ReMMoC: A Reflective Middleware to support Mobile Client Interoperability

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

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

Abstract. Mobile client applications must discover and interoperate with application services available to them at their present location. However, these services will be developed upon a range of middleware types (e.g. RMI and publish-subscribe) and advertised using different service discovery protocols (e.g. UPnP and SLP) unknown to the application developer. Therefore, a middleware platform supporting mobile client applications should ideally adapt its behaviour to interoperate with any type of discovered service. Furthermore, these applications should be developed independently from particular middleware implementations, as the interaction type is unknown until run-time. This paper presents ReMMoC, a reflective middleware platform that dynamically adapts both its binding and discovery protocol to allow interoperation with heterogeneous services. Furthermore, we present the ReMMoC programming model, which is based upon the Web Services concept of abstract services. We evaluate this work in terms of supporting mobile application development and the memory footprint cost of utilising reflection to create a mobile middleware platform.

1 Introduction

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. The popularity of this field, driven by new wireless network and mobile device technologies, has produced a variety of innovative application types (e.g. context aware applications, m-commerce, ad-hoc communities, mobile gaming and many others). To support these, new middleware is emerging that addresses the problems of weak connection, limited device resources and fluctuating network QoS inherent to the domain. These solutions range from extensions to well-established middleware for fixed networks [1–3] and middleware designed explicitly to support mobile applications [4–6]. However, the different solutions introduce the problem of middleware heterogeneity [7]. These platforms offer different communication paradigms, including: remote method invocation, publish-subscribe, message-oriented and tuple

spaces. Furthermore, implementations of individual paradigms vary e.g. SOAP and IIOP for remote method invocation. Therefore, mobile clients implemented upon one middleware type (e.g. SOAP) will not interoperate with discovered services implemented upon different platforms (e.g. IIOP or Publish-Subscribe). As an example, a tourist guide client implemented using publish-subscribe can only interoperate with matching tourist information publishers. Furthermore, 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, services are advertised using one of the contrasting service discovery protocols. At present, there are four main service discovery technologies: Jini, Service Location Protocol (SLP), Universal Plug and Play (UPnP) and Salutation. In addition, new technologies are emerging to better support the discovery of services in mobile environments (e.g. JESA [8] & Centaurus [9]) and across wireless ad-hoc network types (e.g. SDP in Bluetooth and Salutation Lite). Utilising only one of these technologies to discover services will mean that services advertised by the other types will be missed. For example, a set of devices within a room (e.g. lights, video, CD player) advertising their services using UPnP cannot be used by a mobile device looking for services using SLP. 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.

We argue that adaptive middleware is required to support the interoperation of mobile clients with heterogeneous services. Using this approach, the middleware should alter its behaviour dynamically to: i) find the required mobile services irrespective of the service discovery protocol and ii) interoperate with services implemented by different middleware types. We advocate reflection and component technology as well suited techniques to develop middleware with these capabilities. Reflection is a principled method that supports introspection and adaptation to produce configurable and reconfigurable middleware.

For an application to dynamically operate using different middleware implementations it must be programmed independently from them. Hence, an abstract definition of the application services functionality is required. The mobile client application, which requests this service, can then be developed using this interface in the style of IDL programming. A request of the abstract service is mapped at run-time to the corresponding concrete request of the middleware implementation. The emerging Web Services Architecture includes a Web Services Description Language (WSDL) that provides this format of abstract and concrete service definition. We propose that WSDL offers a suitable programming model for such a reflective middleware.

In this paper, we document the design and implementation of a reflective middleware platform, named ReMMoC (Reflective Middleware for Mobile Computing), which combines reflective middleware and the WSDL programming model to provide a solution to the problem of interoperation from mobile clients. Section 2 presents a typical mobile scenario to illustrate the heterogeneous properties of

the mobile environment. The concepts of reflection, component technologies and component frameworks used by ReMMoC are then described in section 3. An overview of ReMMoC is presented in section 4 and a description of the mapping of middleware paradigms to WSDL is given in section 5. Section 6 evaluates the performance of ReMMoC in supporting a typical mobile application and related work in the field of mobile middleware is identified in section 7. Finally, overall conclusions and future work are described in section 8.

