A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments

Paul Grace [a] **Gordon S. Blair** [a] **Sam Samuel** [b]

[a] Computing Department, Lancaster University, Lancaster, UK
[b] Global Wireless Systems Research, Bell Laboratories, Lucent Technologies, Swindon, UK

To operate in dynamic and potentially unknown scenarios a mobile client discovers the local services that match its requirements, and interacts with these to obtain the application functionality. However, mobile environments are populated by heterogeneous mobile service platforms; these range from discovery protocols including SLP, UPnP and Jini to different styles of service interaction paradigms e.g. Remote Procedure Call, Publish-Subscribe and agent based solutions. Therefore given this type of heterogeneity, utilizing single discovery and interaction systems is not optimal as the client will only be able to use the services available to that particular platform. Hence, in this paper we present an adaptive middleware solution to this problem. ReMMoC is a Web-Services based reflective middleware that allows mobile clients to be developed independently of both discovery and interaction mechanisms. We describe the architecture, which dynamically reconfigures to match the current service environment. Finally, we investigate the incurred performance overhead such dynamic behaviour brings to the discovery and interaction process.

I. Introduction

In current mobile applications, users interact with context-based mobile services in both ad-hoc and nomadic wireless networks. For example, querying tourist information services, utilising local business services, collaborating and communicating with other nearby mobile users, and interacting with jukebox players and other computational devices. In these scenarios the client application or mobile user must first discover a service that matches the requirements and then interact with it. To support this behaviour service discovery and interaction platforms have emerged. Generally, these solutions fall into three categories. Firstly, discovery platforms supported by mobile code; examples are Centaurus [1] and Jini [2]. After discovery, the service (either a proxy to the service or the full service) is downloaded onto the mobile device where it then operates. Secondly, the discovery protocol is integrated with a specific interaction protocol, which is used to invoke the service after the service has been discovered. Examples are: Universal Plug and Play (UPnP) [3] with SOAP [4], Salutation [5] with Sun Remote Procedure Call (RPC), and Gaia [6] with Common Object Request Broker Architecture (CORBA) [7]. Thirdly, interaction independent discovery protocols are available e.g. Service Location Protocol (SLP) [8]. These can be integrated with a range of interaction protocols.

However, there is identifiable heterogeneity in these approaches. Heterogeneous discovery protocols (UPnP, Jini, SLP etc.) means that clients using only one discovery protocol will not find all available services as they move from location to location. Furthermore, contrasting implementations of interaction paradigms such as RPC and publish-subscribe, ensures that mobile clients developed upon a single implementation will be unable to interoperate with mobile services implemented upon an alternative. As an example, a tourist guide client implemented using publish-subscribe can only interoperate with matching tourist information publishers. This problem is particularly important to the mobile applications that operate in many locations where the service platform implementations are unknown. Furthermore, the problem is likely to become significantly worse in the future with the emergence of new discovery and interaction protocols.

To address this problem we have developed ReMMoC (Reflective Middleware for Mobile Computing), an adaptive middleware framework, which is independent from particular discovery and interaction protocols. ReMMoC is able to: i) find the required mobile services irrespective of the service discovery protocol and ii) interoperate with services implemented upon different interaction types. The framework monitors the environment and the service types in use and reconfigures itself to mirror the current setup. ReMMoC uses the Web Services abstraction to allow clients to be developed independent from specific service implementation; instead the abstraction is mapped onto the appropriate protocol at run-time.

In this paper, we present the design, implementation and evaluation of ReMMoC. Section 2 presents the overall architecture principles of

components and reflections employed by ReMMoC. The service discovery and binding frameworks are described in section 3. Subsequently, the discovery and interaction abstraction is defined in section 4. Section 5 then evaluates the operation and performance of ReMMoC in supporting typical mobile applications. Finally, related work in this field is identified in section 6, and overall conclusions are drawn in section 7.

II. The ReMMoC Framework

This section describes the design of the reflective middleware framework (ReMMoC), whose key operation is to dynamically adapt discovery and interaction protocols to match the current mobile service environment, and hence overcome platform heterogeneity. This framework is heavily influenced by previous work from Lancaster on reflection and components; we argue that this approach offers an ideal solution to build such a highly dynamic framework. The following section describes the design philosophies that ReMMoC follows. Subsequent sections then document the architectural elements of the framework.

II.A The OpenORB Philosophy

The OpenORB design philosophy [9] promotes a marriage of *reflection*, *component technologies* and *component frameworks*, to develop families of reflective middleware. Components are the building blocks of middleware, where a component is "a unit of composition with contractually specified interfaces, which can be deployed independently and is subject to third party creation" [10]. This technique promotes configurability, re-configurability and re-use at the middleware level. Reflection is then used to provide a principled mechanism to inspect and dynamically adapt the component structure. Finally, component frameworks constrain the design space and the scope for evolution, where a component framework (CF) is defined as a collection of rules and contracts that govern the interaction of a set of components [10].

OpenORB based middleware are built using OpenCOM [11], which is a lightweight, efficient and reflective component model that uses the core features of Microsoft COM to underpin its implementation; these include the binary level interoperability standard, Microsoft's IDL, COM's globally unique identifiers and the IUnknown interface. Each component implements a set of custom interfaces and receptacles. An interface expresses a unit of service provision, a receptacle describes a unit of service requirement and a connection is the binding between an interface and a receptacle of the same type. OpenCOM deploys a standard runtime substrate per address space that manages the creation and deletion of components, acts upon requests to connect/disconnect components and provides service interfaces for reflective operations. The runtime substrate dynamically maintains a system graph of the components currently in use. The explicit maintenance of dynamic dependencies between components provides the support for introspection and reconfiguration of component configurations.

