario:

- *The service provider and service requestor are both implemented against the same abstract WSDL definition. That is, there is an exact syntactic match and hence, type compatibility between the two parties.*
- *There is no guarantee that the service provider offers a semantic match to the requestor's operation; although there is a syntactic match it may not provide the required behaviour and functionality.*
- *Only primitive types and arrays are used in abstract WSDL descriptions.*

5.5.2 Mapping Abstract Operations to Remote Method Invocation

Overview

The mapping of WSDL operations to Remote Method Invocations is based upon the similarities between the abstract messages of WSDL and the concrete messages of RMI. Figure 5.10 informally defines the elements of an abstract WSDL operation; an operation consists of an operation name, an input message and/ or an output message, where each message consists of a set of attribute value pairs.

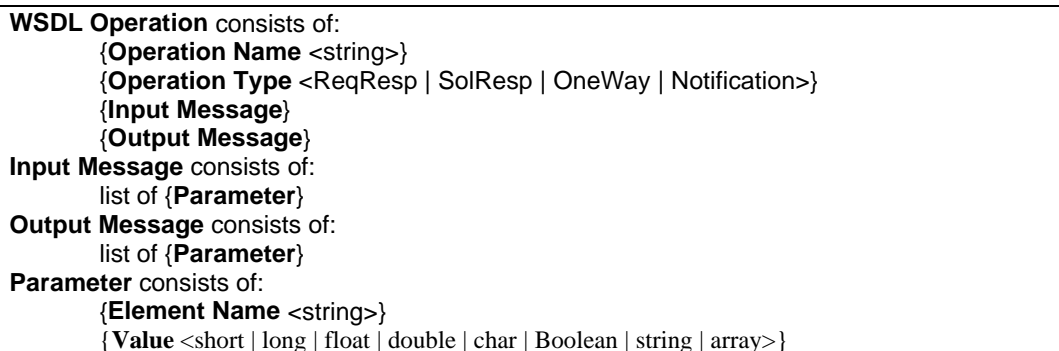


Figure 5.10 Elements of a WSDL operation

Similarly, figure 5.11 informally specifies the format of a typical remote method invocation; the RMI request consists an operation name and a set of input parameters, where a parameter is a name value pair. The method result is synchronously returned as a list of output parameters. To demonstrate a complete mapping between WSDL and the RMI paradigm, this section describes the techniques of each of the four WSDL operations.

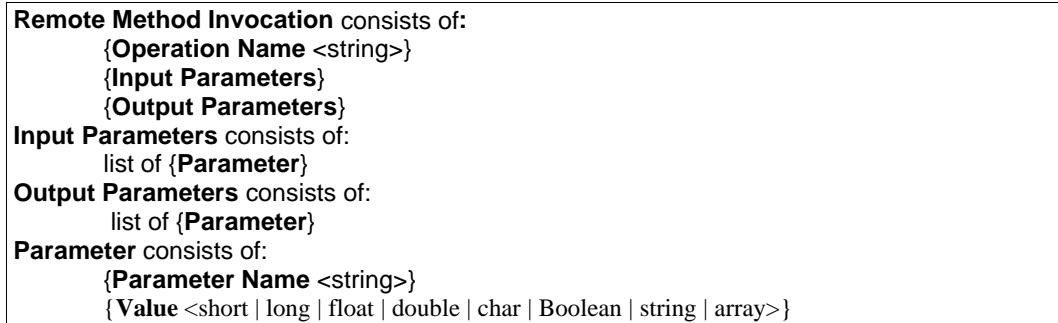


Figure 5.11 Elements of a Remote Method Invocation

Request-Response

The abstract Request-Response operation is mapped directly to a full Remote Method Invocation. That is the service expects a physical input message (RMI request) and will respond with a physical output message (RMI response). Therefore, the input/output messages of abstract Request-Response operations are mapped directly to the corresponding synchronous RMI request and responses typical of implementations including SOAP and IIOP. This technique is illustrated in detail in figure 5.12. The operation name maps to the method name to be invoked, the elements of the input message are used to create the input parameter list and finally the result contained in the output parameter list is mapped to fill the values of the output message elements.

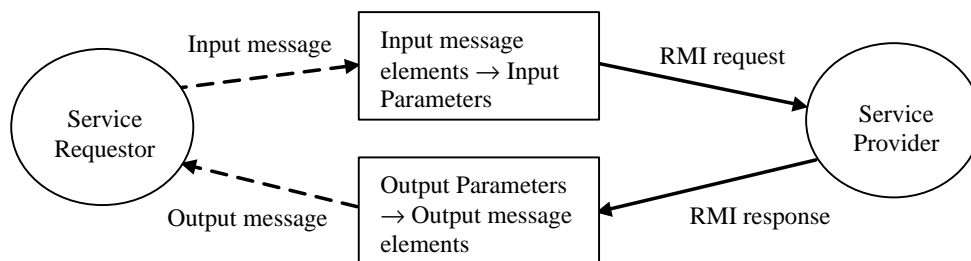


Figure 5.12 Mapping abstract Request-Response to RMI

This technique is based upon the similarities between abstract and concrete messages as previously described in figures 5.10 and 5.11. Therefore the mapping can be made for identical operation names, when the types of the parameters and message elements match. For example, the GetLastestStory operation defined in figure 5.6 is mapped by creating an input parameter of type string with the defined vale. The named operation

is invoked and the resulting output parameter of type string is mapped back to the string element in the output message.

One-Way

An abstract one-way operation states the service provider expects to receive a single concrete input message that it will react to and no response is generated. The abstract operation is defined by an operation name followed by an input message. Therefore, this information is mapped to one-way remote method calls, and this process (shown in figure 5.13) is a subset of the procedure for a full request response operation i.e. the operation is invoked with input parameters only (generated from the elements of the input message). Depending on the RMI implementation, there may be no concrete message returned (e.g. CORBA one-way), or a concrete return with no parameters (e.g. Java RMI). Therefore, it is the responsibility of individual mapping components to ensure consistent behaviour.

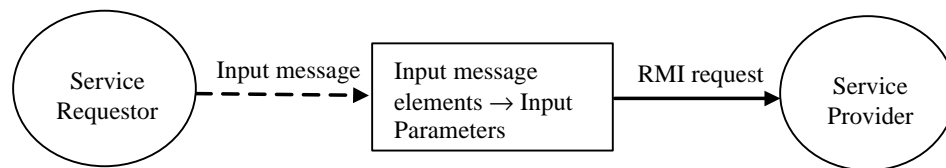


Figure 5.13 Mapping abstract One-Way to RMI

Solicit-Response

A Solicit-Response operation in the service provider definition describes a request response operation carried out by that service, i.e. it doesn't define an operation to invoke, rather the requirement of an operation of another service. ReMMoC exists at the service requestor side; therefore the service provider will invoke an operation hosted by the application running on ReMMoC. As described in the abstract API section, an RMI operation is created to match this Solicit-Response contract i.e. its method name matches the operation name, the set of input parameters match the information from the output message list. The method then produces a result whose output must be in the form of the elements described by the abstract input message. An overview of this mapping is illustrated in figure 5.14; it can be seen that predictably this is the reverse of Request-Response.

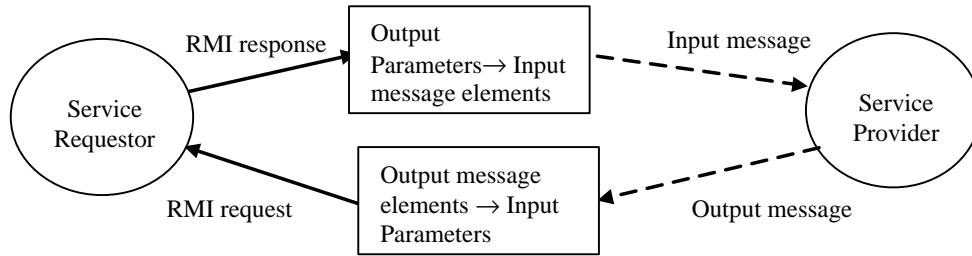


Figure 5.14 Mapping abstract Solicit-Response to RMI

Notification

The abstract Notification operation is similar to Solicit-Response. Service providers define notification messages; these simply state single output messages that are generated by individual services. Service requestors or clients then implement functionality to retrieve these when they are generated. The mapping of this operation to the RMI paradigm is a subset of the mapping for the Solicit-Response operation (illustrated in figure 5.15). The requestor implements a method that is hosted as a one-way RMI operation. It contractually matches the Notification description i.e. the same operation name, and a set of input parameters that map to the output message elements. Therefore, the service will simply receive and react to incoming RMI requests that match the Notification. Like one-way, the mapping component for bindings of these types must ensure that if the invoker expects an empty return one is generated.

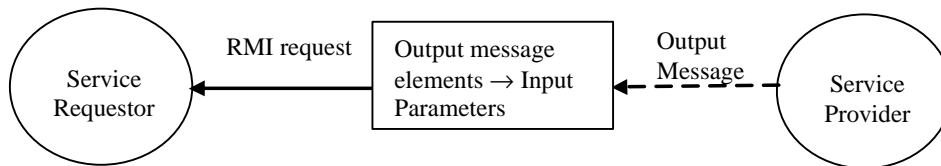


Figure 5.15 Mapping abstract Notification to RMI

5.5.3 Mapping to Publish-Subscribe

Overview

Publish-Subscribe is an alternative communication paradigm whereby there is no direct message exchange between service requestor and provider. A service provider publishes events and a service requestor must filter to receive appropriate events. Therefore, unlike RMI, the mapping of WSDL to publish-subscribe is not a direct correlation. However, the technique employed to perform the mappings is again based

upon the similarities between WSDL messages and published events (the structure of a generic publish-subscribe event is informally specified in figure 5.16). This section then proposes suitable methods to fit each abstract operation to a particular publish-subscribe scenario. Each operation mapping is now described in turn; the goal of these mappings is to ensure that the user of the abstract operation has no idea what paradigm (RMI or Publish-Subscribe) is actually implemented.

<p>Publish-Subscribe Event consists of: {Subject <string>} {Content} Content consists of: List of Attributes Attributes consist of: {Attribute Name <string>} {Value <short long float double char Boolean string array>}</p>

Figure 5.16 **General elements of a produced Publish-Subscribe event**

Request-Response

The abstract request-response operation is a request of a service based upon the information entered in the input message. In publish-subscribe, the subscriber identifies what they wish to receive through a filter. Therefore, this mapping takes the information from the input message to create a new filter. The similarities between the abstract operations and the structure of a filter make this possible. Filters generally take the form of subject and or content filters i.e. these can filter to receive all events of subject A or events with content attributes of type name=value. Hence, the operation name is used as the subject and then the input message elements are used to create each individual content filters (this is a direct mapping as both parties are name-value pairs of the same type). The requestor will then receive one or more events that match this filter. The content of these concrete events become the results of the request-response abstract operation and therefore, these are mapped to the output message. Each content attribute is mapped directly to the corresponding output message element. The complete mapping process is illustrated in figure 5.17.

As an example of this process, a stock quote service produces a set of events. The subject is getQuote and the content is the ticker symbol (e.g. ticker=IBM) along with its value (e.g. value = 89.1). An abstract request response operation of getQuote(IBM) will create a filter with subject equal to getQuote and a single content rule attribute

tickerSymbol equals IBM. The returned event for IBM contains the current value, which is mapped to the element in the output message.

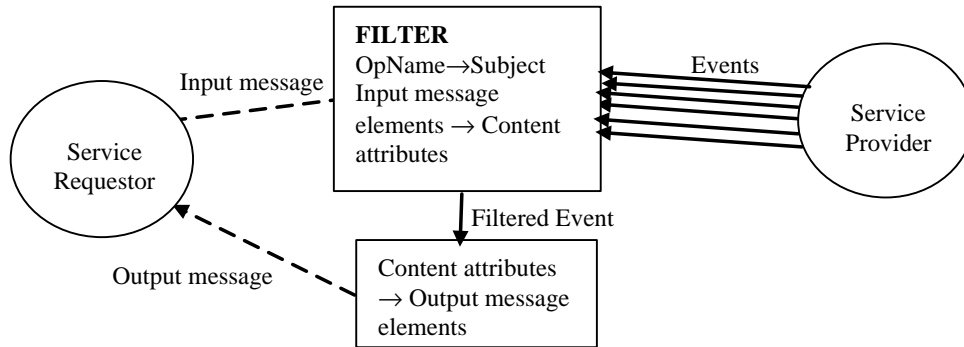


Figure 5.17 Mapping Request-Response to Publish-Subscribe

One-Way

A service provider states in a One-Way message they expect to receive input, and will not return a response. Therefore, to map publish-subscribe to this behaviour requires that the service requester produce events that the service provider is filtering for. The only information available is the operation name and input message. Hence, the technique employed is for the provider to filter on subject name alone; the concrete events which are filtered are then mapped to the input message (i.e. the publisher must create events that meet this contract). The outline of this mapping is illustrated in figure 5.18. For example, a one-way operation for a shares service is addNewShare, which contains the input elements: ticker symbol and starting value. When mapped across publish-subscribe, the service will filter events by subject (addNewShare), and then extract the ticker string and value from the received events.

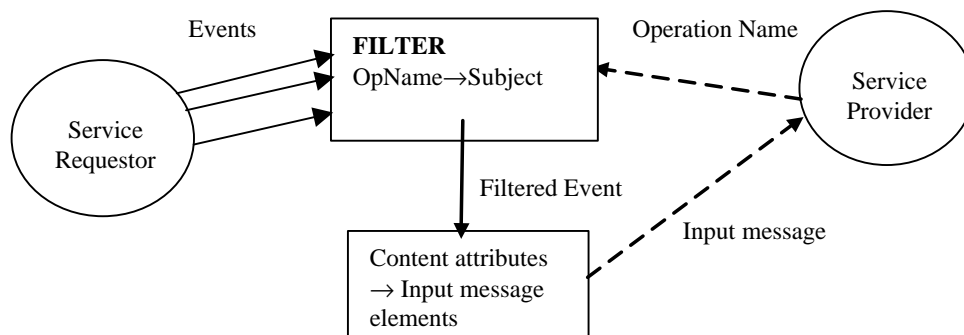


Figure 5.18 Mapping One-Way to Publish-Subscribe

Solicit-Response

Mapping Solicit-Response to publish-subscribe is the reverse of the technique used for Request-Response. This time the service provider expects to receive events of a certain type, and therefore will have created a filter for this; the service requestor role must then publish the correct events to provide a match. Therefore, to meet the Solicit-Response contract agreement, the service requestor creates events with a subject that matches the operation name, and a set of content-attribute values that will first be filtered correctly (it contains the attributes that map to the message elements of the output message) and secondly whose content maps directly to the message elements of the input message (as described by the abstract input message). The complete mapping behaviour is seen below in figure 5.19.

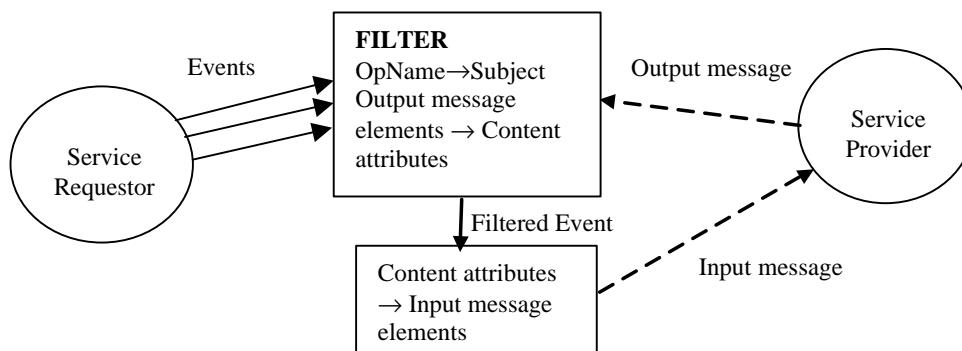


Figure 5.19 Mapping Solicit-Response to Publish-Subscribe

Notification

Notification is the reverse of One-Way. The service provider produces events that the client or requestor filters to receive. The operation name and output message are the information available from the abstract contract. Hence, the client creates a subject filter using the operation name. Matching events are then mapped to output message content (illustrated in figure 5.20); each content attribute value is matched to the corresponding element of the output message. As previously there must be a contractual agreement that the content of the events matches the types and structure of the output message, otherwise this process will fail.

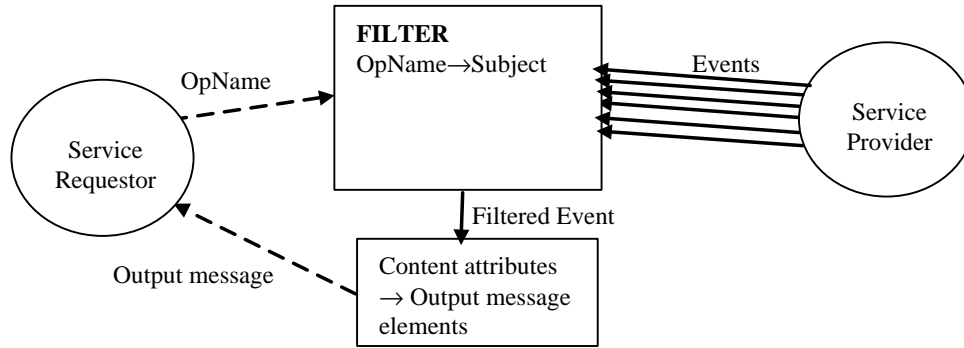


Figure 5.20 Mapping Notification to Publish-Subscribe

5.5.4 Implementation of mapping components

Overview

A single mapping component is required for every middleware binding implementation (e.g. SOAP, CORBA and STEAM publish-subscribe). The role of this component is to implement the abstract to concrete mappings described previously specifically for a particular middleware implementation. These components have a similar format, as seen in figure 5.22 that allows them to be plugged into the framework between the ReMMoC_abstract component and the binding framework. Each component implements one interface (IMap) and one receptacle (IServiceCallback), which are outlined in figure 5.21. There are four operations in the IMap interface: MapInvoke, MapCreate, Receive and EndReceive; each is invoked when the corresponding operation of the abstraction API is called. The IServiceCallback interface is implemented by the ReMMoC_Abstract component, and the mapping component uses the ServiceCallback method to return the results of an operation.

```

interface IMap: IUnknown {
    HRESULT MapInvoke(ServiceReturnEvent sre, WSDLOperation su);
    HRESULT MapCreate(ServiceReturnEvent sre, WSDLOperation su);
    HRESULT Receive();
    HRESULT EndReceive();
};

interface IServiceCallback: IUnknown {
    HRESULT ServiceCallback(WSDLOperation ServiceDescription);
}

```

Figure 5.21 The IMap and IServiceCallback interfaces

For the ReMMoC implementation, a separate mapping component was created for each of the implemented bindings, namely: SOAP, IIOP and STEAM publish-subscribe. The implementations of two of these mapping components (IIOP and publish-subscribe) are now described in detail.

