Interoperating with Services in a Mobile Environment

Paul Grace[1], Gordon S. Blair[1] and Sam Samuel[2]

[1]Distributed Multimedia Research Group, Computing Department,
Lancaster University, Lancaster, LA1 4YR, UK.
p.grace@lancaster.ac.uk, gordon@comp.lancs.ac.uk
[2]Global Wireless Systems Research, Bell Laboratories, Lucent Technologies,
Quadrant, Stonehill Green, Westlea, Swindon, SN5 7DJ.
lsamuel@lucent.com

ABSTRACT

Mobile computing is characterised by users carrying portable devices that allow communication between people and continuous access to networked services independent of their physical location. A mobile application must discover and interoperate with the required application services available to them in their present location. However, these services will be developed upon a range of middleware types (e.g. remote method invocation, publish-subscribe, message-oriented and tuple spaces) and advertised using different service discovery protocols (e.g. UPnP and SLP) unknown to the application developer. Wireless environments are also subject to changes in network QoS and connectivity, and mobile applications operate on devices with limited resources. Given these properties, existing middleware technologies are inappropriate to support mobile client-based applications; developing upon a single middleware platform restricts the possible mobile services that can be interacted with. Therefore, a middleware platform for mobile computing must adapt its behaviour to interoperate with any type of discovered service, provide the best level of service in an environment that is frequently changing and finally, be lightweight in resource use due to the constraints of mobile devices. This paper proposes reflection as a well-suited technology to implement the adaptive features of mobile middleware and identifies what the key requirements of a dynamic middleware platform are. In addition, we describe ReMMoC (Reflective Middleware for Mobile Computing), a middleware platform that dynamically adapts its structure to interoperate with a range of middleware types that may exist in the mobile environment. Finally, the use of ReMMoC in a typical mobile scenario is presented and the memory footprint cost of utilising reflection to create a mobile middleware platform is evaluated.

1. INTRODUCTION

The emergence of new wireless network and mobile device technology has increased the prominence of mobile computing. This has led to novel application types emerging to exploit this domain (e.g. context aware applications, m-commerce, ad-hoc communities, mobile gaming and many others). Consequently, it is envisioned that a variety of application services will be available to the mobile user in their current locale. However, the mobile computing domain presents a number of challenges for middleware, for example, overcoming the problems of weak connection and limited resources. Research into this area has concentrated on addressing these issues, ranging from extensions to well-established platforms for fixed networks [Haahr et al, 00][Seitz et al, 98][Reinstorf et al, 01] and middleware systems designed explicitly to support mobile applications [Meier et al, 02][Davies et al, 98][Capra et al, 01] (for a more detailed description of these see section 2). Therefore, a wide choice of middleware platforms is available to develop mobile application services upon; possible middleware paradigms that may be utilised include: remote method invocation, publish-subscribe, message-oriented and tuple spaces. This demonstrates *middleware heterogeneity* within the mobile environment [Roman, 01]; a client application based upon publish-subscribe cannot interoperate with an RMI-based service of the same application type. Likewise, variations in the implementations of middleware paradigms, e.g. SOAP/ CORBA and different publish-subscribe services, means that *clients developed on one type cannot interoperate with services developed upon another*. As an example, a tourist guide client implemented upon a publish-subscribe system would only be able to interoperate with specific publishers of the same middleware type. Conversely, tourist guide services at a different location implemented using an alternative middleware (e.g. a SOAP service), would require a separate client application and middleware implementation.

Similarly, the services available to the user as they roam from location to location are advertised using one of the contrasting service discovery protocols available. At present, there are four main service discovery technologies: Jini, SLP, UPnP and Salutation; in addition, new technologies are emerging to better support the discovery of services in mobile environments (e.g. JESA [Preuss, 02] & Centaurus [Kagal et al, 01] and across wireless ad-hoc network types (e.g. SDP in Bluetooth and Salutation Lite). Utilising only one of these technologies on a mobile device to discover services will mean that services advertised by the other types will be missed. For example, a set of embedded devices within a room (lights, video, CD player) advertising their services using UPnP cannot be used by a mobile device looking for services using the Service Location Protocol. This problem is likely to become significantly worse in the future with the advent of ubiquitous computing enabled by emerging technologies to discover and interact with the services an embedded device offers.

Furthermore, the mobile computing environment is dynamic in nature, in terms of changes in levels of network bandwidth, periods of connectivity and network type. Additionally, mobile applications operate on devices that are restricted by limited resources, e.g. battery power, CPU size, system memory and screen size. It has been identified therefore, that middleware should be aware of the current environmental context and adapt its behaviour based on this information in order to provide the best level of service to the application [Blair et al, 01]. We extend this argument to include awareness of middleware within the environment; thus we propose that a generalised middleware for mobile computing should be able to adapt its behaviour so that any service available in the current location can be found and communicated with irrespective of its service discovery protocol and underlying middleware type.

We advocate the use of reflection and component technology as a well-suited technique for the development of a mobile middleware platform. Reflection is a principled method that supports introspection and adaptation to produce configurable and reconfigurable middleware. Using this approach, a middleware platform should be able to alter its behaviour dynamically to: i) find the required mobile services irrespective of the service discovery protocol, ii) interoperate with services implemented by different middleware types and iii) provide the best level of service to the application in the current environmental conditions. Given these properties, mobile applications can then be developed independently of the underlying middleware technology. The design and implementation of a reflective middleware platform, named ReMMoC (Reflective Middleware for Mobile Computing), providing this functionality is described. The use of this platform is illustrated within a typical real world mobile scenario and its memory resource use, i.e. static memory footprint is compared against similar technologies.

The paper is structured as follows. Section 2 presents a typical mobile scenario to illustrate the heterogeneous properties of the mobile environment. The concepts of reflection and component technologies used by ReMMoC, followed by the design of the reflective middleware platform itself is described in section 3 and then evaluated in section 4. Future work in this area is identified in section 5. Section 6 examines related work in the field of mobile middleware and finally, some overall conclusions are drawn in section 7.

2. MOBILE SCENARIO

In this section we present a typical mobile computing scenario to illustrate middleware heterogeneity that exists in the mobile domain. In this example, three different application types are presented as services available to mobile users at two locations. Instances of these services are implemented across different types of middleware in the two separate locations and advertised using contrasting service discovery protocols. Application 1 is a *mobile sport news* application, whereby sport news stories are presented to the user based on their current location. For example, users in Manchester could obtain the latest information about the football clubs in that location (Bury, Manchester City etc.). Application 2 is a *tourist guide* application that receives information about local points of interest and acts as a guide through these to the mobile user. Finally, application 3 is a *print* application that allows a given document to be printed to an available printer in the vicinity.

Characteristically, a mobile user will carry a device with computational capacity (e.g., Phone, PDA, Laptop) that is able to connect to wireless networks (e.g. GSM, GPRS, IEEE 802.11b, HomeRF, Bluetooth); in this scenario the user has a Personal Digital Assistant that can connect via an 802.11b network. Figure 1 illustrates two possible locations in the session of a mobile user and the mobile services that can be interacted with via the given mobile device. It can be seen that the same mobile application services are available to the user, but the platforms presenting them differ. For example, the Sport News service is implemented as a publish-subscribe channel on the university campus network and as a SOAP service in the town centre. If fixed middleware were to be used, then two separate applications and middleware implementations would be needed on the device. Similarly, the print service and tourist guide service are implemented across different middleware types.