2 Mobile Scenario

In this section we present a mobile computing scenario to illustrate middleware heterogeneity that exists in the mobile domain. In the example, three application services are available to mobile users at two 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 communicate with one another.

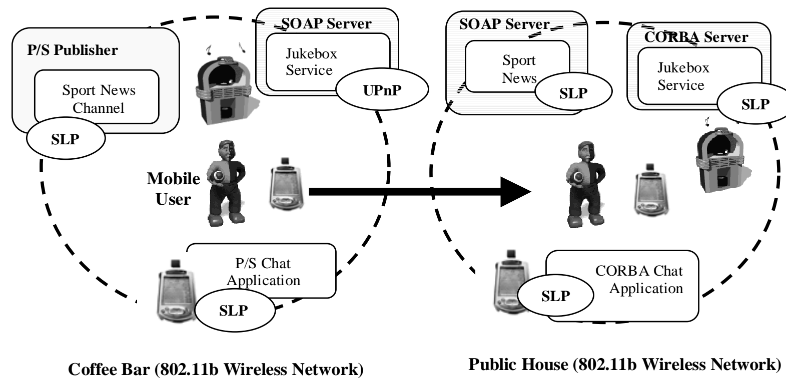

Fig. 1. An mobile computing scenario, populated with heterogeneous middleware.

Figure 1 illustrates two locations (a coffee bar and a public house) in the session of a mobile user and the mobile services that can be interacted with. 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. If fixed middleware were to be used, then two separate applications and middleware implementations would be needed on the device.

Similarly, the chat applications and jukebox services are implemented using different middleware types. However, this is not the only type 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 at the public house are discoverable using SLP and the services at the coffee bar can be found using both UPnP and SLP. If the mobile user utilises only one service discovery protocol then they may miss some available resources and in the worst-case scenario find none.

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 Component Model

3.1 Background on OpenCOM

OpenCOM [10] 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. 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 and disconnect components. Furthermore, a system graph of the components currently in use is maintained to support the introspection of a platform's structure (using the IMetaArchitecture interface).

This component model is used to construct families of middleware. More specifically, each middleware is constructed as a set of configurable component frameworks (more detail on the component framework concept is provided in section 3.2) and reflection is used to discover the current structure and behaviour, and to enable selected changes at run-time. The end result is flexible middleware that can be specialised to domains including multimedia and real-time systems, or in our case mobile computing.

3.2 OpenCOM Component Frameworks

A component framework (CF) is defined as a collection of rules and contracts that govern the interaction of a set of components [12]. The motivation behind component frameworks is to constrain the design space and the scope for evolution. A component framework in OpenCOM is itself an OpenCOM component

that maintains internal structure (a configuration of components) to implement its service functionality. The design of these component frameworks is based upon the concepts of composite components proposed by OpenORB [11]. Therefore, component frameworks can be composed, replaced and connected together in the same manner as components. To provide this capability, each OpenCOM CF implements the base interfaces of an OpenCOM component (IMetaInterface, ILifeCycle, IConnections) in addition to its own interfaces and receptacles. The interfaces and receptacles of internal components can be exposed to create these. The architecture of a component framework is shown in figure 2.

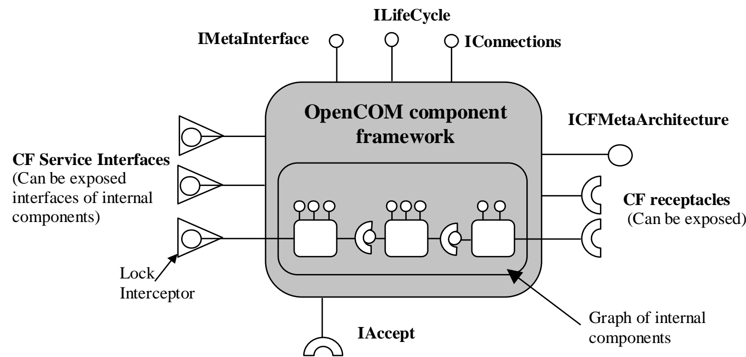

Fig. 2. An OpenCOM component framework.