The IIOP mapping component

The structure of the IIOP map component is seen in figure 5.22; in addition to the standard interface and receptacle, the component implements two additional receptacles: IIIOP and IIIOPServer. These allow the component to invoke operations upon the corresponding binding in the framework. RMI and One-Way operations are called through IIIOP, while the operations to create and host remote objects are made through IIIOPServer.

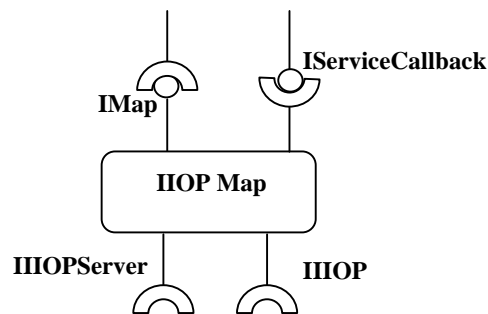


Figure 5.22 The IIOP map component

Application developers use the abstraction API method for Request-Response and One-Way operations. Both manipulate the WSDLOperation data structure (seen in figure 5.4) that holds the operation name, input message, output message and event handler. Note One-way has no handler or output message. The algorithm to then map either is as follows:

```

IOR (unique identifier of service hosting the operation)= ServiceReturnEvent.ior;
RemoteObject = new Object(IOR);
InputSize = Number of elements in abstract input message;
Input Parameters = new Parameter Array (InputSize);
For index = 0 to InputSize{
    Read type and Read value of input message element [index];
    Parameter = new Parameter (type, value);
    Input Parameters [index] = Parameter;
}
If (OperationType==Request-Response){
    OutputSize = number of elements in output message;
    Output Parameters = new Parameter Array (OutputSize);
    For index = 0 to OutputSize{
        Read type and Read value of output message element [index];
        Parameter = new Parameter (type, value);
        Output Parameters [index] = Parameter;
    }
}
Invoke(RemoteObject, OperationName, Input Parameters, Output Parameters);
If (OperationType==Request-Response){
    For index = 0 to OutputSize{
        Read value of Output Parameters [index];
        Output message element [index].value = value;
    }
    WSDLOperation.Handler(Output Message elements);
}

```

The abstract Solicit-Response and Notification operations are called using the CreateOperation. The developer creates a C method named OperationName that takes the WSDLOperation data structure as an in/out parameter. The IIOP map component then creates a generic method dispatcher; i.e. a single Object, referenced by a single IOR, hosts all these registered operations. This object is then hosted as a remote object using the standard IIOPServer implementation; hence the solution is based on a simple two level object adaptor. Incoming IIOP requests are then directed to the corresponding registered method (for both full and one-way requests). However, the content of the concrete request must be mapped first to the WSDLOperation data structure (to be passed as a parameter), using the reverse of the technique used in the algorithm above. Each input parameter is mapped to an element from the output message. The C operation then extracts this information and for solicit-response generates the values placed in the input message. When the response is returned these

input message elements are mapped to create a new output parameter array, which is then used to construct the IIOP response.

The Publish-Subscribe Mapping component

In addition to the standard IMap interface and IServiceCallback receptacle the publish-subscribe mapping component implements two receptacles: IPublish and ISubscribe. These allow the component to be connected to the publish-subscribe personality of the binding framework. All events are currently published upon the same multicast channel. Therefore, no extra information (like IOR in IIOP) is needed by participating parties; however, the mapping may be extended across multiple channels in the future, by adding attributes to the service discovery content.

Application developers call the abstraction operations for Request-Response and One-Way. However, as described previously, Request-Response is a subscribe operation and One-Way is a publish operation. Therefore, the mapping component implements the MapInvoke method to check the operation type and react accordingly. The Request-Response mapping follows the algorithm below. A filter is created to receive events, the operation name matches the subject and the input message elements match the content. The remaining content elements of the received event are then mapped directly to the output message elements, which are returned to the application using the passed event handler. One-Way is invoked with no event handler, only the operation name and input message. Therefore, the mapping component uses this information to create and publish an event to be filtered by the service. Hence, an event of subject operation name is published. Subscribers that agree to the One-Way operation contract (matching subject) can then receive these incoming one-way requests, the content of the event mapped from the input message elements.

```
N = Number of elements in Input Message;  
Filter = OperationName//{{Input Message [0].Name=Input Message [0].Value ...  
Input Message [n].Name=Input Message [n].Value};  
Subscribe(Filter);  
Event = Received Event;  
OutputSize= number of elements in Output Message;  
For index=0 to OutputSize{  
Output message element[index].value=Event attribute [index].value;  
}  
WSDLOperation.Handler(Output Message elements);
```

Application developers use the API method `CreateOperation` for Solicit-Response and Notification operations passing a C method to be invoked from incoming requests. For mapping Solicit-Response, the passed method is used to generate published content. Therefore, the operation is invoked using the output message elements as parameters. The method produces the input message elements as a result. The mapping component then uses this information to create an event that can be filtered for. The operation name is the subject and the output message elements become the content filter attributes, while the resulting input message values form the remaining content. For notification operations, the component filters to receive events that when received force the registered C method to be invoked. Therefore, the operation name is used to form the filter. On notification of a match, the content of the event is mapped to the output message elements, which are passed as parameters to the C method.

5.6 Managing adaptation of the Binding Framework

5.6.1 Overview

The final task of the `ReMMoC_Abstract` component is to manage the adaptation of the binding framework to ensure that the configuration matches both the requirements of the binding operation (client-side or server-side) and the binding type of the service (SOAP, IIOP or Publish-Subscribe). For this purpose, the information returned from service discovery drives reconfiguration. In addition, the style of operation defines rules for reconfiguration (e.g. a create operation for RMI requires service functionality to be configured). The algorithms for configuration are the same as those described for the discovery framework, i.e. each possible binding personality is stored in an XML description (available to the `ReMMoC_Abstract` component); when a change is required the corresponding XML for the personality is parsed and the components are loaded and connected into the binding framework using the `ICFMetaArchitecture` operations.

5.6.2 Rules for Configuration based upon Binding Information

The information returned in `ServiceLookupEvents` is used by `ReMMoC` to configure the binding framework to the correct middleware implementation. Currently, the URL

holds the information to do this; SLP and UPnP return URLs in the format shown in table 5.1. Note to comply with the UPnP standard, UPnP should advertise only SOAP services. ReMMoC then extracts the concrete protocol from the URL (e.g. IIOP, HTTP or steam) to determine which style of binding to configure into the binding framework. This technique is possible because of the descriptive nature of URLs used in SLP and UPnP. However, ReMMoC also accesses attributes, therefore in an alternative discovery protocol the binding can be explicitly defined as an attribute by the advertiser.

Binding type	UPnP URL format	SLP URL format
IIOP	N/A	service:servicename:iiop://hostname
SOAP	service:servicename:http://hostname	service:servicename:http://hostname
STEAM	N/A	service:servicename:steam://hostname

Table 5.1 URL formats for binding types

5.6.3 Rules for Configuring Client and Server Side Bindings

When abstract operations require client side functionality, i.e. IIOP requests, SOAP requests and publish-subscribe subscribes, the required components are configured and are then available to be reconfigured when the request has completed. The component framework lock maintains this behaviour; the lock does not release until an RMI request or subscribe operation completes.

However, when services are hosted over a particular binding style (i.e. publisher or IIOP server) the binding personality must remain in place until these services are released. Therefore, an extra server lease applies in the binding framework and mapping components. Once, a server side personality hosts a set of services that personality cannot be changed until the server lease expires (caused by flushing the personality using the EndReceive() operation, or natural release of all operations). While the server personality is configured it remains possible to change just the client side. Therefore, it is possible to have IIOPServer and Subscribe followed later by IIOPServer and SOAP client.

5.7 Summary

This chapter has presented the abstract programming model of the ReMMoC architecture to hide the application developer from heterogeneous middleware implementations. This chapter proposed the following key features.

- A higher-level generic discovery abstraction based upon the common service type and attribute elements in the majority of discovery protocols.
- A higher-level generic service binding abstraction based upon the Web Services Architecture. Only abstract WSDL descriptions are utilised.
- Mappings from abstract WSDL operations to concrete binding implementations.
- A demonstration that WSDL operations can be mapped to two contrasting communication paradigms: RMI and publish-subscribe.

The next chapter of this thesis qualitatively evaluates the ReMMoC architecture and quantitatively measures its performance.

Chapter 6 Evaluation

6.1 Introduction

This chapter presents an evaluation of the ReMMoC framework. The evaluation methodology adopted by this thesis follows the established combination of qualitative and quantitative evaluation for systems of this type. The fundamental goal of this thesis is to demonstrate that the ReMMoC framework addresses the problem of middleware heterogeneity in mobile computing scenarios. This is examined by a qualitative evaluation (described in section 6.2), which seeks to demonstrate that the required adaptation is performed, and that the higher-level abstraction provides the necessary level of middleware transparency. A typical mobile scenario is presented consisting of three individual application case studies. The behaviour (adaptation) of ReMMoC is then investigated over the time duration of these applications. Furthermore, the development process of middleware-independent, mobile client applications is analysed, i.e. can realistic mobile applications be produced using this abstraction?

In addition, the performance of the ReMMoC framework is evaluated quantitatively in section 6.3. The ability to tackle heterogeneity inevitably comes at the price of increased performance time overhead. This qualitative evaluation examines in detail what this overhead consists of, and compares it to baseline middleware functionality. The framework also operates on mobile devices, therefore it must not consume excessive system resources, or perform considerably worse than existing mobile middleware. However reflection is employed, which has been criticised as an unsuitable technique for mobile devices due to the increase in performance time overhead. Hence, the quantitative evaluation consists of a set of benchmark tests of ReMMoC's performance. These aim to demonstrate that although using reflection to tackle heterogeneity carries an overhead in terms of both resource costs (memory) and performance time, it remains a feasible solution for mobile devices.

6.2 Qualitative Evaluation

6.2.1 Overview

The approach of the qualitative evaluation is to demonstrate that mobile client applications developed using ReMMoC continuously operate across different locations with application services implemented upon different middleware types. This section first presents a typical scenario in the lifetime of a mobile user that consists of a realistic level of heterogeneous middleware implementation. Three case studies (individual mobile applications) are then identified within this scenario. Firstly, an investigation of the adaptation behaviour of ReMMoC in each case study is carried out. Secondly, the operation of the higher-level abstraction is analysed to ensure each application behaves as required. Finally, the process for developing realistic mobile applications and services using the ReMMoC framework is analysed.

6.2.2 Mobile Scenario

A simple mobile scenario (similar to that described in section 1.4) is illustrated in figure 6.1; it is simple in that it consists of only three locations, each populated by two or more of the same three application services. In the future more sophisticated scenarios are likely to be available to mobile users; the user will move between many individual locations that could each be populated with hundreds of applications. However, even in this simple case the problem of middleware heterogeneity arises, and this problem will only escalate in the more sophisticated scenarios.

Within the scenario, the three locations are the user's home, the user's office and a coffee bar close to the office. All three locations are covered by an individual wireless network hotspot; users can then connect to these networks using PDAs or laptops. Three applications reside across the three locations; these applications encompass a range of application styles that are typically utilised in mobile settings. The first application is a stock quote service; this allows the user to request the price of individual shares and view the current status of their portfolio. The style of this application is information retrieval; tourist guides, cinema information, news and weather are examples of other such mobile information applications. The second application is a chat service; this allows the user to communicate with other local

users (who may be connected from a fixed or portable machine). The style of this application is communication-oriented; messaging, video conferencing and collaborative work are examples of other mobile communication applications. Finally, the third application is a jukebox service. At each location a physical device within the environment plays music (typically these are in the form of audio speakers connected to a computational device). The mobile user can display the list of songs available from the jukebox on their mobile device; from here they can then select the song they wish to play. The style of this application is remote control; a mobile device is used to interact with devices in the environment. Alternative applications of this type are video screen displays and light switches (and more generally home automata).

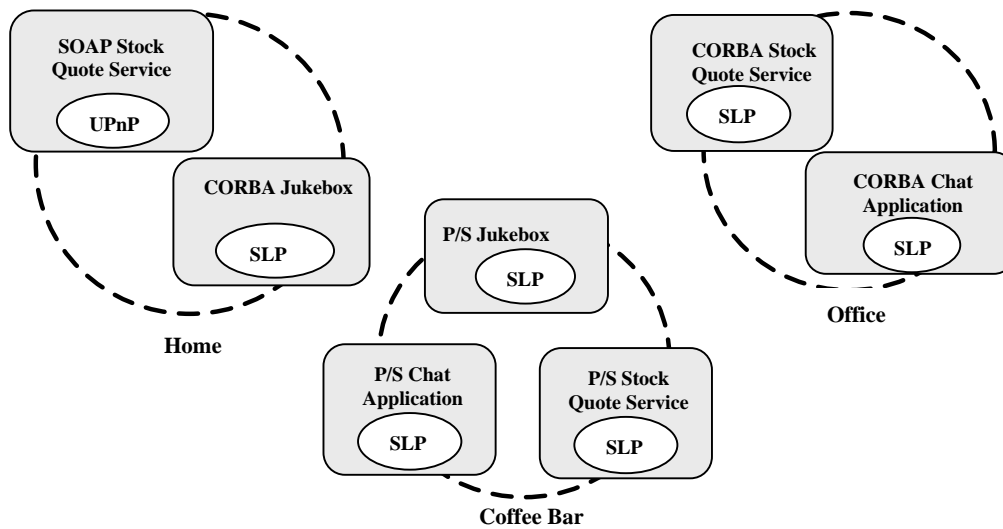


Figure 6.1 The evaluation scenario

Notably, the scenario is populated by heterogeneous middleware implementation. In the office location, the stock quote service and chat service (realised as an application upon another user's mobile device or desktop) are implemented as CORBA services and are advertised using the Service Location Protocol. This exemplifies how at one location the same middleware may be used for all applications. However, the home location does not follow this policy. The stock quote service is implemented as a SOAP service that is advertised using UPnP, and the Jukebox service is implemented using a CORBA implementation, which is advertised by SLP. All three applications reside at the coffee bar and all of them are implemented as publish-subscribe services and advertised using SLP.

6.2.3 Implementing the Scenario

Overview

To evaluate the development of mobile applications using ReMMoC, a test harness was implemented to emulate the previously described scenario. The first step was to create the abstract service descriptions for each of the applications. These three complete WSDL documents are located in Appendix C of this thesis. Three of the four possible WSDL operations are utilised within the services. For example, the stock quote service contains request-response operations (e.g. GetStockQuote). Similarly, the Chat service contains one-way and notification operations; the client uses one-way operations to send chat input, while the other participant (service) responds using notification operations. The Jukebox services use request-response to retrieve play lists (e.g. ListSong) and a one-way operation to play the chosen song.

In the scenario, a wireless network covers each of the three locations; for this purpose, the 802.11b wireless network was used, which has hotspots across the Lancaster University campus. Services operating from fixed machines were hosted using a desktop machine with a 750MHz Pentium processor and 128Mbytes of RAM running the Windows 2000 operating system. Applications operating from mobile devices were hosted upon either a Toshiba e740 Pocket PC or a Compaq iPaq H350 (both with the specification: 206 MHz StrongARM processor, 64 Mbytes of RAM and Windows CE 3.0 OS). To simulate the changes in location by the mobile user, the currently advertised and hosted services are dynamically changed. For example, a move from the home to the office requires that the two available services be removed from the discovery protocols in the network and then physically shutdown, while the two new versions of the applications are started and then advertised. A simple executable was created to manage this process.

Developing CORBA Services

To create the three CORBA application services described in the scenario, each of the WSDL service descriptions were implemented as single CORBA objects. The CORBA services hosted upon fixed machines, e.g. the jukebox and stock quote services were implemented using the Orbacus ORB version 4.0.5, whereas, the CORBA chat application hosted upon the Pocket PC device was developed using the individual CORBA component personality described in section 4.6.3.

The following text describes the method employed to transfer the abstract WSDL definition into a concrete CORBA service. First, each request-response operation and one-way operation is manually defined in a CORBA IDL interface. This IDL is then implemented as a remote CORBA object. However, the service may also contain notification operations (as seen in the chat WSDL); in this case, the CORBA service is implemented to directly send CORBA one-way requests to the client with which it is currently interacting.

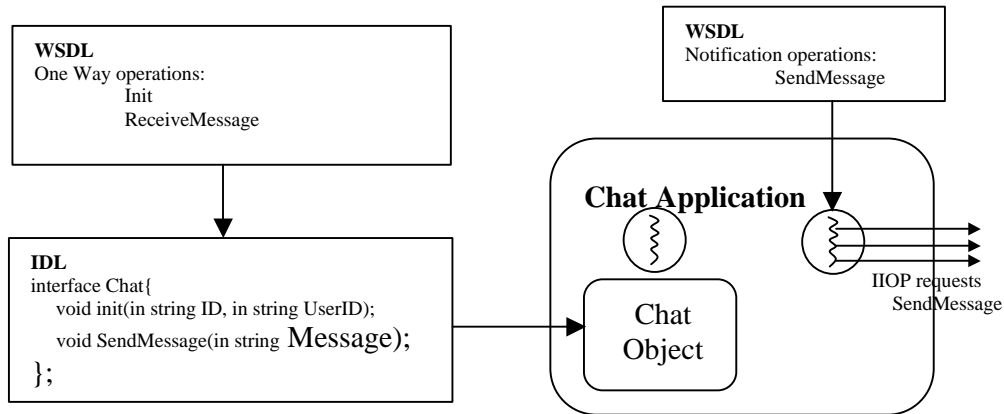


Figure 6.2 **Implementation of the CORBA chat application**

To exemplify this process, the implementation of the chat application service is illustrated in figure 6.2. The WSDL description of the chat service contains two one-way operations (`Init`, `ReceiveMessage`) and one notification operation (`SendMessage`). The ReMMoC client initiates a chat session with the service by calling `Init`; the IOR of the client (which provides a reference for the chat service to direct responses to) and the user's name are passed as string parameters. The `ReceiveMessage` one-way operation implemented by the service receives the incoming chat message from the client and displays this to the screen. These two operations are first manually defined in an IDL interface and then implemented within a single object executing within a single thread. Conversely, the WSDL `SendMessage` notification operation (that sends chat messages back to the client) is implemented within a separate thread that produces one-way CORBA requests; the user of the chat service enters chat messages, which are then passed as the string parameter of the outgoing one-way operation.

