Support and Optimization of Java RMI over Wireless Environments

Cheng-Wei Chen  Chung-Kai Chen  Wang-Zhe Wu  Chien-Tan Ko  Pu-Chen Wei  Jenq-Kuen Lee
Department of Computer Science  National Tsing Hua University  Hsinchu 30043, Taiwan

ABSTRACT
Distributed object-oriented platforms are increasingly important over wireless environments to provide frameworks for collaborative computations and for managing a large pool of distributed resources. Due to limited bandwidths and heterogeneous architectures of wireless devices, efforts are needed to support object-oriented frameworks over heterogeneous wireless environments and optimize system performances. In our research work, we are working towards efficiently supporting object-oriented environments over heterogeneous wireless environments. In this paper, we report early experimental platforms to support Java RMI over Bluetooth, GPRS, and WLAN environments. The Bluetooth layer is done by by incorporating a set of protocol stack layers for Bluetooth, known as JavaBT developed by us and by supporting L2CAP layer with sockets to support RMI socket. In addition, our support for RMI over GPRS/WLAN is done by supporting RMI over mobile IPv4. The roaming among heterogeneous wireless environments are supported by Java proxy mechanisms and exception handle. Experimental results with the prototypes of Java RMI support over heterogeneous wireless environments are given. This platform can serve as a research platform for distributed object-oriented environments over wireless environments.

Keywords
Distributed Object-Oriented Computing; Wireless Computing; Java RMI; Bluetooth Architectures; Collaborative Computing

1. INTRODUCTION
Distributed object-oriented platforms over wireless environments have become important components for distributed computing and collaborative frameworks [2, 3]. Among the distributed object-oriented software, one of the key methods for the Java environments to perform distributed computing is via Java RMI. For example, the Jini network technology, based on Java RMI, provides an administration-less network environment. Devices can collaborate together via remote service, service discovery, etc. The Java RMI supports over wireless environments, such as Bluetooth, GPRS, and CDMA, can provide a software infrastructure for mobile and parallel environments. In the case of Java RMI, parallel computing can be done by asynchronous thread invocations of RMI to the remote site, possibly wireless remote facilities. With the increasing number of small devices in our computing environments, the software support, scalability issues, and optimization issues of such environments are important for investigations.

In an application scenario, the mobile devices, with limited computing power, can communicate with computation servers via RMI to invoke the services for computation-intensive tasks. As illustrated in Figure 1, the RMI services can be adaptive among heterogeneous network facilities, including wired and wireless networks. Java proxy supports and exception handling can also be employed to support Java RMI over heterogeneous networks. Optimization issues then can be the assignments among networks for the connectivity and the quality of service. In our research efforts to hope to support a framework over heterogeneous networks, we have developed several key technologies along the way. In this paper, we report the issues and our research results in one of the key components of our systems to efficiently support Java RMI over Bluetooth environments.

In this paper, we investigate the issues to support Java RMI over wireless environments. Our supports include several key technologies. First, a set of protocol stack layers written in Java for Bluetooth, called JavaBT is developed. In JavaBT, the HCI layer provides a uniform interface of accessing the Bluetooth hardware capabilities. The L2CAP provides connection-oriented and connectionless data services to upper layer protocols with protocol multiplexing capability, segmentation and reassembly operation, and group abstractions. These two layers of protocol driver can help programmers to write Bluetooth applications in the Java programming language. Next, we provide supports in L2CAP layer with socket for RMI socket. This enables the support of Java RMI over Blue-
tooth environments. In addition, we utilize the Java dynamic proxy mechanisms and exception handlings to support the roaming capability of Java RMI among heterogeneous wireless environments, including Bluetooth, GPRS, and WLAN.

We give experimental results to show that we have a prototype model of Java RMI support over Bluetooth, GPRS, and WLAN environments. In our experimental testbed, our implementation of Bluetooth protocol stack for Java platform, JavaBT, is tested with a pair of Ericsson Bluetooth Develop Kit (EBDK) connected to the personal computer. The EBDK is a Bluetooth hardware and software development board, and we connect the EBDK with the computer in the testbed. The current implementation of JavaBT is implemented with JDK 1.1.8 platform with JavaCOMM API 2.0 for Windows. We perform experiments of numerous benchmarks from RMI Benchmark Suite [17], DHPC Java Benchmarks [16], and the Java Grande Forum MPJ Benchmarks [9]. This work is also a part of research excellence projects of our university and of our research efforts to develop and investigate the research issues to advance the technologies for distributed component architectures [6, 7, 8, 12, 13, 19, 23].

The remainder of the paper is organized as follows. Section 2 gives the technical details for our support of Java RMI over L2CAP sockets of Bluetooth environments. Next, Section 3 describes using Java dynamic proxy to support heterogeneous wireless environments. The experimental results are then given in Section 4. Finally, Section 5 gives related work and then concludes this paper.