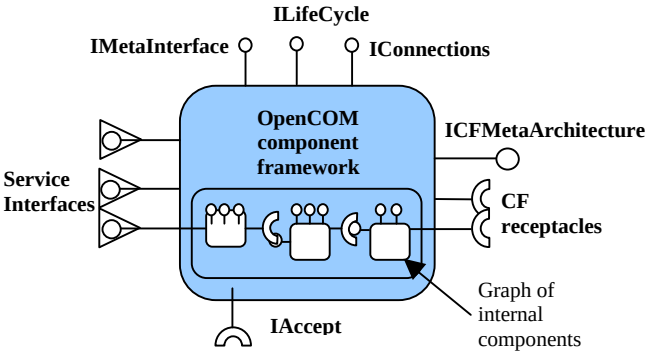

Figure 1: The OpenCOM CF Model

To support the creation of valid software architectures, OpenCOM promotes an additional component framework model [12]. Here, a CF is a single OpenCOM component (seen in figure 1), which contains its own internal structure (a graph of components). Each CF is extended by the ICFMetaArchitecture interface, which provides reflective operations to inspect and dynamically reconfigure the framework's local component architecture.

II.B The ReMMoC Architecture

ReMMoC is designed to reside upon mobile devices for client applications to be developed upon. Hence, the architecture of ReMMoC (illustrated in figure 2) is designed as a minimal set of OpenCOM component frameworks to reduce resource use. ReMMoC is a two-tier architecture consisting of a top-level component framework into which a set of components and component frameworks are then plugged. There are three sections to this top-level framework:

1. The ***concrete middleware section***, which is composed of two component frameworks: (1) a ***binding framework*** for interoperation with mobile services implemented upon different interaction types, and (2) a ***service discovery framework*** for discovering services advertised by a range of service discovery protocols. The

binding framework is configured by plugging in different binding type implementations e.g. SOAP RPC, Event subscriber 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 section.

2. The ***abstract middleware-programming model***, which implements an API for performing service discovery and service interaction independent of protocol implementation.
3. The ***abstract to concrete mapping*** section, which consists of components to map abstract service requests to the current binding and discovery implementations in place.

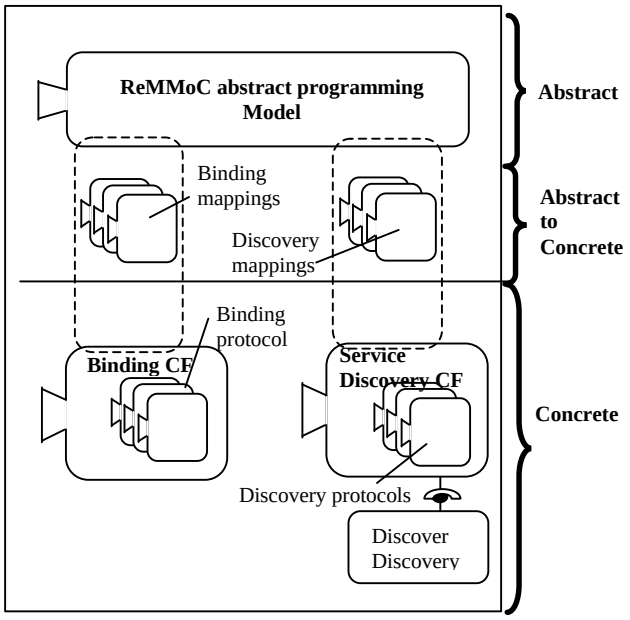

Figure 2: The overall ReMMoC Architecture

ReMMoC is flexible to meet different application developer's requirements. For example, the platform can be configured to just the concrete section, or indeed one of the two component frameworks. This may be required for applications on low resource embedded devices (e.g. wearable computers); memory footprint size is significantly less and the indirection and extra processing overhead is avoided. Similarly, the platform is extensible to allow more component frameworks for other non-functional properties such as security and resource management to be added.

The individual aspects of the architecture and their implementation details are now examined in the subsequent sections.

III. Concrete Middleware

III.A The Discovery Framework

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

- The framework automatically mirrors the current environmental conditions, i.e., which discovery protocols are in use.
- Service lookup is executed across one or more discovery protocols in parallel (depending upon the current setup).

III.A.1 The "Cycle and See" Philosophy

To mirror the current environment, the framework must discover discovery protocols in use. To discover a discover mechanism you must be aware of it in order to test for it. Solutions promoting a fixed point of agreement, e.g. an agreed higher-level discovery mechanism for finding discovery protocols, are infeasible because: 1) not all elements can be guaranteed to use this technology, and 2) the higher-level mechanism itself may change (this simply moves the problem to a higher level). Therefore, ReMMoC uses a "Cycle and See" philosophy. This entails that the framework execute discovery of discovery protocols by cycling through a set of tests for each individual discovery protocol it is aware of. The probability of services being found increases as the number of tests to cycle through increases. "Cycle and See" does not rely on agreement between participating elements, and is evolvable to include future discovery mechanisms.