Developing SOAP Services

In the scenario, only the stock quote service in the user's home is implemented as a SOAP service. Apache SOAP version 2.2 was used to implement this application service and host it upon the fixed desktop machine. The process for implementing SOAP services follows the techniques employed for CORBA application services. The WSDL definition of the stock quote service contains a request response operation (getQuote); this information is used to manually create a Java object with a method that matches the syntax of the operation. This object is then hosted as an individual SOAP service on the Apache server using a standard deployment descriptor.

Developing Publish-Subscribe Services

The chat, jukebox and stock quote services are all implemented as publish-subscribe services in the coffee bar. The implementations of these services use the stand-alone component-based implementation of the STEAM like publish-subscribe personality described in section 4.6.3.

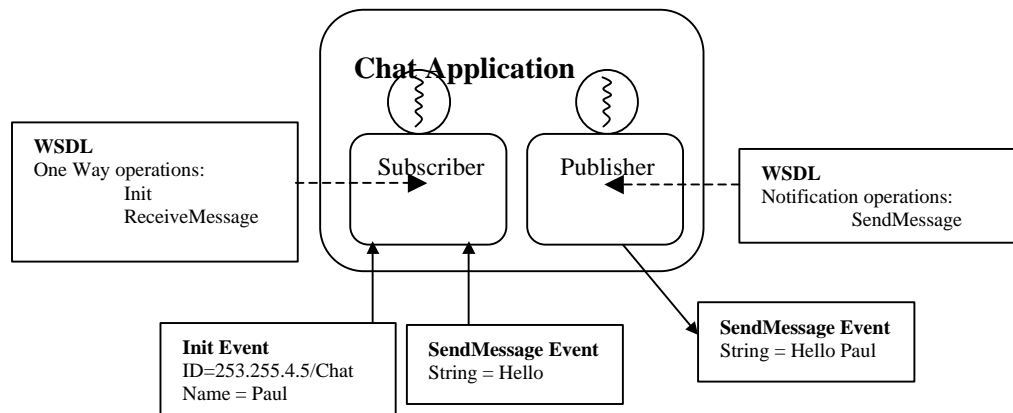


Figure 6.3 Implementation of chat application using publish-subscribe

The following techniques were employed to create the publish-subscribe application services. For request-response operations, a publisher is created to output events that will be matched by subscribers based upon the WSDL information. For example, events with the subject GetQuote and content of the type: {tickerSymbol = IBM, value=4.35} are produced for the stock quote publisher. The one-way operations e.g. PlaySong and ReceiveMessage are implemented by creating a subscriber to receive all messages of matching subject (e.g. subject=ReceiveMessage); the event content is extracted to complete the operation behaviour, e.g. the string sent in the

ReceiveMessage event is displayed to the screen. For notification operations, e.g. SendMessage, the service must publish events to subscribers based upon the WSDL syntax. Figure 6.3 illustrates how these techniques are used in the complete implementation of the chat application service.

Advertising Services

The application services in the scenario are advertised using either UPnP or SLP. SLP advertisement was performed using the OpenSLP toolkit version 1.0.11. A single service agent is initiated on the fixed desktop machine and services were registered using either the agent's command line input or through sending SLP advertisement messages. UPnP advertisement was performed using the Siemens AG UPnP C++ protocol stack. For each service (e.g. stock quote), a UPnP application was developed and an XML device and service descriptors were registered. Table 6.1 illustrates the URLs and service types used to advertise each of the three application services.

Application	Protocol	URL	Attrs	Service Type
CORBA Chat	SLP	Service:ChatService:iiop://148.88.155.209	IOR	ChatService
CORBA Stock quote	SLP	Service:StockService:iiop://148.88.155.209	IOR	StockService
CORBA Jukebox	SLP	Service:MusicService:iiop://148.88.155.209	IOR	MusicService
P/S Chat	SLP	Service:ChatService:steam://255.253.15.8		ChatService
P/S Jukebox	SLP	Service:MusicService:steam://255.253.15.8		MusicService
P/S Stock quote	SLP	Service:StockService:steam://255.253.15.8		StockService
SOAP Stock quote	UPnP	Service:StockService:http://148.88.155.209		StockService

Table 6.1 Discovery protocol advertisements of application services

Developing The Stock Quote Client Application

The stock quote application was developed as a C++ Pocket PC 2002 application. The application provides two types of functionality to the user: 1) the ticker symbol of any stock can be entered to retrieve the current price, and 2) the user can add shares to a portfolio and view the overall value of this. The interface for these interactions is illustrated in the screen shots in figure 6.4.

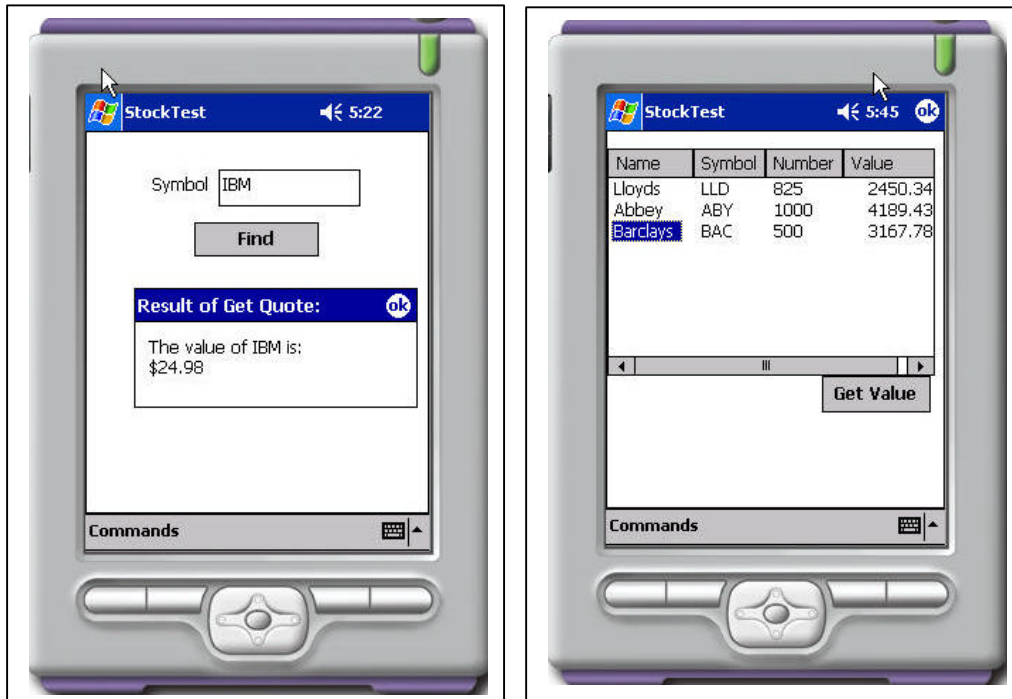


Figure 6.4 Screen shots from the stock quote client application

```

void ResultCallback(WSDLOperation *su){
    TCHAR szError[40];
    VARIANT var, var2;
    ...
    pReMMoC_ICF->GetMessageValue(&WServ, (unsigned char*) "getQuote",
        (unsigned char*)"tickerSymbol", ReMMoC_STRING, RequestResponse, &var);
    ...
    pReMMoC_ICF->GetMessageValue(&WServ, (unsigned char*) "getQuote",
        (unsigned char*)"price", ReMMoC_LONG, RequestResponse, &var2);
    ...
    wsprintf(szError, _T("The value of %s is: %4.2f"), var.bstrVal, var2.floatvalue);
    MessageBox (r_hDlg, szError, TEXT("Result: "), MB_OK);
}
...
FILE *stream = fopen("StockQuote.wsdl", "r+t" );
int numread = fread( xml, sizeof( char), MAX_FILE_SIZE, stream );
...
pReMMoC_ICF->WSDLGet(&WServ, (unsigned char*) xml);
...
pReMMoC_ICF->AddMessageValue(&WServ, (unsigned char*) "getQuote", (unsigned
char*)"tickerSymbol", ReMMoC_STRING, RequestResponse, var );
HRESULT hr = pReMMoC_ICF->OperationCall(WServ, (unsigned char*) "getQuote", 1,
&ResultCallback);

```

Figure 6.5. Code extracts from the stock quote application

To implement these operations the application must invoke the *getQuote* operation of a found service of type *StockService*. The application does not require continued interaction with a specific concrete service implementation, hence the *OperationCall* method of the ReMMoC API can be utilised. Figure 6.5 shows the ReMMoC specific

code to invoke the remote operation. First, the application obtains a local reference to the WSDL interface using *WSDLGet* method. A tickerSymbol value e.g. “IBM” is added to the input message element for the *getQuote* operation and finally the *OperationCall* method is invoked passing the interface, operation name, and finally the event handler for the result (*ResultCallback*). The handler receives the returned information from the individual abstract operation and simply extracts the required abstract elements, in this case the string ticker and float price and displays them to the screen.

Developing the Jukebox Client Application

The jukebox client application allows the user to first list songs available on a nearby music player service, displaying information such as title and artist for each song. The user can then select one of these songs to begin playing on the remote audio output. In addition, the user can also select to stop a currently playing song at any time. The user interface for these operations is illustrated in the screen shot in figure 6.6.



Figure 6.6 Screen shot from the jukebox client application

The implementation of this application differs from the stock quote client. The application requires that a list of songs from a nearby music service be downloaded. However when playing the chosen song the remote invocation must be directed to the

same service that the songs were listed from. Therefore, *OperationCall* is inappropriate; rather *ServiceLookup* followed by a *KnownOperationCall* is used to implement this behaviour. This is described in the code in figure 6.7. The application first performs a service lookup of type *MusicService*. The event handler for lookup (*LookupCallback*) then stores the service reference before invoking the *getNumberOfSongs* abstract operation. The event handler for this operation (*GetNumberCallback*) then finds the song details for each song using *getSongDetails* operations. Finally, *KnownOperationCall* is used to play the song, having added the song ID to the input message element. Note, *PlaySong* and *StopSong* are abstract one-way operations and hence do not need an event handler.

```

boolean LookupCallback(char* ServiceType, ServiceReturnEvent evt, WSDLOperation SU){
    m_evt=evt;
    ...
    // Get the number of Songs on Jukebox
    HRESULT hr = pReMMoC_ICF->KnownOperationCall(m_evt, WServ, (unsigned char*)
        "getNumberOfSongs", 1, &GetNumberCallback);
    ...
    return true;
}
void GetNumberCallback(WSDLOperation *su){

    long value;
    value = su->Output.Body[0].Param.tagged_union.longvalue;

    VARIANT var;
    for (int i = 0; i<value; i++){
        var.IVal = i;
        pReMMoC_ICF->AddMessageValue(&WServ, (unsigned char*) "getSongDetails",
            (unsigned char*)"index", ReMMoC_LONG, RequestResponse, var );
        pReMMoC_ICF->KnownOperationCall(m_evt, WServ, (unsigned char*)
            "getSongDetails", 1, &GetSongCallback);
    }
}
...
pReMMoC_ICF->ServiceLookup(WServ.ServiceType, &LookupCallback);
...
pReMMoC_ICF->AddMessageValue(&WServ, (unsigned char*) "PlaySong", (unsigned char*)"index",
    ReMMoC_LONG, OneWay, (unsigned char*)"input", var );
pReMMoC_ICF->KnownOperationCall(m_evt, WServ, (unsigned char*) "PlaySong", 1, NULL);
...
pReMMoC_ICF->AddMessageValue(&WServ, (unsigned char*) "StopSong", (unsigned char*)"index",
    ReMMoC_LONG, OneWay, (unsigned char*)"input", var );
pReMMoC_ICF->KnownOperationCall(m_evt, WServ, (unsigned char*) "StopSong", 1, NULL);

```

Figure 6.7 Code fragments of the jukebox client application

Developing the Chat Client Application

In the chat client, the user first searches for other chat users who are advertising that they are willing to chat. Once, a user is selected this initiates the chat session with this user. The user then inputs messages, which are sent and displayed on the remote

user's device; conversely all incoming messages are displayed to the chat screen. This behaviour is illustrated in the screen shots shown in figure 6.8.

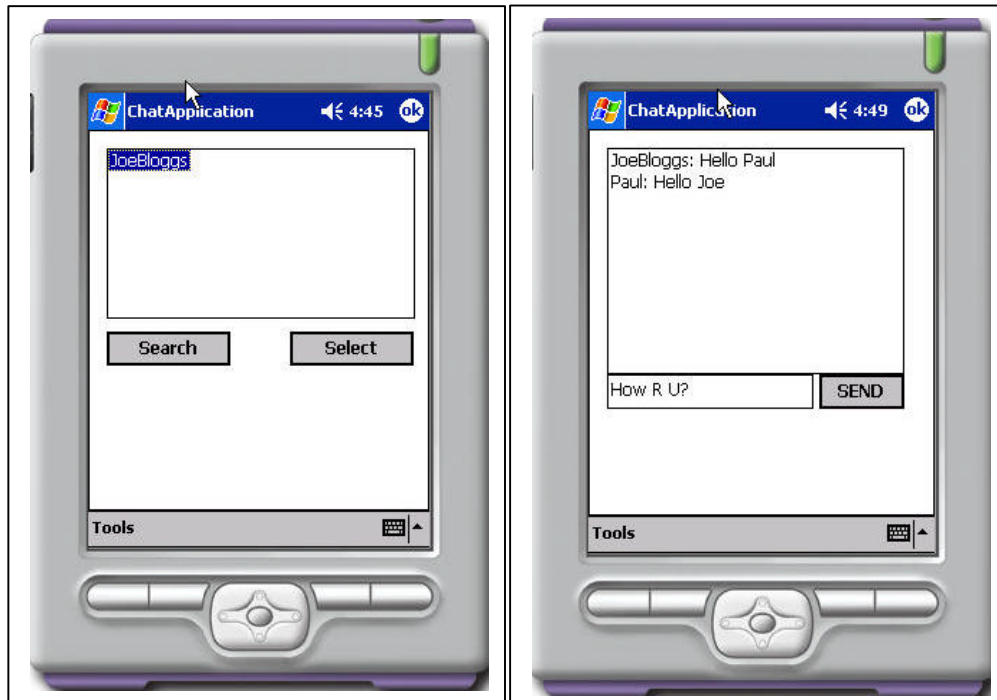


Figure 6.8 Screen shots from chat client application

```

void SendMessage(WSDLOperation *su){
    ...
    pReMMoC_ICF->GetMessageValue(&WServ, (unsigned char*) "getQuote",
        (unsigned char*)"Message", ReMMoC_STRING, Notification, &var);
    ...
    // Display Message to User interface
}

DWORD WINAPI ReceiveThread(LPVOID IParam){
    HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
    if(!SUCCEEDED(hr)) return 0;

    pReMMoC_ICF->Receive();
    return 0;
}

...
pReMMoC_ICF->CreateOperation(ChatList[g_uiLookupItemIndex].evt, WServ, (unsigned char*)
"SendMessage", -1, &SendMessage);
HANDLE hThread = CreateThread(NULL, 0, ReceiveThread, &dwParam, 0, &dwThreadId);

```

Figure 6.9 Code fragments of the chat client application

The implementation of the chat client is similar to the jukebox client; the application first discovers an available chat service, but once a concrete service is selected all subsequent operations are directed to that specific service. Hence, the one-way operations (*Init* and *ReceiveMessage*) are implemented in similar fashion to the code

in figure 6.7. The chat application is notable in that it implements code to respond to a notification operation (*SendMessage*). For this purpose, the user creates an abstract method (*SendMessage*) that will be invoked every time an incoming notification is received; the outline of this method is shown in figure 6.9. The parameter *WSDLOperation* contains the incoming chat message element, which can then be extracted and displayed to screen. The *CreateOperation* method is used to make the abstract method available to the current service the client is interacting with. Hence, this will ensure it is hosted over the same binding type. A single thread is then started to manage the incoming requests (the main thread of the application can then continue sending messages); this thread simply initiates ReMMoC to begin receiving input over the current binding, achieved through the *Receive* operation.

Analysis of the ReMMoC development process

From the experience of developing applications and services within the scenario the following points about the ReMMoC approach to application development can be drawn:

- The ReMMoC API provides a simple environment to create applications to interact with services defined by a WSDL description. To perform an abstract operation, between two and four API methods are required. Hence, the code for distribution and overcoming heterogeneity does not significantly detract from the application logic.
- The ReMMoC API provides two beneficial styles of operation: 1) the traditional method of first looking for a service and then invoking operations on the returned identifier, and 2) an invoke operation that finds any matching service before calling the method. The first method allows interaction with a specific service to be maintained. Furthermore, the second method allows applications to be developed that will continue operating when the user changes location.
- The analysis of the code to develop mobile client applications using the ReMMoC API shows that no concrete middleware implementation information is visible to the application developer. Only, abstract WSDL operations are utilised, hence the appropriate level of middleware transparency is achieved.

