Wireless Sensor Network, **2010, 2, 447-461**
doi:10.4236/wsn.2010.26056 Published Online June 2010 (http://www.SciRP.org/journal/wsn)

Web Services Invocation over Bluetooth

Auletta Vincenzo, Blundo Carlo, De Cristofaro Emiliano, Raimato Guerriero
Dipartimento di Informatica ed Applicazioni, *Università di Salerno*, *Salerno*, *Italy*
E-mail: {*auletta*, *carblu*, *emidec*, *raimato*}*@dia.unisa.it*

Received December 17, 2009; *revised January* 20, 2010; *accepted January* 26, 2010

Abstract

Over the last years, technology evolution is leading the way towards autonomous, ubiquitous and widespread interactions among small computing devices. To this aim, communication technologies that support dynamicity and mobility and work on inexpensive small devices have attracted much attention. The Bluetooth specification particularly fits this idea, providing a free, versatile, and flexible wireless network technology with low power consumption. On the other hand, as the degree of penetration of computational services has increased in everyday life, users' habits have deeply changed, resulting into an increasing request for mobile and ubiquitous services. In a few years, most of the devices accessing the web services will be mobile. Therefore, we need solutions that encompass networking and application issues involved in realizing mobile and ubiquitous access to the services. In this paper, we analyze how Bluetooth can be used to design, develop, and deploy Web Services-based applications that run on mobile devices. We propose and evaluate a framework that allows the interaction with Web Services from mobile devices using Bluetooth as communication channel.

Keywords: Bluetooth, Web Service, Communication Channel

1. Introduction

The technology evolution. Advances in communication technologies drove a deep transformation of users habits, in particular with an increasingly requirement of support to mobility and connectivity. Up to a few years ago mobile devices were very simple and resource limited. As a result, applications produced for these devices were bounded to the device environment. Nowadays, mobile devices such as smartphones or PDAs have enhanced their range of action, turning into fundamental working instruments. Modern applications, however, require connectivity and thus a critical issue for the penetration of mobile devices is the capacity to run network applications, especially Web applications. In the last years, several new protocols have been presented for wireless communications, such as IRDA, WLAN, and GPRS/ UMTS. However, IRDA connections are limited to two devices with a direct line of sight, and thus IRDA is not practically useful for a real intercommunication scheme. WLAN instead has been designed as a powerful technology to support multipoint connections, but penetration of WLAN on mobile devices and particularly on smartphones is still low. GPRS/UMTS are widely supported but they provide connectivity at modest speed and require a personal account with a phone company. At the same time, we witnessed the growth of Bluetooth [1], that is a low-cost, robust, powerful, and flexible short-range wireless link layer technology with low power consumption. It operates in a license-free frequency range, so that user is not charged for accessing the network nor needs an account with any company, thus allowing a relevant decrease of communication costs. Nowadays, the evolution of Bluetooth technology is driven by the Bluetooth SIG, that consists of over 7000 member companies that guarantee a large support to this technology. In fact, Bluetooth technology is used in many wide-spread different devices, such as handhelds, mobiles, smart-phones, laptops, PDAs. A thorough overview on Bluetooth is given in [2] and [3].

Recent work. Lately, a research study [4] has forecasted that, within a few years, most of the devices accessing the Web and Web Services will be mobile and presumably most of them will be Bluetooth-enabled. Therefore, we need solutions that encompass networking and application issues involved in realizing mobile and ubiquitous access to the services. Several research groups are proposing frameworks for developing applications over Bluetooth-based networks (for instance, [5] and [6]) and evaluate the possibility of using this technology for building ad-hoc networks suitable for dedicated applications, such as voice transmission [7], audio streaming [8], context-aware applications [9], and Internet access point [10].

Research goals. In this paper, we analyze how Bluetooth can be used to design, develop, and deploy Web Services-based applications that run on mobile devices. In fact, we propose and evaluate a solution that allows the interaction with Web Services from mobile devices using Bluetooth as communication channel. We consider a scenario where mobile devices consume Web Services but do not offer them. Some preliminary results on the proposed solution can be found in [11] and [12]. Also, in [13], the authors address the problem of deploying Web Services on mobile devices, providing a solution that however relies on the expensive Bluetooth's PAN profile, which is available only on PDAs and requires a preliminary pairing of the devices.

Our solution instead relies on the standard and inexpensive JSR-82 API [14] and on a tunneling mechanism realized by an intermediate software layer to incapsulate HTTP packets into Bluetooth ones. In this way, the connections are state-less and without any preliminary pairing (details are given in Section 5).

Main Contributions. Our work achieves a twofold goal. First, our solution provides ubiquitous Bluetooth-based access to Web Services and it is completely transparent to both users and application programmers. Also, our solution is to be widely supported at no extra cost by mobile devices. To this aim, we have devoted our attention to the free Bluetooth technology as opposed to other wireless technologies.

Typical applications that we have in mind are: information retrieving (e.g., accessing train timetables in a station) or micropayment applications (e.g., buying ticket in a cinema or on a bus), but our solution puts no restricttion on what one user can require. We designed a framework in such a way that, for a programmer, it will be very simple to port a Web Services-client application from a desktop environment (Axis Client API-based) to a mobile Bluetooth-enabled device. Indeed, we developed a J2ME package (named wsbt) exposing the entire Web-service stack to the client devices. For the programmer, it will be enough to change the package to import from org.apache.axis.client to wsbt.

Windows and Linux Implementation. In this work, we present two different implementations of our solution. We can summarize differences between the two implementations as follows: The first one is Windows-based, works on top of a third-party implementation of the Java API for using Bluetooth connections, and operates at a high level; the second one is Linux-based and works on top of our implementation of a Java package for exploiting Bluetooth features giving to the programmer control over several low level parameters of the Bluetooth channel. Several motivations suggested us to provide both implementations. First of all we would like to have our software available on both Windows and Linux, but, to our knowledge, there is no high level implementation of the Java API for Bluetooth for Linux, and, on the other hand, we have no direct access to the Bluetooth stack in Windows. The second motivation is to evaluate whether the low level control of the Bluetooth channel makes the programmers able to tune the communication parameters in order to significantly improve communication performances, such as latency and throughput.

Hence, we remark that the implementation of the Java API for Bluetooth in the Linux environment (that we named JBlueZen) is a programming effort of independently interest.

Efficiency. Our performance evaluations confirm the real applicability and lightness of the framework showing that Bluetooth is well suited to be the transport layer for Web Services accessing from wireless devices. It is worthwhile to notice that our proposed framework has a small footprint. Indeed, the Java code to be put onto the client, to get a web services client application running, needs just 50 KB of memory (including any external library).

Paper Organization. The rest of this paper is structured as follows. In Section 2, we highlight some general concepts about the pertinent technologies, such as Bluetooth, J2ME, and SOAP. In Section 3, we present an overview of our solution, and in Section 4, we show our design choices. The framework implementing our solution is presented and described in Section 5. In Section 6, we present the implementation of the framework client-side; while, in Section 7, two different implementations of the server-side are illustrated. Finally, in Section 8 we present and comment the results of our performance evaluations of the two proposed solutions.

2. Endorsed Technologies

The goal of this paper is to describe the design of a framework that allows Java programmers to easily and directly invoke Web Services from mobile devices over a Bluetooth connection. Hence, the basis for our work are Java 2 Micro Edition (J2ME) [15], the Standard Bluetooth.

Technology [16], and SOAP [17]. J2ME describes how to write Java applications on mobile devices and defines details for the communication between devices. Bluetooth is a low-cost, flexible, robust short-range wireless networking technology with low-power consumption. SOAP is a protocol for exchanging XML-based messages over computer networks. In this section we describe all the technologies that will be used in our framework.

2.1. The Bluetooth Wireless Technology

The Bluetooth specification was introduced in 1994 by Ericsson to provide radio communications between mobile phones, headsets and keyboards. The specifications were then released by the Bluetooth Special Interest Group (SIG) [16] in September 1999. Within this technology, radio communications can take place by mean of integrated and cheap devices with small energy consumption. This technology achieves its goal by embedding tiny, inexpensive, short-range transceivers into electronic devices that are available today. Bluetooth devices operate in a license-free frequency range (starting from 2.4 GHz).

Bluetooth-enabled devices can dynamically *discove*r other devices in their range and their supported services, through an inquiry process.

An overview of the Bluetooth stack is presented in **Figure 1**. The *radi*o level is the lowest one and it defines the technical details of the communication.