To perform these tests the framework implements a plug-in component known as "Discover Discovery". Which is illustrated in figure 2. Example tests for SLP and UPnP are as follows. For SLP you can test the environment for service agents. Therefore, the plug-in component creates an SLP header containing the lookup request "service:service-agents", which is then multicast to the SLP multicast address 239.255.255.253:427. Any response from a service agent is an indication SLP is in use. Similarly, for UPnP a HTTP/SSDP header for

"upnp:rootdevice" lookup is multicast to 239.255.255.250:1900.

We acknowledge this approach is limited in two respects: 1) cycling through discovery protocol tests is both time and resource consuming, and 2) you only find discovery protocols that you are aware of. However, tests can be performed in parallel to reduce time, and knowledge based context information can be used to improve performance. For example, if you know the types of discovery protocol used in an environment (from a previous visit, or through shared knowledge) you can test for only these.

III.A.2 Service Lookup Personalities

Component based implementations of individual service discovery protocols (service lookup personalities) form the core functionality of the discovery framework. These ensure that the physically communicated network messages for service lookup can interoperate with the discovery protocols used by services in the environment. Each individual lookup personality is designed as a reconfigurable configuration of OpenCOM components that implements the functionality of an individual service discovery protocol. In ReMMoC, we have developed two component personalities: SLP lookup and UPnP lookup; both provide service and attribute lookup functionality. Figure 3 demonstrates how an OpenCOM personality implements the UPnP protocol.

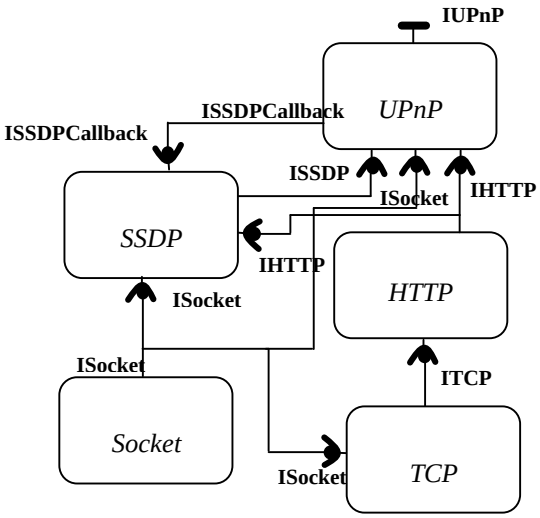

Figure 3: The UPnP Lookup personality

III.A.3 Evolution of the Framework

A key aim of the discovery framework is to be extensible to dynamically incorporate new discovery protocols as they become available. This is especially important in the domain of mobile computing, where much work on creating new discovery solutions for ad-hoc wireless networks and ubiquitous applications is being carried out. To add a new discovery protocol (implemented as a set of OpenCOM components) to the framework, three tasks are carried out: 1) ReMMoC is made aware of the new protocol type by adding its type to an XML list in the ReMMoC repository, 2) A new DiscoverDiscovery component with synchronous and asynchronous tests for the protocol is reconfigured, 3) The XML description for the component personality, used to configure and verify this new personality, is added to the ReMMoC repository.

III.B The Binding Framework

The principal function of the binding framework is to provide a configurable and dynamically reconfigurable binding mechanism that allows mobile clients to bind and interoperate with application services implemented upon particular interaction paradigms (e.g. Remote Method Invocation, Publish-Subscribe, Asynchronous Messaging). To interoperate with a discovered service, the binding framework dynamically reconfigures itself to an identical binding mechanism e.g. if a CORBA service is found the framework becomes a CORBA client side personality; similarly if a particular event publisher is found the framework configures to an event subscriber.

III.B.1 Binding Personalities

We have implemented three interaction protocols for the binding framework: CORBA, SOAP and an event publisher and subscriber based upon the STEAM platform for event publication in ad-hoc networks [13]. The particular component implementation of the event subscriber can be seen in figure 4 .

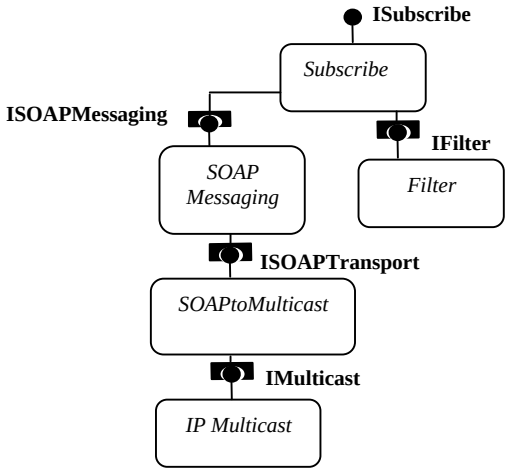

Figure 4: An event subscriber personality

Like the discovery framework, it is possible to add new interaction protocols dynamically to the running framework. This simply involves adding the interaction type to the XML list of known interaction

types and creating the XML architectural description of the protocol, and adding it to ReMMoC repository.