Furthermore, the development of the three applications demonstrated current weaknesses in the ReMMoC approach:

- The Solicit-Response operation is not utilised within the applications. This is because the abstraction to provide this behaviour is not fully implemented for publish-subscribe. Solicit-Response for publish-subscribe requires events to be generated for all possible input parameters (e.g. price events for all possible stock elements). At present the method created by the application developer for solicit-response only maps to RMI operations; future implementation could involve code analysis of the application method to generate events, or to change the implementation style of the abstract method.
- Solicit-Response & Request-Response operations are only effective across publish-subscribe when implemented sensibly. Generally, information retrieval functionality is best, for example get a stock quote or the latest news headline. This is because a finite amount of events can be published to match the service behaviour. However, computation based service methods should only be implemented using RMI i.e. the method add(x, y) cannot be implemented by a publish-subscribe service. Therefore, application service developers must sensibly choose the most appropriate middleware binding to implement their service.
- At present implementing each WSDL interface (e.g. the stock service, chat service etc.) upon a particular middleware (CORBA, SOAP & publish-subscribe) is a complex task that requires a full understanding of the behaviour of WSDL and the implementation details of each middleware. Hence, a better solution would automate stages of this process i.e. take the WSDL service interface and produce template code to be completed by the developer, or produce a WSDL document based upon an existing service.

6.2.4 Results of ReMMoC's Operation within Case Studies

Overview

This section investigates the operation and behaviour of both the ReMMoC framework and the mobile applications described in the scenario. For this purpose, three case studies are presented, which outline possible user movements and application service interactions. At each stage of the sequence of application interactions the current state of ReMMoC, in terms of current component configurations, is analysed. This process evaluates whether or not ReMMoC performs the appropriate dynamic reconfigurations in changing context. Furthermore, the

demonstration of mobile applications (built upon ReMMoC) whose operations meet the requirements of each individual interaction case study will illustrate success in achieving the main aim of this thesis i.e. that middleware heterogeneity has been fully addressed.

Dynamic Interaction Case Studies within the Scenario

The following text describes three case studies based upon mobile users changing location while using the three applications presented in the main scenario.

- **Case Study One.** The mobile user is at home and uses the stock quote client application on their Pocket PC device to retrieve the latest value of their portfolio. Later the user moves to their office, and again checks the share prices from the same client application. Finally, they move to the coffee bar and when a friend wishes to know a latest share price the user again uses their application. To perform the operations of this interaction the application must perform identically in all three scenarios, the user is unaware of the changing middleware implementation.
- **Case Study Two.** At work the user wishes to arrange a meeting with a colleague. The chat client application is used to search for and then chat with this staff member. Later in the day, the user is sitting alone in the coffee bar so they search for other nearby, like-minded individuals to communicate with.
- **Case Study Three.** The user is in the coffee bar and wishes to play a song on the jukebox, therefore the users opens their music player application on their mobile device and selects a song from the list available. Similarly, at home the user uses the same application to play songs on their own music player.

Results of Interaction in Case Study One

The sequence of operations for the Stock Quote interaction case study is described in figure 6.10; the application is first opened in the home location, therefore ReMMoC *Startup* is initiated. This forces the discovery framework to configure itself. A UPnP device and SLP agent respond to protocol discovery, therefore SLP and UPnP components are configured. The application then invokes an *OperationCall* method to find the price of IBM. This forces ReMMoC to perform lookup for a StockService over the two protocols, however only UPnP responds. The identified binding type is SOAP, therefore the binding framework is configured appropriately. The request

response operation is carried out as a SOAP method call and the resulting price is returned. The user then moves to their office and again invokes the same operation to find the price of BT (the application is not shutdown and re-started); then in the coffee shop they request the price of BA. Figure 6.10 shows how the ReMMoC middleware changes and behaves correctly in each case.

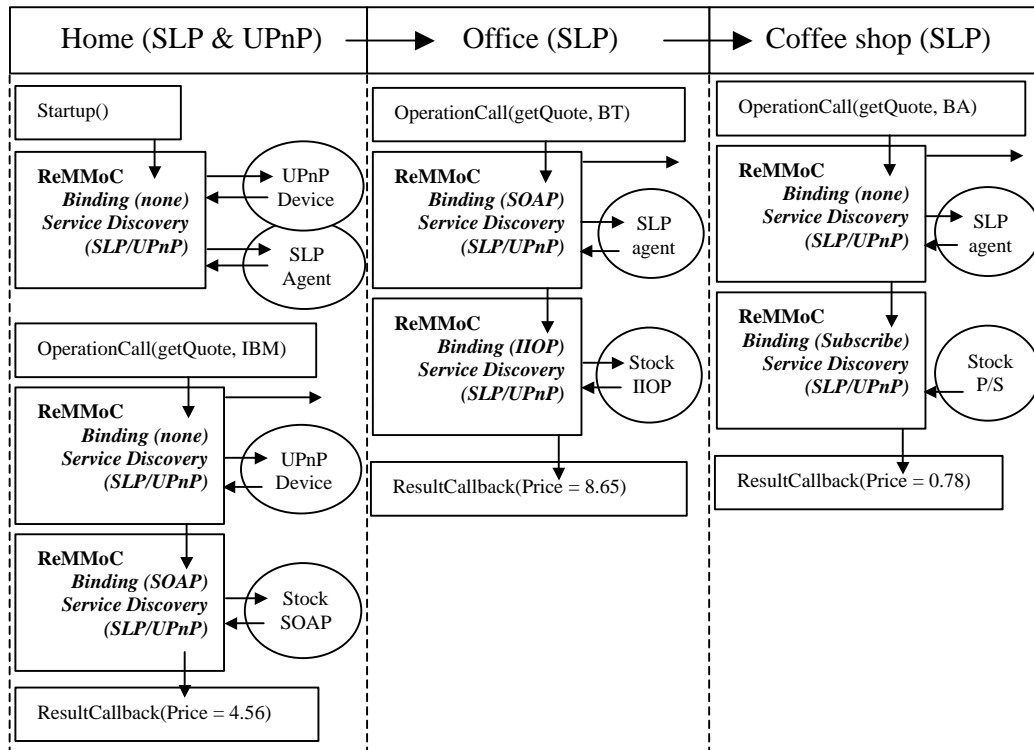


Figure 6.10 Illustration of stock application behaviour across changing locations

Results of Interaction in Case Study Two

The sequence of operations for the jukebox interaction case study is described in figure 6.11; the application is initiated in the coffee shop, where only SLP is found in use. The discovery framework is then configured as a single personality. The jukebox application first performs *ServiceLookup* for a nearby jukebox; the information returned configures the binding framework to a publish-subscribe personality. A *KnownOperationCall* is made to request the *SongDetails* of a song; ReMMoC correctly subscribes to receive the matching event. Another *KnownOperationCall* to play a song forces a *PlaySong* event to be published; on reception the jukebox begins audio output. Notably, when the user moves to their home location, the *DiscoverDiscovery* component detects that UPnP is in use in this environment, and therefore UPnP is configured into the discovery framework personality. When the

application performs service lookup the CORBA implementation is found using SLP. The binding framework is appropriately configured and the subsequent KnownOperation calls are invoked as IIOP requests.

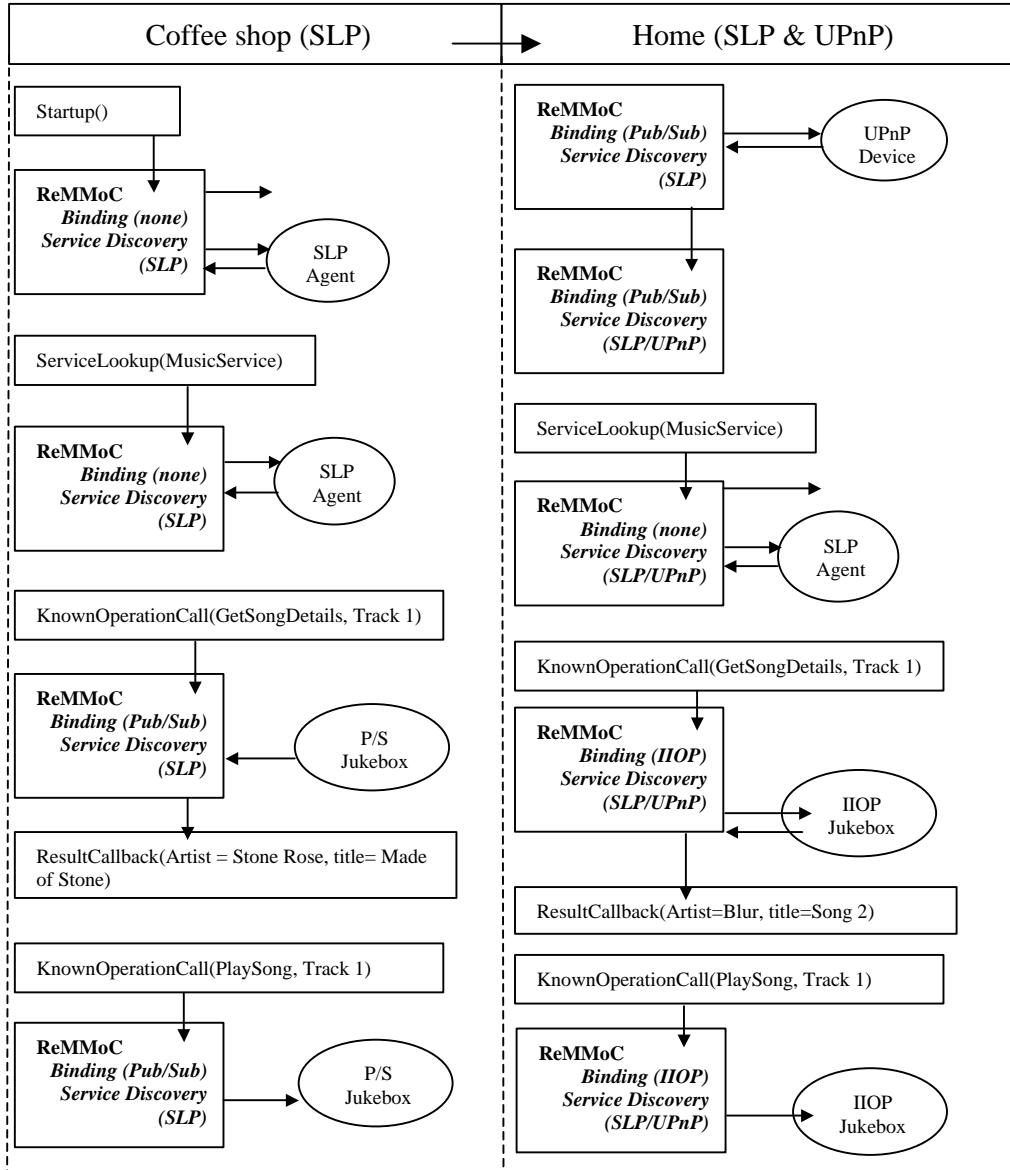


Figure 6.11 Illustration of the jukebox application behaviour across changing locations

Results of Interaction in Case Study Three

The behaviour of ReMMoC during the operation of the chat application is illustrated in figure 6.12. This case study illustrates the behaviour of ReMMoC when using the *CreateOperation* method over two contrasting middleware. In the Office location, the chat session is initialised and then the *CreateOperation* is called to allow the

application to receive notifications of the abstract SendMessage operation. The chat service is implemented as an IIOP application therefore, ReMMoC configures the service side personality to IIOPServer (alongside the current IIOP client personality). The following call of *Receive()* initiates the binding so that it can begin to receive and respond to incoming requests. Finally, when the chat between the two parties is ended the *EndReceive()* method is invoked. This clears the hosted service side personality to ensure that the application will operate in the next location.

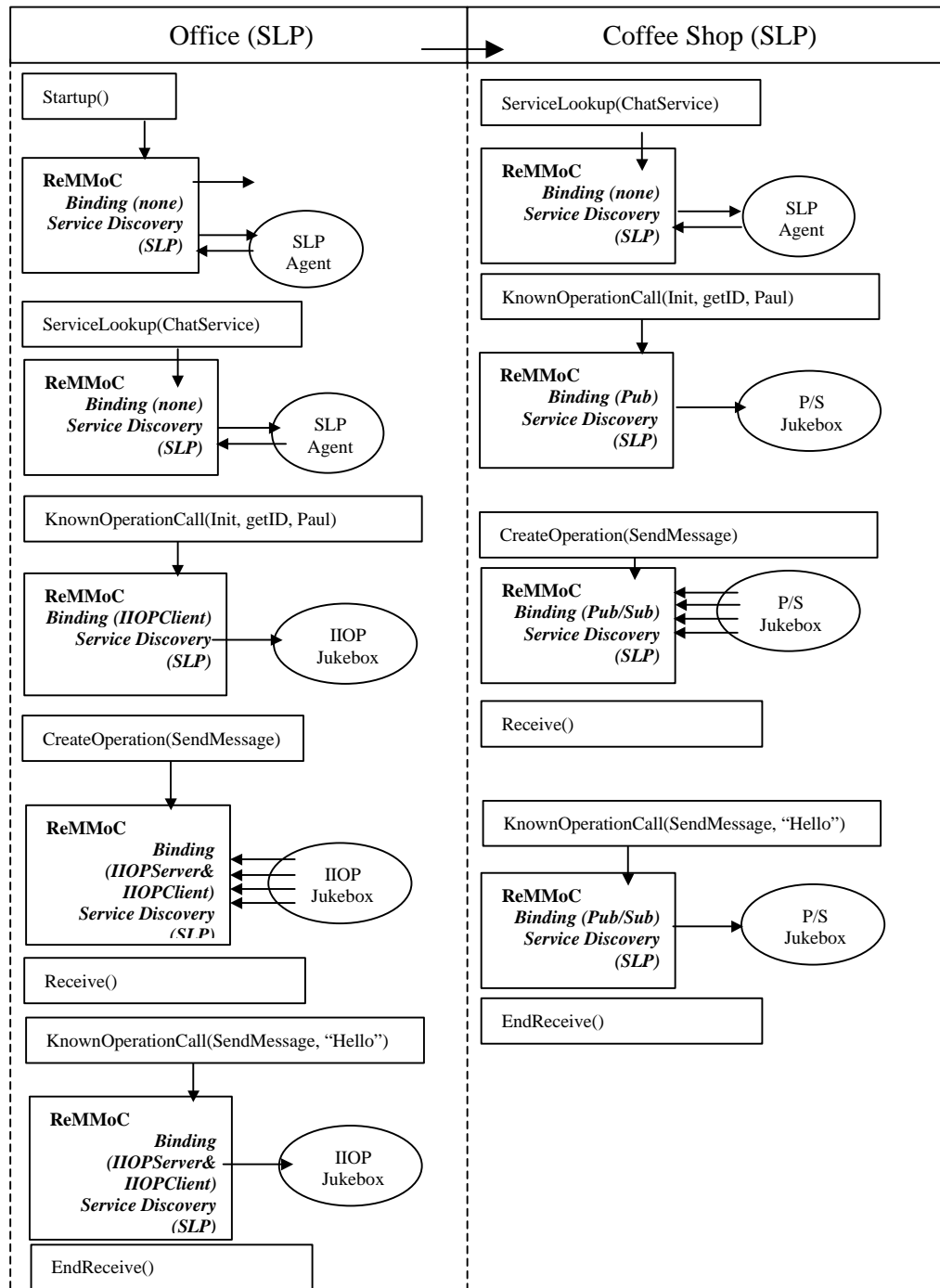


Figure 6.12. **Illustration of the chat application behaviour across changing locations**

When the user moves to the coffee bar the lookup request returns a chat application of type publish-subscribe. In this case, *CreateOperation* forces ReMMoC to configure the Subscribe personality as the fixed service side personality. The *Receive()* method call then allows the incoming subscribed messages to be passed to the same abstract method.

6.2.5 Analysis of Qualitative Evaluation

The qualitative evaluation of the ReMMoC platform in the chosen case studies has shown the following benefits of the ReMMoC approach:

- The ReMMoC platform correctly discovers service discovery mechanisms within the environment. The case studies show that when the service discovery framework is initiated in different locations it adapts itself to the current environmental context.
- The application developer can perform generic service lookup for a required service. Hence, the same service type advertised using two different protocols in separate locations is found by the application. The application is unaware of discovery protocol implementation.
- The binding framework follows the rules for configuration in a number of situations presented within the scenario. Using information from the service discovery results, the correct type of binding is initialised e.g. IIOP, SOAP or publish-subscribe. In addition, the more specific requirements driven by the abstract operation type perform the correct adaptation e.g. request response over IIOP equals IIOP client and over publish-subscribe equals the subscribe personality.
- The three case studies demonstrate that ReMMoC can be used by a number of styles of application. Information retrieval applications, e.g. News, Weather, Tourist information can be implemented using the demonstrated techniques. Similarly, the evaluation illustrates that remote device control and mobile communication applications can be easily implemented using ReMMoC.
- ReMMoC cannot currently handle migration during operation invocation. A location change will cause the “OperationCall” to fail. Therefore, a potential

improvement to ReMMoC would detect location change, and perform reconfiguration (i.e. a restart of the discovery and interaction phase based upon current environmental conditions) to allow currently requested operations to complete.

6.3 Quantitative Evaluation

6.3.1 Overview

The approach of the quantitative evaluation is to demonstrate both the performance measures and the overhead costs of the ReMMoC framework. These seek to illustrate that the core operations of ReMMoC (i.e. service calls) have a small performance overhead (incurred as the cost for overcoming heterogeneity) compared to similar operations within mobile middleware platforms. However, measurements of service discovery mechanisms (i.e. abstract Service Lookup) are not included. This is because meaningful comparisons cannot be made with concrete service discovery implementations. ReMMoC performs service lookup for a period of time fixed by the application e.g. search for 2 seconds. Once this time period has expired the matching service is returned to the application. Hence, the abstraction overhead is bound into this fixed time period.

In addition, these benchmarks also evaluate the cost of using reflection on mobile devices. The flexibility provided by reflection is extremely valuable (the qualitative evaluation has demonstrated that it provides the necessary level of dynamic behaviour, openness and extensibility required to tackle heterogeneous environments), however it comes at the expense of both performance and resource costs. First, the typical coarse-grained reflective operations (personality configuration) that are composed of meta-architecture and meta-interface operations are analysed. The analysis of fine-grained (individual) reflective operations is outside the scope of this evaluation. However, an analysis of the performance of reflective, OpenCOM based middleware platforms can be found in [Coulson04]. Secondly, the additional resources utilised (in this case system memory) is investigated.

All tests within this evaluation were executed on the following equipment setup: a stand-alone Compaq iPaq Pocket PC device (with a 206MHz StrongARM processor and 64 Mbytes of system memory) running the Windows CE 3.0 Operating system, and a Desktop PC (Windows 2000) with 128Mbytes RAM and 750MHz processor. The devices were connected via an IEEE 802.11b wireless network at 11 Mbytes/s.