The *baseban*d layer handles channels and physical links, providing services such as error correction and security. It supports multipoint communications through FH/TDMA (Frequency Hopping/Time Division Multiple Access). The master device is in charge of defining the hopping sequence to all the slave devices. A physical channel is shared between the master and a slave using a time division scheme in which data are transmitted in one direction at time, with transmissions alternating between the two directions.

Up in the stack we find: the *Lin*k *Managemen*t *Protoco*l (LMP) handling link setup, authentication, and link configuration; the *Hos*t *Controlle*r *Interfac*e (HCI) which provides a uniform method of accessing the Bluetooth baseband capabilities; the *Logica*l *Lin*k *Contro*l *an*d *Adaptatio*n *Protoco*l (L2CAP) which deals with data multiplexing and segmentation. Finally, on top of L2CAP, we find several data communication protocols. The main protocols are:

1) SDP (Service Discovery Protocol), which handles the discovery of devices and services within the device's transmission range.

2) RFCOMM, which implements emulation of serial connections, setting up point-to-point connections. It supports framing and multiplexing and achieves all the required functions for serial data exchange.

3) OBEX (Object Exchange), which is built on the top of RFCOMM to implement exchange of objects, such as files and vCards. Originally, it was developed by IrDA (Infrared Data Association) for IR-enabled devices.

4) TCS (Telephony Control protocol Specification), which defines ways to send audio calls between Bluetooth devices.

The Bluetooth technology is also composed by a set of profiles. Bluetooth profiles describe several scenarios where Bluetooth technology is responsible of transmission. Each scenario is described by a user model and the corresponding profile gives a standard interface that applications can use to interact with the Bluetooth protocols. The profile concept is used to decrease the risk of interoperability problems between different manufacturers' products.

In order to interface applications to the physical layer, a Bluetooth Stack implementation is necessary. The stack provides a standard interface between the application layer and the Bluetooth specification. This interface is used to overcome the compatibility problems between the application and different Bluetooth devices. Indeed, Bluetooth stacks are responsible of implementing the Bluetooth wireless specifications. There are several different stacks targeted to different devices, applications, and operating systems. Currently available Bluetooth stack implementations are:

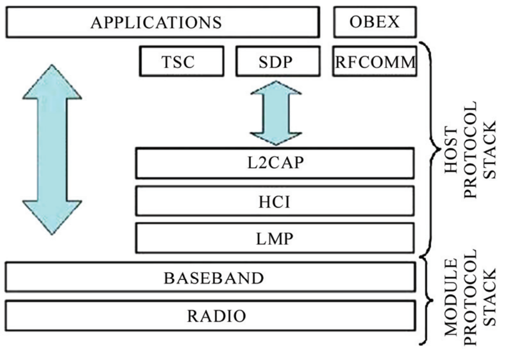

Figure 1. The bluetooth stack.

1) Mobile devices vendors' embedded stacks. Vendors providing Bluetooth-enabled devices have to build their own Bluetooth stack; for smartphones stack implementations obviously depend on the OS (e.g., Symbian).

2) Broadcom BTW (not free) [18]. It is addressed to PC OEMs and accessory manufactures to quickly and easily add Bluetooth technology to desktop PC and notebooks running Windows.

3) Microsoft BT Stack [19]. It is the Microsoft version of the Bluetooth stack and it is embedded in Windows XP SP 2. It provides the support for most of Bluetooth profiles, essentially the ones based on the RFCOMM protocol.

4) BlueZ (free and open-source) [20]. It is the Linux Bluetooth Stack. The code is licensed under the GNU General Public License and is included in the Linux 2.4 and Linux 2.6 kernel series. It provides a direct access to the transmission layer and allows developers to set several parameters of the communication.

2.2. J2ME

The J2ME (Java Platform Micro Edition) is a collection of Java APIs for developing applications targeted to resource-constrained devices such as PDAs and smartphones. Formally, J2ME is an abstract specification, but the term is frequently used also to refer to runtime implementations. The advantages of using Java as programming language are code portability and an increase of mobile devices flexibility. In particular, it provides support for deploying dedicated applications, named MIDlets, on the mobile device. They allow programmers to increase available features and capabilities of mobile devices. Since the range of micro devices is so diversified and wide, J2ME was designed as a collection of configurations, where each configuration is tailored to a class of devices. Each configuration consists of a Java Virtual Machine and a collection of classes that provide a programming environment for the applications. Configurations are completed by profiles, which add classes to provide additional features suitable to a particular set of devices. J2ME defines two configurations: the *Connecte*d *Devic*e *Configuratio*n (CDC) [21] and the *Connecte*d *Limite*d *Devic*e *Configuratio*n (CLDC) [22].

CDC is addressed to small, resource-constrained devices such as TV set-top boxes, auto telematics. It can add a graphical user interface and other functionalities; CLDC, instead, is addressed to devices with limited memory capacity. In this paper, we restrict our attention to the CLDC configuration. CLDC is a low level specification that includes a set of APIs providing basic features for resource-constrained devices, such as smartphones and PDAs. Producers should add features to CLDC by providing new libraries and thus creating a Profile. The first profile proposed for CLDC was the MIDP (Mobile Information Device Profile) [23]. MIDP is a set of Java libraries that permits to create an application environment for mobile devices with limited resources. Here, limitations include: amount of available memory, computational power, network communications with strong latency, and low bandwidth. MIDP 1.0 specification was produced by MIDPEG (MIDP Expert Group), as part of the JSR-37 [24] standardization effort; while, the MIDP 2.0 specification was released with the JSR-118 [25] standardization effort. MIDP 2.0 devices have to meet the following requirements:

1) *Memory*, 250 KB of non volatile memory for MIDP components, 8 KB for user data.

2) *Display*, 96 × 54 resolution, 1-bit color depth, 1:1 aspect ratio.

3) *Networking, bidirectional and wireless communication, limited bandwidth.*

2.3. JSR-82

Although the synergy between MIDP and J2ME technologies supplies a large number of communication schemes, it does not provide support for the Bluetooth technology. Therefore, the Java Expert Group JSR-82 [14] introduced the *Jav*a *APIs* *f*or *Bluetoot*h *Wireles*s *Technolog*y (JABWT) that provides a standard and high-level support for handling Bluetooth communications in Java applications. These APIs operate on top of CLDC to extend MIDP functionalities. Their development is still in progress, but about twenty mobile vendors have adopted them in their devices. The last released version (Version 1.1) provides support for:

1) Data transmission on the Bluetooth channel (audio and video are not supported).

2) Protocols: L2CAP, RFCOMM, SDP, OBEX.

3) Profiles: GAP, SDAP, SPP, GOEP

The Generic Access Profile (GAP) defines the generic procedures related to discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices. The Service Discovery Application Profile (SDAP) defines the features and procedures for an application in a Bluetooth device to discover services registered in other Bluetooth devices and retrieve any desired available information pertinent to these services. The Serial Port Profile (SPP) defines the requirements for Bluetooth devices' necessary for setting up emulated serial cable connections using RFCOMM between two peer devices. The Generic Object Exchange Profile (GOEP) defines the requirements for Bluetooth devices necessary for the support of the object exchange usage models.

The interaction between the J2ME environment and the Bluetooth API is shown in **Figure 2**. Using JABWT, it is possible to interact with the Bluetooth stack in a Java application. In particular, it is possible to call services such as device and service discovery, establishment of RFCOMM, L2CAP, and OBEX connections.

In order to use the Java APIs for Bluetooth, a real implementation of the JSR-82 specification is necessary on the device. The current JSR-82 implementations are:

1) Mobile devices vendors' embedded JSR-82 implementations.

2) Atinav aveLink suite (not free) [26]. It offers both an implementation of the Bluetooth stack and the implementation of all the standard profiles for ANSI C, JSR-82 for J2SE Java, JSR-82 for J2ME, Windows and Windows CE.

3) Impronto Rococo (not free) [27]. It is a complete product that provides the Bluetooth Stack and the integration layer, the JVM and the JSR-82 implementation layer both for J2SE and J2ME.

4) Avetana (not free) [28]. It enables writing J2SE applications to access the Bluetooth layer; it is available for Windows, MacOS X, and Linux platforms.