To inspect component configurations, the OpenCOM runtime IMetaArchitecture interface examines the external structure of a component or CF (i.e. what it is connected to). However, it does not inspect or dynamically adapt the internal structure of a component framework. Therefore, every CF implements the ICFMetaArchitecture interface; this provides operations to inspect the internal structure and change the component configuration. To implement this interface a graph of local components is maintained, which is simply a view of a subset of the OpenCOM runtime system graph to avoid replicating data.

A component framework constrains the configuration of components to a valid implementation within its domain. To enforce this policy, each component framework implements a receptacle called IAccept. When a change to the existing implementation has been made, a call to the IAccept interface is performed. This executes a check of the component architecture; if the Accept component verifies the architecture then 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 can be dynamically changed. The implementation may have no checking (no component connected), simply check against a list

of configurations (described in XML) or alternatively incorporate architectural style rules proposed by [28].

Finally, if a change to the configuration is attempted while one or more service calls of the component framework are executing then the results of these invocations would be compromised or lost. Therefore, each framework utilises a readers/writers lock to access the local CF graph. Standard interface calls access the lock as a reader (there can be multiple concurrent readers) and every call to alter the CF configuration, accesses the lock as a writer (a single writer accesses the lock when there are no readers). The algorithm to implement this property is a standard readers/writers solution with priority for readers. To enforce this, every exposed interface automatically has an interceptor, to access and release the lock, attached.

4 The Design and Implementation of ReMMoC

4.1 Overview

This section describes ReMMoC, 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 only two 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 frameworks are illustrated 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 frameworks is given 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 ReMMoC component, seen in figure 3, performs reconfiguration management and provides a generic API to develop mobile applications upon (see section 5.3).

4.2 The Binding Component Framework

The primary function of the binding framework is to interoperate with heterogeneous mobile services. 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. Different middleware paradigms, synchronous or asynchronous (e.g. tuple spaces, media streams, RPC,

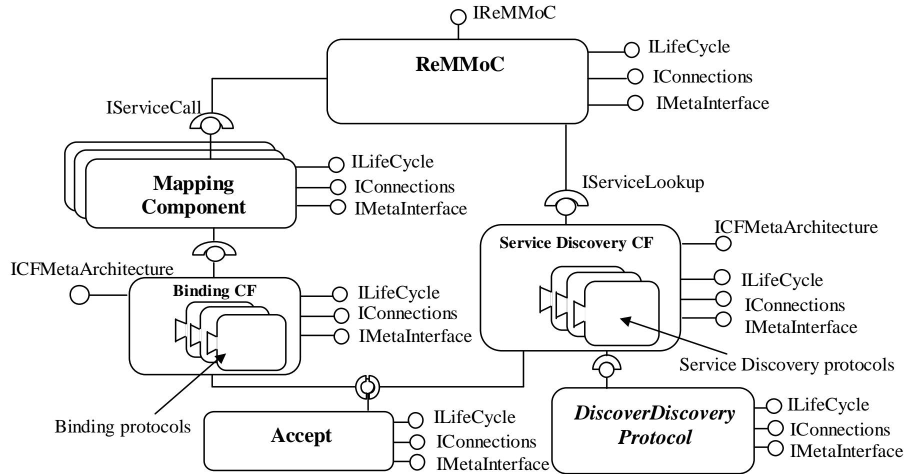

Fig. 3. Overview of the ReMMoC platform.

publish-subscribe or messaging), can be plugged into the binding framework if they have been implemented using OpenCOM components.