6.3.2 Abstract Operation Overhead in ReMMoC

Overview

This section investigates the cost of invoking abstract services in ReMMoC. That is, what are the extra-incurred performance costs for overcoming heterogeneity? In this case, three benchmark tests are executed that analyse how ReMMoC's operation compares to similar operations in concrete middleware platforms. The first experiment identifies the overall percentage overhead of abstract service calls. The second experiment then investigates this deeper, examining the overhead of mapping operations in the overall system call overhead. Finally, the third experiment investigates what impact dynamic reconfiguration has on the significance of the performance overhead.

Benchmark Test One: Abstract versus Concrete Operation Invocations

This experiment demonstrates the overhead incurred when invoking abstract service operations (in this case KnownOperationCall methods are used). For this purpose, two operations were implemented upon both a SOAP and an IIOP service: an empty NULL method (that performs no operation and takes no parameters) and a getQuote operation that retrieves stock data from a remote web site. The empty method was invoked 100 times (using four different component setups) from a mobile client connected via the wireless network. From this measure, the operations invoked per second was calculated. The four set-ups were: 1) a concrete IIOP client implementation, 2) a concrete SOAP client implementation, 3) the ReMMoC platform configured when the IIOP service has been found, and 4) the ReMMoC platform when the SOAP service has been found. The underlying middleware for SOAP and IIOP is identical in the ReMMoC and non-ReMMoC set up, therefore ReMMoC's overhead can be evaluated. The same experiment was then repeated for the getQuote remote method.

The incurred overhead documented in figure 6.13 is composed of two factors:

- The time required to initially reconfigure the binding framework to the correct personalty
- The time to map the abstract operations onto the concrete invocations.

The results of the four tests are illustrated in figure 6.13. The NULL method results demonstrate the maximum percentage overhead of the ReMMoC platform (i.e. in addition to the cost of performing invocation across the network). These results show that for NULL IOP operations there is a 54% decrease in invocation per second throughput for abstract calls compared to concrete calls. Similarly for SOAP, there is an 11% throughput decrease for NULL operations. The SOAP decrease is less because SOAP invocations are more expensive than IOP invocations; therefore the overhead of the reconfiguration time has less of an impact.

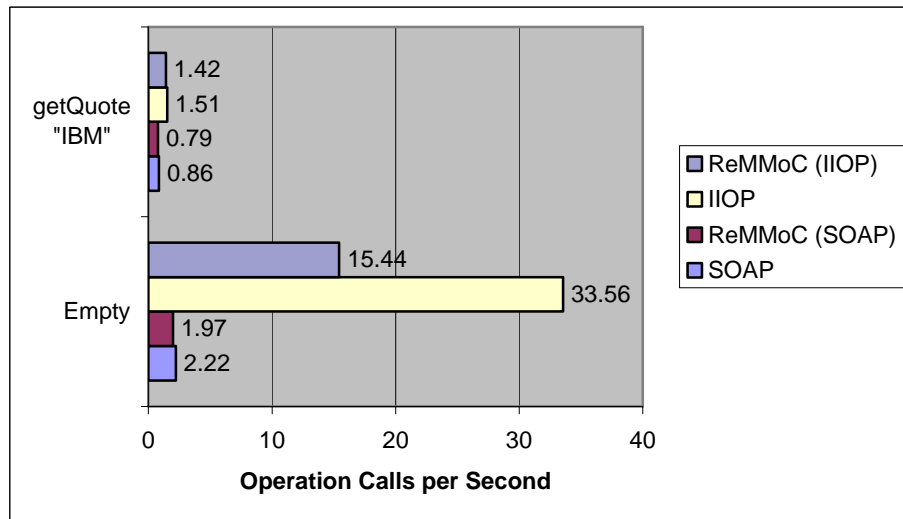


Figure 6.13 Comparison of service invocations

The results for GetQuote IIOp operations demonstrate that there is a 6% decrease in invocations per second throughput for abstract operations compared to concrete. Similarly for SOAP there is an 8% decrease. This illustrates that the impact of the overhead is reduced when realistic application operations are executed. Hence, the initial cost of reconfiguration becomes less of a factor for operations whose logic takes longer to perform, i.e. there is only a small decrease in invocation throughput.

However, there remains a small, fixed, in-band overhead on each operation call due to the abstract-to-concrete mapping; this is investigated further in the next experiment.

Benchmark Test Two: Investigating Abstract-to-Concrete Mapping

The previous test demonstrated the overhead of ReMMoC for a fixed number of method invocations. This experiment investigates the in-band overhead of mapping abstract operations to concrete invocations during ReMMoC's operation. For this purpose, the same four tests used in the last benchmark test (using NULL and GetQuote operations on IIOP and SOAP services) were carried out. However, in this case the initial reconfiguration is not measured, only the time for 100 invocations; from this the invocations per second value was calculated.

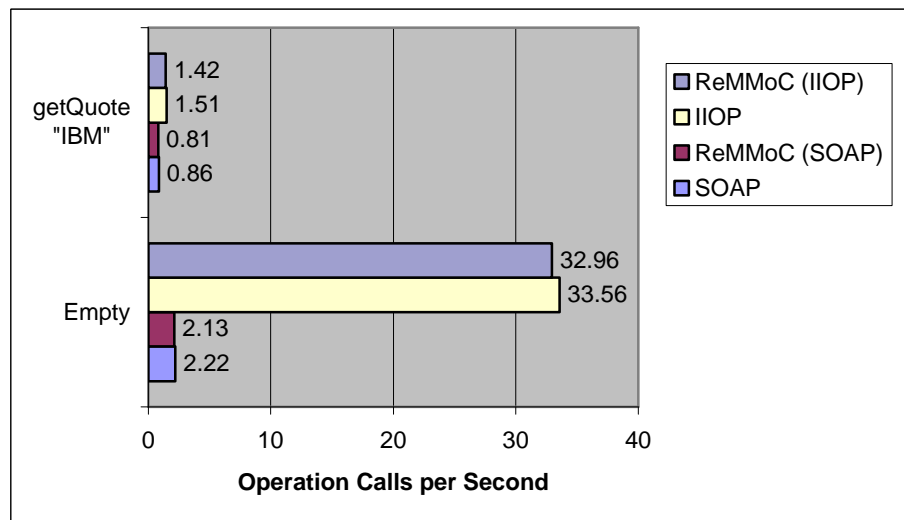


Figure 6.14 Abstract-to-concrete mapping costs during service invocation

The results in figure 6.14 show that as expected for NULL operations, there is only a small overhead for abstract invocations. For IIOP there is a 2% decrease in throughput, and a 2% decrease for SOAP. This is because there is no abstract data to map, and the overhead is simply the extra indirection due to ReMMoC's component architecture. Conversely, the getQuote operation requires a mapping of one input and one output parameter. Hence, there is an additional in-band overhead. For IIOP there is a 5% decrease in throughput (an additional 3% to the NULL measure) and 7% for SOAP. Therefore, an extra mapping overhead is attached to each invocation, and this

is dependent on the complexity of the operation call, i.e. an operation with more parameters will take longer to map.

Benchmark Test Three: Impact of Dynamic Reconfiguration

The final test of ReMMoC's overhead investigated the impact of dynamic reconfiguration. That is, how does frequent reconfiguration affect service invocation? For this purpose, the binding framework was used to invoke 1000 operations of both SOAP and IIOP methods, repeatedly switching between the two with varying levels of frequency. In this experiment only the binding framework of ReMMoC was utilised, this allowed the abstraction overhead to be minimised. In addition the IIOP and SOAP services were hosted on the same Pocket PC as the binding framework to remove the network communication overhead.

Test Description	Time (milliseconds)	Calls/Second	% Time increase from test 1
1. 500 SOAP invocations + 500 IIOP invocations	55505	18	0
2. 500 SOAP then 500 IIOP	64543	15.49	16.3
3. 250 SOAP then 250 IIOP (x2)	69679	14.35	20.3
4. 100 SOAP then 100 IIOP (x5)	84067	11.89	51.46
5. 50 SOAP then 50 IIOP (x10)	114476	8.74	106.2

Table 6.2 **Cost of dynamic reconfiguration**

The first test involved no reflection; this is a simulated base test (using base components, rather than the ReMMoC framework) of the time taken to perform 500 SOAP invocations and 500 IIOP invocations. Subsequent tests used reflective operations on the binding framework to switch invocation types between SOAP and IIOP; the frequency of reconfiguration was changed for each test. In test two a SOAP personality was configured and 500 invocations were performed, the framework was then dynamically reconfigured to IIOP and 500 invocations were made. Similarly, test three performed 250 SOAP invocations then 250 IIOP invocations and this was repeated once.

The results of the five tests performed are shown in table 6.2. It can be seen that as the frequency of reflective operations increases the time taken to perform 1000

invocations increases. For behaviour where reconfiguration is generally out-of-band, i.e. infrequent compared to the number of invocations, the additional overhead is less significant (a 16.3% increase in time). However, as the reconfiguration becomes more frequent, e.g. 10 reconfigurations in 1000 invocations, the overhead becomes significantly expensive (a 106% increase in time).

6.3.3 Measurements of Coarse-Grained Reflective Operations

Overview

This section describes three benchmark tests that illustrate the performance costs incurred by the key coarse-grained, reflective mechanisms that are performed during ReMMoC's principle operations. The first test examines the cost of loading components. The second test illustrates the cost of configuring middleware personalities into either the binding or service discovery framework. Finally, the third test measures the actual cost of dynamic reconfiguration, i.e. changing from one personality to another in the service discovery framework. These experiments demonstrate where the actual overheads that have been previously described occur.

Benchmark Test 1: Component Insertion

These experiments measure the time taken to instantiate a new component and then insert it into a framework. This is the single most expensive fine-grained reflective operation and consumes a large part of the overhead incurred in personality configuration. The experiments measure the time taken for the first insertion of a particular component, and then the time taken for subsequent insertions.

The results in table 6.3 demonstrate that the initial insertion takes more time. This is because each component is stored in a separate Dynamic Link Library (DLL) that must be first loaded. In windows CE, once a DLL is loaded then it remains loaded throughout the lifetime of the application; therefore, subsequent component creations take less time. Table 6.3 also illustrates that the size of the component (DLL) has no relation to the component insertion time, which remains constant.

Component Name	Size (bytes)	Initial Time (mSecs)	Subsequent Time (mSecs)
Socket	16896	66	54
TCP	14336	62	54
HTTP	18432	68	55
SLPMessage	45638	68	55
GIOP	20480	67	54
SOAP	30720	66	54

Table 6.3 Component insertion measurements

Benchmark Test 2: Configuring Middleware Personalities

The measurements in table 6.4 illustrate the time taken to configure each of the binding personalities into the binding framework. This is a measurement of the time taken from when the ReMMoC framework initiates the new configuration, until the configuration has been verified as a correct personality by the framework. The two times represent the time taken for the initial configuration, and then the time for subsequent configurations. The additional overhead is explained by the time to load new DLLs as described in the previous benchmark tests.

Personality Name	Total Initial Time (mSecs)	Total Subsequent Time (mSecs)
IIOP Client	2949	2754
SOAP Client	3876	3552
IIOP Server	2976	2733
IIOP Client and Server	6589	6291
Publish	3069	2810
Subscribe	2584	2387
Publish-Subscribe	5208	4929

Table 6.4 Binding framework configuration measurements

Table 6.5 illustrates the results of experiments breaking down the total time to configure personalities into the binding framework. This consists of the time to insert the personality into the framework (using the algorithm described in figure 4.15 to first insert the components and then connect them together based upon an XML

configuration description), and then to check that the personality is valid. It can be seen that increasing the complexity of the personality (in terms of number of components and number of connections) increases the time to first configure the personality and then verify it is valid. Connecting the components is the most expensive operation; this is because the interfaces must be searched for (using introspection operations) before the connections are dynamically made.

Personality Name	No. Components	No. Connections	Time to Insert Components (mSecs)	Time to Connect Components (mSecs)	Time to check (mSecs)
IIOP Client	5	6	628	2080	263
SOAP Client	6	6	747	2375	273
IIOP Server	5	6	640	2086	271
IIOP Client and Server	7	11	880	4962	521
Publish	6	5	841	1979	315
Subscribe	5	4	660	1578	234
Publish-Subscribe	7	7	900	3113	345

Table 6.5 Detailed binding framework configuration measurements

Table 6.6 illustrates the time taken to configure personalities into the service discovery framework. Again, these results show that the same factors as for the binding framework (e.g., number of components and connections) affect performance time. However, these results show a significant improvement in configuration time i.e. the more complex SLP & UPnP personality takes less time to configure than the simpler SOAP client. This is because of the differences in implementation of the two frameworks. The binding framework is implemented for extensibility. Each personality has an XML description that is used to build the configuration; this allows new personalities to be dynamically added to the ReMMoC framework without re-implementation. However, the discovery framework configuration is driven by the DiscoverDiscovery component that knows in advance which components to configure; therefore this process is optimised to perform the minimum reflective

operations. Adding a new discovery protocol to ReMMoC requires re-implementation of the DiscoverDiscovery component. Hence, there is a trade-off between performance and extensibility.

In addition, the time to verify each service discovery personality is illustrated in table 6.6. This measure demonstrates that a large part of the overhead incurred during configuration of the service discovery framework is for ensuring valid operation in the face of dynamic change. An unsafe version of ReMMoC (i.e. where there is no architectural checking of the component framework graph against XML descriptions in the face of reconfiguration) would perform significantly better; for example, an optimised, unsafe configuration of SLP takes only 1.06 seconds, compared to 3.87 seconds for the XML-based, safe SOAP client personality configuration.

Personality	No. Comps.	No. Conns.	Time to Configure (mSecs)	Time to Check (mSecs)	Total Time to Configure (mSecs)
SLP	4	9	1066	563	1629
UPnP	5	8	1070	432	1502
SLP & UPnP	8	17	1956	997	2953

Table 6.6 Service Discovery framework configuration measurements

Benchmark Test 3: Dynamic Reconfiguration

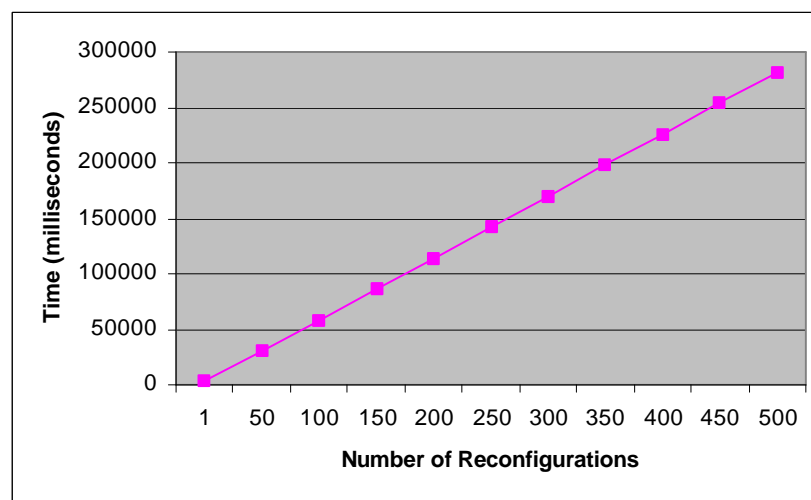


Figure 6.15 Performance of dynamic reconfigurations in discovery framework

This experiment investigated the cost of dynamic reconfiguration. In particular, the service discovery framework was repeatedly reconfigured between SLP and UPnP. A reconfiguration is defined as a change from one personality e.g. SLP to the other i.e. UPnP. The graph in figure 6.15 shows that the time to reconfigure is consistent; that is, the time to reconfigure remains the same irrespective of the number of reconfigurations. This is an area of potential optimisation. When the personality has been configured before, this can be remembered and simply swapped back in. Hence, initial configurations are more expensive than reconfigurations (see [Coulson04] for an example of this process).

6.3.4 System Memory Costs incurred when using Reflection

At present mobile devices have a limited amount of system memory, which can quickly be consumed by user's applications; therefore it is important to minimise the amount of memory needed to store a middleware implementation. This section examines the resource costs (in terms of system memory) in building the reflective middleware framework.

Table 6.7 documents the static memory footprint sizes of the separate parts of the platform i.e. configurations for the binding and service discovery frameworks (e.g. IIOP client, SOAP client etc.), and the base elements of ReMMoC. Two measurements are taken for each architecture personality: the ARM reflective and ARM non-reflective memory footprint size. The non-reflective personality is the basic component implementation, whereas a reflective personality maintains meta-information about the structure of each component and supports the subsequent introspection of this data. These reflective personalities can be used within ReMMoC, the non-reflective counterparts cannot. Non-reflective base elements of ReMMoC are meaningless (they cannot be used alone), and hence these are not measured.

Table 6.7 illustrates the cost in terms of extra memory requirements of the reflective personalities as opposed to their non-reflective counterparts. For the implemented configurations this ranges between a 41.7% and 71.8% increase in the amount of extra memory consumed by the reflective version of the personality. The storage of type

libraries and an additional 20 lines of C++ code for each component in the configuration accounts for the extra memory cost. This can be calculated by: *Personality Type Library Size + (Number of Components * 1.5K)*. The size of each type library is dependent on the complexity of interface descriptions used on that component; hence, the cost per component varies.

Function	Reflective	Non-Reflective	% Overhead
	ARM (Bytes) (a)	ARM (Bytes) (b)	
Base Elements of ReMMoC			
OpenCOM	28160	n/a	n/a
Binding CF	16896	n/a	n/a
Service Discovery CF	19968	n/a	n/a
ReMMoC_Abstract	37376	n/a	n/a
CORBA_Map	24064	n/a	n/a
SOAP_Map	19968	n/a	n/a
Subscribe_Map	23040	n/a	n/a
Binding Framework Personalities			
IIO Client	96768	56320	71.8
IIO Server	99840	58880	69.6
IIO Client & Server	140288	82944	69.1
SOAP client	97792	64512	51.6
Publish	76800	55645	41.7
Subscribe	85504	58368	46.5
Publish & Subscribe	105984	74752	41.8
Service Discovery Framework Personalities			
SLP Lookup	85504	53248	60.6
SLP Register	80896	48128	68.1
SLP Lookup & Register	103936	65024	59.8
UPnP Lookup	80384	56320	42.7

Table 6.7 Memory footprint sizes of component configurations in ReMMoC

The results also illustrate that the reflective configurations are suited to mobile devices, as minimum configurations of the binding framework and service discovery framework are less than 100Kbytes. For example, the reflective ARM measurements of IIO client, SOAP client, subscribe, UPnP lookup and SLP lookup are each individually less than 100Kbytes. These are comparable to related systems; for example, the non-reflective ARM IIO client implementation (55K) compares with the 29K SH3 CORBA client personality of the Universal Interoperable Core (UIC)

implementation [Roman01] and the 48K non-pluggable GIOP client Zen implementation [Klefsad03], which have similar capabilities. The difference between the ReMMoC and the UIC value can be attributed to a different processor, for example ARM implementations are larger than both x86 and SH3 implementations (because it is a RISC processor, rather than CISC) and using a COM based implementation.