III.B.2 Configuration and Reconfiguration

Configuration and dynamic reconfiguration of the binding framework is controlled by higher-level elements. In ReMMoC's case the top level ReMMoC CF receives information from the service discovery framework to drive the correct configuration i.e. it finds a SOAP service therefore reconfigures to SOAP. From discovery mechanisms that return Universal Resource Identifiers (URI) to identify services e.g. SLP and UPnP, ReMMoC extracts the protocol information directly e.g. "http" for SOAP and "iiop" for CORBA. ReMMoC can also extract the protocol information from the service attributes for services that utilise a non-universal identifier scheme. Once the type has been determined ReMMoC uses the reflective operations of the binding framework to configure this new configuration, based upon the XML based architectural definition stored in the ReMMoC repository.

ReMMoC also supports fine-grained reconfiguration. For example, when the mobile device switches from an infrastructure based wireless network to an ad-hoc network the lookup and interaction protocols can be reconfigured accordingly. For example, both SLP and the event subscriber personality utilise an IP multicast component, however this can be replaced by a probabilistic multicast component that operates by intelligently flooding the ad-hoc network. Local event publishers in the ad-hoc network can be discovered and their events received [13].

IV. The ReMMoC Abstraction

Using dynamic reconfiguration to mirror protocols in the current environment does not provide a complete solution to the discovery and interaction problem. A programmer using this technology would need to explicitly program for each dynamic change, e.g. when the discovered service is of type SOAP a SOAP RPC invocation must be made, then when an event publisher is found the client must subscribe for events. Program code of this nature is inevitably repetitive, overly long (unnecessarily consuming memory resources) and detracts from the application logic. Furthermore, it is impossible to predict in advance the course of a mobile user; they are unlikely to encounter predictable middleware implementation, especially in newly entered locations.

Therefore, ReMMoC promotes an overriding discovery and interaction abstraction, which has the following properties:

- Applications perform general service lookup, stating the service type with attributes that they wish to discover.
- Applications invoke operations on abstract mobile services. That is, ReMMoC follows the Web Services [14] concept of separating the description of a service's behaviour from its interaction protocol.

ReMMoC takes the information from the programming API and then maps them onto the concrete binding and discovery protocols. We now examine in turn both the abstraction and abstract to concrete mappings of ReMMoC.

IV.A The Discovery Abstraction

The abstract service discovery model provides a generic service lookup interface that hides the details of heterogeneous discovery protocols. This takes the form of a custom API, which is based upon the generic features of the majority of discovery protocols. This API is then mapped by individual mapping components onto the implemented interfaces exported from the discovery framework. ReMMoC concentrates on service lookup; other common features including leasing and service events are not considered because they are not available in all protocol implementations.

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

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

HRESULT ServicesLookup(char* ServiceType, Attributes[]
  attrs,  int TimeToSearch,  ReMMoCServiceFindHandler cback,);
HRESULT GetAttributes(ServiceReturnEvent ServiceID,
        AttributeList* list);
```

Figure 5: IDL definition of Discovery Interface

The IReMMoC interface provides the developer with a generic lookup API, as described by the interface in figure 5. This consists of two methods: *Servicelookup* and *GetAttributes*. The required service type and list of attributes are passed to the ServiceLookup operation together with a handler to receive a *ServiceReturnEvent* and an integer stating the time to search for. The items of information returned are the ServiceType, the URL (used to identify the service location), and the Attribute list. The GetAttributes operation returns all attributes for the identified service.

This abstraction relies on each protocol describing a service by a service type as a named string, and service attributes (properties of the service) as a name

value pair. Furthermore, this technique relies upon the assumption that all services of the same service type provide the same service functionality. The abstract service binding (described later) utilises WSDL abstract service descriptions; hence, the same service type identifies services with the same WSDL description. For example, the SLP and UPnP mapping components use these assumptions to directly map from the abstract to the concrete.

IV.B The Binding Abstraction

The ReMMoC binding abstraction is based upon the concepts of abstract Web Services. Each service is described by a Web Service Definition Language (WSDL) description [14], containing the abstract operations provided by the service. These operations can then be implemented upon the developers choice of concrete binding. The Web Service abstraction was chosen for the abstract binding model of ReMMoC for the following reasons:

- Web Services are already being heavy utilised as the key technology in integrating existing heterogeneous middleware platforms [15].
- Web Services are simple, compared to complex modelling tools and languages. The simplicity of the technique has driven the current interest in Web Services.