Within the binding framework changes are made at two distinct levels. Firstly, each binding type implementation can be replaced; e.g. a SOAP client is replaced by a publish-subscribe subscriber (illustrated in figure 4). This dynamic reconfiguration is performed by receiving information from the service discovery framework describing the type of binding; an XML description of the component configuration for this binding is then parsed to create the new configuration. Hence, new binding protocols can be dynamically added to the framework at a future date. Multiple personalities can also be created, e.g. 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. Secondly, fine-grained changes to each configuration can be made in light of environmental context changes, such as those involving quality of service, or changes in the applications requirements. For example, an application may require IIOP server side functionality, in addition to the existing client side; therefore components implementing server side functionality are added. In order to test and evaluate the binding framework, we have implemented IIOP client and server, SOAP client and Publish-Subscribe personalities.

4.3 The Service Discovery Framework

The Service Discovery framework allows services that have been advertised by different service discovery protocols to be found. The framework is configured to discover protocols currently in use in the environment. For example, if SLP is in use, the framework configures itself to an SLP Lookup personality. However, if SLP and UPnP are found then the frameworks configuration will include

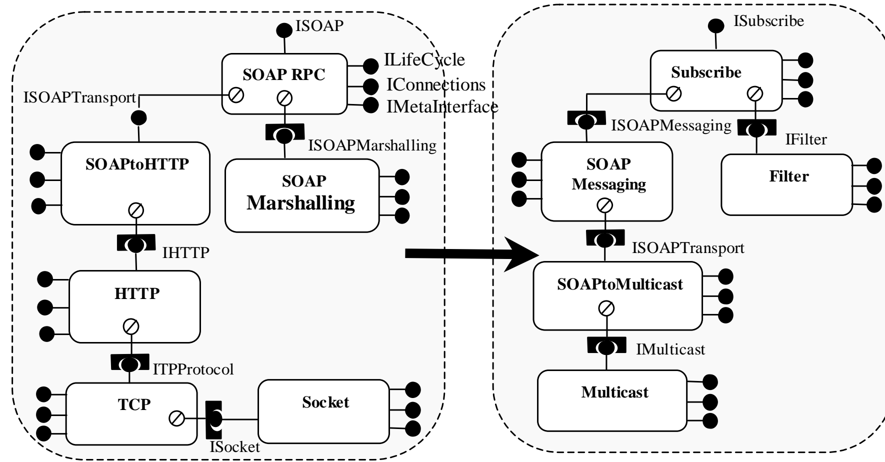

Fig. 4. A dynamic reconfiguration from a SOAP client to a subscriber implementation.

component implementations to discover both. Like the Binding CF, fine-grained component changes can be made. 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.

The service discovery framework offers a set of generic service discovery methods through the IServiceLookup Interface. This includes a generic service lookup operation that returns the information from different service discovery protocol searches in a generic format. For example, a lookup of a weather service across two discovery configurations, e.g. UPnP and SLP, returns a list of matched services from both types. It is this information (the description of the service returned by the lookup protocol) that is used to configure the binding framework.

Initially, the discovery protocol(s) that are currently in use at a location must be determined. The DiscoverDiscoveryProtocol component, which is plugged into the framework, tests if individual service discovery protocols are in use, either upon a synchronous request or by continuously monitoring the environment and generating an event on detection. Continuous monitoring will quickly use up resources (e.g. battery power); therefore in some cases synchronous checking may be appropriate. The service discovery framework utilises this behaviour to automatically reconfigure itself. Other methods for discovering discovery protocols, not currently included in the implementation, may utilise the devices context information, e.g. if the device is currently using a Bluetooth connection then an SDP personality is configured. Furthermore, the middleware 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.

We have implemented the service discovery framework with two service lookup protocol implementations: SLP and UPnP, allowing 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 new discovery protocols to be dynamically integrated into the framework at a later date. This requires a new version of the DiscoverDiscoveryProtocol component, which can detect the new protocol, to be plugged into the framework.