2. SUPPORTING JAVA RMI OVER BLUETOOTH ENVIRONMENTS

2.1 Support Java RMI over Bluetooth Layers

In this section, we present our methodologies in supporting Java RMI over Bluetooth environments. In our research framework, we have implemented two set of software infrastructures. First, we implement a set of layers for Bluetooth protocol stacks in Java. The software is called JavaBT. Second, we then implement the Java RMI layer on top of our software with Bluetooth protocol stacks. Traditionally, the Java RMI implementation in the Sun JDK is running over the TCP/IP network architecture. However, the current TCP/IP support in Bluetooth needs to go through several protocol layers (see Figure 2(a)). It will downgrade the communication performance and spend additional resources. One can run the TCP/IP over L2CAP layer to ease this problem (see Figure 2(b)).

With the above mechanism, we can make Java environment over the Bluetooth wireless network by running the Java RMI on the TCP/IP over Bluetooth wireless network. This will still need the TCP/IP support of the Bluetooth protocol stack (see Figure 3(a)). In our research software framework, we have the Java RMI over the L2CAP layer directly to reduce the protocol stack implementations overhead and to improve the performance (see Figure 3(b)). This reduces the number of layers in the protocol stack as well as the well-known overhead of TCP/IP layers.

We develop a custom Java RMI library that uses Bluetooth L2CAP layer instead of TCP/IP Socket. A custom RMI socket factory is used to transfer data at Bluetooth L2CAP layer. Under the Java RMI layer, we also develop the Java Bluetooth protocol stack including L2CAP layer, HCI layer, and a replaceable transport layer to communicate with the Bluetooth hardware through Java Native Interface (JNI).

2.2 The Bluetooth Socket for Java RMI

We build a family of classes to provide functionalities for the Java Socket over our Bluetooth driver. These classes are shown in Figure 4. They are BluetoothServerSocket, BluetoothSocket, BluetoothInputStream, BluetoothOutputStream, and BluetoothInetAddress. Our BluetoothServerSocket class extends from java.net.ServerSocket to provide the same interface as the original ServerSocket, but listens to the L2CAP layer. It also implements the L2CAPEventIndicationInterface to receive the L2CAP events. Detailed functions for BluetoothServerSocket class are given below.

- To listen the connection request

  BluetoothServerSocket overloads the constructor methods for initializing the L2CAPService and waiting for connections. After creating an instance of BluetoothServerSocket class, it will listen to a special Bluetooth L2CAP PSM and wait for the BluetoothSocket's connection requests.

- To accept the connection request

  BluetoothServerSocket overrides the accept method of java.net.ServerSocket to accept the connection from the BluetoothSocket. BluetoothServerSocket implements the
Figure 4: Bluetooth Socket Supports for Java RMI.

$L2CAPEventIndicationInterface$ to receive the $L2CAP$ Connection event. After receiving the Connection event, $BluetoothServerSocket$ will create a $BluetoothSocket$ with the specific $L2CAP$ CID from that event and return it.

- To retrieve the local Bluetooth address
  $BluetoothSocket$ overrides the $getInetAddress$ method of $java.net.ServerSocket$. In the original $ServerSocket$, this method will return the local address. In the $BluetoothServerSocket$, it will return a $BD_ADDR$ object that represents the Bluetooth address of local Bluetooth device by calling the $getLocalBDADDR$ method of $L2CAPService$.

Next, our $BluetoothSocket$ class extends from $java.net.Socket$ to provide the same interface as the original Bluetooth Socket but transmits data over the $L2CAP$ layer. Detailed functions for $BluetoothSocket$ class are given below.

- To connect to the remote host
  $BluetoothSocket$ overrides the constructor methods for initializing the $L2CAPService$ and connecting to the specific Bluetooth address and PSM.

- To read data from $BluetoothSocket$
  $BluetoothSocket$ overrides the $getInputStream$ method of $java.net.Socket$. This method will return an instance of $BluetoothInputStream$ that reads data from the $L2CAP$ channel.

- To write data to $BluetoothSocket$
  $BluetoothSocket$ overrides the $getOutputStream$ method of $java.net.Socket$. This method will return an instance of $BluetoothOutputStream$ that writes data to the $L2CAP$ channel.

- To retrieve the local and remote Bluetooth address
  $BluetoothSocket$ overrides the $getLocalAddress$ method and the $getInetAddress$ method of $java.net.Socket$. In the original socket, the $getLocalAddress$ method will return the local address. In the $BluetoothSocket$, it will return a $BD_ADDR$ object that represents the Bluetooth address of local Bluetooth device by calling the $getLocalBDADDR$ method of $L2CAPService$. The $getInetAddress$ method of $BluetoothSocket$ will return remote Bluetooth address by calling the $getRemoteBDADDR$ method of $L2CAPService$.