```
HRESULT WSDLGet(WSDLService* servDesc, char* XML);
HRESULT AddMessageValue(WSDLOperation *op, char*
        elemName, VARIANT value, ReMMoC_TYPE type);
HRESULT GetMessageValue(WSDLOperation *op, char*
        elemName, VARIANT *value, ReMMoC_TYPE type);
HRESULT KnownOperationCall(ServiceReturnEvent
        retLookupEvent, WSDLOperation op, int iterations,
        ReMMoCResultHandler * handler);
HRESULT OperationCall(WSDLOperation op, int iterations,
        ReMMoCResultHandler* handler);
```

Figure 6: IDL definition of Interaction Interface

Therefore, the potential benefit of Web Services is that they will be the most frequently used technology for interoperability, which is the most important factor when attempting to tackle heterogeneity. However, there remains the possibility that Web Services will become one of many competing open standards to follow the predictable trends of previous middleware standards. However, with Web Services there is not the company driven competing standards (there is already worldwide agreement on technologies like XML), rather these companies are collaborating on these meta-standards. Hence, by complying with Web Service standards ReMMoC is less likely to become simply another middleware.

Figure 6 illustrates the operations provided by ReMMoC for interacting with services. WSDLGet takes a WSDL description and creates a data structure to be used to invoke operations. There are two types of operation OperationCall and KnownOperationCall; OperationCall performs service lookup and interation in one operation, wheras KnownOperationCall uses the events returned from service lookup to perform invocations on particular service instances.

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

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

Figure 7 illustrates how abstract messages (input and output) that constitute each WSDL operation map to the RMI and publish-subscribe communication paradigms. We assume that each paradigm understands the set of types used by the abstract defnition. 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, Notifcation 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 fillter attributes.

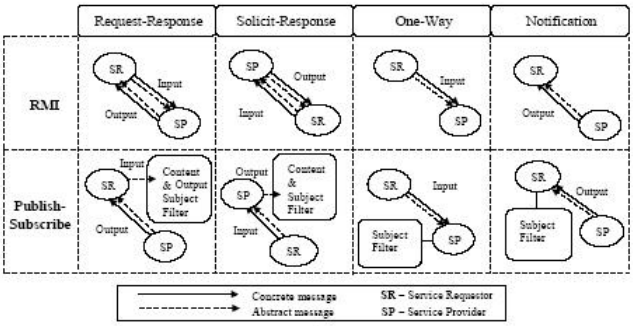

Figure7: Mapping WSDL to abstract operations

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

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

V. Evaluation

V.A A typical mobile scenario

To demonstrate the capabilites of ReMMoC we present a typical mobile scenario illustrated in figure 8. There are three locations: the user's home, the user's office and a coffee bar close to the office. All three locations are covered by an individual wireless network hotspot; users can then connect to these networks using PDAs or laptops. Three applications reside across the three locations. The first application is a stock quote service; this allows the user to request the price of individual shares and view the current status of their portfolio. The second application is a chat service; this allows the user to communicate with other local users (who may be connected from a fixed or portable machine). Finally, the third application is a jukebox service. At each location a physical device within the environment plays music (typically these are in the form of audio speakers connected to a computational device). The mobile user can display the list of songs available from the jukebox on their mobile device; from here they can then select the song they wish to play.

To evaluate ReMMoC, a test harness was implemented to emulate the described scenario. The first step was to create the abstract service descriptions for each of the applications. In the scenario, a wireless network covers each of the three locations; for this purpose, the 802.11b wireless network was used, which has hotspots across the Lancaster University campus. Services operating from fixed machines were hosted using a desktop machine with a 750MHz Pentium processor and 128Mbytes of RAM running the Windows 2000 operating system. Applications operating from mobile devices were hosted upon either a Toshiba e740 Pocket PC or a Compaq iPaq H350 (both with the specification: 206 MHz StrongARM processor, 64 Mbytes of RAM and Windows CE 3.0 OS).

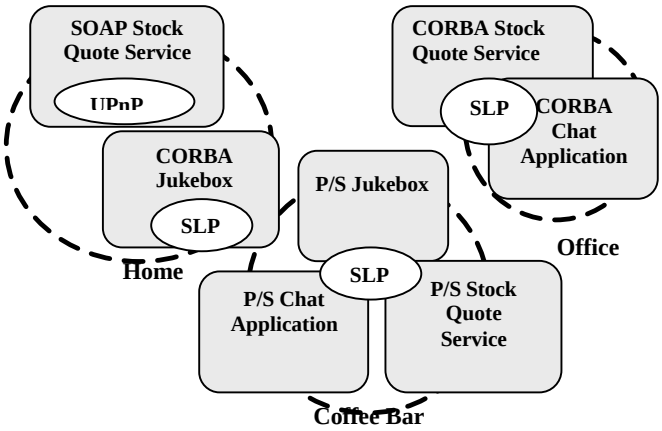

Figure8: Typical mobile scenario

V.B Evaluating ReMMoC's behavior

We now describe the operation of ReMMoC for one of the applications in the scenaro. The mobile user is first at home and uses the stock quote client application on their Pocket PC device to retrieve the latest value of their portfolio. Later the user moves to their office, and again checks the share prices from the same client application. Finally, they move to the coffee bar and when a friend wishes to know a latest share price the user again uses their application. To perform the operations of this interaction the application must perform identically in all three scenarios, the user is unaware of the changing middleware implementation. The user interface showing the developed stock quote application is shown in figure 9.

The sequence of operations for the Stock Quote interaction case study is described in figure 10; the application is first opened in the home location, therefore ReMMoC *Startup* is initiated. This forces the discovery framework to configure itself. A UPnP device and SLP agent respond to protocol discovery, therefore SLP and UPnP components are configured. The application then invokes an *OperationCall* method to find the price of IBM. This forces ReMMoC, to perform lookup for a StockService over the two protocols, however only UPnP responds. The

