Mobile and Dynamic Web Services

Elena Sánchez-Nielsen, Sandra Martín-Ruiz, Jorge Rodríguez-Pedrianes

Dpto. E.I.O. y Computación – Escuela Técnica Superior de Ingeniería Informática
Universidad de La Laguna, 38271 La Laguna, Spain
enielsen@ull.es

Abstract. Making mobile phones capable of consuming Web services over wireless networks is a challenging task because of the different issues to be addressed and the limited resources of mobile devices. In this paper, we focus on the issue of how to perform dynamic discovery and invocation of Web services from mobile phones when a J2ME wireless middleware is used. In order to solve the limitations of the middleware platform when mobile phones act as Web services requestor we propose a Web service based dynamic proxy between service providers and mobile consumers. With this approach, we provide the following features to mobile devices: (1) support of dynamic binding, (2) support of UDDI specification, (3) support of SOAP messages with encoded representation and (4) handling of complex data types. The paper includes the description of the dynamic proxy, implementation and experimental results with the performance of the approach proposed.

1 Introduction

The use of Web services (WSs) in mobile phones allows users to discover and access to digital content and services at anywhere and anytime. The access to these resources in a wired-wireless system involves: service provisioning, service discovery and service execution.

The need of service providers to add new capabilities at anytime and in turn give mobile consumers a huge choice of available services at runtime requires a dynamic discovery and invocation process. The use of this process brings a number of benefits to mobile users such as require no prior knowledge of available services nor require updating clients applications when new services are incorporated at runtime.

Dynamic adaptive middleware for mobile computing has been proposed with the purpose of adapting applications to the current context [18, 19], frameworks for Web services provisioning in a static environment of fixed and mobile computing have been described in [22] and approaches for provisioning mobile services in critical environments have been outlined in [23]. However, no significant frameworks with experimental results have been carried out to allow access to Web services from mobile phones at runtime without prior knowledge of available services due to the current limitations of extending the Web service technology into the wireless world by the key commercial players.

This paper describes our approach to addressing dynamic discovery and invocation when mobile phones act as WS requestor at runtime, its implementation and performance when J2ME middleware platform [10] is used.

The remainder of this paper is organized as follows. Section 2 introduces the different issues related to invoke Web services from mobile phones and the related work about mobile devices acting as WS requestor. Section 3 describes the current restrictions to design mobile client applications to access to Web services when J2ME development platform is used. Focused on these limitations, we propose to introduce a Web service based dynamic proxy. The use of this component allows service providers to create, update and change services at anytime and mobile users to locate new services at runtime without adapting the application of their devices. Section 4 describes our approach and conceptual model to dynamically discover and invoke Web services from mobile phones. Section 5 illustrates the implementation and performance of the approach proposed. Comparisons with common scenarios based on the use of static stubs are performed. Discussion of the advantages and disadvantages of the use of J2ME as wireless middleware is included. Section 6 gives concluding remarks and future work.

2 Related Work

This section provides a brief summary of Web services standards related to our work, scenarios of using Web services in mobile phones, and ongoing specifications related to it. Discussion about if is appropriate to adopt common scenarios to support services on wired networks in mobile phones is included.

2.1 Web Services Standards

The WS paradigm [6] involves three types of participants: WS provider, WS requestor (also referred to as service consumer or client) and WS registry or broker. The infrastructure necessary to implement a WS based approach requires: a way to communicate (SOAP) [7], a way to describe services (WSDL) [8], and a name and directory server to publish and advertise available services (UDDI) [9]. In middleware terms, a service is a procedure, method or object with a published interface by a service provider that can be invoked by service clients. Using SOAP-based interaction, the client makes a procedure call that looks like a local call. As a result, clients can invoke Web services by means of standardized conventions to convert procedure calls into an XML message, to exchange this message trough HTTP or other protocols, and to turn the XML message back into an actual service invocation. The structure of a SOAP message is influenced by: two different interaction styles and encoding rules. Then, four different types of SOAP messages are possible: RPC/encoded, RPC/literal, document/literal and document/encoded.

WSDL is an XML-based interface definition language. This interface is specified in terms of methods supported by the Web service. This interface can be compiled into the appropriate programming language to generate the stubs and intermediate layers that make calls to the Web services transparent. Invoking Web services with

clients can be carried out by static stubs, dynamic proxies and dynamic invocation interface (DII) according to client applications have knowledge of the WSDL URL at development-time or runtime.