6.3.5 Analysis of Quantitative Evaluation

The following points can be extracted from the quantitative evaluation of ReMMoC's performance.

- Abstract service invocation incurs a performance overhead compared to the same operation performed by a concrete middleware platform. The configuration of the binding personality (loading and connecting of components) and the mapping of abstract operations to concrete invocations cause this. A potential optimisation of ReMMoC is to pre-load configurations in advance before service invocation is requested therefore, removing a significant performance overhead.
- The significance of the service invocation overhead is reduced when realistic service operations are performed. The throughput of ReMMoC IIOP invocations per second is reduced from the maximum 54% decrease to a 6% decrease (compared to base IIOP invocations) for a realistic mobile application operation.
- Mapping abstract operations to concrete operations incurs an in-band operation overhead. For NULL operations where there is no mapping, a 2% decrease in ReMMoC invocation throughput (compared to base IIOP) is observed. This is caused by additional indirection. Mapping a single input and output parameter incurs an extra 3% decrease in throughput for ReMMoC IIOP, and an extra 5% decrease for SOAP.
- Dynamic reconfiguration adds an additional "out-of-band" overhead. Infrequent reconfiguration e.g. 1 reconfiguration during 1000 invocations suffers a 16% decrease in performance time. Frequent reconfiguration e.g. 10 reconfigurations during 1000 invocations suffers a 106% decrease in performance time. Therefore, where reconfiguration is performed infrequently it has less of an impact on overall throughput.

- Algorithms implemented to improve platform extensibility (e.g. configuring personalities in the binding framework) are significantly more expensive than optimised configuration algorithms (e.g. in the service discovery framework). The configuration of the less complex SOAP client personality takes over three times longer than the SLP personality. Hence, a trade-off between extensibility and performance can be made when implementing middleware platforms.
- Checking the validity of component frameworks adds another overhead. A trade-off can again be made between platform safety and system performance. A non-safe SLP personality can be configured in 1.066 seconds compared to 1.629 for the safe version.
- Reflective component configurations can be created that fit on devices with limited memory capacity. A multi-personality instantiation of ReMMoC (IIOP, SOAP, SLP and UPnP) can be created that is smaller than 500 Kilobytes.
- On average, utilising reflective component personalities provides between a 42% and 71% increase in memory resource usage compared to standard component implementations.
- Components will need to be transmitted across the network (for example, when the platform discovers it needs components not currently on the device). Therefore, personalities less than 100K (a typical size of a binding or discovery personality as seen in table 6.7) in size can support this behaviour.

These results generally indicate that utilising a reflective middleware framework upon mobile devices is expensive in terms of both performance time and storage costs, and that these may be detrimental to the uptake of such an approach in real world scenarios. However, closer analysis of the results suggests that the approach is not prohibitive; reconfiguration and architectural verification are the primary expenses, yet these are dependent upon environmental context and the platform only changes upon detected middleware heterogeneity. Therefore, in scenarios where there is little heterogeneity ReMMoC will perform similarly to a base middleware platform. Furthermore, ReMMoC is not designed as an optimised implementation and the results indicate possible areas to improve its performance measures. Finally, overcoming heterogeneity must come at an extra performance cost, but the author

believes this is acceptable in order to allow mobile applications to continue operating in any environment.

6.4 Summary

This chapter has provided an evaluation of the ReMMoC platform. The qualitative evaluation described in section 6.2 has shown that the ReMMoC framework achieves the main goal of this thesis. That is, mobile applications can be developed, which continue operating in locations populated by heterogeneous middleware implementation. In addition, the ReMMoC development process has shown that ReMMoC can be used for a range of application types. Furthermore, the ReMMoC API presents the appropriate level of middleware transparency and developers need not be aware of concrete middleware implementation.

The quantitative evaluation described in section 6.3 documented the comparison of ReMMoC to traditional concrete middleware implementation. The ability to overcome heterogeneity adds an additional performance time overhead. The significance of this overhead is reduced when the framework is utilised in realistic application scenarios. Furthermore, the breakdown of this performance time overhead into coarse-grained reflective operations was examined. It was seen that the safety and extensibility of the ReMMoC framework contributes to the additional overhead. Finally, the cost of reflection was investigated. Using reflection increases memory footprints between 42% and 71%, and reflective operations add to a 6% decrease in ReMMoC IIOP throughput of invocations/second and 8% decrease for SOAP, compared to base middleware behaviour.

Chapter 7 Conclusions

7.1 Introduction

This thesis has investigated the problems that middleware heterogeneity pose to the developers of the next generation of mobile applications. More specifically, the ReMMoC middleware framework has been described in detail. This platform demonstrates that reflective middleware offers a good solution for developing a higher-level (or meta) middleware to solve the problems of middleware heterogeneity. The combination of components, component frameworks and reflection supports appropriate adaptation of middleware behaviour in the domains of service binding and service discovery. In addition, ReMMoC promotes a higher-level abstraction that provides middleware transparency to mobile application developers. Web Services form the base of this abstraction, a standard the author believes will become a widely used technology for addressing middleware heterogeneity and middleware integration.

The remainder of this chapter is structured as follows. Section 7.2 provides a summary of the arguments presented within this thesis in a chapter-by-chapter fashion. Section 7.3 reviews the major results that have emerged as a result of the work carried out, and the other notable contributions are found in section 7.4. The order of these results is arbitrary and does not indicate relative importance. Finally, section 7.5 describes pointers to future work that may be carried out based upon the research presented in this thesis.

7.2 Thesis Overview

The thesis began with chapter 1, which introduced the main areas of research undertaken. The characteristics of mobile computing were first briefly introduced, including surveys of mobile applications, wireless networks, mobile devices and the well-identified challenges that mobile computing poses to middleware developers. The different styles of mobile middleware that have so far been developed to meet these challenges were then briefly described. Furthermore, a new problem of middleware heterogeneity in mobile environments was identified, specifically the need to develop mobile applications that continue operation across multiple locations

independent of middleware implementation in the environment. The remainder of this chapter then outlined the main aims of the thesis.

Chapter 2 presented an in-depth survey of middleware for mobile computing. Each particular middleware style was examined in turn: extensions to well-established middleware (e.g. CORBA), asynchronous middleware, data-sharing middleware, mobile agents, reflective middleware, policy-based adaptive middleware and finally service discovery. Each particular style was analysed for effectiveness in overcoming the challenges posed in chapter 1. It was argued that, although these solutions have been successful in solving these initial challenges, they do not offer any solutions to the problems of middleware heterogeneity, rather the range of available solutions exacerbates the problem.

Chapter 3 examined the initial solutions in the area of middleware heterogeneity. The wide-ranging solutions covered: Web Services, mobile code, platform independent modelling, bridging and adaptive middleware. It was argued that none of these solutions supports the dynamic interaction scenarios found in mobile computing applications, nor allows for the many different middleware styles and service discovery mechanisms. Hence it was argued that a suitable higher-level abstraction must be complemented by a combination of dynamic service binding and service discovery mechanisms.

The design of the ReMMoC framework was introduced in chapter 4. The combination of components, component frameworks and reflection was presented as the basis for the design. To complement this, the choice of the OpenCOM component platform was explained, and the design of a novel component framework architecture for this platform was provided. The remainder of the chapter then concentrated on the reflective operations of the ReMMoC framework. The design and implementation of the two key component frameworks: binding and service discovery were described in detail. The component implementations of individual middleware personalities were discussed together with the algorithms for dynamic adaptation. Notably, the “cycle-and-see” philosophy was presented as a solution to the problem of discovery protocol discovery.

Chapter 5 documented a higher-level abstraction for mobile middleware based upon the abstract services of the Web Services Architecture. The choice of Web Services, as opposed to other heterogeneity solutions, was explained; the deciding factor was that this standard is currently the front-runner for integrating heterogeneous middleware technologies. The design and implementation of this abstraction was then described in detail. The operations available from the event-based ReMMoC API were presented. Furthermore, a description of the mapping process was explained; each of the four abstract WSDL operation was mapped to two contrasting middleware paradigms: RMI and publish-subscribe.

Chapter 6 presented an evaluation of the ReMMoC framework. The evaluation approach combined two techniques. Firstly, the facilities for distributed mobile application development were evaluated in a qualitative manner. A case study of application services implemented upon heterogeneous middleware across multiple locations was carried out, and the benefits and flexibility of the ReMMoC adaptation framework was demonstrated. Secondly, the basic performance of the framework was measured using quantitative measures. The importance of this evaluation is that it demonstrated comparable performance to existing mobile middleware in addition to overcoming heterogeneity. Furthermore, quantitative evaluation identified that the cost of reflection does not impact on the suitability of utilising the technique of reflection on mobile devices.

7.3 Major Results

7.3.1 Identification of Middleware Heterogeneity in Mobile Computing

An important contribution of this thesis is the identification of the problem of middleware heterogeneity in the domain of mobile computing. It is now well identified that the range of middleware paradigms and implementations available to developers is causing problems across all application domains. However, this thesis takes the view that the dynamic nature of the mobile environment, e.g. the user constantly changing location, magnifies this problem; a significant range of middleware implementations will be encountered by a mobile application during its execution. In addition, many more types of middleware may be used within mobile

computing settings e.g. proprietary solutions for individual locations (e.g. smart spaces), whereas a large percentage of the middleware used in the fixed environment will be of established types (e.g. CORBA, SOAP and Java RMI). State of the art mobile middleware has generally ignored the middleware heterogeneity problem, instead focussing on the specific challenges of mobile computing. However, middleware heterogeneity is a problem that must be addressed, otherwise mobile applications will only be able to interoperate in particular locations and situations.

7.3.2 The ReMMoC Approach

The most important contribution of this thesis is the design and implementation of the ReMMoC framework. A three part architecture was designed and implemented consisting of: 1) a dynamically adaptable concrete middleware framework, 2) an abstract programming API to hide middleware heterogeneity, and 3) mappings between abstract and concrete operations. The resulting qualitative evaluation of this framework demonstrated that client applications developed upon ReMMoC would continue operating in locations populated with heterogeneous middleware implementation. Within the evaluation scenario, different styles of mobile client applications (information retrieval, device control and communication) interfaced with application services implemented upon heterogeneous middleware (e.g. CORBA, SOAP, SLP, UPnP and publish-subscribe). Hence, the ReMMoC platform is shown to tackle middleware heterogeneity in the mobile domain and meet the main aim of this thesis.

The implementation of the ReMMoC framework consists of the adaptive service binding component framework and the adaptive service discovery component framework. The required application service must be found using a discovery protocol that matches the advertising mechanism of that service. Hence, the service discovery framework dynamically changes its internal personality when new discovery protocols are in use in the environment; for this purpose, service lookup can be executed over multiple protocols to ensure the service is found. The information returned from service lookup is then used to control the service-binding framework that performs the communication with a service. Notably, the binding framework within the ReMMoC architecture adapts between multiple communication paradigms e.g. publish-subscribe

and RMI. The combination of these two adaptive frameworks is an important contribution of this thesis, as it provides a novel solution that has not previously been applied to the problem of middleware heterogeneity.

7.3.3 A Higher-level Middleware Abstraction

The final important contribution of this thesis is the higher-level middleware abstraction promoted by ReMMoC. A reflective architecture for changing middleware behaviour is not enough to solve heterogeneity. A developer would find it impossible to predict the middleware a mobile application may encounter in newly entered locations. Therefore, an abstraction above current middleware programming models is required. This thesis identifies that Web Services already offers an interesting and popular abstraction in this domain, which is becoming widely used as a method to integrate heterogeneous middleware. ReMMoC uses the base concepts of abstract services (defined in WSDL) and adds a generic service discovery abstraction to create a higher-level middleware abstraction for mobile computing middleware. Through an API providing middleware transparency, mobile applications can then be developed that will operate in unknown locations populated with unknown types of middleware.

7.4 Other Significant Results

7.4.1 The OpenCOM Component Framework Model

The design of the ReMMoC framework is based upon the concept of component frameworks; these manage the adaptation of components within particular domains of middleware functionality. The available component framework methods for OpenCOM were identified as unsuitable for the complex hierarchical architectures required by ReMMoC. Therefore, a new component framework model for OpenCOM was designed based upon the concept of composite components in OpenORB [Blair01]. Each component framework is composed of the component configuration that implements its behaviour. Dynamic inspection and alteration of the architecture of the framework is then made through an additional meta-object protocol (whose design is based upon the OpenORB reflective APIs [Blair01]). In addition, the use of locking interceptors and graph checking components provide methods to maintain integrity in the face of dynamic changes to the framework. The resulting component framework

model is generic in nature, and therefore is usable by other OpenCOM based platforms and not just ReMMoC.

7.4.2 The “Cycle and See” Philosophy

A significant problem that emerged during the course of this research was that of discovering discovery protocols. In order for the discovery framework to operate correctly, the environment must be searched for discovery mechanisms in use. For this purpose, the “Cycle and See” method was developed. This involves the framework searching for known protocols in parallel; when a matching response is returned this forces the framework to be reconfigured appropriately. This method is preferable to a higher-level discovery mechanism, as it does not require any conformance between advertised services. Hence, existing environments and discovery protocols can be used. However, when an unknown discovery protocol is encountered this method will fail. Therefore, the implementation is componentised to allow a new “cycle and see” component (containing new protocols) to be dynamically added.

7.4.3 The use of Reflection on Mobile Devices

The technique of reflection is often criticised for its poor performance and increased utilisation of system resources. Hence very few mobile middleware systems are available that use reflection, even though it is ideally suited to many of the challenges of mobile middleware [Capra02]. The quantitative evaluation results produced in this thesis demonstrate the following properties. ReMMoC does consume more system resources (memory) than a corresponding non-reflective implementation (between a 42% and 71% increase). However, the complete solution (approximately 1 Megabyte) will comfortably operate on today’s mobile devices (e.g. the Compaq iPaq H3870 has 64Mbytes of system memory). In addition, the results show that reflection adds additional overhead to the performance of a middleware. However, the impact of this overhead is lessened during the operation of realistic mobile application services. For example, there is only a 6% decrease in IIOP invocations per second in ReMMoC compared to base IIOP behaviour.

7.4.4 Abstract-to-Concrete Mappings

The final significant contribution of this thesis is the abstract to concrete mappings that form part of the solution to middleware heterogeneity. The abstract services described in WSDL can only be utilised if the abstract operations are mapped to concrete middleware messages e.g. IIOP requests and publish-subscribe events. This thesis describes in details the mappings to contrasting communication paradigms: remote method invocation and publish-subscribe. The similarities between abstract and concrete message content form the basis of these mapping e.g. message elements and RMI parameters. In addition, these event-based mappings (i.e. all mappings return results as events) ensure that a consistent flow of information is maintained to the application irrespective of the computation model of the underlying middleware.

7.5 Future Work

7.5.1 Additional Middleware Personalities

At present, only a small number of binding and service discovery protocols have been developed. Increasing the encompassed types of middleware personalities will strengthen the argument that ReMMoC fully addresses middleware heterogeneity. For service discovery, only two protocols are implemented. Initial analysis illustrates that alternative discovery protocols like Jini and Salutation have the required properties to be included within the generic service discovery architecture. Further investigation through implementation of component-based personalities for these will verify that this is the case. Furthermore, this will identify if additional discovery protocols have a significant impact on the performance of ReMMoC.

Only two middleware paradigms are implemented within the ReMMoC framework: RMI and publish-subscribe. To further investigate the general nature of the ReMMoC framework, new bindings of each style of mobile middleware should be investigated to see if it is applicable in the framework model. For example, tuple spaces, mobile agents and data-sharing communication paradigms offer diverse and interesting challenges in terms of applying them within this domain. As seen within the analysis of the publish-subscribe model not all of the WSDL operations need to be mapped to each paradigm, hence it is feasible that these bindings can be applied.

7.5.2 Security Component Framework

Security is an important issue in mobile computing. In the scenarios presented in this thesis, access to services may be restricted to authentic users. In addition, secure methods of communication will be required for services that communicate private information or for which the user pays for the service. Therefore, an interesting piece of future work is a security component framework to extend the ReMMoC framework. To maintain the ReMMoC philosophy the security framework would ideally need to operate across different security mechanisms implemented by each service. For example, a generic user authentication service may be physically implemented by a number of contrasting authentication protocols. An investigation within this realm would identify if such an approach is feasible, or if single fixed mechanisms are more appropriate.

7.5.3 Resource Management Component Framework

A challenge of mobile computing is the limited resources found upon mobile devices. At present ReMMoC does not attempt to reduce system resource consumption during operation. Hence, in the future ReMMoC could be extended by a resource management component framework, which effectively controls the resource consumption by each component. An OpenCOM resource management framework is currently available [Duran00], which looks at controlling system memory and threads. However, battery power is a more valuable resource on mobile devices, particularly as ReMMoC's operation involves frequent communication over wireless networks. Therefore, future work integrating this resource management framework to effectively improve battery lifetime would be a significant result.

7.5.4 Web Service Extensions

There is ongoing work into the definition of the Web Services Architecture. This includes the standardisation of new languages that offer more complex behaviour than WSDL. For example, the Web Services Flow Language (WSFL) [Leyman01] allows developers to describe complex interactions patterns between groups of participating services. In addition, new languages are emerging that allow the abstract service descriptions to be extended to include non-functional aspects including Quality of

Service and Security e.g. the Web Services Endpoint Language (WSEL) [Hung02]. An investigation of how the base ReMMoC abstraction must be extended to support languages of this type would be required. It is likely, that the implementation would rely upon the two previously described component frameworks (i.e. security and resource management).