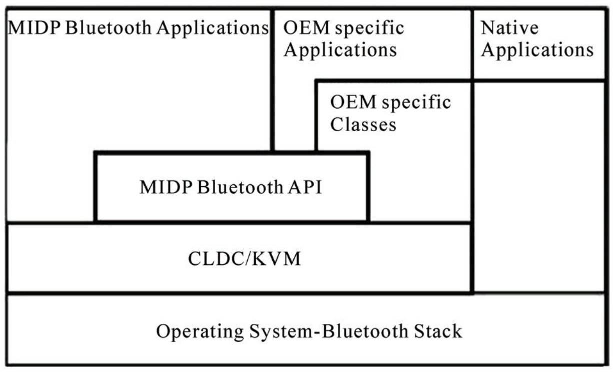
MIDP Bluetooth Applications
OEM specific Applications
Native Applications
OEM specific Classes
MIDP Bluetooth API
CLDC/KVM
Operating System-Bluetooth Stack

Figure 2. J2ME-Bluetooth API interaction architecture.

5) BlueCove (free) [29]. It provides the Java JSR-82 support for J2SE applications over the Windows XP SP2 Bluetooth stack.

2.4. SOAP

SOAP is a lightweight protocol for exchanging information in a distributed environment. It is an XML based protocol consisting of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for ex-pressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses [30].

In our framework the client application runs on a mobile device; it is implemented as a MIDlet and communicates with the Web Services Container through SOAP messages. Thus, we need a J2ME-compatible library that can be used in a MIDlet to serialize/deserialize messages according to the SOAP specification. Our framework uses the kSOAP library [31] to serialize/deserialize the messages on the client side. The kSOAP library provides highlevel classes to construct the envelope, the body, and the header of the SOAP message and to specify their elements. Such a library supports a subset of SOAP 1.1 features. Indeed, the library supports only simple data types such as strings, integers, etc. However, it is possible to extend these basic functionalities so that any complex datatype can be included in the SOAP message. One has to simply provide custom classes for these data types. Such classes have to implement the Java KvmSerializable interface.

3. Architecture Overview

In this section we will present a lightweight framework that allows an application programmer to design client applications running on mobile Bluetooth-enabled devices that invoke remote Web Services. We assume a client-server interaction, where client and server communicate over a Bluetooth channel using SOAP [17] as messaging system. We refer to the common scenario shown in **Figure 3**.

We refer to the usual architecture, where a client exchanges SOAP messages with a server using HTTP as the transport protocol to invoke a Web Service. We observe that wireless communication by means of HTTP over GPRS and wLAN is widely supported. On these channels it is easy to create an HTTP connection and invoke remote Web Services using HTTP as a transport protocol. However, to the best of our knowledge, there is no implemented support for executing an HTTP POST operation over a Bluetooth channel within a J2ME MIDlet. To overcome this limitation, we introduced in our framework a new entity (*i.e.*, a distributed proxy) that takes care of binding SOAP messages to the Bluetooth transport protocol. Such entity interfaces clients to Web Services Containers. In this way, we can maintain the same server-side architecture and guarantee the interoperability of the application running on the mobile device with any Web Service (see **Figure 4**). Therefore, the architecture of our framework is based on three different entities:

1) CLIENT. It runs on a Bluetooth-enabled mobile device and it invokes the Web Service on a Bluetooth channel.

2) PROXY. It interfaces clients with the Web Services Container.

3) WSC. The Web Services Container replies to clients' requests communicating through the PROXY.

4. Design Choices

Our design choices descend from our prerequisites of having a framework to invoke web services over a Bluetooth connection that is:

1) *Transparent*: in the sense that it should allow programmers to develop device-independent applications.

2) *Data independent*: in the sense that client applications could exchange any kind of data, even user-defined.

Moreover, as implementation constraint, we restricted our attention to license free implementation of the Bluetooth stack.

In order to obtain transparency to programmers, we have to use the JSR-82 API standard [14], allowing applications to run both on mobile devices with limited computation power using J2ME as environment and on powerful computer machines using J2SE environment. Then, to our knowledge, we have only two alternatives that are license free.

1) To use the BlueCove JSR-82 implementation that interfaces with the Microsoft Bluetooth stack found in Windows XP starting from the SP2 version.

2) To use BlueZ, the Linux Bluetooth stack, and provide a JSR-82 implementation for BlueZ.

Obviously, the first choice is an off *the shelf* solution; while, the second one requires an *in-house* development.

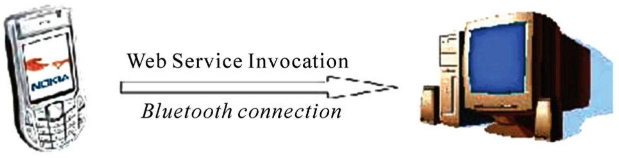

Figure 3. Our scenario.

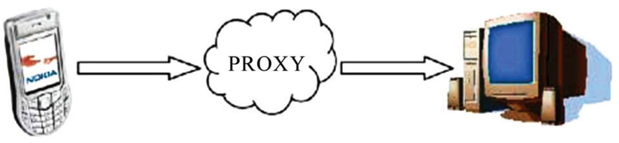

Figure 4. Proxy invocation scenario.

But the latter allows us to operate in a completely free and open environment. Besides, BlueCove, when used on winsock, supports only RFCOMM as communication protocol and it does not allow setting low level parameters on the connection as BlueZ does. Since our goal is to perform remote procedure calls using the Web Services technology, we have considered two solutions for the message exchanging protocol: XML-RPC [32] and SOAP [17]. XML-RPC is a way to perform procedure calling using HTTP as transport protocol and XML as encoding protocol; while, SOAP is a lightweight protocol for exchange of information in a decentralized and distributed environment. To have a complete data independent infrastructure, we have chosen JAX-RPC [33]. In fact, even if XML-RPC is lighter, easier to use for developing purposes and more suitable to the bandwidth limitation of the Bluetooth channel, the JAX-RPC solution provides more powerful features and advantages:

1) SOAP passes parameters by name while XML-RPC passes parameters by position, resulting in a dependance on the order of parameters.

2) SOAP allows user-defined record types by extending the XML document using XML Schemas; while, XML-RPC only allows the base types defined in the specification.

3) Both SOAP and XML-RPC support passing binary data in an XML document using Base-64 encoding, but XML-RPC defines string parameters as being ASCII text. Some XML-RPC servers will enforce this, forcing the user to pass internationalized text as Base-64 encoded data.

4) XML-RPC is defined as operating over an HTTP connection, while SOAP describes the envelope format for an RPC request which may be sent over HTTP, SMTP, or any other protocol.

In particular, we consider the *Axi*s *Clien*t API [34] as the model for carrying out client-side interactions. AxisClient is part of the *Axi*s API [35]. This API is license free and it is the most diffused free SOAP engine. Since in our scenario client applications are mobile and run on resource-constrained devices, we consider the kSOAP [31] library, a SOAP API suitable for the Java 2 MicroEdition, based on kXML [36]. The feature set of kSOAP is a subset of SOAP 1.1 features. It provides classes and methods to construct the envelope, the body, and the header of the SOAP message and to specify their elements.

5. The WSBT Framework

The goal of our work is to investigate the possibility of using Bluetooth as the communication channel for Web Service invocation from a mobile client. J2ME and JSR-82 give an appropriate programming environment for achieving our goal, but they do not provide the required support for directly accessing a Web Services Container over a Bluetooth connection from a mobile client. This is due to the impossibility of addressing IP-based transport protocols (e.g., HTTP, FTP, ...) using Bluetooth as the physical layer. Bluetooth SIG does define appropriate profiles and protocols, (e.g., PAN profile) but their use requires a preliminary pairing of the devices. Thus, a middleware needs to be created on top of the operating system level to incapsulate HTTP packets in the Bluetooth ones. However, Bluetooth communications established within the JSR-82 API are state-less and without any preliminary pairing. They only allow sending and receiving bytes over L2CAP or RFCOMM connections. For this reason, we implemented a tunneling mechanism by introducing an intermediate software layer which acts as a distributed proxy to achieve Web Services invocation over Bluetooth. Our PROXY allows application developers to deploy Web Services client applications with no extra work required to use Bluetooth as the physical layer. The PROXY is a double sided software entity:

1) MASTER (server-side of the PROXY). It extends the Web Service Container (WSC) features to accept requests coming over the Bluetooth channel. It acts as a master device, accepting incoming requests and sending back responses.

2) SLAVE (client-side of the PROXY). It lies on the smartphone and essentially it binds SOAP messages to the Bluetooth physical layer.

The result is a lightweight framework, which we named WSBT (Web Service over Bluetooth). Our proposed framework has a small footprint. Indeed, the code of the jars that need to be put onto the client to get a web services client application running needs just 50 KB of memory (this figure also include the kSOAP library whose size is 41 KB). The entities of WSBT are shown in **Figure 5**, where a CLIENT addresses Web Services on the Container through the use of the two-sided proxy.