- Static stubs: a procedure call of a client application is an invocation of a proxy procedure located in a stub appended to the client at compile time. Then clients invoke methods of a WS directly via the stub.
- Dynamic proxies: the client application calls a remote procedure through a dynamic proxy that is created at runtime. The dynamic proxy needs to be re-instated whenever the service endpoint interfaces are changed.
- Dynamic invocation interface (DII): this approach enables dynamic invocation of Web services without having to know interface details at compile time.

2.2 Scenarios of using Web services in mobile phones

The possible scenarios of using Web services in mobile phones are [17]: (i) mobile device acting as WS requestor, (ii) mobile device acting as WS provider and (iii) a mixed combination of the previous approaches. The approach proposed in this paper is related to the first scenario.

In the following sections we describe existing and ongoing work related to the two possible architectural configurations for the first scenario.

2.2.1 WS-aware mobile device

In this architectural configuration, the entity that plays the role of the WS requestor is the mobile device itself. This device needs to dispose a WS client application in order to enable the provision of services to mobile users. It interacts with the service provider and the service broker using WS-aware protocols over the wireless network (eg., WLAN, GSM/GPRS). TinyXML [20] can be used to present data and VoiceXML [21] allows user to listen to data instead of viewing it. Figure 1 illustrates how to access to Web services functionalities from mobile phones using a WS based service oriented architecture with static stubs.

2.2.2 WS-agnostic mobile device

This configuration introduces a proxy entity that plays the role of the mobile device representative in the fixed network infrastructure. This scenario is applicable in the case where the mobile user moves into an unfamiliar environment and obtain services for which it has no previous knowledge. For example, we could consider a mobile user entering an airport and obtaining access to services such as flight information, special offers and promotions in the duty free shops, etc. The proxy interacts via WS-aware protocols with the service broker and the service provider and returns the results to the mobile device using WS-agnostic protocols such as WAP/WML, i-Mode/cHTML over a wireless network [17]. The proxy may also perform various tasks such as conversion and content adaptation in order to adjust the WS result to different terminal and network environments.

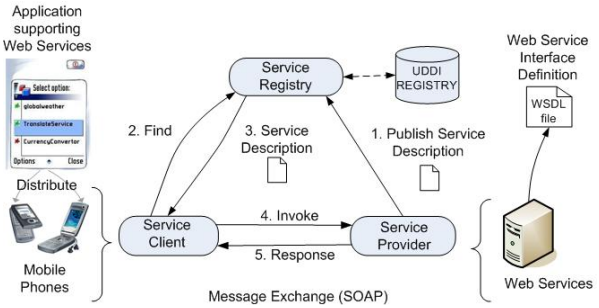


Fig. 1. Mobile phone as a service requestor. The device hosts WS client code

2.3 Specifications

Two specifications related to implement services in mobile phones are being developed: (i) OSGi Alliance [2] and Liberty Alliance Project [3]. The OSGi Service Platform defines a standardized, component oriented, computing environment for networked services, where software components can be installed, updated or removed. These components are libraries or applications that can dynamically discover and use other components. The design of this platform is not targeted to Web services solutions. Therefore, there is ongoing work in order to provide the OSGi Service Platform as a platform for Web services such is illustrated by Hall and Cervantes work [4]. On the other hand, the Liberty Alliance Project proposes a federated network with an authentication mechanism that makes use of a Web services framework. However, the usage, advantages and disadvantages of dynamic binding are not mentioned in this specification.

2.4 Discussion

In order to provide an admissible solution to dynamic services from mobile devices, the following considerations must be taken into account:

- Standard WS infrastructures to support services on wired networks (e-services) [5] are not appropriate because they are based on the use of static stubs. As a result, the slightest change of Web service definition leads to the stub being useless and a generation of a new stub. Also, each WS to be invoked by a client application requires a stub appended to the client at compile time. Therefore, in order to support a dynamic infrastructure where new services can be provided to mobile clients at runtime requires that the client downloads a new application to its device each time a new service is provided to the marketplace when a static stub based approach is used.
- The main usage mode of UDDI today is focused on design-time discovery and not on dynamic binding [5]. That is, users browse or search the content of a registry

for services of interest, read the service descriptions, and subsequently write clients that can interact with the discovered services.

- The WS-aware mobile device based configuration presents several issues which come from the fact that mobile devices are characterized by limited resources such as processing power and memory. Also, CPUs in mobile phones are restricted to handle complex XML parsing and in general to handle the processing need of Web services.
- The WS-agnostic mobile device based configuration is characterized by an increase of the amount of interactions between the mobile device and the network. Also, at the present time commercial middleware based solutions make not possible a DII based approach.
- Also, the specifications supporting these scenarios are still or just emerging.