5 The ReMMoC Programming Model

5.1 Background on Web Services

The web services architecture [13] consists of three key roles: a service provider, a service requestor and the discovery agency, which the requestor uses to find the service description. Each service is described in WSDL [14]; this is an XML format for documenting the exchange of messages (containing typed data items) between the service requestor and service provider. The key property of WSDL is that it separates the abstract description of service functionality from the concrete details of the service implementation. Hence, the aim of Web Services is to allow different service providers to implement an abstract service description upon their chosen concrete middleware binding. For example, a news service may be implemented using SOAP by one vendor while another may use publish-subscribe.

In our context, WSDL offers the ability to develop mobile clients, based upon agreed abstract service descriptions, thus hiding the developer from the problem of middleware heterogeneity encountered across different locations. Hence this offers an attractive solution to ReMMoC, with the added benefit that WSDL is a recognised international standard. However, the approach does not offer any support to the dynamic adaptation of the underlying concrete implementations as required by our platform. We return to this in section 5.3. Firstly, we illustrate in the next section the mapping of abstract WSDL descriptions to different bindings e.g. RMI and publish-subscribe. This shows that WSDL can be mapped to the diverse paradigms that are encountered within mobile environments.

5.2 Mapping Abstract Services to Concrete Binding Types

In this section we demonstrate how the abstract operations of WSDL can be mapped to two contrasting binding types exposed by a reflective middleware (RMI and publish-subscribe). The following four abstract operations can be described in WSDL. (1) Request-Response (input, output), a service receives a request of its functionality and responds to it. (2) Solicit-Response (output, input), a service provider acts as a service requestor. (3) One-Way (input), a service receives a notification message. (4) Notification (output), a service outputs a notification message.

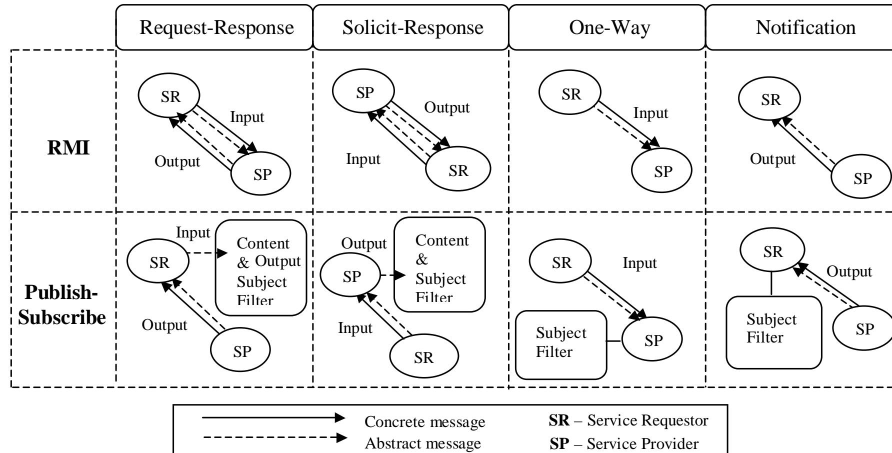

Fig. 5. Mapping WSDL operations to different middleware paradigms.

Figure 5 illustrates how abstract messages (input and output) that constitute each WSDL operation map to the RMI and publish-subscribe communication paradigms. The service requestor is the mobile client. We assume that each paradigm understands the set of types used by the abstract definition. In RMI, the input/output messages of Request-Response and Solicit-Response operations can be mapped directly to the corresponding synchronous RMI messages of SOAP and IIOP. The operation name maps to the method name, the input message to the input parameter list and the output message to the output parameter list. Similarly, Notification and One-Way operations can be mapped as one-way messages e.g. one-way IIOP invocations and asynchronous SOAP messages.