The invocation of a service on WSC is executed by following the next steps which are summarized in **Figure 6**:

1) The CLIENT uses the classical mechanism to invoke a Web Service, ignoring the presence of the Proxy and all the details about the Bluetooth communication.

2) The SLAVE is in charge of serializing data to prepare SOAP requests for a remote Web Service.

3) The SLAVE discovers the MASTER and establishes a Bluetooth connection (either L2CAP or RFCOMM) to send the SOAP request message as a raw byte stream.

4) The MASTER receives the byte stream representing the SOAP message and posts it to the WSC.

5) The WSC elaborates the SOAP request message and returns a SOAP response message that is bound in the HTTP POST response packet.

6) The MASTER forwards back responses over the Bluetooth channel as rough byte stream, without interpreting them.

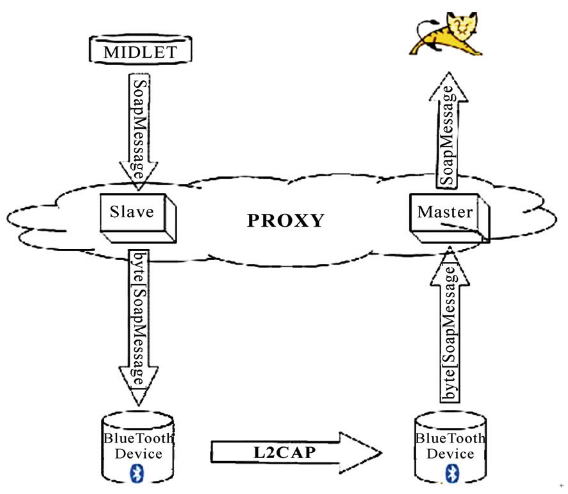

Figure 5. The entities of the WSBT framework.

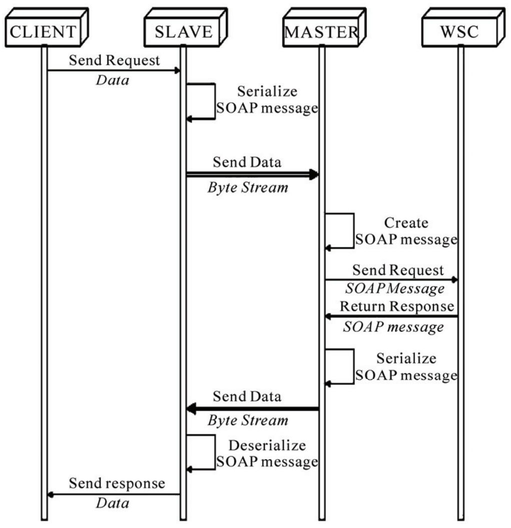

Figure 6. Sequence diagram of the WSBT framework.

7) The SLAVE receives the byte stream from the Bluetooth channel, reconstructs the SOAP return message, deserializes it, and returns results to the CLIENT.

6. Proxy's Client-Side (SLAVE)

In order to guarantee transparency to the programmers, we want to design APIs such that all details related to the use of Bluetooth as the communication channel are hidden. Our APIs are modeled on the AXIS Client API and implement Web Services invocations in a JAX-RPC [33] compliant way. To this aim, we developed a J2ME package (wsbt) whose structure is given in **Figure 7**. The Call class is the core of our framework and it is in charge of implementing the invocation mechanism. It provides methods of the Axis Client's Call interface [37]. Therefore, porting a Web Services-client application from a desktop environment (Axis Client API-based) to a mobile Bluetooth-enabled device is very simple. Indeed, it is enough to change the package to import form org. apache.axis.client to our wsbt.

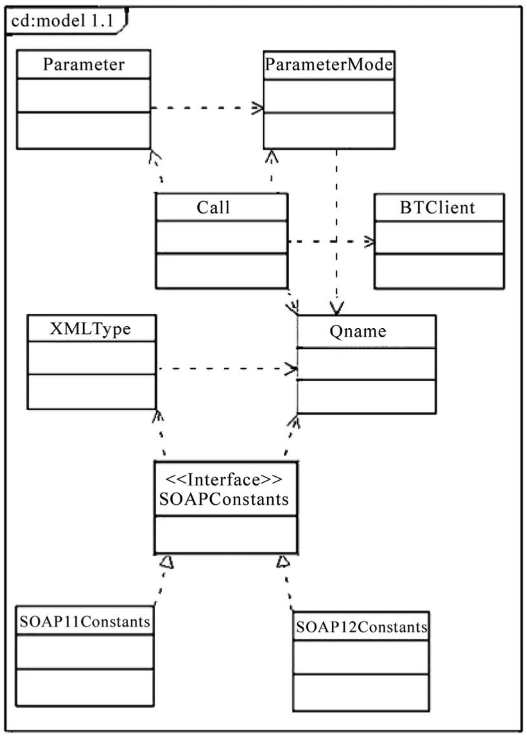

Figure 7. Structure of the wsbt Java Package.

Figure 8 depicts the fragment of a Java code for the invocation of a simple EchoString service. Invocations involving user-defined classes are not much harder to code: they only require the registration of the type mapping.

The actual invocation is carried out by the invoke method. This method, whose code is given in **Figure 9**, takes as input the list of parameters and constructs a SOAP message. This SOAP message is serialized and then sent over the Bluetooth channel, as a raw array of bytes. Finally, the method waits for the SOAP response message, deserializes it and returns it. More precisely, the invoke method performs the following operations:

1) Create a SoapObject through the use of the kSOAP library. We need to specify the namespace for the soap object and the operation name associated to the Call object.

2-5) Add to the SoapObject, through the method addProperty, all the input parameters for the operation associated with the Call object.

6) Serialize the SoapObject, *i.e.*, create an XML message representing the SOAP request. Details are hidden into the private method serialize.

7-8) Create an instance of the BTClient class and use the performInvocation method to send the SoapRequest message over the Bluetooth channel and to get the Soap-

Response.

9) Deserialize the response by invoking the private method deserialize and return the result.

We remark that the Bluetooth communication is handled by the *BtClient class*. When the *performInvocation* method of the *BtClient* object is invoked the following operations are performed:

1) Discovery of the Bluetooth Master;

2) Instauration of a L2CAP or RFCOMM channel;

3) Transmission of the request message over the channel and reception of the response message.

The code of performInvocation is given in **Figure 10**.

7. Proxy's Server-Side (MASTER)

The server-side of the proxy will run as a J2SE application on a desktop computer. Design of this part has been driven by the requirements described in Section 4.

As previously discussed, the MASTER accepts client Bluetooth connections, gets SOAP requests, posts them to the WSC, and sends back obtained SOAP responses. It consists of two main classes: the BTServer and the Poster.

The BTServer class takes care of:

1) setting the device in *discoverabl*e mode

2) activating a listening connection

3) accepting incoming connections

4) performing I/O on the Bluetooth channel

5) instantiate a Poster object, passing to it the URL of the service to invoke and the SOAP request to send.

The Poster class is in charge of:

1) performing an HTTP post operation on the WSC, using the Apache *Common*s *Http-Clien*t package

2) giving back the SOAP request to the BTServer object

```
(1) Call call = new Call();
(2) String address = "http://localhost:8080/axis/EchoService";
(3) call.setTargetEndpointAddress(address);
(4) call.setOperationName("echo");
(5) call.addParameter("toEcho", XMLType.XSD STRING, ParameterMode.IN);
(6) call.setReturnType(XMLType.XSD STRING);
(7) String result = (String)
(8) call.invoke(new Object[] [] { new String("Hello World")});
```

Figure 8. Invocation of the EchoString service.

```
public Object invoke(Object[] args) {
(1) SoapObject method = new SoapObject("urn:method", operation);
(2) for (int i = 0; i < params.size(); i++) {
(3)     Parameter tmp = (Parameter) params.elementAt(i);
(4)     QName qname = tmp.getXmlType();
(5)     method.addProperty(tmp.getType(), args[i]); }
(6) String soapRequest = serialize(method);
(7) BTClient bt = new BTClient();
(8) String soapResponse = bt.performInvocation(address, soapRequest);
(9) return deserialize(soapResponse);
}
```

Figure 9. Java code for invoke.

```
public String performInvocation(String address, String soapRequest) {

        /* Check if the address is already in cache */
        String BT_url = queryCache(address);

        /* If not in cache, perform discovery and update the cache */
        if (BT_url == null)
           BT_url = discoverMaster(address);

        /* If the discovery has failed, throw an Exception
        if (BT_url == null)
           throw UnavailableMasterException;

        /* Once the BT url has been recovered perform the connection */
        /* Get a L2CAP or RFCOMM connection */
        _connection = Connector.open(BT_url);

        /* Send the SOAP request and get the SOAP response over the BT channel */
        String soapResponse = null;
        try {
                send(address, soapRequest);
                soapResponse = receive();
                _connection.close();
        } catch (IOException ioe) { }

        return soapResponse;
}
```

Figure 10. Java code for the performInvocation method of BTClient class.