In this context, we propose a framework based on a dynamic proxy entity. The main contribution of our approach is that we propose the proxy component as a Web service that makes use of dynamic binding and that act as client over the network of services and as server to the mobile devices. With this approach, we compute at runtime WS descriptions from service providers, UDDI registry and invoke services selected by mobile user using WS technology

3 Client Applications with J2ME

The Java Platform, Micro Edition (Java ME) provides an environment for applications running on consumer devices, such as mobile phones. This platform is divided into configurations, profiles and optional packages. Configurations are specifications that detail a virtual machine and a set of class libraries which provide the necessary APIs that can be used with a certain class of device. A profile is a set of higher-level APIs that further define the application life-cycle model, the user interface, persistent storage and access to device-specific properties. Optional packages extend the Java ME platform by adding functionality to Web services.

MIDP profile with CLDC configuration, KVM virtual machine and JSR-172 specification is required as development environment to design mobile client applications to access to Web services using Java ME. JSR-172 specification provides the necessary APIs to access from J2ME applications to remote SOAP/XML services and parsing XML data. This specification provides two optional packages based on XML: Java API for XML Processing (JAXP) and Java API for XML-based RPC (JAX-RPC). JAXP provides the XML parsing functionality to process XML data received in a mobile phone. JAX-RPC is an implementation of RPC technology (Remote Procedure Call), where the client makes a procedure call that looks like a local call. This call is an invocation of a proxy procedure located in a stub appended to the client at compile time. Currently, designing client applications using JSR-172 specification presents the following restrictions:

- There is no support for dynamic proxies or dynamic invocation interface. That is, the Java ME subset supports only static stubs. The developer is responsible of

generating the stubs using a WSDL-to-Java mapping tool. Figure 2 illustrates the process. The Sun Java Wireless Toolkit includes a stub generator. In this context, so many stubs are generated and appended to the client application as different services are provided to the service client.

- There is only support to the document style of operation with literal use.
- Neither capabilities for standard service registration and discovery nor support to UDDI 2.0 specification are provided.
- There is no support to the use of a mobile phone as server of Web services. That is, the JAXP-RPC for Java ME subset doesn't support the service endpoint model, only the client service consumer model is supported.

JAX-RPC for Java ME doesn't support all of the JAX-RPC 1.1 basic types. For example, there is partial support for complex value types, and mapping of floating-point types depends on the Java ME configuration you use.

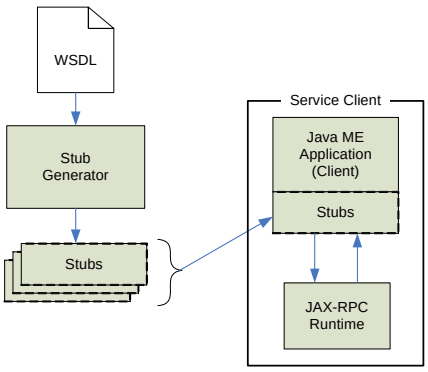


Fig. 2.. Generating the JAX-RPC Stub

4. Mobile Web Services Framework

Commercial middleware such as J2ME and basic infrastructures to support WSs are addressed by employing a static stub approach that guarantee the execution of WSs in a static environment. However, this middleware platform does not take into account basic features that characterize today's mobile phones environments. In this context, we aim to modify traditional WS SOA based approach to enable dynamic discovery and invocation for mobile phones. To be precise, we propose:

- To introduce an intermediate entity between service providers and service clients. This entity consists of a service manager that operates as a dynamic discovery and invocation (DDI) client of the distributed network of Web services offered by the different providers and as server to the mobile phones. With this approach, we delegate the business logic to service managers, solving the problems faced by

direct access from mobile devices to Web services and at the same time reducing the number of interactions between mobile phones and the network.

- An XML based infrastructure as format data exchange with two purposes: (i) to define a service registry structure to locate services that allows service providers to create, update and change services at anytime and (ii) to establish the communication between the service manager and mobile phones. This infrastructure is described in [1].

Figure 3 illustrates the framework proposed. In the following sections, we describe the components of our approach illustrated in Figure 3, the interactions among the different components and the UML class diagram for the conceptual model of the service manager.