identified binding type is SOAP, therefore the binding framework is configured appropriately. The request response operation is carried out as a SOAP method call and the resulting price is returned. The user then moves to their office and again invokes the same operation to find the price of BT (the application is not shutdown and re-started).

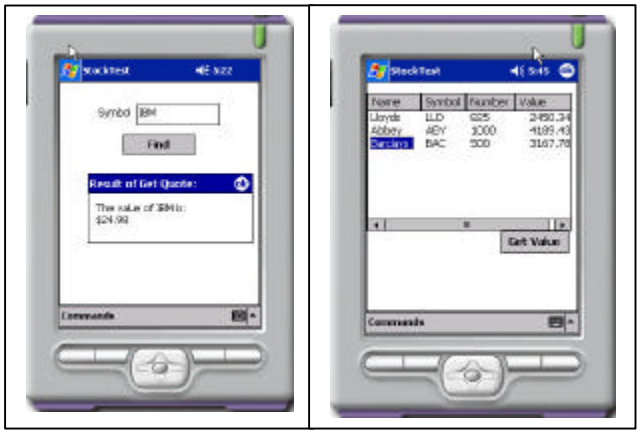

Figure 9: Stock Quote client application

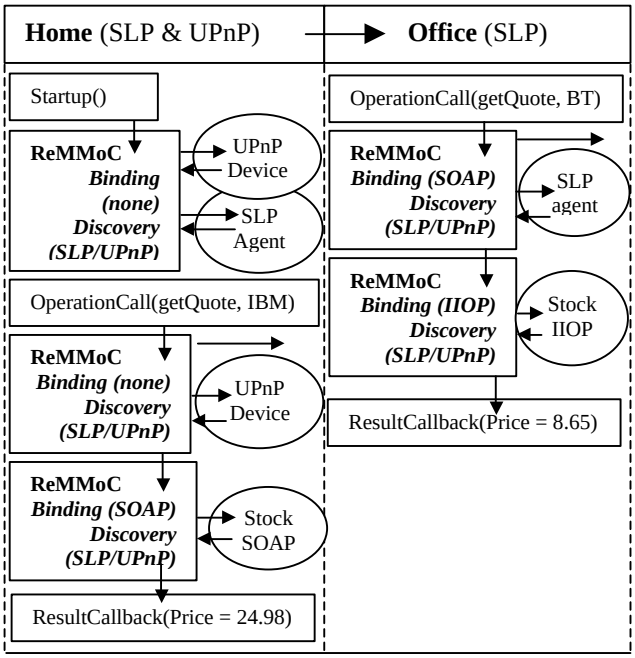

Figure 10: Operation of ReMMoC for stock quote client application

V.C Investigating performance

This section describes the a set of tests to illustrate the performance measures and the overhead costs of the ReMMoC framework. These show that the core operations of ReMMoC (i.e. service calls) have a small performance overhead (incurred as the cost for overcoming heterogeneity) compared to similar operations within related technologies.

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

V.C.1 Abstract v Concrete Operation Invocations

This experiment demonstrates the overhead incurred when invoking abstract service operations (in this case KnownOperationCall methods are used). For this purpose, two operations were implemented upon both a SOAP and an IIOP service: an empty NULL method (that performs no operation and takes no parameters) and a getQuote operation that retrieves stock data from a remote web site. The empty method was invoked 100 times (using four different component setups) from a mobile client connected via the wireless network. From this measure, the operations invoked per second was calculated. The four set-ups were: 1) a concrete IIOP client implementation, 2) a concrete SOAP client implementation, 3) the ReMMoC platform configured when the IIOP service has been found, and 4) the ReMMoC platform when the SOAP service has been found. The underlying interaction protocols for SOAP and IIOP is identical in the ReMMoC and non-ReMMoC set up, therefore ReMMoC's overhead can be evaluated. The same experiment was then repeated for the getQuote remote method. The incurred overhead documented in figure 11 is composed of two factors:

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

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

The results for GetQuote IIOP operations demonstrate that there is a 6% decrease in invocations per second throughput for abstract operations compared to concrete. Similarly for SOAP there is an 8% decrease. This illustrates that the

impact of the overhead is reduced when realistic application operations are executed. Hence, the initial cost of reconfiguration becomes less of a factor for operations whose logic takes longer to perform, i.e. there is only a small decrease in invocation throughput. However, there remains a small, fixed, in-band overhead on each operation call due to the abstract-to-concrete mapping; this is investigated further in the next experiment.

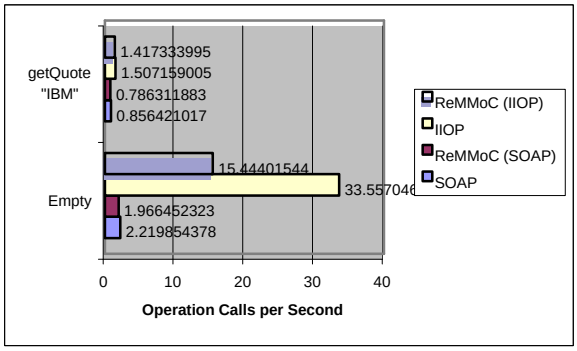

Figure 11: Comparison of service invocations

V.C.2 Investigating Abstract-to-Concrete Mapping

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

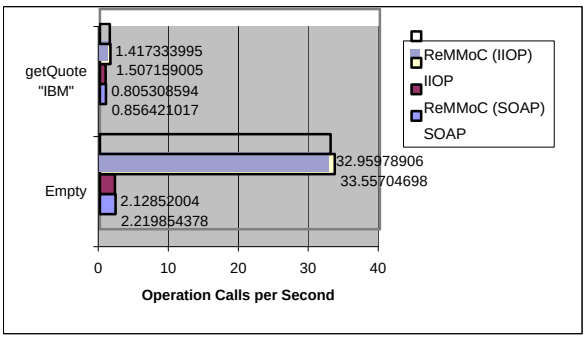

Figure 12: Abstract-to-concrete mapping costs during service invocation

The results in figure 12 show that as expected for NULL operations, there is only a small overhead for abstract invocations. For IIOP there is a 2% decrease in throughput, and a 2% decrease for SOAP. This is because there is no abstract data to map, and the overhead is simply the extra indirection due to ReMMoC's component architecture. Conversely, the getQuote operation requires a mapping of one input and one output parameter. Hence, there is an additional in-band overhead. For IIOP there is a 5% decrease in throughput (an additional 3% to the NULL measure) and 7% for SOAP. Therefore, an extra mapping overhead is attached to each invocation, and this is dependent on the complexity of the operation call, i.e. an operation with more parameters will take longer to map.

V.C.3 Dynamic Reconfiguration

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

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

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

Table 1: Cost of dynamic reconfiguration

The results of the five tests performed are shown in table 1. It can be seen that as the frequency of reflective operations increases the time taken to

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

V.C.4 Configuration Times

The measurements in table 2 illustrate the time taken to configure each of the binding personalities into the binding framework. This is a measurement of the time taken from when the ReMMoC framework initiates the new configuration, until the configuration has been verified as a correct personality by the framework. The two times represent the time taken for the initial configuration, and then the time for subsequent configurations. The additional overhead is explained by the time to load new components (DLLs) into memory.

Table 3 then illustrates the results of experiments breaking down the total time to configure personalities into the binding framework. This consists of the time to insert the personality into the framework (using the algorithm to insert the components and then connect them together based upon an XML configuration description), and then to check that the personality is valid. It can be seen that increasing the complexity of the personality (in terms of number of components and number of connections) increases the time to first configure the personality and then verify it is valid. Connecting the components is the most expensive operation; this is because the interfaces must be searched for (using introspection operations) before the connections are dynamically made.