In order to have a licence-free and JSR-82 compliant implementation of our Proxy's server-side, we have considered two alternatives:

To use BlueCove [29], the free implementation of the JSR-82 API is within the Microsoft Windows XP SP2.

To use BlueZ [20], the Linux Bluetooth stack, and provide a JSR-82 implementation for BlueZ.

7.1. BlueCove-Based Implementation

BlueCove [29] is a free implementation of the JSR-82 API that runs over the Windows XP SP2 Bluetooth stack. When used on winsock, it only provides the support for RFCOMM protocol, which is the emulation of a serial port communication enabling programmers to open inbound DataInputStream and outbound DataOutputStream. According to the JSR-82 API, the Bluetooth listener is implemented by a Notifier object, which handles a StreamConnection. **Figure 11** shows main steps of the proxy's server-side.

The Java-code in **Figure 11** executes the following operations:

1) Set the device in discoverable mode.

2-3) Activate a listening connection on localhost, on the channel 1, named "rfcomm test".

4) Accept incoming connections.

5) Open an InputStream on the connection.

6-7) Read data on the stream.

8-9) Post the SOAP request at the specified address, using the Poster class, to get the Soap response.

10) Open an OutputStream on the connection.

11) Write data, *i.e.*, the Soap response.

7.2. JBlueZen: BlueZ-based JSR-82 API Implementation

BlueZ is the implementation of the Bluetooth stack included in the Linux kernels 2.4 and 2.6. It gives to programmers direct access to the transmission layers and allows to set up several communication parameters to tune the transmission to the application characteristics. To our knowledge, there is no JSR-82 implementation that runs over BlueZ and thus we have written our own. As presented in **Figure 12**, the BlueZ stack offers to programmers a Berkeley socket interface (C-based) to handle L2CAP, RFCOMM, and SDP features. As a result, we had to implement a set of C functions which access Berkeley sockets in order to accomplish data exchange and service discovery. These functions will be then interfaced with Java applications through the use of the Java Native Interface (JNI) [38]. The resulting scheme is the Java-JNI package presented in **Figure 13**, which we named JBlueZen. Programming on RFCOMM sockets to build a RFCOMM server is very similar to programming on TCP/IP sockets, with some relevant differences like the maximum number of allowed ports (65536 for TCP/IP, 30 for RFCOMM) and different functions for the byte-ordering (big-endian, littleendian). The main steps performed by our implementation to open a listening RFCOMM socket are shown in **Figure 14**.

1) Create a socket: AF BLUETOOTH indicates that we are using the Bluetooth communication channel, SOCK STREAM that it is stream-oriented service, and BTPROTO RFCOMM that we are using RFCOMM.

2-4) Populate the loc addr structure used to set information over the Bluetooth adapter in the bind system call: use Bluetooth as the communication channel, accept connection from any device and on the specified channel (*i.e.*, the RFCOMM mechanism for implementing a port).

5) Bind the socket on a listen port.

6) Listen on the socket for incoming connections.

7) Accept incoming connections and get a communication socket through.

We remark that using the *setsockop*t system call, it is also possible to set some socket options, such as the authentication and encryption to use on the Bluetooth transmission, or the device role (MASTER or SLAVE). For L2CAP, it can also be set the maximum amount of consecutive bytes that can be sent/received on the connection (MTU).

The mechanism for L2CAP is similar to RFCOMM and it is shown in **Figure 15**. The only difference here is the type of socket to be used: SOCK SEQPACKET instead of SOCK STREAM.

The last protocol to implement is the Service Discovery, which requires a bit more coding effort. BlueZ provides a set of C functions to address service discovery

```
(1)  (LocalDevice.getLocalDevice).setDiscoverable(DiscoveryAgent.GIAC);
(2)  (StreamConnection) notifier = (StreamConnection)
(3)  Connector.open("btspp://localhost:1;name=rfcommtest;master=true;encrypt=false;
                    authorize=false;authentication=false;receiveMTU=512;transmitMTU=512");
(4)  notifier.acceptAndOpen();
(5)  InputStream input = notifier.openInputStream();
(6)  /* Perform buffered readings to get the sopaRequest */
(7)  String soapRequest = input.read();
(8)  Poster poster = new Poster(address);
(9)  String soapResponse = poster.doPost(soapRequest);
(10) OutputStream output = notifier.openOutputStream();
(11) output.write(soapResponse.getBytes());
```

Figure 11. Java code for RFCOMM implementation proxy's server-side.

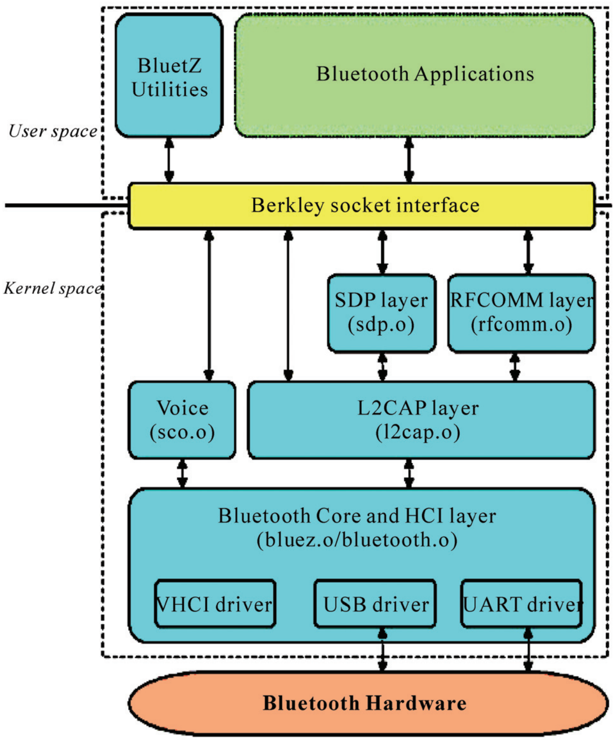

Figure 12. The BlueZ Bluetooth stack.

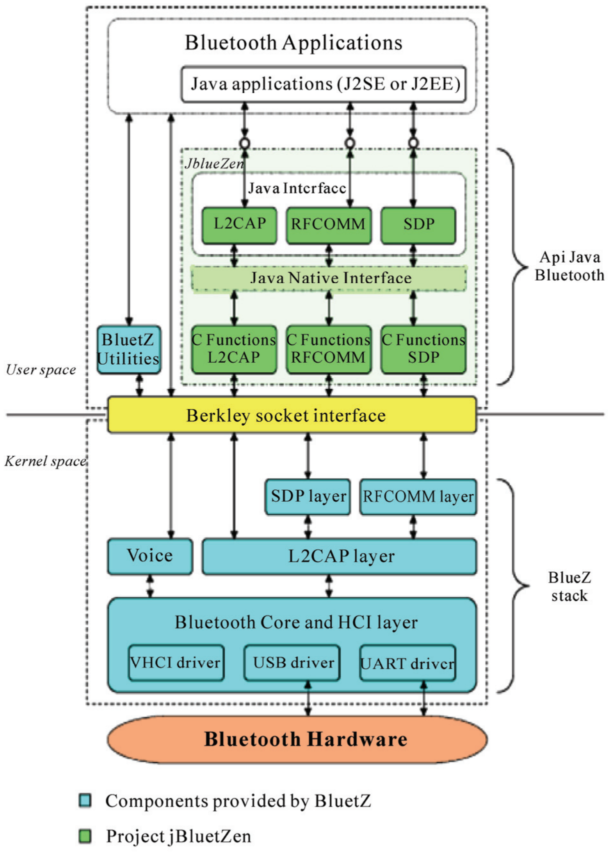

Figure 13. The JBlueZen package.

features both on the 'server' side and on the 'client' side. To implement the discovery in the JSR-82 standard way, on the server-side, we had to handle the service record and register services on it; on the client-side we had to access the HCI to perform inquiry and connect to the SDP server running on the remote device to get all the info. The last step was to provide a Java interface (JBlueZen) to the described functions we implemented in C. This was made through the use of the JNI to implement all the communication (L2CAP and RFCOMM) and the discovery features in the JSR-82 standard way.