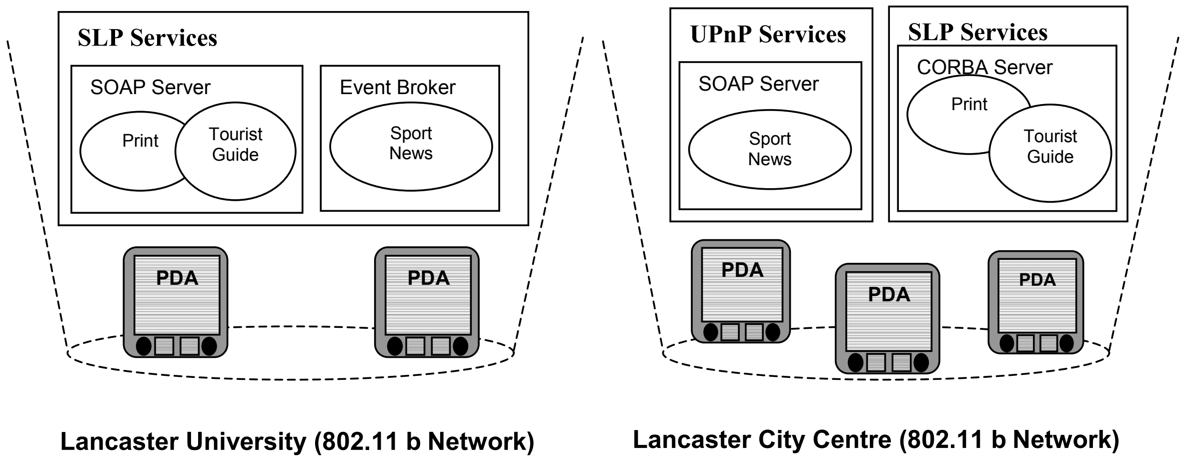

Figure 1. A typical mobile application scenario

However, this is not the only level of heterogeneity in the scenario, the services themselves must first be discovered by the mobile application before interaction can occur. Nevertheless, in this setting the service discovery technologies are different, i.e. the services available to campus users are discoverable using the SLP and the services across the city centre can be found using both UPnP and SLP. If the middleware can only perform one type of service discovery then it may miss some available resources and in the worst-case scenario find none of them, for example if a client used Jini as its discovery protocol it would not find any of the services in this scenario.

Given scenarios of this type, the authors argue that 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.

3. ReMMoC (A REFLECTIVE MIDDLEWARE FOR MOBILE COMPUTING)

This section describes the design of a reflective middleware platform (ReMMoC) for supporting generic classes of distributed mobile applications. The platform is influenced by the work on OpenORB and OpenCOM that has been carried out at Lancaster University (the following section describes the features of these in more detail), i.e. we utilise two key technologies: reflection and components in order to produce a configurable and reconfigurable middleware that meets the requirements we have previously identified.

3.1 Background on OpenCOM

OpenCOM is a lightweight, efficient and reflective component model, built atop a subset of Microsoft's COM. Higher level features of COM, including distribution, persistence, transactions and security are not used, whilst core aspects including the binary level interoperability standard, Microsoft's IDL, COM's globally unique identifiers and the IUnknown interface are the basis of the implementation. However, the initial version of OpenCOM [Clarke et al, 01] was developed for Windows-based desktop machines and is unsuitable for resource-constrained devices such as mobile phones, Pocket PCs and embedded devices. Therefore, OpenCOM has been updated to also execute on devices running the Windows CE Operating System.

The fundamental concepts of OpenCOM are *interfaces*, *receptacles* and *connections* (bindings between interface and receptacles). An interface expresses a unit of service provision and a receptacle describes a unit of service requirement. OpenCOM deploys a standard runtime substrate that manages the creation and deletion of components, and acts upon requests to connect/disconnect components. Furthermore, a system graph of the components currently in use is maintained to support the introspection of a platform's structure. Initially, each OpenCOM component implemented five standard interfaces: two component management interfaces (*ILifeCycle* and *IReceptacle*) and three meta-interfaces (*IMetaInterception*, *IMetaArchitecture* and *IMetaInterface*). However, the implementation of these interfaces increases the size of each component and in some cases may never be used. Therefore, OpenCOM now requires only three interfaces to be implemented by each component:

- ILifeCycle provides operations called *startup* and *shutdown* that are called when a component is created or destroyed.
- IReceptacle 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.

In addition, the OpenCOM runtime component provides meta-interception and meta-architecture interfaces. Interception enables pre and post methods to be associated with a given interface on a component, which 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. The OpenCOM architecture is illustrated in figure 4. The final change to OpenCOM is the removal of locking receptacles to reduce the implementation size and increase portability; instead we now advocate the use of higher-level structures (i.e. component frameworks, see section 3.2) for maintaining system integrity. The memory footprint size of the OpenCOM component platform for Pocket PC devices running the Windows CE 3.0 operating system on StrongARM processors is 17Kbytes.

3.2 Background on OpenORB

OpenORB is a reflective middleware platform that was designed to overcome the problems of black-box middleware implementations, which hide their structure and are unable to adapt their performance in the face of environmental changes. OpenORB can be configured and reconfigured through a marriage of reflection, component technologies and component frameworks. A component framework is defined as a collection of rules and contracts that govern the interaction of a set of components [Szyperski, 98]. The motivation behind component frameworks is to constrain the design space and the scope for evolution. Moreover, they simplify component development and assembly, enable lightweight components and increase the understandability 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 well-defined APIs that encompass the operations of the CF's constituent components. In OpenORB, a component framework is explicitly represented by a single component (called a component framework representative), which is responsible for implementing the meta-interfaces while enforcing the architectural constraints.

The middleware architecture of OpenORB is decomposed into an extensible set of specialised and focused domains of concern, such as buffer management and binding establishment, each based on a component framework as shown in figure 2. That is, OpenORB is structured as a set of (configurable) 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 can be specialised to domains including, multimedia and real-time systems.

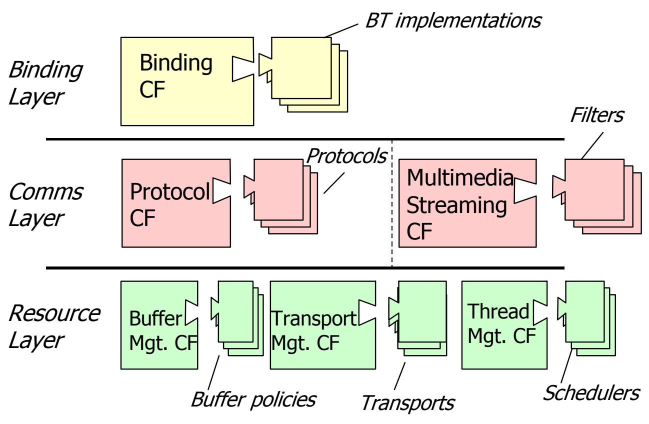

Figure 2. *The component frameworks of Open ORB*

3.3 The Design and Implementation of ReMMoC

3.3.1 Overview

This section describes the ReMMoC platform, a configurable and reconfigurable reflective middleware that supports mobile application development and overcomes the heterogeneous properties of the mobile environment. ReMMoC uses OpenCOM as its underlying component technology and it is built as a set of component frameworks. Using many component frameworks (e.g. as found in OpenORB) increases the size of the middleware implementation; extra management functionality for managing reconfiguration exhausts the constrained resources of a mobile device. Therefore, ReMMoC consists 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. These two component frameworks can be seen in figure 3. 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. Adding more component frameworks for other non-functional properties such as security and resource management can extend the platform at a later stage. The top level CF of ReMMoC manages the reconfiguration of the underlying frameworks and provides an interface for discovering and interoperating with services. However, the application can also interact directly with either of the frameworks to avoid indirection and extra processing overhead; for example, if the application knows the binding type it wants to use.