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

Table 2: **Binding configuration measurements**

The frameworks are implemented for extensibility. Each personality has an XML description that is used to build the configuration; this allows new personalities to be dynamically added to the ReMMoC framework without re-implementation. However, the discovery framework was changed for testing purposes to perform optimised reconfiguration i.e. the XML archirectural descriptions are replaced by hand coded configurations the minimum. Hence, we demostrate the trade-off between performance and extensibility. Table 4 illustrates the time taken to configure these optimised personalities into the service discovery framework.

Personality Name	Comps	Conns	Time to Insert (mSecs)	Time to Connect (mSecs)	Time to check (mSecs)
IIOP Client	5	6	628	2080	263
SOAP Client	6	6	747	2375	273
IIOP Server	5	6	640	2086	271
IIOP Client and Server	7	11	880	4962	521
Publish	6	5	841	1979	315
Subscribe	5	4	660	1578	234
Publish-Subscribe	7	7	900	3113	345

Table 3: **Binding configuration measurements**

Personality	Comps	Conns.	Time to Configure (mSecs)	Time to Check (mSecs)
SLP	4	9	1066	563
UPnP	5	8	1070	432
SLP & UPnP	8	17	1956	997

Table 4: **Optimised framework measurements**

Again, these results show that the same factors as for the binding framework (e.g. number components and connections) affect performance time. However, these results show a significant improvement in configuration time i.e. the more complex SLP & UPnP personality takes less time to configure than the simpler SOAP client. This measure demonstrates that a large part of the overhead incurred during configuration of the frameworks is for ensuring valid operation in the face of dynamic change. An unsafe version of ReMMoC would perform significantly better; for example, an optimised, unsafe configuration of SLP takes only 1.06 seconds, compared to 3.87 seconds for the XML-based, safe SOAP client personality configuration.

VI. Related Research

Much work has been carried out on various discovery and interaction protocols e.g. UPnP, Jini, Centaurus and many others, which have already been described in this paper. These platforms although solving the problem of discovery and interaction are creating a new heterogeneity problem, which hinders the creation of dynamic mobile applications that can operate in new unknown settings. ReMMoC addresses this issue using a Web Services, reflective

approach. Other platforms that examine this heterogeneity platform are Universal Interoperable Core, the Web Service Invocation Framework and SATIN. Each is now analysed in turn.

The Web Service Invocation Framework (WSIF) [16] is a Java API for invoking Web Services irrespective of how and where these services are provided. Its fundamental goal is to achieve a solution to better client and Web Service interoperability by freeing the Web Services Architecture from the restrictions of the SOAP messaging format. WSIF utilises the benefits of discovery and description of services in WSDL, but applied to a wider domain of middleware, not just SOAP and XML messages. WSIF follows the discovery model of web services, and requires new and existing services to be available through advertising of the WSDL file (e.g. in a UDDI registry). Like Web Services, the performance of the WSIF platform will suffer due to its reliance on XML in discovery. This doesn't account for heterogeneous discovery mechanisms and downloading the service description consumes bandwidth. Furthermore, services will be implemented and advertised without exposing a WSDL file; these cannot be interacted with, as the message exchange format cannot be determined. Hence, the technique requires that all providers follow this solution, which cannot be guaranteed.

SATIN [17] is a low footprint component based middleware, which aims to address the problem of heterogeneous service implementations in dynamically changing mobile environments. At the heart of SATIN is the ability to advertise and discover service implementations that may be advertised using different techniques; each discovery mechanism is represented by a different capability that can be added to the host when needed in the environment. SATIN then utilises its own "higher level" XML based discovery mechanism for initialisation. For example, a host uses SATIN to find the discovery capabilities being used and then downloads these. The required application services are looked up and their interaction capabilities are downloaded to complete the cycle. The use of logical mobility provides an elegant solution to the problem of heterogeneity; applications do not need to know in advance the implementation details of the services they will interoperate with, rather they simply use code that is dynamically available to them at run-time. independently of SATIN, can in theory still be utilised. SATIN relies on participants conforming to their non-standardised architecture i.e. the SATIN abstract discovery mechanism. Therefore, the solution does not scale to include application services not implemented with knowledge of these techniques.