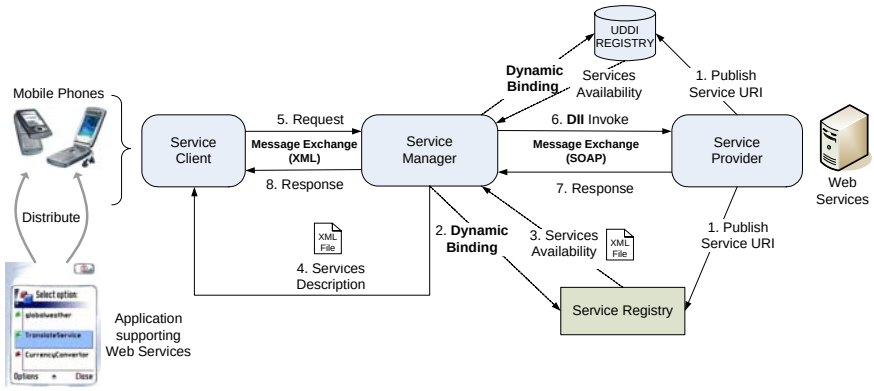


Fig. 3. Mobile Web services framework

4.1. Service Providers and Service Clients

Service providers are the owners of different enterprises or a single enterprise who made up the marketplace that offer services. They define descriptions of their services using WSDL specifications [8].

Service clients are mobile phones-oriented users interested in diverse services such as search engine tools, language translation facilities, newspaper reports, weather forecast, airport services, mobile shopping, mobile banking and m-government services.

4.2. Service Managers

Service managers act as a mediator layer between service providers and mobile clients. They are responsible for information flow between both components. A service manager is a Web service entity that uses dynamic binding to compute service

descriptions and dynamic invocation interface (DII) to query for services to service providers.

With the use of DII, we allow service managers to invoke WS without knowing their communication interface at compile time. As a result, we obtain several advantages: (i) invocations of Web services not known prior can be computed by the service manager (ii) service providers can create, update and change their services at runtime, (iii) no static stub generated manually for the service manager at compile time is required and (iv) a single stub appended to the Java ME client application is required. This appended stub corresponds to the service manager.

According to the structure of marketplace, one or multiple service managers can be supported. The use of a single service manager involves a centralized marketplace. If multiple service managers are used, different operators or third parties can be incorporated at anytime, where each one can support different service providers. The integration of service managers into a service oriented architecture leads to mobile client applications to only interact with these components and no with the different service providers. This way, a single stub corresponding to the service manager is needed to be appended to the client application and no several stubs corresponding to the different services available on the marketplace. At the same time, the interactions between mobile phones and the network are considerably reduced. The conceptual model of a service manager is described in section 4.5.

4.3. UDDI Registry

UDDI service directory can be used by mobile users to locate new services. Discovery is computed at runtime by the service manager, once the user has sent their request of new services at UDDI registry.

4.4 Interactions

Interactions between service providers and mobile clients using a service manager consist of the following processes:

- **Start up:** When the service manager starts up, it processes a service registry. This registry is a structure that enables service providers to store their list of URL addresses (URI) of accessible services made available. New URI can be incorporated at anytime. The service manager maintains an XML based structure as registry. Dynamic binding is used by the service manager in order to obtain the service descriptions at runtime.
- **Service delivery descriptions:** the description (operations provided, parameter…) of available Web services set is sent from service manager to mobile client according to an XML format.
- **Request Service:** once mobile clients have received the description of available services, they send requests for services of their interest.
- **Service invocation:** service manager receives a request encoded as an XML message with the necessary information (Web service name, selected operation,

parameter values introduced…) from a mobile device when a user is interested in some service. Dynamic invocation is used by service manager in order to invoke Web services functionalities to service providers.

- **Results transmission:** the service manager sends the information encoded as an XML message to the mobile user, when it receives the response of the corresponding service provider. This information is shown on the screen display of the mobile device.
- **UDDI services:** mobile clients can also demand services supported by UDDI registry. In this context, a client makes a request to the service manager using keyword in order to discover a particular service at UDDI registry. Then, the service manager uses dynamic binding to discover services at UDDI registry that match with the user search criterion. The description of these services is sent from the service manager to the mobile application. The user selects the service of its interest and finally the service manager processes this request at the same way as the request and invocation of services previously described.

4.5 Conceptual Model of Service Manager

Figure 4 depicts the UML class diagram for the conceptual model of the service manager. The different classes and relationships are illustrated. Following, the different classes are described.