7.3. JBlueZen-Based Implementation

Using the Java interface of our package JBlueZen, we are able to implement the MASTER (proxy's server-side) on Linux. The mechanism is similar to the Windows implementation, but in this case we can choose among two different communication protocols: L2CAP and RFCOMM and we can set many parameters on the Bluetooth connection which were fixed in the Windows implementation, such as the device role (MASTER or SLAVE) or the types of packet to use (DH-DM 1-3-5).

Obviously, the code fragment that uses the RFCOMM implementation is identical to the code given in **Figure 14**. The L2CAP implementation is instead implemented as shown in **Figure 16**.

The Java code in **Figure 16** executes the following operations:

1) Set the device in discoverable mode.

2) Activate a listening connection on the localhost, on the psm 1001, named "testl2cap", and with the desired settings. Remark that psm is the Protocol Service Multiplexer and is the mechanism used by L2CAP to implement multiplexing on connections.

3) Accept incoming connections.

4-5) Get the in/outbound maximum transfer unit.

6) Read data on the stream.

7-8) Post the SOAP request at the specified address, using the Poster class, to get the Soap response.

9) Write data on the connection.

8. Performance Evaluation

In this section, we present the results of several experiments that we ran to evaluate the real applicability and the lightness of the deployed framework. We also compare the performances of the two different implementations that we described in the previous sections. For our experiments, we set up the following small test-bed:

1) (Linux) Server-side: Workstation HP XW6000, Xeon 2.8 GHz Dual-Processor, 2 GB RAM, with Trust BT 180 Bluetooth USB dongle, running Fedora Core 4 with 2.6.11 Linux Kernel.

```
int sock = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
loc_addr.rc_family = AF_BLUETOOTH;
loc_addr.rc_bdaddr = *BDADDR ANY;
loc_addr.rc_channel = channel;
bind(sock, (struct sockaddr *)&loc_addr, sizeof(loc_addr));
listen(sock, MAX_NO_CONNECTIONS);
int sockfd = accept(sock, (struct sockaddr *)&rem_addr, &opt);
```

Figure 14. C-code to open a RFCOMM listening socket on BlueZ.

```
int sock = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
loc_addr.l2_family = AF_BLUETOOTH;
loc_addr.l2_bdaddr = *BDADDR_ANY;
loc_addr.l2_psm = psm;
bind(sock, (struct sockaddr *)&loc_addr, sizeof(loc_addr));
listen(sock, MAX_NO_CONNECTIONS);
int sockfd = accept(sock, (struct sockaddr *)&rem_addr, &opt);
```

Figure 15. C-code to open a L2CAP listening socket on BlueZ.

```
(LocalDevice.getLocalDevice).setDiscoverable(DiscoveryAgent.GIAC);
(L2CAPNotifier) notifier = (L2CAPNotifier)
        Connector.open("btl2cap://localhost:1001; name=testl2cap;
                         master=true;encrypt=false;authorize=false;
                         authentication=false;receiveMTU=512;
                         transmitMTU=512");
 L2CAPConnecction conn = (L2CAPConnection) notifier.acceptAndOpen();
 int inMTU = conn.getReceiveMtu();
 int outMTU = conn.getTransmitMtu();
 /* Perform cycling readings over the inMTU to get the soapRequest */
 String soapRequest <- conn.read();
 Poster poster = new Poster(address);
 String soapResponse = poster.doPost(soapRequest);
 conn.write(soapResponse.getBytes());
```

Figura 16. Java code for L2CAP implementatio of Proxy's server-side.

2) (Windows) Server-side: Workstation HP XW6000, Xeon 2.8 GHz Dual-Processor, 2 GB RAM, with Trust BT 180 Bluetooth USB dongle, running Windows XP SP2.

3) Client-side: Nokia 6630 Smartphone with Symbian OS, MIDP 2.0 and JSR-82 APIs support.

Our tests were aimed to evaluate the effciency of our framework in terms of used bandwidth and of serialize-tion/deserialization performances. Thus, we had to meas-ure times for serializing and deserializing messages and transmission times over the Bluetooth channel. Noticing that on a Bluetooth channel transmission and receiving times are significantly different. In fact, usually Bluet-ooth stack implementations assign a much larger bandw-idth to master-slave communications than to slave-master ones. Thus, the same message will require more time to be transmitted from the mobile device to the container than to be transmitted from the container to the mobile. In our experiments, we invoked a test Web Service impl-ementing just an echo service and measured only the transmission time from the mobile device to the cont-ainer. Clearly, these times are an upper bound to the times we could measure for transmissions in the other direction. We also measured the serialization/deserializa-tion times on the mobile device. Noticing that these op-erations are local to the mobile device and their execu-tion times do not depend on our framework but only on the implementation of the Java Virtual Machine and of the KSOAP library that are used on the mobile device. However, we have measured these times to evaluate the lightness and effectiveness of our framework for devel-oping applications invoking web services from mobile devices with limited resources.

To have a comprehensive analysis of the performances of our framework several experiments were run. In the first experiment, we measured times necessary to comp-lete a client request. We repeated the same experiment using both the Blue-Cove-based implementation and the JBlueZen-based implementation in order to compare the effciency of the two implementations. The second expe-riment was aimed to isolate and evaluate the impact of serialization and deserialization on the performance of the framework. The third experiment was aimed to measure the discovery delay. In fact, our framework allo-ws clients to dynamically discover Bluetooth servers by handling Bluetooth specification policies for devices and services discovery. In our last experiment, we tested framework's performances with different Bluetooth com-munication modalities, taking advantage of the capacity of the JBlueZenbased implementation to modify some low-levels Bluetooth communication parameters.

8.1. Transmission Times

Our first experiment was aimed to evaluate the time required to transmit a message on the mobile-container channel with respect to the size of the message. We measured the time taken to complete the invocation of the send method in the performInvocation method (see **Figure 10**). Notice that, since the IO is blocking, the send method returns only when all input data is sent and an ACK is received from the container for the correct reception of the last data byte. To compare our two implementations we repeated the experiment in three different cases: using the JBlueZen-based implementation both with a RFCOMM or L2CAP connection and using the BlueCove-based implementation with a RFCOMM connection. We remark that BlueCove, when used on winsock, does not allow setting a L2CAP connection.

We distinguished two cases: unstructured and structured messages. In the first case, we assumed that the message consists of a string (array of bytes) and we ran tests for strings of size ranging from 0.5 KB to 30 KB (the size is increased by 0.5 KB in each test). We were not able to perform tests for larger strings since in the JVM deployed on the mobile device used in our experiments (*i.e.*, Nokia 6630) it is not possible to instantiate String objects of size greater than 30 KB. In the second case, we assumed that messages contain complex data types. Our framework represents complex data types as JavaBeans. We assumed that the request message contains an array of Address objects, which consist of six String objects of fixed length and two Integer objects. We measured transmission times with respect to the length of the array and we ran tests for array lengths ranging from 1 to 65. **Figure 17** shows message lengths with respect to the length of the array. For each test, we repeated the invocation 50 times and computed the average times. To guarantee that for each iteration the device were in the same initial conditions, we used dedicated threads. In fact, for each invocation a new thread was created and destroyed after the operation.

Figures 18 and **19** show transmission times for unstructured messages with respect to the size of the string. Notice that the size of the string is not equal to the size of the message sent on the Bluetooth channel (we have to consider additional bytes inserted by KSOAP and by the serializer). However, we have observed that this overhead is constant (approx. 500 bytes) and it does not depend on the string's size. It can be seen that our two implementations are equivalent when using the RFCOMM communication modality, while L2CAP communication modality is 20% more effcient.

Figure 20 shows transmission times for structured messages with respect to the length of the array using a RFCOMM connection. Our experiments show that the two implementations are equivalent but the BlueCove-

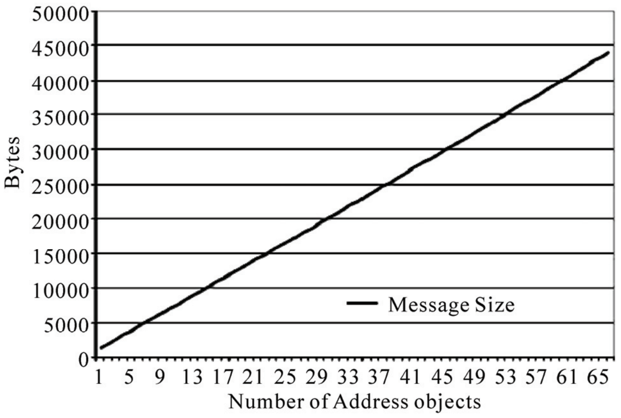

Figure 17. Array of address objects: message length (in bytes) vs. array length.