7.5.5 Semantic Service Matching

ReMMoC relies upon the assumption that if the service type matches the lookup request then that service provides the functionality required. However, services of the same type and with the same syntactic definition may still behave differently. Therefore, to remove this reliance, semantic matching of services is required. The mechanisms of service discovery could be extended to base service selection upon information that semantically describes the service's behaviour. Semantic services are an emerging hot topic within the Web Services community. A number of technologies have already emerged. OWL [OWL03], the Web Ontology Language, allows explicit meaning to be attached to information allowing machines to automatically process information. Future work could examine how these technologies may provide application service selection to ensure the behaviour of the service matches the application requirements.

7.5.6 Dynamic Component Downloading

The evaluation section of this thesis describing memory costs demonstrated that single middleware personalities could be stored on mobile devices. However, each device cannot store every possible middleware component that may be needed. Over the lifetime of a mobile device, upwards of ten middleware implementations (bindings and service discovery) would exhaust the system memory of present day devices. Therefore, a method for dynamically downloading components to the device when needed is required. However, component downloading introduces additional performance overhead; therefore techniques to ensure the component is available to start-up before the application requests it are required. Predictive caching based upon context information is an interesting option for this. For example, the user is moving towards a particular location were they previously used the SLP discovery protocol

and the publish-subscribe middleware binding, therefore download all of the components for these personalities.

7.5.7 Ubiquitous Computing Environments

ReMMoC has been specifically designed for, and applied within the domain of mobile computing applications. However, the framework has the potential to be utilised in many more applications domains, including ubiquitous computing and Smart Home Environments. These are applications where the computer becomes part of the environment e.g. intelligent devices and wearable computers. A framework such as ReMMoC would then ideally support the discovery of and communication between heterogeneous elements within these scenarios. The application of ReMMoC to these domains would also provide a sterner evaluation of ReMMoC in terms of the complexity of applications that it can fully support.

7.6 Concluding Remarks

This thesis has identified that there are now many mobile middleware solutions available, each of which address one or more of the original challenges of mobile computing. However, these heterogeneous solutions create the problem of middleware heterogeneity. The author envisages the next generation of mobile applications will support the philosophy of “use anywhere” irrespective of middleware implementation i.e. a restaurant table booker will work in any city in the world. Hence, the ReMMoC framework has been presented, whose goal is to allow mobile applications to be developed independently of middleware implementation. These applications will then continue operation in new, unknown locations. ReMMoC currently relies upon the Web Services Architecture; an initiative that the author believes will become the driving technology in higher-level middleware frameworks and middleware integration. This thesis has demonstrated that the combination of Web Services with a dynamically adaptable middleware framework successfully overcomes the problem of dynamic middleware heterogeneity.

References

- [**Ankolekar01**] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara and H. Zeng, “Daml-s: Semantic markup for web services”, Proceedings of the International Semantic Web Working Symposium (SWWS), pp. 39-54, 2001.
- [**Apache03**] The Apache Software Foundation, “Web Services – Axis”, <http://ws.apache.org/axis/>, 2003.
- [**Apple94**] Apple Computer Inc., “OpenDoc: White Paper”, Apple Computer Inc., 1994.
- [**Arnold99**] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo and A. Wollrath, “The Jini Specification”, Addison Wesley, 1999.
- [**Asthana94**] A. Asthana, M. Cravatts and P. Krzyzanowski, “An Indoor Wireless System for Personalized Shopping Assistance”, Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, US, 1994.
- [**Bacon00**] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel and M. Spiteri, “Generic Support for Distributed Applications”, IEEE Computer, 33(3), pp. 68-76, March 2000.
- [**Bakker00**] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen and A. Tanenbaum, “The Globe distribution network”, Proceedings of the {USENIX} Annual Conference”, pp. 141-152, 2000.
- [**Biegel02**] G. Biegel, V. Cahill and M. Haahr, “A Dynamic Proxy-Based Architecture to Support Distributed Java Objects in Mobile Environments”, Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2002), Lecture Notes in Computer Science, volume 2519, R. Meersman and Z. Tari (Eds.), pp. 809-826, October 2002.
- [**Blair01**] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas and K. Saikoski, “The design and implementation of Open ORB 2”, IEEE Distributed Systems Online, 2(6), Sept 2001.
- [**Bluetooth99**] The Bluetooth specification, <http://www.bluetooth.com/developer/specification/specification.asp>, 1999.
- [**Bluetooth99b**] Bluetooth Specification Part E, “Service Discovery Protocol (SDP)”, <http://www.bluetooth.com/>, 1999.
- [**Booth03**] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard, “Web Services Architecture”, W3C Working Draft, <http://www.w3.org/TR/ws-arch/>, August 2003.

- [**Boulkenafed03**] M. Boulkenafed and V. Issarny, “AdHocFS: Sharing Files in WLANS”, Proceeding of the 2nd IEEE International Symposium on Network Computing and Applications, Cambridge, MA, USA, April 2003.
- [**Box00**] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte and D. Winer, “Simple Object Access Protocol (SOAP) 1.1. Technical Report”, <http://www.w3.org/TR/SOAP>, May 2000.
- [**Brewer98**] E. Brewer et al., “A Network Architecture for Heterogeneous Mobile Computing”, IEEE Personal Communications, October 1998.
- [**Bustamante00**] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener, “Efficient wire formats for high performance computing”, Proceedings of the 2000 International Conference on Supercomputing (SC2000), Dallas, Texas, USA, November 2000.
- [**Campadello00**] S. Campadello, H. Helin, O. Koskimies and K. Raatikainen, “Wireless Java RMI”, Proceedings of The 4th International Enterprise Distributed Object Computing Conference, pp. 114-123, Makuhari, Japan, September 2000
- [**Capra01**] L. Capra, W. Emmerich and C. Mascolo, “Reflective Middleware Solutions for Context-Aware Applications”. Proceedings of REFLECTION 2001- The Third International Conference on Meta-level Architectures and Separation of Crosscutting Concerns, September 2001.
- [**Capra02**] L. Capra, G. Blair, C. Mascolo, W. Emmerich and P. Grace, “Exploiting Reflection in Mobile Computing Middleware”, ACM SIGMOBILE Mobile Computing and Communications Review, 6(4), pp. 34-44, October 2002.
- [**Capra02b**] L. Capra, W. Emmerich and C. Mascolo, “A Micro-Economic Approach to Conflict Resolution in Mobile Computing”, Proceedings of the Foundations of Software Engineering (ACM SIGSOFT/FSE-10), pp. 31-40, Charleston, South Carolina, USA, November 2002.
- [**Carzaniga01**] A. Carzaniga, D. Rosenblum and A. Wolf, “Design and Evaluation of a Wide-Area Event Notification Service”, ACM Transactions on Computer Systems, 19(3), pp. 332-383, 2001.
- [**Chan01**] W. Chan, “Project Voyager: Building an Internet Presence for People, Places, and Things”, Masters Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2001.
- [**Chinnici03**] R. Chinnici, M. Gudgin, J. Moreau and S. Weerawarana, “Web Services Description Language (WSDL) Version 1.2”, W3C Working Draft, <http://www.w3.org/TR/wsd112/>, March 2003.
- [**Clarke01**] M. Clarke, G. Blair, G. Coulson and N. Parlavantzas, “An Efficient Component Model for the Construction of Adaptive Middleware”, Proceedings of Middleware 2001, Heidelberg, Germany. November, 2001.

- [**Cole03**] A.Cole, S. Duri, J. Munson, J. Murdock and D. Wood, “Adaptive Service Binding to Support Mobility”, Proceedings of the ICDCS International Workshop on Mobile Computing Middleware, Providence, RI, US, May 2003.
- [**COM95**] Microsoft Corporation, “The Component Object Model Specification, Version 0.9”, <http://www.microsoft.com/Com/resources/comdocs.asp>, October 1995.
- [**Coulouris00**] G. Coulouris, J. Dollimore, and T. Kindberg, “Distributed Systems, Concepts and Design”, Addison-Wesley (3rd Edition), 2000.
- [**Coulson04**] G. Coulson, G. Blair and P. Grace, “On the Performance of Reflective Systems Software”, Proceeding of International Workshop on Middleware Performance (MP 2004), Phoenix, Arizona, USA, April 2004 (to appear).
- [**Cugola01**] G. Cugola, E. Di Nitto and A. Fuggetta, “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS”, IEEE Transactions on Software Engineering, 9(27), pp. 827-850, September 2001.
- [**Cunnings01**] R. Cunnings, S.Fell and P. Kulchenko, “SMTP Transport Binding for SOAP 1.1”, <http://www.pocketsoap.com/specs/smtpbinding/>, November 2001.
- [**Davies98**] N. Davies, A. Friday, S. Wade and G. Blair, “L²imbo: A Distributed Systems Platform for Mobile Computing”, ACM Mobile Networks and Applications (MONET) - Special Issue on Protocols and Software Paradigms of Mobile Networks, 3(2), pp. 143-156, August 1998.
- [**Davies99**] N. Davies, K. Cheverst, K. Mitchell and A. Friday, “Caches in the Air: Disseminating Information in the Guide System”, Proceedings of the 2nd Workshop on Mobile Computing Systems and Applications (WMCSA '99), 1999.
- [**Davis02**] D. Davis and M. Parashar, “Latency performance of SOAP implementations”, Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 407–412, Berlin, Germany, May 2002.
- [**DCOM96**] Microsoft Corporation. “Distributed Component Object Model Protocol-DCOM/1.0, draft”, <http://www.microsoft.com/Com/resources/comdocs.asp>, November 1996.
- [**Delamaro02**] M. Delamaro and G. Picco, “Mobile Code in .NET: A Porting Experience”, Proceedings of the 6th International Conference on Mobile Agents, Barcelona, Spain, October 2002.
- [**Demers94**] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch, “The Bayou Architecture: Support for Data Sharing among Mobile Users”, Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, pp. 2–7, Santa Cruz, California, December 1994.
- [**Dey99**] A. Dey, M. Futakawa, D. Salber and G. Abowd, “The Conference Assistant: Combining Context-Awareness with Wearable Computing”, Proceedings of the 3rd International Symposium on Wearable Computers (ISWC '99), pp. 21-28, October 1999.

- [**Dey00**] A. Dey and G. Abowd, "CybreMinder: A Context-Aware System for Supporting Reminders", Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC), pp. 172-186, Bristol, UK, September 2000.
- [**Duftler01**] M. Duftler, N. Mukhi, A. Slominski and S. Weerawarana, "Web Services Invocation Framework (WSIF)", Proceedings of OOPSLA 2001 Workshop on Object Oriented Web Services, Tampa, Florida, October 2001.
- [**Duran00**] H. Duran and G. Blair, "A Resource Management Framework for Adaptive Middleware", Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC'2K), Newport Beach, California, USA, March 2000.
- [**Efstratiou02**] C. Efstratiou, A. Friday, N. Davies and K. Cheverst, "A Platform Supporting Coordinated Adaptation in Mobile Systems", Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, pp. 128-137, Callicoon, New York, June, 2002.
- [**Fassino02**] J. Fassino, J. Stefani, J. Lawall and G. Muller, "Think: A software framework for component-based operating system kernels", Proceedings of Usenix Annual Technical Conference, Monterey (USA), June 2002.
- [**Fell03**] S. Fell, "Pocket SOAP", <http://www.pocketsoap.com>
- [**Fiege03**] L. Fiege, F. Gärtner, O. Kasten, and A. Zeidler, "Supporting Mobility in Content-Based Publish/Subscribe Middleware", Proceedings of Middleware 2003, Rio de Janeiro, Brazil, June 2003.
- [**Finney96**] J. Finney and N. Davies, "The FLEXible Ubiquitous Monitor Project", Proceedings of the Third Computer Networks Symposium, July 1996.
- [**Flinn01**] J. Flinn, E. de Lara, M. Satyanarayanan, D. Wallach and W. Zwaenepoel, "Reducing the Energy Usage of Office Applications", Proceedings of Middleware 2001, Heidelberg, Germany. November, 2001.
- [**Forman94**] G. H. Forman and J. Zahorjan, "The Challenges of Mobile Computing", IEEE Computer, 27(4), pp. 38-47, 1994.
- [**Friday96**] A. Friday, "Infrastructure Support for Adaptive Mobile Applications", Ph.D Thesis, Computing Department, Lancaster University, 1996.
- [**Gamma95**] E. Gamma, R. Johnson, R. Helm and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.
- [**Gelernter85**] D. Gelernter, "Generative Communications in Linda", ACM Transactions on Programming Languages and Systems, 7(1), pp80-112, January 1985.
- [**Goland99**] Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright, "Simple Service Discovery Protocol 1.0", http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt, Internet Draft, June 1999.

- [**Govindaraju02**] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon, "Requirements for and evaluation of RMI protocols for scientific computing", Proceedings of the 2000 Conference on Supercomputing (SC2000), Dallas, Texas, November 2000.
- [**Haahr00**] M. Haahr, R. Cunningham and V. Cahill, "Towards a Generic Architecture for Mobile Object-Oriented Applications", Proceedings of SerP 2000: Workshop on Service Portability, San Francisco, December 2000.
- [**Hayton98**] R. Hayton, A. Herbert and D. Donaldson, "Flexinet: a flexible, component oriented middleware system", Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications, Sintra, 1998.
- [**Hung02**] P. Hung, "Specifying Conflict of Interest in Web Service Endpoint Language (WSEL)", ACM SIGecom Exchanges, 3(3), pp. 1-8, August 2002.
- [**IBM00**] IBM Corporation, "AlphaWorks: Web Services Toolkit", <http://www.alphaworks.ibm.com/tech/webservicestoolkit>, July 2000.
- [**IEEE03**] The IEEE 802.11 Working Group, <http://www.ieee802.org/11/>
- [**IKV99**] IKV++ GmbH Informations und Kommunikationssysteme. "Grasshopper: The Agent Platform - Technical Overview", February 1999.
- [**IONA99**] Iona Technologies, "OrbixCOMet ", <http://www.iona.com/support/whitepapers/ocomet-wp.pdf>, 1999.
- [**IrDA01**] IrDA Serial Infrared Data Link Standard Specifications, <http://www.irda.org/>, 2001.
- [**Jacobsen99**] K. Jacobsen and D. Johansen, "Ubiquitous Devices United: Enabling Distributed Computing Through Mobile Code", Proceedings of the Symposium on Applied Computing (ACM SAC'99), February 1999.
- [**Johansen95**] D. Johansen, R. van Renesse and F. Schneider, "Operating system support for mobile agents", Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems, Orcas Island, Wa, USA, May 1995.
- [**Joseph95**] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford and M. Kaashoek, "Rover: A Toolkit for Mobile Information Access", Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95), Colorado, U.S., pp. 156-171, December 1995.
- [**Kagal01**] L. Kagal, V. Korolev, H. Chen, A. Joshi and T. Finin, "Centaurus: A framework for intelligent services in a mobile environment", Proceedings of the International Workshop on Smart Appliances and Wearable Computing (IWSAWC), April 2001.
- [**Klefstad03**] R. Klefstad, S. Rao and D. Schmidt, "Design and Performance of a Dynamically Configurable, Messaging Protocols Framework for Real-time CORBA", In Proceedings of Distributed Object and Component-based Software Systems part of the Software Technology Track at the 36th Annual Hawaii International Conference on System Sciences, Big Island of Hawaii, January, 2003.

- [**Kon00**] F. Kon, F. M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes and R. Campbell, “Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB”, Proceedings of Middleware 2000, ACM/IFIP, April 2000.
- [**Kon00b**] F. Kon and R. Campbell, “Dependence Management in Component-Based Distributed Systems”, IEEE Concurrency, 8(1), pp.26-36, 2000.
- [**Kristensen00**] T. Kristensen and T. Plagemann, “Enabling Flexible QoS Support in the Object Request Broker COOL”, Proceedings of International Workshop on Distributed Real-Time Systems (IWDRS 2000), April 2000.
- [**Lange98**] D. Lange and M. Oshima, “Programming and Deploying Java Mobile Agents with Aglets”, Addison-Wesley, 1998.
- [**Leyman01**] F. Leymann, “Web Service Flow Language (WSFL 1.0)”, IBM Document, www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.
- [**Liljeberg97**] M. Liljeberg, K. Raatikainen, M. Evans, S. Furnell, K. Maumon, E. Veldkamp, B. Wind and S. Trigila, “Using CORBA to Support Terminal Mobility”. Proceedings of TINA '97, 1997.
- [**Lin01**] Y. Lin and I. Chlamtac, “Wireless and Mobile Network Architectures”, John Wiley & Sons, 2001
- [**Long96**] S. Long et al., “Rapid Prototyping of Mobile Context-aware Applications: The Cyberguide Case Study”, Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom'96), 1996.
- [**Mamei03**] M. Mamei, F. Zambonelli and L. Leonardi, “Tuples On The Air: a Middleware for Context-Aware Computing in Dynamic Networks”, Proceedings of 1st International ICDCS Workshop on Mobile Computing Middleware (MCM03) Providence, Rhode Island, May 2003.
- [**Marmasse00**] N. Marmasse and C. Schmandt, “Location-aware information delivery with comMotion”, Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing (HUC), pp. 157 – 171, Bristol, UK, September 2000.
- [**Mascolo02**] C. Mascolo, L. Capra, and W. Emmerich, “Middleware for Mobile Computing (A Survey)”, Advanced Lectures in Networking, Editors E. Gregori, G. Anastasi, S. Basagni, Springer, LNCS 2497, 2002.
- [**Mascolo02b**] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich, “XMIDDLE: A Data-Sharing Middleware for Mobile Computing”, International Journal on Personal and Wireless Communications, April 2002.
- [**McHugh03**] J. McHugh “Low Bandwidth SOAP”, webservices.xml.com, <http://webservices.xml.com/pub/a/ws/2003/08/19/ksoap.html>, August 2003.