Publish-Subscribe however is an alternative communication paradigm whereby there is no direct message exchange between service requestor and provider. The 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. The request-response operation is a request of a service based upon the input message. The input message can be used to filter published messages and receive the correct event, whose content maps to the output message. The operation name maps to the event subject, while the input message maps to the content filter attributes. Similarly, for Solicit-response the service filters to receive events from other services. For One-way operations and Notifications, services subscribe and publish events based upon subject filtering only, with the content of the concrete message mapping to the abstract message.

5.3 The ReMMoC API

The ReMMoC programming model is based upon the concept of WSDL described abstract services. Application developers must utilise these WSDL defi-

nitions in the style of IDL programming. To maintain a consistent information flow to the application an event-based programming model, that overrides the different computational models of each paradigm, is offered. Each abstract service operation is carried out and its result is returned as an event. For example, if that operation is executed by an RMI invocation or an event subscription the result is always an event. Similarly, service lookup operations return results as events. Figure 6 documents the API of ReMMoC, which consists of operations to: lookup services, lookup then invoke abstract WSDL operations, invoke operations on known services, or create and host service provider operations.

```
interface ReMMoC_ICF : IUnknown {
    HRESULT WSDLGet (WSDLService* ServiceDescription, char* XML);

    HRESULT FindandInvokeOperation (WSDLService ServiceDescription, char*
                        OperationName, int Iterations, ReMMoCOPHandler Handler);

    HRESULT InvokeOperation (WSDLService ServiceDescription, ServiceReturnEvent
        ReturnedLookupEvent, char* OperationName, int Iterations, ReMMoCOPHandler   Handler);

    HRESULT CreateOperation (WSDLService ServiceDescription, ServiceReturnEvent
          ReturnedLookupEvent, char* OperationName, int Iterations, ReMMoCOPHandler Handler);

    HRESULT AddMessageValue(WSDLService *ServiceDescription, char* OperationName,
                  char* ElementName, ReMMoC_TYPE type, char* direction,  VARIANT value);

    HRESULT GetMessageValue(WSDLService *ServiceDescription, char* OperationName,
                  char* ElementName, ReMMoC_TYPE type, char* direction,  VARIANT value);
}
```

Fig. 6. Interface definition for the ReMMoC API.

ReMMoC maps these API calls to the binding framework through the use of a reconfigurable mapping component, illustrated in figure 3. For example, an IIOP mapping component maps abstract WSDL operation calls to IIOP invocations through the interface exposed by the binding framework; it can be replaced by a subscribe mapping component that maps to subscribe requests. These components carry out the mapping of abstract to concrete operations described in section 5.2.

6 Evaluation

6.1 The Cost of Reflection

At present mobile devices have a limited amount of system memory, which can quickly be consumed by users applications; therefore it is important to minimise the amount of memory needed to store a middleware implementation. Utilising reflection to change between protocols allows only the minimum required number of components to be stored on the device, rather than store complete

multi-middleware implementations. 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. Figure 7 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

Fig. 7. Size of component configurations in ReMMoC.

The results in figure 7 illustrate that the 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 IIOP 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 IIOP client implementation (55K) compares with the 29K SH3 CORBA client personality of the Universal

Interoperable Core (UIC) implementation [7] and the 48K non-pluggable GIOP client Zen implementation [15], which have similar capabilities. The difference between the ReMMoC and the UIC value can be attributed to a different processor, as illustrated by ARM personalities being larger than x86 (RISC versus CISC) and using a COM based implementation.

The table also illustrates the cost in terms of extra memory requirements of the reflective personalities as opposed to their 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; hence, 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 properties described in the previous sections.