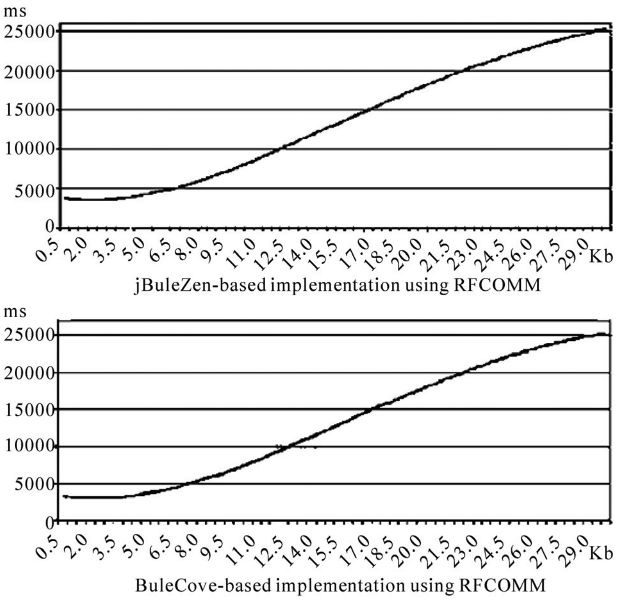

Figure 18. Times to send unstructured data using the RFCOMM communication modality.

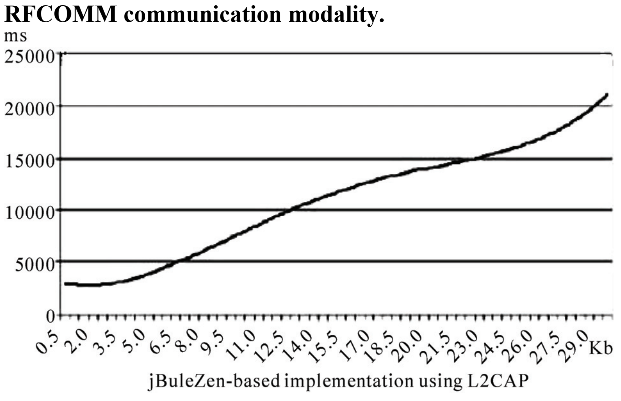

Figure 19. Times to send unstructured data using the L2 CAP communication modality.

based implementation is a little bit more effcient. Moreover, the average throughput is around 10 KBps. Finally, **Figure 21** shows transmission times for L2CAP connections (in the BlueZen-based implementation). As expected, performances are slightly more effcient using L2 CAP than usign RFCOMM.

8.2. Serialization-Deserialization

The second experiment was aimed to measure the processing times for serialization and deserialization to evaluate lightness of our framework. **Figure 22** shows times to serialize and deserialize SOAP messages containing arrays of Address objects with respect to the length of the array. Notice that serialization/deseraliziation times are equal for both our implementations since they depend only on the implementation of the Java Virtual Machine and of KSOAP library. We can observe that, when the array length is large enough, serialization is heavier than deserialization: differences between serialization and deserialization times are due to the different behavior of the kSOAP when parsing and serializing XML documents.

It can be seen that the serialization time increases linearly with the length of the message and it weights less than 1/3 of the transmission time. Thus, we can state that our framework is suffciently light and it has no dramatic impact on the effciency of the framework.

8.3. Discovery

This experiment was aimed to measure the discovery delay. In fact, our framework allows the client to dynamically discover Bluetooth servers and not to be bounded to hard-coded settings. For this reason, the framework handles Bluetooth specification policies for devices and services discovery.

We ran 50 discovery operations, getting only 1 timeout error (see **Figure 23**). In all the other cases, we measured a discovery delay that is around 14.5 seconds. However, introducing a cache mechanism to store addresses of recently used Bluetooth devices we were able to reduce delays to a few hundreds milliseconds (see **Figure 24**).

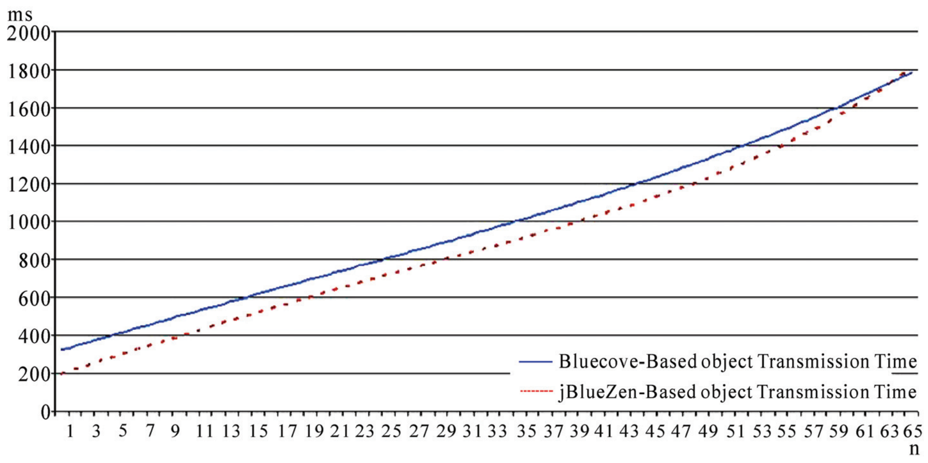

Figure 20. Total times to send structured data using RFCOMM communication modality.

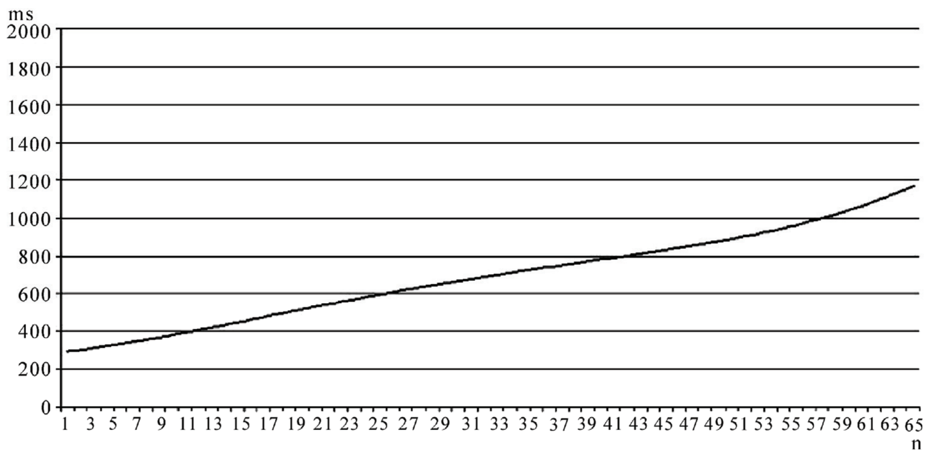

Figure 21. Total times to send structured data using L2CAP communication modality.

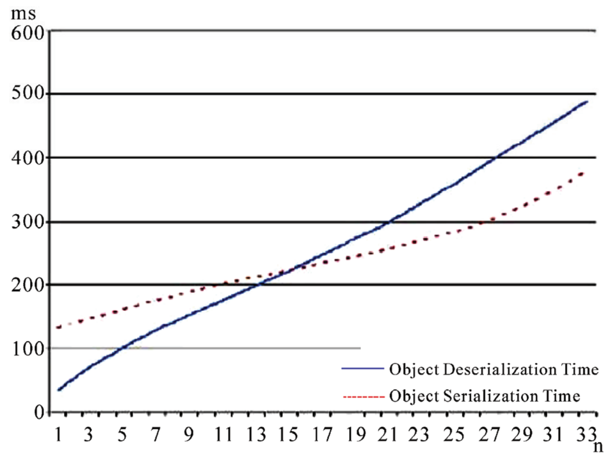

Figure 22. Serialization and deserialization times for an array of Address objects.