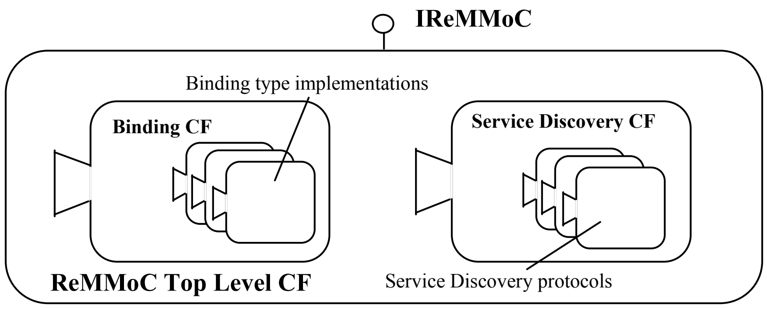

Figure 3. Overview of the ReMMoC platform

3.3.2 The Binding Framework

The primary function of the binding framework is to provide interoperation with heterogeneous mobile services that exist in the mobile environment. Therefore, over time it may be configured as an IIOP client configuration and make a number of IIOP requests, or change to a subscribe configuration and wait to receive events of interest. Fundamentally, any type of middleware paradigm, synchronous or asynchronous, can be plugged into the binding framework if it has been implemented using OpenCOM components. The component framework structure manages the configuration and dynamic reconfiguration of these bindings and ensures that a correct binding type is in place before operation occurs.

Figure 4 illustrates the binding framework and also presents the generic features of component frameworks within ReMMoC (functionality that is common to every CF). Each component framework in the platform implements three meta-interfaces that can be utilised by a higher-level CF or by an application component. These interfaces allow the inspection of the current structure of the framework and the ability to dynamically alter its behaviour. The three meta-interfaces are: (i) ICFMetaInterface, (ii) ICFMetaArchitecture and (iii) ICFMetaInterception. These interfaces differ from their counterparts in OpenCOM (i.e. IMetaInterface, IMetaArchitecture and IMetaInterception) by offering extra reflective capabilities and constraining their operation to each framework structure. The ICFMetaInterface allows the programmer to discover the types of the interfaces and receptacles that are implemented by the CF and also that exist on components currently within the component framework. Furthermore, it allows the inspection of the methods available on any given interface within the framework; these methods can then be dynamically invoked. The ICFMetaArchitecture interface allows access to the current component framework graph to inspect the components in use and the connections between components; the structure of the graph can also be dynamically altered using this interface. Finally, the ICFMetaInterception interface provides functions to insert pre and post method behaviour onto a specified interface within the framework. A detailed list of the functions offered across the meta-space can be found in appendix A. The meta-space implementation is encapsulated within a single component (ReMMoC Meta Implementation) that is included in each component framework implementation. It is important to note that the implementation of this component relies on the reflective properties of the OpenCOM platform; in turn this reduces the component's size.

Each component framework constrains the configuration of components to a valid implementation within the framework domain. Therefore, in the binding framework only valid binding type implementations are allowed after reconfiguration. To enforce this policy, each component framework implements a receptacle called IAccept. When configuration has been initiated or a change to the existing binding implementation has been made, a call to the IAccept interface is performed. This carries out a check on the component architecture; if the Accept component verifies the architecture then a true response is generated and the platform can continue its operation, otherwise, an exception is generated and the framework rolls back to the previous configuration. The complexity of checking depends upon the implementation of the Accept component, which may have no checking (no component connected), simply check against a list of components or alternatively incorporate pattern-based strategies; by changing the component implementation the strategy is changed. For the purpose of the ReMMoC implementation, the CF graph is checked against known component configurations (e.g. a SOAP client's graph structure) described in XML.

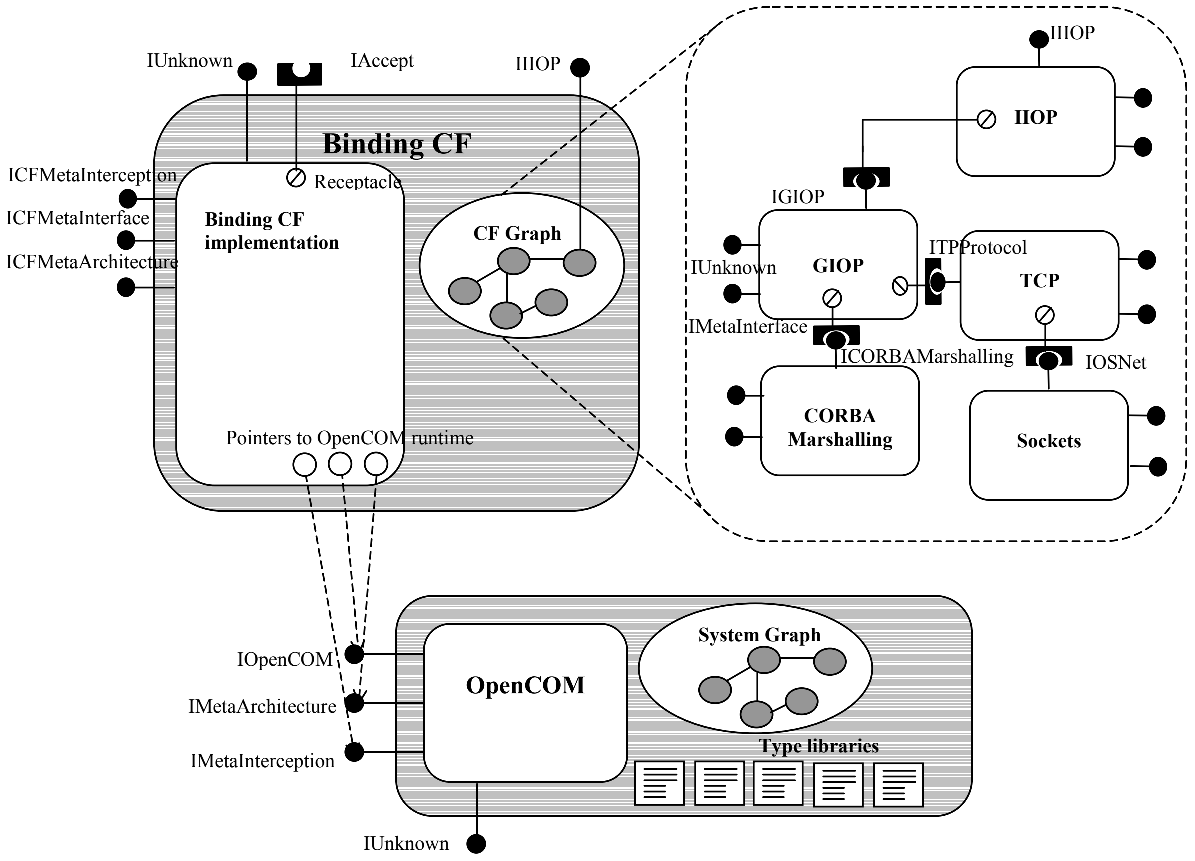

Figure 4. *The ReMMoC binding component framework*

Moreover, every component framework maintains a graph of the components that comprise the current implementation. This graph is a subset of the OpenCOM runtime system graph. To reduce resource use this information is not replicated and the framework maintains a simple view of the underlying graph. Any actual changes to the CF graph are reflected directly at the system graph level. Therefore, component frameworks cannot rely on components within another separate framework, only on components and frameworks within that framework; this technique is put forward by [Clarke et al, 01].