- [**Meier02**] R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks", Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02), pp. 639-644, Vienna, Austria, 2002.
- [**Microsoft00**] Microsoft Corporation, "The .NET Framework", <http://www.microsoft.com/net/>, 2000.
- [**Microsoft00b**] Microsoft Corporation, "Universal Plug and Play Device Architecture", Version 1.0, http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.
- [**Microsoft01**] Microsoft Corporation, "CE.NET", <http://www.microsoft.com/windows/embedded/ce.net/>
- [**Miller01**] J. Miller and J. Mukerji (eds.), "Model Driven Architecture", OMG Document number ormsc/2001-07-01, July 2001.
- [**Mitchell00**] S. Mitchell, M. Spiteri, J. Bates and G. Coulouris, "Context-Aware Multimedia Computing in the Intelligent Hospital", Proceedings of SIGOPS EW2000, the Ninth ACM SIGOPS European Workshop, Kolding, Denmark, September 2000.
- [**Monson-Haefel00**] R. Monson-Haefel, "Enterprise Java Beans", O'Reilly UK (2nd Edition), 2000.
- [**Moreira01**] R. Moreira, G. Blair and G. Carrapatoso, "Reflective Component-Based & Architecture Aware Framework to Manage Architecture Composition", Proceedings of 3rd International Symposium on Distributed Objects & Applications (DOA 2001), Rome, Italy, September 2001.
- [**Muhl02**] G. Mühl, L. Fiege, F. Gärtner and A. Buchmann, "Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems", Proceeding of 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, pp.167-176, Fort Worth, Texas, October 2002.
- [**Muratore00**] F. Muratore (Ed), "UMTS: Mobile Communications for the Future", John Wiley & Sons, 2000.
- [**Murphy01**] A. Murphy, G. Picco and G. Roman, "LIME: A Middleware for logical and Physical Mobility", Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), May 2001.
- [**Narayanaswami00**] C. Narayanaswami and M. Raghunath, "Application design for a smart watch with a high resolution display", Proceedings of the Fourth International Symposium on Wearable Computers, pp. 7-14, Atlanta, GA, October 2000.
- [**Newcomer02**] E. Newcomer, "Understanding Web Services: XML, WSDL, SOAP and UDDI", Addison-Wesley, 2002.
- [**Oasis02**] Oasis Technical Committee, "Universal Description, Discovery and Integration of Web Services", <http://www.uddi.org>, 2002.

- [**Oberon97**] Oberon Microsystems Inc., “Blackbox Developer and Blackbox Component Framework”, Oberon Microsystems, <http://www.oberon.ch>.
- [**OMG95**] Object Management Group, “The common object request broker: Architecture and specification”, Tech. Report. Version 2.0, July 1995.
- [**OMG97**] Object Management Group, “COM/CORBA Interworking Specification Part A & B”, 1997.
- [**OMG98**] Object Management Group Telecom Domain Task Force, “Wireless access and terminal mobility in CORBA (draft white paper)”, OMG Document: telecom/98-06-01, May 1998.
- [**OMG98b**] Object Management Group, “Event Service”, OMG Document formal/2001-03-01.
- [**OMG02**] Object Management Group, “CORBA Component Model v3.0”, OMG Document: formal/2002-06-65.
- [**OMG03**] Object Management Group, “Wireless Access & Terminal Mobility in CORBA, v1.0”, OMG Document formal/03-03-64.
- [**OWL03**] Web Ontology Language Working Group. <http://www.w3.org/2001/sw/WebOnt/>
- [**POMA03**] “POMA headset”, www.xybernaut.com/Solutions/product/poma_product.htm
- [**Preuss02**] S. Preuss, “JESA Service Discovery Protocol”, Proceedings of Networking 2002, pp. 1196-1201, Pisa, Italy, May 2002.
- [**Rahnema93**] M. Rahnema, “Overview of the GSM System and Protocol Architecture”, IEEE Communication Magazine, 31(4), pp. 92-100, April 1993.
- [**Roman01**] M. Roman, F. Kon and R. Campbell, “Reflective Middleware: From Your Desk to Your Hand”, IEEE Distributed Systems Online, 2(5), August 2001.
- [**Roman02**] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K.Nahrstedt, “Gaia: A Middleware Infrastructure to Enable Active Spaces”, IEEE Pervasive Computing, 1(4), pp. 74-83, Oct-Dec 2002.
- [**Rysavy98**] P. Rysavy, “General Packet Radio Service (GPRS)”, GSM Data Today online journal, September 1998.
- [**Salutation98**] Salutation Consortium. “White Paper: Salutation Architecture Overview”, <http://www.salutation.org/whitepaper/originalwp.pdf>, 1998.
- [**Salutation00**] The Salutation Consortium, “Salutation-Lite Open Source”, <http://www.salutation.org/lite/litesource.htm>, 2000.
- [**Satyanarayanan90**] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. “Coda: A Highly Available File System for a Distributed Workstation Environment”, IEEE Transactions on Computers, 39(4), pp. 447–459, April 1990.
- [**Satyanarayanan96**] M. Satyanarayanan, “Mobile Information Access”, IEEE Personal Communications, 3(1), pp. 26-33, February 1996.

- [**Satyanarayanan96b**] M. Satyanarayanan, "Fundamental Challenges in Mobile Computing", Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, May 1996.
- [**Schmidt99**] D. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware", IEEE Communications Magazine Special Issue on Design Patterns, 37(4), pp. 54-63, 1999.
- [**Segall97**] B. Segall and D. Arnold, "Elvin has left the building: a publish/subscribe notification service with quenching", Proceedings of AUUG97, September 1997.
- [**Seitz98**] J. Seitz, N. Davies, M. Ebner and A. Friday, "A CORBA-based Proxy Architecture for Mobile Multimedia Applications", Proceedings of the 2nd IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS '98), Versailles, France. November, 1998.
- [**Shah03**] P. Shah, B. Bryant, C. Burt, R. Raje, A. Olson and A. Mikhail, "Interoperability between Mobile Distributed Components using the UniFrame Approach", Proceedings of the 41st Annual ACM Southeast Conference, pp. 30-35, Savannah, Georgia, March 2003.
- [**Silva97**] A. Silva, M. Mira da Silva and J. Delgado, "Motivation and Requirements for the AgentSpace: A Framework for Developing Agent Programming Systems", Proceedings of the Fourth International Conference on Intelligence in Services and Networks, Cernobbio, Italy, 1997.
- [**Sivaharan02**] T. Sivaharan, "A Publish-Subscribe Service for Mobile Client Applications", MSc. Thesis, Lancaster University, September 2002.
- [**Sivaharan04**] T. Sivaharan, G. Blair, A. Friday, M. Wu, H. Duran-Limon, P. Okanda, C. Sørensen, "Cooperating Sentient Vehicles for Next Generation Automobiles", Proceedings of the First ACM International Workshop on Applications of Mobile Embedded Systems (WAMES'04), Boston, June 2004.
- [**Srinivasan95**] R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2", Internet RFC 1831, August 1995.
- [**Stallings02**] W. Stallings, "Wireless Communications and Networks", Prentice-Hall, 2002.
- [**Storey02**] M. Storey, G. Blair and A. Friday, "MARE: Resource Discovery and Configuration in Ad Hoc Networks", Mobile Networks and Applications, 7(5), pp. 377-387, October 2002.
- [**Sun97**] Sun Microsystems Corporation, "Java RMI Specification", <ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.pdf>, 1997.
- [**Sun97b**] Sun Microsystems, "The Java Beans Specification 1.01", <http://java.sun.com/products/javabeans/docs/spec.html>, August 1997.
- [**Sun02**] Sun Microsystems, "Java Reflection API", <http://java.sun.com/j2se/1.3/docs/guide/reflection/index.html>, 2002.

- [**Sutton01**] P. Sutton, R. Arkins and B. Segall, "Supporting Disconnectedness - Transparent Information Delivery for Mobile and Invisible Computing", IEEE International Symposium on Cluster Computing and the Grid, Brisbane, Australia, May 2001.
- [**Szyperski98**] C. Szyperski, "Component Software, Beyond Object-Oriented Programming", ACM Press/Addison-Wesley, 1998.
- [**TCD00**] Trinity College Dublin, "The K-ORBs Project", <http://www.dsg.cs.tcd.ie/research/minCORBA/>, 2000
- [**Thatte01**] S. Thatte, "XLANG: Web Services for Business Process Design", www.gotdotnet.com/team/xml_wsspecs/xlang-c/, 2001.
- [**Veizades97**] J. Veizades, E. Guttman, C. Perkins and S. Kaplan, "Service Location Protocol (SLP)", Internet RFC 2165, 1997.
- [**Vinoski02**] S. Vinoski, "Toward Integration - Web Services Interaction Models", IEEE Internet Computing Online, 6(3), pp. 89-91, May/June 2002.
- [**Vinoski03**] S. Vinoski, "It's just a Mapping Problem", IEEE Internet Computing Online, 7(3), pp. 88-90, May/June 2003.
- [**W3C99**] World Wide Web Consortium, "Resource Description Framework (RDF)", www.w3.org/RDF/, 1999.
- [**Wade99**] S. Wade, "An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments," Ph.D. Thesis, Computing Department, Lancaster University, 1999.
- [**Waldo98**] J. Waldo, "Javaspaces specification 1.0", Sun Microsystems Technical report, March 1998.
- [**Wall01**] T. Wall and V. Cahill, "Mobile RMI: Supporting Remote Access to Java Server Objects on Mobile Hosts", Proceedings of the Third International Symposium on Distributed Objects and Applications", pp. 41-51, Rome, Italy, September 2001.
- [**Watanabe87**] T. Watanabe and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language", Proceedings of OOPSLA'88, Vol. 23 of ACM SIGPLAN Notices, pp. 306-315, ACM Press, 1988.
- [**Weiser91**] M. Weiser, "The Computer for the 21st Century", Scientific American, 265(3), pp. 94-104, September 1991.
- [**Wong97**] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young and B. Peet, "Concordia: An Infrastructure for Collaborating Mobile Agents", Proceedings of the 1st International Workshop on Mobile Agents, Berlin, Germany, April 1997.
- [**Wyckoff98**] P. Wyckoff, S. McLaughry, T. Lehman and D. Ford, "Tspaces", IBM Systems Journal, 37(3), pp. 454-474, 1998.

[**Yokote92**] Y. Yokote, “The Apertos Reflective Operating System: The Concept and Its Implementation”, Proceedings of OOPSLA’92, ACM SIGPLAN Notices, Vol. 28, pp. 414-434, ACM Press, 1992.

[**Zachariadis03**] S. Zachariadis, C. Mascolo and W. Emmerich. “Adaptable Mobile Applications: Exploiting Logical Mobility in Mobile Computing”. Proceedings of 5th International Workshop on Mobile Agents for Telecommunication Applications”, Marrakech, Morocco, October 2003.

Appendix A Component Framework Meta Interfaces

Operations for Inspection

```
/* ***** */
* Returns a list with the identifiers of the components that constitute the
* base-level configuration.
* ***** /
HRESULT get_internal_components([out] IUnknown** ppComps[], [out] int *pcElems);

/* ***** */
* Returns a list with information (component id and interface names) of all
* components bound to the one identified as the argument.
* ***** /
HRESULT get_Bound_Components([in] IUnknown* comp, [out] ConnectedComponent**
                             ppConnections[], [out] int *pConnectedElements);

/* ***** */
* Returns a list with the ids of all connections that are part of the
* base-level composition.
* ***** /
HRESULT get_internal_bindings([out] unsigned long *ppConnIDs[], [out] int *pcElems);
```

Operations for Reconfiguration

```
/* ***** */
* Establish a local binding on the interface between two components.
* ***** /
HRESULT local_bind([in] IUnknown *pIUnkSource, [in] IUnknown *pIUnkSink, [in]
                   REFIID iid, [out] unsigned long *pConnID);

/* ***** */
* Break the local binding between the two Components.
* ***** /
HRESULT break_local_bind([in] unsigned long connID);

/* ***** */
* Create and insert a new component into the base-level configuration,
* with the given name.
* ***** /
HRESULT insert_component([in] CLSID clsid, [in, string] const char *name, [out]
                          IUnknown **ppIUnknown);

/* ***** */
* Delete the component from the configuration.
* ***** /
HRESULT remove_component([in] IUnknown *pIUnknown);

/* ***** */
* Replace an existing component with a new component of the
* given type.
* ***** /
HRESULT replace_component([in] IUnknown *pOldComponentIUnk, [in] IUnknown
                           *pNewComponentIUnk);
```

```

/*****
* Map the interface of an internal component as a new interface of the *
* composite CF. *
*****/
HRESULT Expose_Interface([in] IID r_intf, [in] IUnknown *pComp);

/*****
* Remove an exposed interface. *
*****/
HRESULT UnExpose_Interface([in] IID r_intf, [in] IUnknown *pComp);

/*****
* Map the receptacle of an internal component as a new receptacle of *
* the composite component. *
*****/
RESULT Expose_Receptacle([in] IID r_intf, [in] IUnknown *pComp, [in]
                           OCM_RecpType_t recpType);

/*****
* Remove an exposed Receptacle. *
*****/
HRESULT UnExpose_Receptacle([in] IID r_intf, [in] IUnknown *pComp);

/*****
* Replace the current graph of components with a new graph. *
*****/
HRESULT ReplaceConfiguration([in] IUnknown *pComponents[], [in] int cCmps);

/*****
* Start the transaction for architecture reconfiguration. *
*****/
HRESULT init_arch_transaction();

/*****
* Completes the reconfiguration. *
*****/
HRESULT commit_arch_transaction();

/*****
* Rolls back any changes made during an architectural transaction. *
*****/
HRESULT rollback_arch_transaction();

```


Appendix B Example XML Component Configuration

The following is an XML based architectural description of the IIOP client binding personality.

```
<ReMMoC_Configuration>
  <Interfaces>
    <Interface>{D692671C-F14C-4f27-9646-07A6E7EC013A}</Interface>
  </Interfaces>
  <Components>
    <Component>
      <Name>ReMMoC_OSNet</Name>
      <ID>{15E7D7CF-5750-46de-9924-D219DDD7CA8E}</ID>
    </Component>
    <Component>
      <Name>ReMMoC_TCP</Name>
      <ID>{8CB1DB64-ED4F-4486-8578-96017825F6DD}</ID>
      <Connections>
        <Interface>{D993631C-FD4C-4f27-9646-07E6E7EC098A}</Interface>
      </Connections>
    </Component>
    <Component>
      <Name>ReMMoC_CORBAMarshaling</Name>
      <ID>{12C7D7CF-5451-43de-9924-D219DED2CB2A}</ID>
    </Component>
    <Component>
      <Name>ReMMoC_GIOP</Name>
      <ID>{14C7E7CF-5750-46de-9924-D219DED7CB2A}</ID>
      <Connections>
        <Interface>{D892611A-F14B-4f27-9646-07A6E7EC013A}</Interface>
        <Interface>{ABFC5317-BF1D-4644-A19C-1A6766AA8349}</Interface>
      </Connections>
    </Component>
    <Component>
      <Name>ReMMoC_IIOP</Name>
      <ID>{842AC4E9-CC84-4bb6-9673-EDAC639F106D}</ID>
      <Connections>
        <Interface>{D892611A-F14B-4f27-9646-07A6E7EC013A}</Interface>
        <Interface>{ABFC5317-BF1D-4644-A19C-1A6766AA8349}</Interface>
        <Interface>{D293611A-FD4C-4f27-9646-07A6E7EC013A}</Interface>
      </Connections>
    </Component>
  </Components>
</ReMMoC_Configuration>
```

Appendix C WSDL of Application Services

1. Stock Quote Service

```
<?xml version="1.0"?>
<definitions name="StockQuote"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="long"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>

<message name="GetLastTradePriceInput">
  <part name="body" element="TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
  <part name="body" element="TradePrice"/>
</message>

<portType name="StockQuotePort">
  <operation name="getQuote">
    <input message="GetLastTradePriceInput"/>
    <output message="GetLastTradePriceOutput"/>
  </operation>
</portType>

<service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="StockQuoteBinding"></port>
</service>

</definitions>
```

2. Chat Service

```
<?xml version="1.0"?>
<definitions name="Chat" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <element name="InitRequest">
      <complexType>
        <all>
          <element name="ID" type="string"/>
          <element name="Nickname" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="InitResponse">
      <complexType>
        <all>
          <element name="ID" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="ChatData">
      <complexType>
        <all>
          <element name="Message" type="string"/>
        </all>
      </complexType>
    </element>
  </types>

  <message name="InitRequestInput">
    <part name="body" element="InitRequest"/>
  </message>
  <message name="InitResponseOutput">
    <part name="body" element="InitResponse"/>
  </message>
  <message name="ChatMessage">
    <part name="body" element="ChatData"/>
  </message>

  <portType name="ChatPort">
    <operation name="Init">
      <input message="InitRequestInput"/>
      <output message="InitResponseOutput"/>
    </operation>
    <operation name="ReceiveMessage">
      <input message="ChatMessage"/>
    </operation>
    <operation name="SendMessage">
      <output message="ChatMessage"/>
    </operation>
  </portType>
</definitions>
```

3. Music Player Service

```
<?xml version="1.0"?>
<definitions name="MusicService" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <element name="GetNumberOfSongsResponse">
      <complexType>
        <all><element name="Number" type="long"/></all>
      </complexType>
    </element>
    <element name="GetSongRequest">
      <complexType>
        <all><element name="index" type="long"/></all>
      </complexType>
    </element>
    <element name="SongResponse">
      <complexType>
        <all>
          <element name="Title" type="string"/>
          <element name="Artist" type="string"/>
        </all>
      </complexType>
    </element>
  </types>
  <message name="GetNumberOfSongsInput">
    <part name="body" element="GetSongRequest"/>
  </message>
  <message name="GetNumberOfSongsOutput">
    <part name="body" element="GetNumberOfSongsResponse"/>
  </message>
  <message name="GetSongInput">
    <part name="body" element="GetSongRequest"/>
  </message>
  <message name="GetSongDetailsOutput">
    <part name="body" element="SongResponse"/>
  </message>
  <portType name="MusicServicePort">
    <operation name="getNumberOfSongs">
      <input message="GetNumberOfSongsInput"/>
      <output message="GetNumberOfSongsOutput"/>
    </operation>
    <operation name="getSongDetails">
      <input message="GetSongInput"/>
      <output message="GetSongDetailsOutput"/>
    </operation>
    <operation name="PlaySong">
      <input message="GetSongInput"/>
    </operation>
    <operation name="StopSong">
      <input message="GetSongInput"/>
    </operation>
  </portType>
</definitions>
```