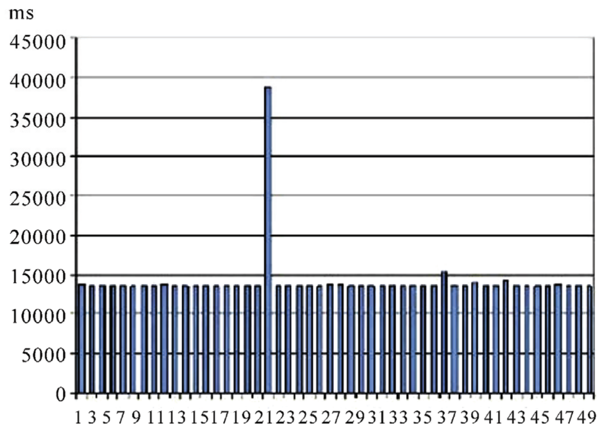

Figure 23. Discovery delay.

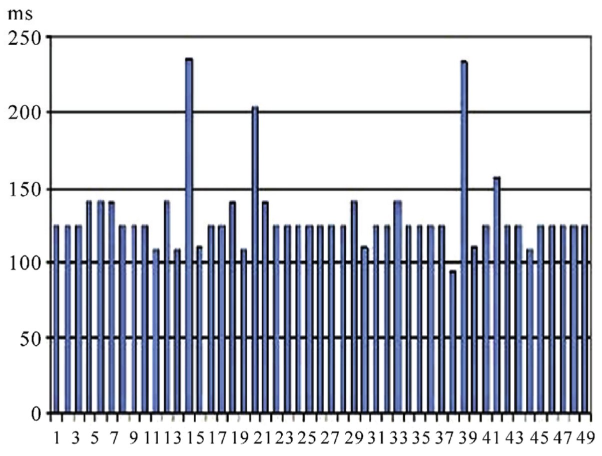

Figure 24. Discovery delay using cache.

9. Conclusions

In this paper, we presented a framework to allow applications running on mobile devices to invoke Web-Services over a Bluetooth-connection. We gave two different implementations of our framework, and give an extensive analysis of their performances. We can summarize differences between the two implementations as follows: the first implementation is Windowsbased and works on top of BlueCove, that is a third-party free implementation of the JSR-82 API that runs over the Windows XP SP2 Bluetooth Stack; the second one is Linux-based and works on top of our implementation of a Java package for exploiting Bluetooth features giving to the programmer control over several low level parameters of the Bluetooth channel.

Our experiments confirm the real applicability and lightness of the framework showing that Bluetooth is well suited to be the transport layer for Web Services accessing from wireless devices. Moreover, our tests show that the Windows-based implementation is a little bit more effcient when using the RFCOMM communication modality, but the Linuxbased implementation obtains the best performances when using the L2CAP communication modality.

We think that results presented in this paper show that Bluetooth is a good candidate to be the leading communication technology to provide access to the Web from mobile, low cost devices.

10. References

[1] "The Bluetooth Technology," March 2008. http://www.bluetooth.com

[2] "Bluetooth Wireless Technology," March 2008. http://www.ericson.com/technology/techarticles/Bluetooth.shtml

[3] C. Bisdikian, "An Overview of the Bluetooth Wireless Tecnology," *IEEE Communication Magazine*, Vol. 39, No. 12, 2001, pp. 86-94.

[4] O. P. Association, "Going Mobile: An International Study of Content Use and Advertising on the Mobile Web," March 8, 2007. http://www.onlinepublishers.org/media/176W opa going mobile report mar07.pdf

[5] J. Beutel and O. Kasten, "A Minimal Bluetooth-Based Computing and Communication Platform," Technical Report, Engineering and Networks Lab, Swiss Federal Institute of Technology, 2001.

[6] J. Misic, K. L. Chan and V. B. Misic, "Tcp Traffic in Bluetooth 1.2: Performance and Dimensioning of Flow-Control," *Proceedings of the* 2005 *IEEE Wireless Communications and Networking Conference*, New Orleans, 2005, pp. 1798-1804.

[7] F. Kargl, S. Ribhegge, S. Schlott and M. Weber, "Bluetooth-Based Ad-Hoc Networks for Voice Transmission," *Proceedings of the* 36*th Annual Hawaii International Conference on System Sciences*, Hawaii, 2003, pp. 314-322.

[8] S. Zeadally and A. Kumar, "Protocol Support for Audio

Streaming between Bluetooth Devices," *Proceedings of the* 2004 *IEEE Radio and Wireless Conference*, Atlanta, 2004, pp. 303-306.

[9] J. Cano, D. Ferrandez-Bell and P. Manzoni, "Evaluating Bluetooth Performace as the Support for Context-Aware Applications," *Proceedings of the* 12*th IEEE International Conference on Computer Communications and Networks*, Dallas, 2003, pp. 333-347.

[10] Y. Lim, J. Kim, S. L. Min and J. S. Ma, "Performance Evaluation of the Bluetoothbased Public Internet Access Point," *Proceedings of the* 15*th International Conference on Information Networking*, Beppu City, Oita, 2001, pp. 643-648.

[11] V. Auletta, C. Blundo, E. D. Cristofaro and G. Raimato, "A Lightweight Framework for Web Services Invocation over Bluetooth," *Proceedings of the* 2006 *IEEE International Conference on Web Services* (*ICWS'*06), Chicago, 2006, pp. 331-338.

[12] V. Auletta, C. Blundo, E. D. Cristofaro and G. Raimato, "Performance Evaluation of Web Services Invocation over Bluetooth," *Proceedings of the ACM International Workshop on Performance Monitoring*, *Measurement and Evaluation of Heterogeneous Wireless and Wired Networks*, Terromolinos, Spain, 2006, pp. 1-8.

[13] S. Berger, S. McFaddin, C. Narayanaswami and M. Raghunath, "Web Services on Mobile Devices-Implementation and Experience," *Proceedings of the* 5*th IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, California, 2003, pp. 100-109.

[14] "JSR 82: Java APIs for Bluetooth," March 2008. http://www.jcp.org/en/jsr/detail?id=82

[15] "Java 2 Platform, Micro Edition (J2ME)," March 2008. http://java.sun.com/j2me/

[16] "The Official Bluetooth Membership Site," March 2008. http://www.bluetooth.org

[17] "SOAP Version 1.2," March 2008. http://www.w3.org/TR/soap/

[18] "Broadcom Bluetooth Solutions," March 2008. http://www.broadcom.com/products/Bluetooth/Bluetooth-RF-Silicon-and-Software-Solutions

[19] "Windows Support for Bluetooth," March 2008. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/bluetooth/bluetooth/about bluetooth.asp

[20] "Bluez: Official Linux Bluetooth Protocol Stack," March 2008. http://www.bluez.org/

[21] "JSR 36, JSR 218: Connected Device Configuration (CDC)," March 2008. http://java.sun.com/products/cdc/

[22] "JSR 30, JSR 139: Connected Limited Device Configuration (CLDC)," March 2008. http://java.sun.com/products/cldc/

[23] "Mobile Information Device Profile (MIDP): JSR 37, JSR 118," March 2008. http://java.sun.com/products/midp/

[24] "Mobile Information Device Profile (Midp): JSR 37," March 2008. http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html

[25] "Mobile Information Device Profile 2.0 (Midp 2.0): Html, JSR118," March 2008. http://jcp.org/aboutJava/communityprocess/final/jsr118/index

[26] "Bluetooth Solutions by Atinav Avelink," March 2008. http://www.avelink.com/bluetooth/index.htm

[27] "Impronto Rococo Software," March 2008. http://www.rococosoft.com/

[28] "Avetana Jsr-82 Implementation," March 2008. http://www.avetanagmbh.de/avetanagmbh/produkte/jsr82.eng.xml

[29] "Blue Cove Jsr-82 Implementation," March 2008. http://code.google.com/p/bluecove/

[30] "Webservices-Soap," March 2008. http://ws.apache.org/soap/

[31] "kSOAP 2," March 2008. http://kobjects.org/

[32] "XML-RPC," March 2008. http://www.xmlrpc.com/

[33] "Java API for xML-Based RPC," March 2008. http://java.sun.com/webservices/jaxrpc/

[34] "The axis Client Api," March 2008. http://ws.apache.org/axis/java/apiDocs/org/apache/axis/client/package-summary.html

[35] "Web Service Axis," March 2008. http://ws.apache.org/axis/

[36] "kXML," March 2008. http://kxml.sourceforge.net/

[37] "The Call Class JavaDoc," March 2008. http://ws.apache.org/axis/java/apiDocs/index.html

[38] "Java Native Interface," March 2008. http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html