Figure 4 also illustrates that no fixed interface for the binding framework service is implemented i.e. the application programmer does not use the same interface to utilise the framework. Instead, the interface for the given configuration (e.g. ISOAP, IIIOP, ISubscribe or multiples of these) is exported as the means to use the framework and this is discoverable by the application or higher-level framework using the framework's ICFMetaInterface. Reflection is then used to discover the methods on this interface and dynamically invoke them. An example of this is shown in the diagram, where the IIIOP interface of the current configuration is exported. In order to implement a generic binding interface, two directions can be taken. The basic components can implement an agreed plug-in interface (e.g. the IIOP client, SOAP client and publisher all implement an IBind interface) [Blair et al, 01]. This allows unknown binding type implementations to be added dynamically. However, the components must be developed directly for the middleware, which does not allow re-usability of components between middleware implementations. Furthermore, the generic interface cannot express all of the functionality exposed by a component configuration. The second direction is to implement a mapping from a higher-level framework interface to the component interface. In our case this is done in the ReMMoC top-level framework. We can now re-use any OpenCOM based binding implementation and all of its features. The disadvantage is that code to do this for each type is needed. Therefore, this is dynamically plugged into the framework for the current implementation, allowing any type to be utilised at run-time. Hence, if resources are scarce the application can include just a binding framework and control this itself; otherwise a higher-level framework that simplifies the programming across different binding types (ReMMoC) can be used.

The binding framework allows changes to be made at two distinct levels. Firstly, the current binding type implementation can be replaced. This is illustrated in the diagrams shown in figure 5. A configuration of components that implement a SOAP client binding type, for performing SOAP RPC invocations, and a subscriber configuration for use as part of a publish-subscribe service is illustrated; in this case the subscriber component subscribes for XML events delivered over a multicast protocol. Each of these is a "single" personality, but it is also feasible for multiple personalities to be created i.e. SOAP and IIOP clients together, or a publish-subscribe publisher and SOAP client together; their implementation is simply a configuration of components, but more than one interface is exposed by the framework. The creation of the component configuration for each binding type is designed so as to minimise the number of components used (each component adds extra overhead) while maintaining a level of re-usable components and scope for fine-grained reconfiguration (discussed later). The ICFMetaArchitecture of the binding framework is used to dynamically change between middleware roles; specifically, the *ReplaceConfiguration* method is invoked passing the new component configuration; for example, figure 5 shows the SOAP binding being replaced by the subscribe binding.

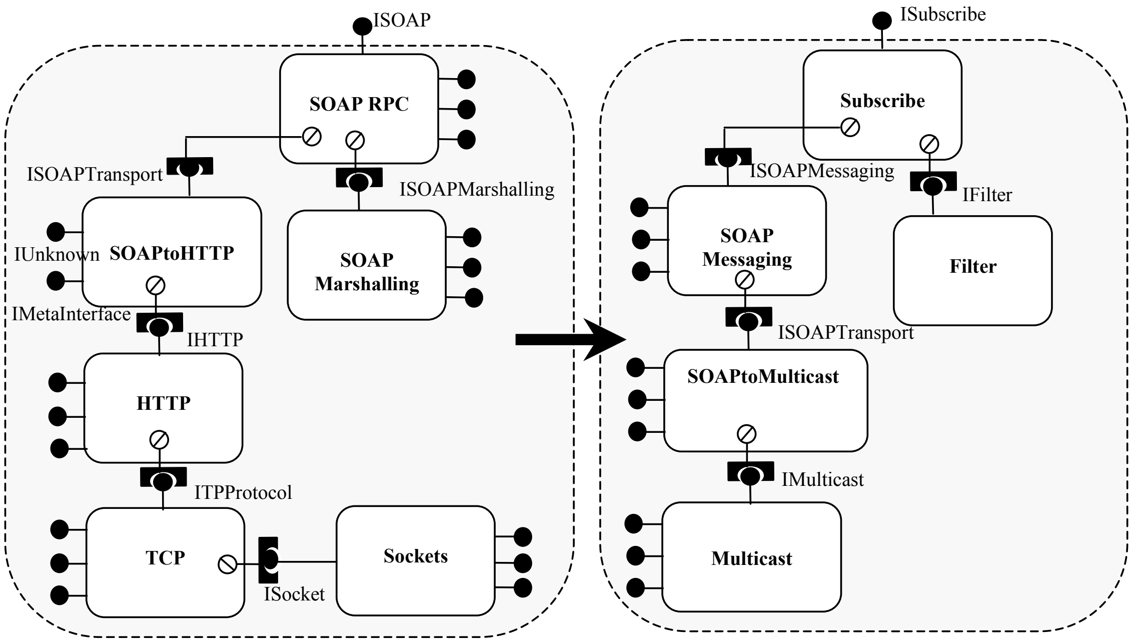

Figure 5. *A dynamic change from a SOAP client binding framework implementation to a publish-subscribe subscriber implementation.*

Changes to the framework can also be made at the component level; that is, more fine-grained changes to the configuration can be made in light of environmental context changes, such as those involving quality of service, or changes in the application's requirements. For example, if the device encountered an environment where frequent disconnection occurred then the SOAP client's transport binding could be changed from HTTP-based components to SMTP components. Furthermore, an application may require IIOP server side functionality, in addition to the existing client side; therefore components implementing server side functionality are added. Finally, the subscriber configuration may choose to switch from a reliable multicast component to an unreliable one, if network congestion is detected. To perform this functionality the methods of the ICFMetaArchitecture interface are called.

We have implemented the binding framework and a set of binding type implementations to illustrate its functionality. It is feasible, for any binding type to be utilised by the framework e.g. tuple spaces and media stream bindings, provided it is implemented as a configuration of OpenCOM components. However, we have implemented IIOP client and server, SOAP client and Publish-Subscribe personalities for evaluation and demonstration purposes.

3.3.3 The Service Discovery Framework

The principal function of the Service Discovery framework is to allow services that have been advertised by different service discovery protocols to be discovered. This is performed by changing the component configuration depending on what type of discovery technology is currently used in the environment. For example, if only SLP is currently in use, the framework's configuration will be an SLP Lookup personality. However, if SLP and UPnP are both being utilised at a location then the framework's configuration will include component implementations to discover both. An application may also require services to be advertised; therefore, the personality can be changed to include service registration functionality using one or more protocols of choice. As in the Binding CF, the framework allows individual components to be changed, added or deleted. This is beneficial due to the range of functionalities that service discovery technologies offer. For example, in SLP you may wish to perform lookup using just the multicast protocol if no directory agent is present, but at a later stage if a directory agent is discovered the configuration can be changed to direct requests to it, rather than send multicast requests.