6.2 Operating in a Mobile Scenario

To illustrate that ReMMoC performs its primary function of discovering and interoperating with heterogeneous services, we evaluated the platform using the scenario in section 2. The test environment included Compaq iPaq h3870 Pocket PCs running the Windows CE 3.0 operating system and fitted with wireless LAN cards. With the exception of the chat services, the remainder executed on desktop machines; SOAP services were developed upon the Apache SOAP 2.0 implementation and the IIOP services were developed using ORBacus 4.05. We successfully created three applications that operated in both locations, irrespective of the underlying middleware implementations. These examples proved it was possible to discover services across different discovery platforms and interoperate with them through the appropriate binding. We briefly describe how ReMMoC dynamically changes to support the sport news application.

In the first location, the sport news service was implemented using a publish channel and advertised using SLP, in the second location the service was implemented as a SOAP service and registered with UPnP. In location one, the service discovery framework detects SLP and configures itself to an SLP lookup personality. When a lookup of the sport news service is executed a single result is returned. The returned information is used by ReMMoC to configure the binding framework to a subscribe personality. The subscribe mapping component is connected to the binding framework, which then creates a filter to receive requested events. In the same lifecycle of the application, the user moves to the second location and UPnP is detected; therefore the discovery framework changes from SLP to UPnP. This time the discovery operation detects the service is implemented by a SOAP binding. Therefore, the binding framework changes from subscribe to a SOAP client. The SOAP mapping component is connected and the abstract operations are communicated as SOAP invocations.

7 Related Work

7.1 Asynchronous Mobile 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, 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 [16] 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 [17], Siena [18] and the Cambridge Event Architecture [19]. 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 [20], 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 [5] 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 [21] 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 [22] and Lime [6], however, none of these adapt their behaviour like L^2imbo, to support context changes.

7.2 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, 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 [11].

Recently, a group of reflective middleware technologies have emerged to meet these requirements: OpenORB [11], DynamicTAO [23], Multe-ORB [24] and OpenCORBA [25]. 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, [7] 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 [7] 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 [4] 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 [26].

7.3 Others

Alternatively, other projects have extended traditional platforms to make them effective over wireless networks. For example, ALICE [27] presents a layered architecture for managing the movement of mobile hosts and ensures that CORBA connections remain established transparently. Alternatively, DOLMEN [3] offers a special Light-Weight Inter-ORB Protocol for object communication over a wireless link. RAPP [2] allows proxies to be inserted between distributed CORBA

objects to manage poor levels of network service and disconnection. Finally, [1] 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 [15] 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 [7], 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.

8 Concluding Remarks and Future Work

Research into mobile middleware has addressed specific concerns such as poor network QoS, weak connection and limited device resources. We argue however that the crucial problem of cascading levels of heterogeneity has been largely ignored. We have proposed that a middleware for mobile computing must provide support to applications for discovering and interoperating with heterogeneous services in the mobile environment. We identify that a marriage of web services with reflective middleware offers a solution to mobile client interoperability. 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. 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.

ReMMoC was designed specifically for the mobile environment and has been fully developed and tested using simple applications e.g. chat, news and stock quote clients across IIOP, SOAP and Publish-Subscribe bindings. We also recognise that the properties of our platform are usable in domains other than mobile computing, hence ongoing work includes an evaluation of this method on larger, complex applications (e.g. Grid computing, ubiquitous computing and intelligent

home environments) and across a range of further middleware bindings including data sharing and tuple spaces.

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. predictive caching based upon context information is an interesting option.

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 the development of additional component frameworks.