The Universally Interoperable Core (UIC) [18] is a reflective middleware. The goal of the middleware is to support interactions with multiple service platforms from a mobile device in ubiquitous environments. UIC provides the capability to interact with a service implemented in CORBA, and also with the same service type implemented in Java RMI and SOAP. UIC uses dynamic adaptation to directly tackle the problem of heterogeneous middleware in mobile environments. This technique has the potential to address the changing middleware heterogeneity as the user moves location. However, the design of the platform defines a standard skeleton structure targeted to only object-oriented request brokers (CORBA, Java RMI, and DCOM); it offers no solution to the different interaction paradigms e.g. publish-subscribe, data-sharing etc.). In addition, there is no higher-level abstraction to invoke heterogeneous services. The platform will operate for all RMI based implementations, but it cannot be extended to include contrasting communication paradigms. Furthermore, UIC does not address heterogeneous service discovery. It is utilised within a framework that offers a single discovery mechanism.

VII. Conclusions

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

Furthermore, the following points can be extracted from the evaluation of ReMMoC's performance. Abstract service invocation incurs a performance overhead compared to the same operation performed by a concrete middleware platform. The significance of the service invocation overhead is reduced when realistic service operations are performed. The throughput of ReMMoC IIOP invocations per second is reduced from the maximum 54% decrease to a 6% decrease (compared to base IIOP invocations) for a realistic mobile application operation.

Mapping abstract operations to concrete operations incurs an in-band operation overhead. For

NULL operations where there is no mapping, a 2% decrease in ReMMoC invocation throughput (compared to base IIOP) is observed. This is caused by additional indirection. Mapping a single input and output parameter incurs an extra 3% decrease in throughput for ReMMoC IIOP, and an extra 5% decrease for SOAP. Hence, this in-band overhead increases when more parameters are mapped.

Dynamic reconfiguration adds an additional "out-of-band" overhead. Infrequent reconfiguration e.g. 1 reconfiguration during 1000 invocations suffers a 16% decrease in performance time. Frequent reconfiguration e.g. 10 reconfigurations during 1000 invocations suffers a 106% decrease in performance time. Therefore, where reconfiguration is performed infrequently it has less of an impact on overall throughput.

Algorithms implemented to improve platform extensibility (e.g. configuring personalities in the binding framework) are significantly more expensive than optimised configuration algorithms The configuration of the less complex SOAP client personality takes over three times longer than the SLP personality Hence, a trade-off between extensibility and performance can be made when implementing middleware platforms.

References

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

[**2**] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo and A. Wollrath, "The Jini Specification", Addison Wesley, 1999.

[**3**] Microsoft Corporation, "Universal Plug and Play Device Architecture", Version 1.0, http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.

[**4**] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte and D. Winer, "Simple Object Access Protocol (SOAP) 1.1. Technical Report", http://www.w3.org/TR/SOAP, May 2000.

[**5**] Salutation Consortium. "White Paper: Salutation Architecture Overview", http://www.salutation.org/whitepaper/originalwp.pdf, 1998.

[**6**] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K.Nahrstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces", IEEE Pervasive Computing, 1(4), pp. 74-83, Oct-Dec 2002.

[**7**] Object Management Group, "The common object request broker: Architecture and specification", Tech. Report. Version 2.0, July 1995.

[**8**] J. Veizades, E. Guttman, C. Perkins and S. Kaplan, "Service Location Protocol (SLP)", Internet RFC 2165, 1997.

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

[**10**] C. Szyperski, "Component Software, Beyond Object-Oriented Programming", ACM Press/Addison-Wesley, 1998.

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

[**12**] P.Grace, G. Blair and S. Samuel, "ReMMoC: A Reflective Middleware to Solve Mobile Client Interperability", Proceeding of International Symposium of Distributed Objects and Applications (DOA'03), Catania, Sicily, November 2003.

[**13**] R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks", Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02), pp. 639-644, Vienna, Austria, 2002.

[**14**] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard, "Web Services Architecture", W3C Working Draft, http://www.w3.org/TR/ws-arch/, August 2003.

[**15**] R. Chinnici, M. Gudgin, J. Moreau and S. Weerawarana, "Web Services Description Language (WSDL) Version 1.2", W3C Working Draft, http://www.w3.org/TR/wsdl12/, March 2003.

[**16**] M. Duftler, N. Mukhi, A. Slominski and S. Weerawarana, "Web Services Invocation Framework (WSIF)", Proceedings of OOPSLA 2001 Workshop on Object Oriented Web Services, Tampa, Florida, October 2001.

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

[**18**] M. Roman, F. Kon and R. Campbell, "Reflective Middleware: From Your Desk to Your Hand", IEEE Distributed Systems Online, 2(5), August 2001.