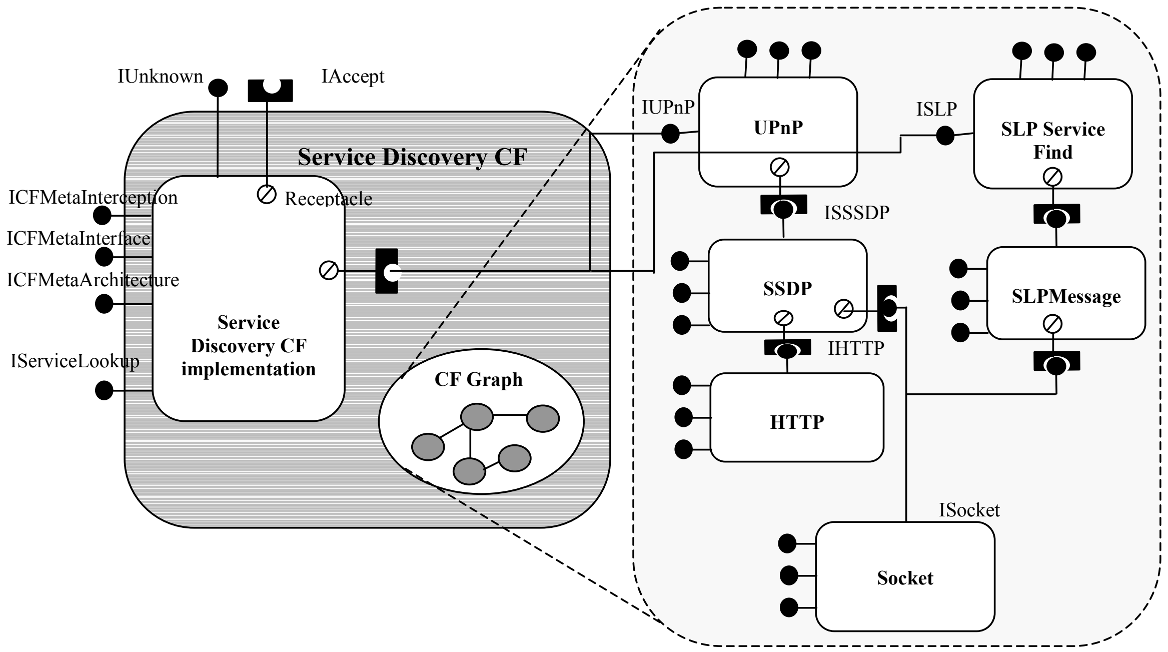

Figure 6. The Service Discovery Component Framework

The basic constituents of this component framework are identical to the binding CF; i.e. the same three meta interfaces are implemented. Also, the IAccept receptacle connection is utilised to check the validity of the discovery configuration and the framework maintains a graph of the components. The service discovery framework differs in that it presents its own custom interface of the service it provides (IServiceLookup); the IDL for this interface is listed in figure 7. The interface offers three key method calls. The first makes a generic service lookup call and will return the information from any service discovery technology lookup call in a generic format. For example, if a generic lookup of a weather service is called and two discovery configurations are implemented by the framework (UPnP and SLP) then the framework will return a list of matched services, from both types in a generic format. It is this information that then drives the use of the binding framework. The interface also provides methods to determine if a particular service discovery technology is available in the environment or not. Furthermore, the properties of the framework ensure that direct interaction with the underlying component interface e.g. IUPnP can be carried out in similar fashion to the binding CF, if the application needs to improve performance in light of constrained resources.

```
Interface IServiceLookup{
        Service[] Lookup(ServiceType st, Attributes[] attrs);
        Boolean SDTinUse(SDTType sdt);
        SDTType FindSDTs();
}
```

Figure 7. IDL definition of IServiceDiscovery Interface

However, the issue of which discovery protocol to use in the current environment must still be addressed. We argue that this information will be made available by higher-level, context-based mechanisms that present information about the environment. This may come from information about the device, e.g. if the device is currently using a Bluetooth connection then an SDP personality should be used. Furthermore, the device may use prior knowledge to select an appropriate protocol, i.e. the platform stores context information per location that details which service discovery protocols were used at that point previously. The use of a context service within the environment is discounted because it would need to be discovered. Therefore, the framework presents the minimum functionality required to obtain the discovery protocol if none of the previous methods can be used. This involves configuring the platform so that checks for each type of protocol can be made e.g. configure components to check for Jini, SLP, UPnP and Salutation. Invoking the *SDTinUse*() and *FindSDTs*() methods on the *IServiceLookup* interface will then use these configurations to return information to the application. The final requirement of the platform is to provide knowledge of when to check for environmental changes; the obvious technique is to synchronously check for new protocols when no results are returned from a lookup. Alternatively, the environment could be continuously monitored for protocol types and then an asynchronous event is generated when the protocol is detected.

We have implemented the service discovery framework with two service discovery protocol implementations: SLP and UPnP. Therefore, this allows us to demonstrate how to overcome the problems of the availability of multiple service discovery protocols. However, as with the binding framework, it is feasible for any discovery protocol to be integrated into the framework.

4. EVALUATION

This section provides an evaluation of the ReMMoC platform in terms of the memory footprint sizes that the platform utilises and in providing the functionality required to interoperate with heterogeneous application services in a typical mobile scenario.

4.1 The Cost of 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. In the future, storing components on the device is likely to be less of a problem as mobile devices with much higher memory capacity become available. However, components will still need to be transmitted across the network (for example, when the platform discovers it needs components not currently on the device). Therefore, the implementation of middleware personalities still needs to be minimised. We have implemented the components used to build the ReMMoC platform with the aim of reducing the storage space they occupy. Table 1 documents the static memory footprint sizes of the separate parts of the platform i.e. configurations for the two frameworks (IIOP client, SOAP client etc.). Four measurements were taken for each personality: the ARM and x86 implementations for reflective and non-reflective personalities. 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.

	Reflective		*Non-Reflective*	
Function	***ARM (Bytes)***	***x86 (Bytes)***	***ARM (Bytes)***	***x86 (Bytes)***
Platform Core				
OpenCOM	28160	18432	n/a	n/a
Binding CF	16896	11776	n/a	n/a
Service Discovery CF	19968	16384	n/a	n/a
Binding Configurations				
IIOP Client	96768	79872	56320	38912
IIOP Server	99840	82432	58880	40960
IIOP Client & Server	140288	114688	82944	56832
SOAP client	97792	80896	64512	47104
Publish	92160	74752	65024	49152
Subscribe	85504	71168	58368	46080
Publish & Subscribe	105984	86016	74752	56320
Service Discovery Configurations				
SLP Lookup	85504	68608	53248	36352
SLP Register	80896	65536	48128	33792
SLP Lookup & Register	103936	83456	65024	45056
UPnP Lookup	80384	64724	56320	39424

Table 1. Size of component configurations in ReMMoC

The results illustrate that the possible configurations are suited to mobile and embedded devices with limited memory resources, as minimum configurations of the binding framework, service discovery framework and individual implementations of these total less than 100Kbytes. For example, the reflective ARM measurements of IIOP client, SOAP client, subscribe, UPnP lookup and SLP lookup are all individually less than 100Kbytes. These are comparable to related systems. For example, the non-reflective ARM IIOP client implementation (55K) compares with the 29K SH3 CORBA client personality of the Universal Interoperable Core (UIC) implementation [Roman, 01] and the 48K non-pluggable GIOP client Zen implementation [Klefstad et al, 02], which have similar capabilities. Similarly, the IIOP Client/Server personality compares with the full UIC CORBA implementation of 44.5K. The table also illustrates the actual cost in terms of extra memory requirements of the reflective personalities utilised in the ReMMoC framework as opposed to their corresponding non-reflective counterparts. For the implemented configurations (ARM) this ranges between an extra 23.5K and 56K. The storage of a type library and an additional 20 lines of C++ code for each component in the configuration, accounts for the extra memory cost. The size of each type library is dependent on the complexity of interface descriptions used on that component. Therefore, the cost per component varies. Our results show that configurations can be created that fit on devices with limited capacity and still retain the dynamic inspection and reconfiguration described in the previous sections.