References

1. Reinstorf, T., Ruggaber, R., Seitz, J., Zitterbart, M.: A WAP-Based Session Layer Supporting Distributed Application in Nomadic Environments. In Proceedings of Middleware 2001, Heidelberg, Germany, November 2001.
2. Seitz, J., Davies, N., Ebner, M., Friday, A.: A CORBA-based Proxy Architecture for Mobile Multimedia Applications. In Proceedings of the 2nd International Conference on Management of Multimedia Networks and Services, Versailles, France, November 1998.
3. Liljeberg, M., Raatikainen, K., et al.: Using CORBA to Support Terminal Mobility. In Proceedings of TINA 1997.
4. Capra, L., Emmerich, W., Mascolo, C.: Reflective Middleware Solutions for Context-Aware Applications. In Proceedings of REFLECTION 2001, Kyoto, Japan, September 2001.
5. Meier, R., Cahill, V.: STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In Proceedings of the International Workshop on Distributed Event-Based Systems, Vienna, Austria, 2002.
6. Murphy, A., Picco, G., Roman, G.: LIME: A Middleware for logical and Physical Mobility. In Proceedings of the 21st International Conference on Distributed Computing Systems, Arizona, USA, May 2001.
7. Roman, M., Kon, F., Campbell, R. H.: Reflective Middleware: From Your Desk to Your Hand. IEEE Distributed Systems Online, 2(5), 2001.
8. Preuss, S.: JESA Service Discovery Protocol. In Proceedings of Networking 2002, pp 1196-1201, Pisa, Italy, May 2002.
9. Kagal, L., Korolev, V., Chen, H., et al.: 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.

10. Clarke, M., Blair, G., Coulson, G., Parlavantzas, N.: An Efficient Component Model for the Construction of Adaptive Middleware. In Proceedings of Middleware 2001, Heidelberg, Germany. November, 2001.
11. Blair, G. et al.: The design and implementation of Open ORB 2. IEEE Distributed Systems Online, 2(6) , Sept 2001.
12. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison Wesley, 1998.
13. W3C.: Web Services Architecture. W3C Working Draft, http://www.w3.org/TR/ws-arch/, November, 2002.
14. W3C.: Web Services Description Language (WSDL) Version 1.2. W3C Working Draft, http://www.w3.org/TR/wsdl12/, March, 2003.
15. Klefstad, R., Rao, S., 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, Big Island of Hawaii, January, 2003.
16. Joseph, A., deLespinasse, A., Tauber, J., Gifford, D., Kaashoek, M.: Rover: A Toolkit for Mobile Information Access. In Proceedings of the 15th Symposium on Operating Systems Principles, Colorado, U.S., pp 156-171, December 1995.
17. Segall, B., Arnold, D.: Elvin has left the building: a publish/subscribe notification service with quenching. In Proceedings of AUUG97, September 1997.
18. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3), pp 332-383, 2001.
19. Bacon, J., Moody, K., Bates, J., et al.: Generic Support for Distributed Applications. IEEE Computer, pp 68-76, March 2000.
20. Cugola, G., Di Nitto, E., 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.
21. Davies, N., Friday, A., Wade, S., Blair, G. S.: L2imbo: A Distributed Systems Platform for Mobile Computing" ACM Mobile Networks and Applications (MONET), 3(2), pp 143156, August 1998.
22. Waldo, J.: Javaspaces specification 1.0. Sun Microsystems Technical report, March 1998.
23. Kon, F., Roman, M., Liu, P., et al.: Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In Proceedings of Middleware 2000, New York, USA, April 2000.
24. Kristensen, T., Plagemann, T.: Enabling Flexible QoS Support in the Object Request Broker COOL. Proceedings of International Workshop on Distributed Real-Time Systems, April 2000.
25. Ledoux, T.: OpenCorba: a Reflective Open Broker. In 2nd International Conference on Reflection and Meta-level Architectures, St. Malo, France, July 1999.
26. Capra, L., Emmerich, W., Mascolo, C.: A Micro-Economic Approach to Conflict Resolution in Mobile Computing. In Proceedings of the 10th International Symposium on the Foundations of Software Engineering, South Carolina, USA, November, 2002.
27. Haahr, M., Cunningham, R., Cahill, V.: Towards a Generic Architecture for Mobile Object-Oriented Applications. SerP 2000: Workshop on Service Portability, San Francisco, December 2000.
28. Moreira, R., Blair, G., Carrapatoso, G.: Reflective Component-Based & Architecture Aware Framework to Manage Architecture Composition. In 3rd International Symposium on Distributed Objects & Applications. Rome, Italy, September, 2001.