The *WebService class* corresponds to the service manager that processes the requests of mobile clients. In order to avoid a new creation of instance every time the user performs an invocation of some operation of this class and reduce computational costs, two different classes were developed (*StandardServices* and *UDDIServices*). These classes were implemented using a *singleton* pattern. As a result, a single instance is present every time. The client application invokes the corresponding operation according to the selection achieved by the mobile user:

- **updateStandardServices:** this operation allows mobile users to check the current version of downloaded services. If the version of the mobile application doesn't correspond to the service manager version, a new version with the new services incorporated at runtime is sent to the mobile client in an XML format.
- **invokeStandardService:** this operation allows mobile users to invoke any WS registered at the service registry of the service manager. Input parameters are: URL address of WSDL file, QName of portType, operation name to invoke and parameter values of the operation. Once the results of the invocation of the Web service have been processed by the service manager, this one transforms these results to an appropriate XML format, which is sent to the mobile application.
- **processUDDISearch:** this operation allows mobile users to request a search at UDDI registry. Once the search is computed by the service manager using dynamic binding, the results of available services are encoded as an XML message to the client application.
- **computeUDDISearch:** once client application receives services located at UDDI registry as a result of the **processUDDISearch** operation, the user selects the appropriate Web service for its interest. The input parameter of this operation is

the service selected. An XML document is generated with the service description which is sent to the client application.

- **invokeUDDIService:** with this operation, services searched at UDDI registry are invoked using dynamic invocation.

The *StandardServices* class is responsible of the management of all the operations related to the processing of Web services, such as: computation of the Web services to offer to mobile clients, making of the XML document to be sent to the mobile application with the new services incorporated at runtime and invocation of the operation of a specific WS selected by a mobile user.

The *UDDIServices* class corresponds to the operations related to Web services searched at UDDI registry: Web services list that matches with the search criterion of the user and invocation of the corresponding operation. The search of Web services at UDDI registry is computed by *locateUDDI* method. This method makes use of the *locateUDDIService* class that computes the search at UDDI registry according to the search criterion detailed by the mobile user. After the search has been performed, services found at UDDI registry are checked in order to detect inconsistencies.

The *ManagerOperations* class manages the operations related to Web services such as operations of incorporation, searching and invocation. The attributes of this class are the *operations* set and the *dynamicInvoker* object. Attribute *operations* contains the description of operation of Web services: URL address of WSDL document, parameters required and description of the operation. The *dynamicInvoker* object is an instance of the class *DynamicInvoker.*

The *DynamicInvoker* class is designed with the purpose of achieving the description of Web services from WSDL files and invocating them at runtime. Apache Axis [11] is used to implement this class. In order to make dynamic invocation, we use the *call* interface provided by Axis. Invocation of an operation implies to generate an instance of this class. With the purpose of reducing computational costs and avoiding the generation of a new instance with every invocation, we use a structure with a *call* instance for every operation created. This way, a single instance is generated for every operation, using this instance every time that it is necessary to invoke the same operation.

Following, the parameters of the operation selected must be checked. The parameters to be processed can basically be simple or complex. However, the *call* interface does not provide support to handling complex data type. That is, if the *call* object receives as input parameter a complex type, this object has not the sufficient information to the serialization and deserialization process of this parameter.

At the present, Axis interface provide the *invoke* method to compute the invocation of the service. Once the operation and service to be computed has been indicated, the method *invoke* can basically be used by two different ways:

- *invoke (Object[] arg0):* service invocation is computed by means of the use of arg0, which represents a set of parameters of the operation to be computed. Every element of this array is an instance of a Java class that represents every parameter of the operation.

- *invoke (Message arg0):* service invocation is computed by a message, which represents a SOAP message, that is, an XML format with the operation to be invoked and their parameters.

To handling the complex data types, we use the first option of the *invoke* method. However, there is not a representation for a complex data type in a structure that can be used by the *invoke* method. Therefore, in this situation, the service cannot be invoked. To solve this problem, we generate at runtime a *JavaBeans* software component that represents each one of the different complex types that appears on the WSDL files analyzed. As a result, the bean associated to each complex parameter type allows that the *call* instance will be able to catch and modify the fields of the parameter at the invocation time by means of the instance of Axis *BeanSerializerFactory* and *BeanDeserializerFactory* classes.

The *BeanJavassistUtils* class produces at runtime the beans necessary to make invocations of operations that contain parameters with complex data types. In order to produce a class at runtime, we use the class library *Javassist (Java Programming Assistant)* [16]. It enables Java programs to generate a new bytecode at runtime. *Makebean* method of *BeanJavassistUtils* class (Figure 4) is responsible of generating the corresponding beans. It is implemented in a similar way as the beans are generated by the mapping tools of stubs of Axis.