4.2 Functionality in a Real World Mobile Scenario

To illustrate that the ReMMoC platform can perform its primary function of discovering and interoperating with heterogeneous services, a scenario consisting of a set of differing mobile applications was developed. Figure 9 illustrates the test-bed scenario that was created to evaluate the functionality of the ReMMoC platform. In this scenario, three applications exist. The first of these is a mobile sport news service that is accessed by local users who wish to find out the latest sport news in their current location. This information comes from two sources: i) a publish-subscribe service that publishes sport stories, and at a different time: ii) an IIOP service that lets you obtain the current feature story for a given topic (football, cricket etc). The second application is a weather service that displays the latest weather information to the user; in the test environment the UPnP weather device captures the current weather conditions and publishes this through a SOAP service. The final application is a chat application between multiple users (mobile devices) within the environment. Each device presents a chat publish-subscribe system; the user publishes their input and interested parties (members of the chat) subscribe to receive this input. The test environment includes Compaq iPaq h3870 Pocket PCs running the Windows CE 3.0 operating system; each device is fitted with a Wireless LAN PC Card so that it can connect to the local wireless network. With the exception of the chat services the remainder operate on desktop machines running the Windows 2000 operating system; the SOAP services were developed upon the Apache SOAP 2.0 implementation and the IIOP services were developed upon ORBacus 4.05.

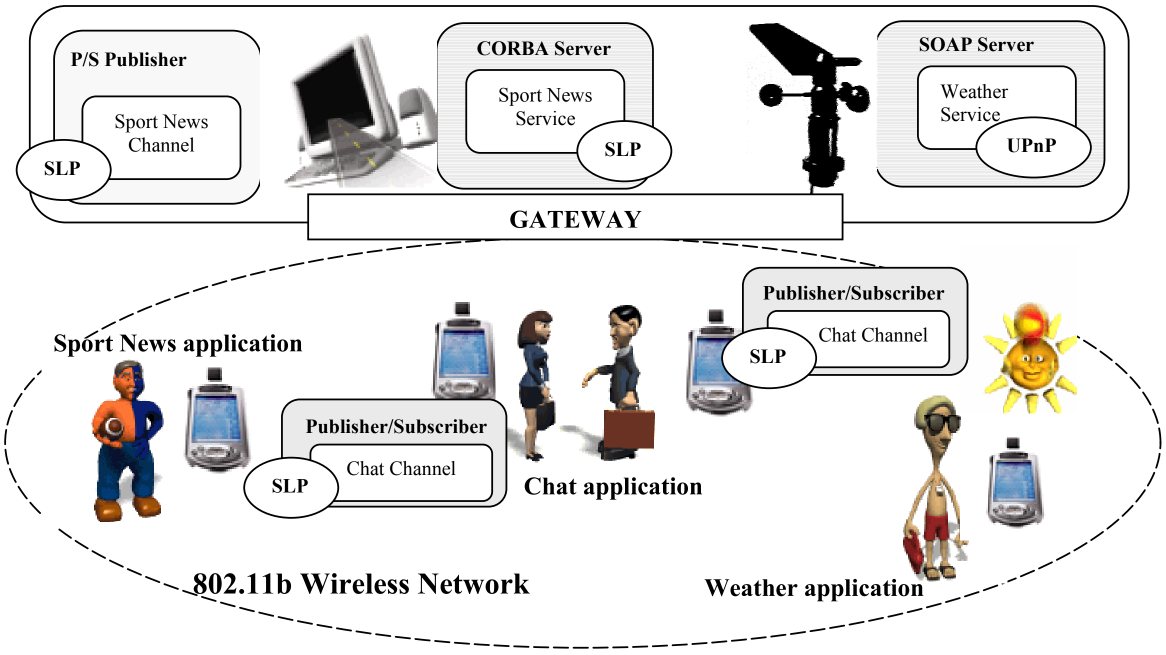

Figure 9. Test bed for ReMMoC Evaluation

To illustrate the functionality of the ReMMoC platform, we step through the operation of one of the applications in the scenario. The sport news application was developed to discover sport news stories and display them to the user. Firstly, the application invokes the lookup interface of the service discovery framework requesting services of type "*Sport_News*". UPnP and SLP are in use in this scenario, therefore the framework is configured to discover services advertised by these. Initially, only the IIOP service is available and the binding type is configured to an IIOP client type and an invocation requesting the latest football story is made. Later, only the publish-subscribe service is available (to simulate heterogeneity), therefore a different middleware type is detected when the lookup is next called. Following this the binding framework is reconfigured to a subscribe personality and the application subscribes to the latest football stories. This shows that an application can continue operating with services in different locations that are implemented on different middleware types. Similar tests were carried out with the other applications to demonstrate the functionality of the ReMMoC platform. These examples illustrate the ability to discover services across different discovery platforms and interoperate with them through the appropriate binding.

However, the code to make the service requests and reconfiguration of the two frameworks is implemented within the application code. This is not the most suitable solution; knowledge of the frameworks is required and code is replicated across each application implementation. Furthermore, knowledge of the type of middleware that may be encountered is needed. A higher-level framework, to manage this complexity is needed in order to simplify the development of applications of this type; this is discussed further in future work.

5. FUTURE WORK

The main goal of the ReMMoC project is to allow mobile applications to be developed independently of the heterogeneous platform types of the mobile services that they intend to interact with. At present, developing mobile applications requires knowledge of the underlying framework structure and possible middleware bindings that may be encountered; but these are not known until run-time. Therefore, work is required on the specification of an XML-based language for describing the application's requirements from mobile services. The ReMMoC top level framework will then receive these XML requests and perform the appropriate reconfiguration to find and interact with the services that meet these requests, while managing the configuration of the required middleware functionality. We also aim to extend our test bed to include a greater array of embedded devices and services, in order to better demonstrate the operation of this platform in a heterogeneous environment.

The evaluation of memory use has illustrated that single middleware personalities can exist on mobile devices. However, each device cannot store every possible middleware component that may be needed. Therefore, a method for dynamically downloading components when needed is required. Furthermore, techniques to ensure the component is available to start-up before it needs to be used, e.g. caching, are necessary

Finally, the work does not address a number of key issues in distributed systems development that are important within this application domain. Firstly, security needs to be added to the system in order to deal with access control of services. Furthermore, resource management to control use of memory, CPU and battery power is important. Also, the use of context information for driving underlying adaptation needs to be considered, i.e. how best to integrate this information with the middleware and how to deal with conflicting requests. We envisage that these orthogonal aspects will be integrated into the platform through use of separate component frameworks.

6. RELATED WORK

The mobile computing domain offers a number of new challenges for middleware to overcome. In this section we examine existing mobile middleware platforms that have been developed to support these new requirements, which include: methods for overcoming the poor characteristics of wireless networks, lightweight platforms for devices with limited memory and techniques for adapting to changing environmental context.

6.1 Asynchronous Mobile Middleware

The properties of a wireless network means that the mobile device may become disconnected involuntarily, or otherwise choose to become disconnected to save resources such as battery power. Furthermore, 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. For example, the Rover platform [Joseph et al, 95] was one of the very first to address this issue; the toolkit provides queued remote procedure calls that allows an application to continue making invocations asynchronously while disconnected from the network. Other asynchronous styles include publish-subscribe systems and tuple spaces. Within a publish-subscribe system, interaction takes the form of event notification; namely, consumers register for the events they are interested in and are informed when they occur. Logically, the two parties do not have to be connected simultaneously to interact. Examples of these are Elvin [Segall, 98], Siena [Carzaniga et al, 01], the Cambridge Event Architecture [Bacon et al, 00] and Gryphon [IBM, 98]. However, these platforms were designed for fixed networks and do not take into account the dynamic connection of mobile hosts. This has enforced the emergence of some preliminary solutions. For example, Elvin has been extended to incorporate proxy servers to support the persistency of events, so that clients who disconnect repeatedly do not lose events; but it requires that clients connect to the same proxy, which cannot be guaranteed in mobile networks. An alternative is JEDI [Cugola, 01], which includes a dynamic tree of dispatchers (the client can reconnect to any) for ensuring publish-subscribe information is retained as members connect and reconnect. Nevertheless, both of these rely on centralised entities holding event information, which cannot be guaranteed within ad-hoc wireless networks. Consequently, STEAM [Meier et al, 02] is a scalable, publish-subscribe system designed to operate in ad-hoc networks; the platform is based upon the concepts of group communication with publishers and subscribers belonging to the same group. The communication is scaled by the proximity of publisher to subscriber; any subscribers out of range do not receive the events.

The tuple space is an alternative asynchronous communication model that is effectively a shared distributed memory spread across all participating hosts that processes can concurrently access; hence communication is decoupled in time and space. The L^2imbo platform [Davies et al, 98] is based upon the classic tuple space architecture but includes a number of extensions for operation within a mobile environment. Multiple tuple spaces can be created and used, removing the need for all operations to go through a central global tuple; this is an important factor in an environment where communication links are unreliable. Furthermore, QoS attributes can be added to a tuple, including delivery deadline allowing the system to re-order to make the best use of network connectivity.

Alternative technologies are JavaSpaces [Waldo, 98], T-spaces [Wyckoff et al, 98] and Lime [Murphy et al, 01], however, none of these adapt their behaviour like L^{2}imbo, to support context changes.

6.2 Adaptive Middleware

Established middleware technologies and those described in the previous section offer a fixed black-box implementation whose underlying structure and behaviour is hidden from the programmer and cannot be altered at run-time to cope with changes that occur in the mobile environment. Therefore, [Blair et al, 01] believe that future middleware platforms, for domains such as multimedia and mobile computing, should be *configurable* to match the requirements of a given application domain and dynamically *reconfigurable* to enable the platform to respond to changes in its environment.

Recently, a group of *reflective middleware* technologies have emerged to meet these requirements: OpenORB [Blair et al, 01], DynamicTAO [Kon et al, 00], Multe-ORB [Kristensen & Plagemann, 00] and OpenCORBA [Ledoux, 99]. A reflective system is one that provides a representation of its own behaviour that is amenable to inspection and adaptation, and is *causally connected* to the underlying behaviour it describes. The key to the approach is to offer a *meta-interface* supporting the inspection and adaptation of the underlying structure. However, these existing systems are built for application domains, such as multimedia and real-time; they do not address the issue of middleware heterogeneity in mobile computing. Consequently, [Roman, 01] identifies that the key property in supporting mobile computing is the ability to seamlessly interoperate with the range of ubiquitous devices that are encountered by the mobile device as it changes location. Therefore, the Universal Interoperable Core [Roman, 01] has been developed; this reflective middleware is loosely based on the reconfiguration techniques of DynamicTAO. The platform can change between different middleware *personalities* e.g. a SOAP client, a CORBA server and a SOAP server. The implementation of UIC concentrates on synchronous middleware styles and does not implement all paradigm types that could be encountered in a ubiquitous environment, i.e. it is likely that asynchronous platforms would be as prominent given their suitability to the environment, nor does it address the issue of heterogeneous discovery protocols.

Furthermore, middleware and applications need to be aware of context information to support adaptation. Work at University College London [Capra et al, 01a] examines the use of reflection in managing a repository of application meta-data that stores each application's requirements for adaptation. They then use reflection to inspect and adapt this so that behaviour can be altered dynamically. They also look at managing the conflicting requests for adaptation based on the amount of differing context information available [Capra et al, 01b].

6.3 Others

Alternatively, other projects have extended traditional platforms to make them effective over wireless networks. For example, ALICE [Haahr et al, 00] presents a layered architecture for managing the movement of mobile hosts and ensures that CORBA connections remain established transparently. Alternatively, DOLMEN [Liljeberg et al, 97] offers a special Light-Weight Inter-ORB Protocol for object communication over a wireless link. RAPP [Seitz et al, 98] allows proxies to be inserted between distributed CORBA objects to manage poor levels of network service and disconnection. Finally, [Reinstorf et al, 01] implements a session layer that allows CORBA invocations to be made over the Wireless Application Protocol.

The memory footprint size of a middleware implementation is often large, especially that of traditional types like CORBA, RMI and DCOM. This becomes a critical problem in the domain of mobile computing where mobile and embedded devices have a small, fixed amount of ROM and RAM available. Therefore, middleware platforms designed for mobile devices must ensure they minimise the amount of memory they utilise. OrbacusE and e*ORB are examples of commercially available CORBA ORBs optimised for memory size and performance. Nevertheless, these remain static over time and cannot alter their behaviour and performance when the available resources change. Consequently, Zen [Klefstad et al, 02] is a real-time CORBA ORB that reduces the memory footprint by allowing the selection of a minimal subset of ORB capabilities used by an application, this can then be altered dynamically when the applications requirements change. However, due to middleware heterogeneity in the mobile environment, utilising multiple minimum footprint platforms is unsuitable. An improved solution is the Universal Interoperable Core [Roman, 01], which is an example of a platform whose configuration can be dynamically altered over time to offer different functionality, while minimising the memory resources used.

7. CONCLUDING REMARKS

We have identified that a middleware for mobile computing must provide support to applications for discovering and interoperating with heterogeneous services in the mobile environment. The middleware must dynamically adapt its behaviour to perform this primary function. We have proposed the use of reflection and components as the key underlying technologies. The middleware itself should offer dynamic reconfiguration to provide the best level of service to the application based upon context changes such as network QoS, but fundamentally it should reconfigure to overcome the level of heterogeneity in mobile environments. A mobile middleware platform for supporting general classes of mobile application must be able to discover services advertised by a range of service discovery protocols and reconfigure between all possible middleware paradigms in order to interact with these newly found services.

This paper presents ReMMoC, a configurable and dynamically reconfigurable middleware platform that supports interoperation in heterogeneous mobile environments. The use of component frameworks within this design offers a technique to ensure that only valid component implementations are utilised in the platform's operation. As shown by the discussion of the frameworks, reflection can be used to make changes to the component framework to allow a new "personality" to be plugged-in, or fine-grained component changes to be made depending on environmental context changes. The functionality of this platform has been illustrated in a real world mobile scenario. Finally, a middleware platform for mobile and embedded devices must minimise its memory size, so memory resources are not exhausted and its components can be passed easily across networks; the component configuration sizes of the ReMMoC implementation compare with other minimum middleware implementations, but are larger due to extra reflective functionality.

ACKNOWLEDGEMENTS

The research described in this paper is funded by UK Bell Labs, Lucent Technologies together with the EPSRC. The authors would also like to acknowledge the work carried out by Thirunavukkarasu Sivaharan on the design and implementation of the component configuration for a publish-subscribe service.

8. REFERENCES

[Bacon et al, 00] Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O. and Spiteri, M. *"Generic Support for Distributed Applications"*. IEEE Computer, pp 68-76, March 2000.