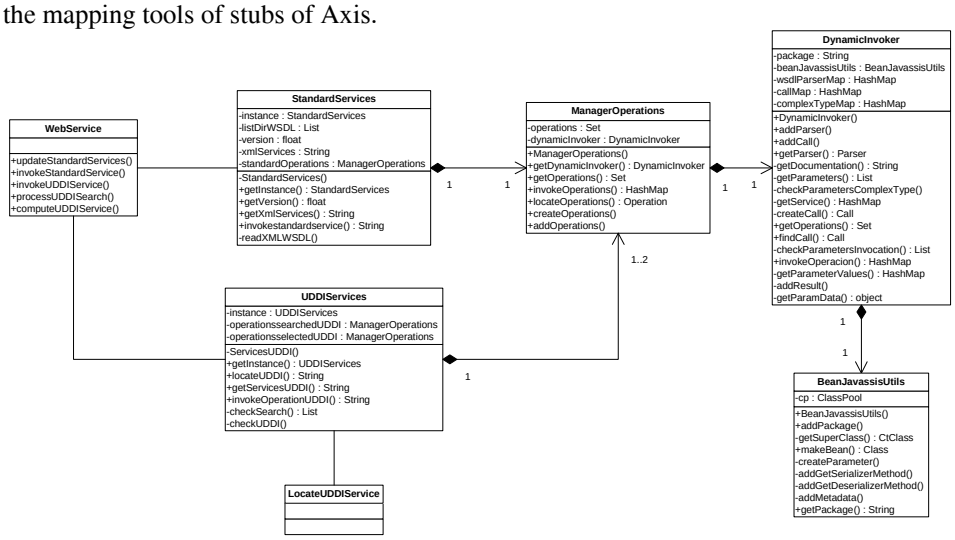


Fig. 4. Class diagram of service manager

5. Development and Results

The framework described in section 4, allows mobile users to access to Web services published at World Wide Web by means of requesting a service manager entity. No update of client application is required when new services are provided at runtime.

The framework has been implemented using the following open source software: Apache Tomcat 5.0.28 for application server [15], J2ME Wireless Toolkit (WTK) [13] for developing wireless applications and designed to run on cell phones, and Eclipse 3.1 development platform with WTP (Web Tools Platform) plug-in [14] for building software and developing Web applications. Axis [11] and UDDI4J [12] Java libraries have been used as SOAP motor and Java implementation of UDDI protocol. Javassist (Java Programming Assistant) [16] has been used as a class library for editing bytecodes in Java. It enables to define a new class at runtime and to modify a class file when the JVM loads it. The client application has been implemented as a MIDlet using J2ME Wireless Toolkit. That is, a Java application developed with MIDP profile and CLDC configuration.

In order to test the Web services framework for mobile devices, we have implemented on mobile phones different scenarios services using a service manager entity. The following services have been implemented and tested: (1) searching with Google engine, (2) text translation from one language to another, (3) newspaper reports from different sources, (4) temperature converter, (5) weather forecast, (6) calculator operations and (7) dynamic binding with UDDI registry.

We have developed the Java client application and tested it on the Sun emulator. Also, with the purpose of testing correct performance, we have tested the client application with mobile emulators of commercial trademarks.

5.1 Performance

The core of our framework is the Web service based proxy (service manager). An important part of this component is the ability of processing requests from mobile phones. In order to measure and compare the running of our proxy, we compute the performance of the service manager invocating services over the wired network under two approaches: (i) static sub and (ii) DII.

Different qualities or properties defined by Quality of service (QoS) [24] are used in order to evaluate the performance of the Web service based proxy from the perspective of the users of services (in this case, the users with mobile phones).

WSTest [25], a benchmark developed at Sun Microsystems is employed with the purpose of computing two aspects of QoS. WSTest benchmark simulates a multi-thread server program that makes multiple Web services calls in parallel. WSTest reports the *Throughput* (average number of Web service operations per second) and the *Response Time* (average time taken to process a request).

5.1.1 Test Description

With the purpose of computing the performance of the service manager, we consider the invocation of three operations of a Web service with the following types of parameters:

- **echoVoid:** sends and receives an empty message.
- **echoStruct:** sends and receives an array of size 20, where each element is a structure composed of one element each of an integer and string data type.
- **echoSynthetic:** sends and receives a string and a complex parameter (struct).

For the results reported, WSTest was run with the following parameters set, specified in an initialization file:

- **Agents:** this is the number of client threads and is set to maximize CPU utilization and system throughput. The number of concurrent threads is set to 8.
- **Execution time:** 300 seconds.
- **The same number of calls** for each of the 3 types of operations tested.

WSTest was run on the following system configuration:

- **Service manager and Web service invoked:** Intel Pentium 2GHz. 1GB DDR2, 1 processor.
- **Client of service manager:** Intel Pentium 2GHz. 512MB DDR2, 1 processor.