[Blair et al, 01] Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. "*The design and implementation of Open ORB 2*". IEEE Distrib. Syst. Online, 2(6) , Sept 2001.

[Capra et al, 01a] Capra, L., Emmerich, W. and Mascolo, C. **"***Reflective Middleware Solutions for Context-Aware Applications*". In Proc. of REFLECTION 2001- The Third International Conference on Meta-level Architectures and Separation of Crosscutting Concerns, September 2001.

[Capra et al, 01b] Capra, L., Emmerich, W., and Mascolo, C. "*A Micro-Economic Approach to Conflict Resolution in Mobile Computing*". Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE-10), Charleston, South Carolina, USA. November, 2002.

[Carzaniga et al, 01] Carzaniga, A., Rosenblum, D. and Wolf, A. "*Design and Evaluation of a Wide-Area Event Notification Service*". ACM Transactions on Computer Systems, 19(3), pp 332-383, 2001.

[Clarke et al, 01] Clarke, M., Blair, G., Coulson, G and Parlavantzas, N. "*An Efficient Component Model for the Construction of Adaptive Middleware*". In Proceedings of Middleware 2001, Heidelberg, Germany. November, 2001.

[Cugola, 01] Cugola, G., Di Nitto, E., and Fuggetta, A. "*The JEDI event-based infrastructure and its application to the development of the OPSS WFMS*". IEEE Transactions on Software Engineering, 9(27), pp827-850, September 2001.

[Davies et al, 98] Davies, N., Friday, A., Wade, S. and Blair, G. S. "*L^2imbo: 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.

[Haahr et al, 00] Haahr, M., Cunningham, R. and Cahill, V. "*Towards a Generic Architecture for Mobile Object-Oriented Applications".* SerP 2000: Workshop on Service Portability, San Francisco, December 2000.

[IBM, 98] IBM research. *"Gryphon: An Information Flow Based Approach to Message Brokering*". http://researchweb.watson.ibm.com/gryphon/home.html, 1998.

[Joseph et al, 95] Joseph, A., deLespinasse, A., Tauber, J., Gifford, D. and Kaashoek, M. *"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.

[Kagal et al, 01] Kagal, L., Korolev, V., Chen, H., Joshi, A., and Finin, T. "*Centaurus: A framework for intelligent services in a mobile environment*". In Proceedings of the International Workshop on Smart Appliances and Wearable Computing (IWSAWC), April 2001.

[Klefstad et al, 02] Klefstad, R., Rao, S., and Schmidt, D. "*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.

[Kristensen & Plagemann, 00] Kristensen, T. and Plagemann, T. *"Enabling Flexible QoS Support in the Object Request Broker COOL", Proceedings of International Workshop on Distributed Real-Time Systems* (IWDRS 2000), April 2000.

[Kon et al, 00] Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L., and Campbell, R. "*Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB*". In Proceedings of Middleware 2000, ACM/IFIP, April 2000.

[Ledoux, 99] Ledoux, T. *"OpenCorba: a Reflective Open Broker".* In 2nd International Conference on Reflection and Meta-level Architectures, St. Malo, France, July 1999.

[Liljeberg et al, 97] Liljeberg, M., Raatikainen, K., Evans, M., Furnell, S., Maumon, K., Veldkamp, E., Wind, B., Trigila, S. **"***Using CORBA to Support Terminal Mobility".* In Proceedings of TINA '97, 1997.

[Meier et al, 02] Meier, R. and Cahill, V. "*STEAM: Event-Based Middleware for Wireless Ad Hoc Networks*". In Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02), Vienna, Austria, 2002.

[Murphy et al, 01] Murphy, A., Picco, G. and Roman, G. "*LIME: A Middleware for logical and Physical Mobility*". In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), May 2001.

[Preuss, 02] Preuss, S. "*JESA Service Discovery Protocol*". In Proceedings of Networking 2002, pp 1196-1201, Pisa, Italy, May 2002.

[Reinstorf et al, 01] Reinstorf, T., Ruggaber, R., Seitz, J. and Zitterbart, M. "*A WAP-Based Session Layer Supporting Distributed Application in Nomadic Environments*". In Proceedings of Middleware 2001, Heidelberg, Germany, November 2001.

[Roman, 01] Roman, M., Kon, F. and Campbell, R. H. *"Reflective Middleware: From Your Desk to Your Hand".* IEEE DS Online, Special Issue on Reflective Middleware, 2001.

[Segall, 98] Segall, B and Arnold, D. "*Elvin has left the building: a publish/subscribe notification service with quenching*". In Proceedings of AUUG97, September 1997.

[Seitz et al, 98] Seitz, J., Davies, N., Ebner, M. and Friday, A. "*A CORBA-based Proxy Architecture for Mobile Multimedia Applications*". In Proceedings of the 2nd IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS '98), Versailles, France. November,1998.

[Szyperski, 98] Szyperski, C. "*Component Software: Beyond Object-Oriented Programming*". Addison Wesley, 1998.

[Waldo, 98] Waldo, J. "*Javaspaces specification* 1.0". Sun Microsystems Technical report, March 1998.

[Wyckoff et al, 98] Wyckoff, P., McLaughry, S., Lehman, T. and Ford, D. "*Tspaces*". IBM Systems Journal, 37(3), pp 454-474, 1998.

APPENDIX A. ReMMoC Component Framework Meta-Interfaces

ICFMetaInterface:

```
HRESULT get_exposed_interfaces([out] IID* intfseq[], [out] int* count);
HRESULT get_comp_interfaces([out] IID* intfseq[], [in] IUnknown *Comp,
            [out] int* count);
HRESULT get_interactions_list([in] IUnknown *pIUnkSink,REFIID riid,
                        [out] FunctionInfo* list[]);
HRESULT call_operation([in] IID intf, [in] IUnknown* pIUnkSink, [in] const
                char *name, [in] ParameterInfo argseq[], [in] int cparams,
                [out] VARIANT* result);
```

ICFMetaInterception:

```
HRESULT Add_Interceptor([in] IID iid, [in] const char *DLLName, [in] IUnknown
            *pComp, [in] const char *methodName, [in] char* type);
HRESULT Delete_Interceptor([in] IID iid, [in] const char *methodName,
            [in] IUnknown *pComp, [in] char* type
HRESULT ViewIntMethods([in] IID iid, [out] char *methodNames[], [in] IUnknown
            *pComp, [in] unsigned char *type);
```

ICFMetaArchitecture:

```
HRESULT get_Components([out] IUnknown  **ppComps[], [out] int *pcElems);
HRESULT get_Bound_Components([in] IUnknown* comp, [out] IUnknown** ppComps[],
            [out] int *pcElems);
HRESULT get_Bindings([in] IUnknown * comp, [out] unsigned long  **ppConnInfo,
            [out] int *pcElems);
HRESULT Bind([in] IUnknown *pIUnkSource, [in] IUnknown *pIUnkSink, [in] REFIID
            iid, [out] unsigned long *pConnID);
HRESULT unBind([in] unsigned long connID);
HRESULT Insert_Comp([in] CLSID clsid, [out] IUnknown **ppIUnknown, [in] const
            char *name);
HRESULT Remove_Comp([in] IUnknown *pIUnknown, [out] int *pcElems);
HRESULT Expose_Interface([in] IID rintf);
HRESULT ReplaceCFConfiguration([in] IUnknown *pIUnkSource[], [int] int cCmps ,
            [in] IID intfseq[], [in] int cIntfs);
```