5.1.2 Results

The measured throughput and response times were computed for four different scenarios:

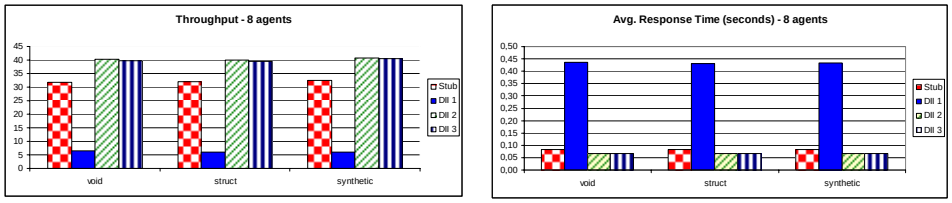
- **Static stub:** invocation of the Web service from the service manager using a static stub approach.
- **DII1:** invocation of the Web service with a standard DDI service manager.
- **DII2:** invocation of the Web service with a DDI service manager. The first time the service manager employs standard DII. In the successive times the service manager computes and caching the *call* class. This consideration will allow the invocation for each of the different operations that support the service. As a result, it will be not necessary to generate the *call* objects every time a client of the service manager processes a request of the same Web service.
- **DII3:** it is assumed that the service manager has a cache memory with the *call* objects.

The graph in Figure 5 shows the performance for each of the four scenarios. The x-axis indicates the types of parameters such as void, struct and synthetic. The y-axis in Figure 5.a indicates the throughput, the number of Web service operations executed per second (higher is better) and the y-axis in Figure 5.b indicates the response times measured in seconds (lower is better).

In order to compare the results between a static stub approach and a DII improved approach with cache memory, Figure 6 shows the results performed with both approaches.

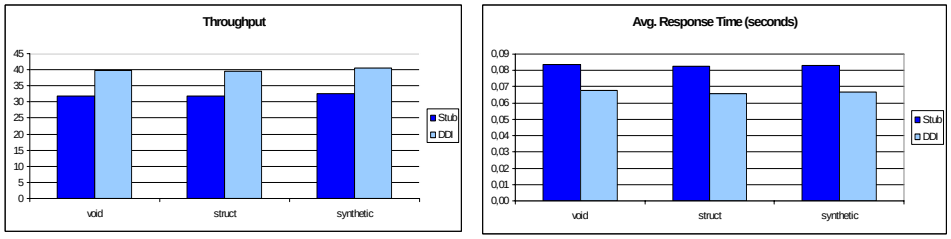
Although a better performance could be supposed with a static stub approach, a DII approach with cache memory offers better results. Initially, the DDI service manager client needs an additional cost for discovering the service to be invoked, for processing the WSDL document for obtaining the description of the service and the generation of the necessary structures for the invocation such is illustrated in Figure 5 with the DII1 scenario. However, this computational cost is only assumed the first time by the service manager for every service invoked. After, all the knowledge acquired by means of the use of a cache memory of *call* objects is used in successive

invocations. With this approach, we compute a better performance in contrast to a static stub approach such is illustrated in Figure 6.



(a) (b)

Fig. 5. Throughput and average response time for service manager using three different parameter types (echo, struct and synthetic) and four different approaches: (1) static stub, (2) standard DII (DII1), (3) standard DII for the first time and a memory cache for the successive times (DII2) and (4) DII with a cache memory (DII3)



(a) (b)

Fig. 6. Throughput and average response time for service manager using three different parameter types (echo, struct and synthetic) and two approaches: (1) static stub, and (2) DII with a cache memory (DII)

5.2 Discussion

At present, we found that open source development tools for building, deploying and testing production quality work well together. Using a dynamic binding based approach and MIDlet's as client applications allows users to download a single time the application directly to their device over-the-air or via their PC. As a result, no update of the client application is required when new services are provided. In order to develop client applications using J2ME middleware platform, we have found the following drawbacks:

1. The Java Specification Request 172 (JSR-172) required for invocating Web services from a mobile J2ME application does not support UDDI specification and SOAP encoded messages.

2. There is only support to static stubs. Therefore, new stubs must be manually generated when new services are incorporated at runtime and clients need to download a new application in order to incorporate the new services.
3. Specific implementations must be developed in order to handling complex data types when dynamic invocation interface is used.

Also, we have found that the use of UDDI registry provides high rate of time responses and many of the services published at UDDI registry are not correctly published. Thus, all the services located through UDDI registry must be verified by the service manager, before the results are sent to the mobile user. However, these checking operations increase the response time to mobile users.

In order to provide flexibility in an environment with high rate of change we design a dynamic discovery and invocation (DDI) service manager. We have found that the use of DII is more complex for the software developer because a more complex interface is required in relation to the use of static stubs or dynamic proxies. We have compared the performance of the service manager under a static stub and a DII approach. The results show that DII approach offers better performance than static stub approach when a cache memory is used.

6. Conclusions and Future Work

Standard Web services infrastructures are focused on static stub based invocation of Web services. However, this scenario is not appropriate for mobile environments, where services and clients have a high rate of change. In order for Web services to expand across the mobile phones, users need to be able to efficiently discover and access to Web services at runtime. In this paper, we address the issues, challenges, implementation and performance in the use of dynamic discovery and invocation of Web services in mobile phones using J2ME middleware platform. We propose a Web service based proxy that acts as a DDI client over the network of services and as server to the mobile devices. With this approach, mobile consumers may locate new services at runtime without updating their client application. Also, interactions between the mobile phones and the network are reduced. Making DII is programmatically more complex than using a static stub. However, the advantage of using DII is that make the code easy to modify if the Web service details change and/or new services are offered at anytime. We have measure the performance using Sun benchmark with the purpose of comparing Web service proxy performance under a static stub and a DII approach. The results show that DII approach offers better throughput and average response time than the static stub approach when a cache memory is used.

Once we have tested the viability of dynamic mobile services, our future work will be focused on extending our approach in order to explore other approaches to handling complex data types based on XML messages when dynamic invocation is used, to perform more complex services and incorporate semantics, context-awareness and security aspects.

References

1. Elena Sánchez-Nielsen, Sandra Martín-Ruiz, Jorge Rodríguez-Pedrianes. "An open and dynamical service oriented architecture for supporting mobile services". Proceedings of ACM ICWE 2006, pp. 121-128, Palo Alto, California, July 2006.
2. OSGi Alliance. http://www.osgi.org/
3. Liberty Alliance Project. https://www.projectliberty.org/
4. R.S. Hall and H. Cervantes. "Challenges in Building Service-Oriented Applications for OSGi", IEEE Communications, Volume 42, Number 5, May 2004.
5. Gustavo A.., Casati, F., Kuno H., Machiraju, V. "Web Services: concepts, architectures and applications". Springer-Verlag Publications, Berlin 2004.
6. Vinoski, S,. "Web Services Interactions Models, Part 1: Current Practice". IEEE Internet Computing, 6(3), 2002.
7. W3C: World Wide Web Consortium. Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap/
8. W3C: World Wide Web Consortium. Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl
9. UDDI. Universal Description, Discovery and Integration. http://www.uddi.org/
10. Java Platform, Micro Edition (Java ME). http://java.sun.com/javame/index.jsp
11. Apache Axis. http://ws.apache.org/axis
12. UDDI4j. http://uddi4j.sourceforge.net
13. Sun Java Wireless Toolkit. http://java.sun.com/products/sjwtoolkit
14. Eclipse. http://www.eclipse.org
15. Apache tomcat. http://tomcat.apache.org/index.html
16. Javassist – Java Programming Assistant. http://www.jboss.org/products/javassist
17. T. Pilioura, A. Tsalgatidou, S. Hadjiefthymiades. "Scenarios of using Web Services in M-Commerce". ACM SIGecom Exchanges, Vol. 3, Nº 4, January 2003, pp. 28-36.
18. V. Sacramento, M. Endler, H. K. Rubinsztejn, L.S. Lima, K. Gonalves, and F.N. do Nascimento. MoCA: A Middleware for Developing Collaborative Applications for Mobile Users. IEEE Distributed System Online, 2004.
19. J. keeney, V. Cahill. Chisel: A Policy-Driven, Context-Aware, Dynamic Adaption Framework. IEEE 4th International Workshop on Policies for Distributed Systems and Networks, June 2003.
20. TinyXML. http://www.grinninglizard.com/tinyxml/
21. VoiceXML. http://www.w3.org/TR/voicexml20/
22. Z. Maamar, Q.Z. Sheng and B. Benatallah. "On composite Web Services Provisioning in an Environment of Fixed and Mobile Computing Resources". Information Technology and Management 5, 251-279, 2004. Kluwer Academic Publishers.
23. F. Papadopoulos, A. Zarras, E. Pitoura, P. Vassiliadis. "Timely Provisioning of Mobile Services in Critical Pervasive Environments". Lectren Notes in Computer Sciences LNCS 3760, pp. 864-881, 2005.
24. Daniel A. Menascé. "QoS Issues in Web Services". IEEE Internet Computing, pp. 72-75, December 2002.
25. WSTest. Sun Microsystems. https://wstest.dev.java.net/