Next, we design a $BluetoothInputStream$ class as an input stream for reading data from a $L2CAPChannel$. It extends from $java.io.InputStream$. In addition, a $BluetoothOutputStream$ class is implemented as an output stream for writing data to a $L2CAPChannel$.

It extends from $java.io.OutputStream$. Next, we have the $BluetoothInetAddress$ class extends from $java.net.InetAddress$ to represent the Bluetooth address. The $BluetoothInetAddress$ wraps the $BD_ADDR$ and provides the same interface as $InetAddress$.

Finally, to run the $RMI$ over the Bluetooth, we create a class, $BluetoothRMIAddress$ that extends from $java.rmi.server.RMIClassUnicastFactory$, and implement $createServerSocket$ method and $createSocket$ method. The $createServerSocket$ method will create a $BluetoothServerSocket$ with a specific port and return it. The $createSocket$ method will create a $BluetoothSocket$ that connects to a specific Bluetooth address with a specific port and return it. To use our $BluetoothRMIAddress$ class in the $RMI$, we call the static method, $setSocketAddress$ of the $RMIAddress$ class and create an instance of the $BluetoothRMIAddress$ for the parameter.

3. USING JAVA PROXY TO SUPPORT HETEROGENEOUS WIRELESS ENVIRONMENTS

Despite its many attractive features, Java RMI lacks support for some essential services including system monitoring, profiling, message logging, secure communication and fault tolerance. In the section, we describe the efforts on fault tolerance of Java RMI to support roaming among heterogeneous wireless environments by using Java proxy mechanisms and exception handlings.

The dynamic proxy [20] mechanisms exploit the reflection capabilities of the Java platform to generate a wrapper at runtime for any object that implements a set of interfaces, it can subsequently be cast to any of the supported interfaces in a type-safe manner. The dynamic proxy’s functionality is made up by the $java.lang.reflect.Proxy$ class and the $java.lang.reflect.InvocationHandler$ interface.

- $java.lang.reflect.Proxy$
  The $ProxyMetaObject$ is generate at runtime by invoking the method $newProxyInstance$ of $java.lang.reflect.Proxy$. The method requires three parameters to construct the $MetaObject$: a class loader, a list of interfaces that the $MetaObject$ will implement, and an object that implements $InvocationHandler$. In the internal, it uses Reflection API to investigate the inheritance relationships of interfaces, dynamic generate the $MetaObject$’s classfile, and load it into the JVM via the class loader specified in parameter list.

- $java.lang.reflect.InvocationHandler$
  The $InvocationHandler$ interface declares only one method $invoke$. An object implements $InvocationHandler$ override the method to provide its own functionality. When a method invocation is fired to a $MetaObject$, it forward the invocation to the object mentioned in $newProxyInstance$. Note that the method is declared in one of the $BaseInterfaces$. The user-defined $invoke$ method then gains access in the execution path. In general, there is a $ConcreteComponent$ associated to the $InvocationHandler$ object, so that it can further forward the invocation to that $ConcreteComponent$. $InvocationHandler$ provides a generic mechanism to intercept the execution path at the points of method invocations.

In the Java RMI model, the static proxies or stubs act as a local
delegate for the remote server at client JVM. The stubs implement the same interfaces as remote servers, and contain code for network connection, so the client can invoke the services of remote servers via stubs. But the stubs are generated by the RMI compiler at compile-time, thus the behaviors are fixed and hard to customize.

In the other hand, we can generate dynamic proxies at run time, their behaviors are more extensible.

To support the roaming capability of Java RMI among heterogeneous wireless environments, including Bluetooth, GPRS, and WLAN, the dynamic proxy should implement the corresponding set of Remote interfaces the RMI server implemented. Thus the dynamic proxy can be invoked as remote services instead of the original stub. When the dynamic proxy is created, it finds a connected wireless network facility with highest bandwidth, and binds the RMI services to the corresponding wireless facility. The remote reference of the RMI server is then stored in the InvocationHandler. Any invocation to the dynamic proxy will be dispatched to the customized invoke() method in the InvocationHandler. The code in the invoke() method can check the parameters and returned results/exceptions, so the InvocationHandler can intercept the invocation and forward it to remote reference it contains as well as return the results from the remote server to the client.

In the roaming among heterogeneous wireless environments, the mobile device may be out of the coverage of the original wireless network facility, in such circumstance the forwarded remote invocation can raise a java.rmi.RemoteException, which can be caught in the invoke() method of InvocationHandler. Thus the InvocationHandler can find another wireless network facility within its coverage and with highest bandwidth to rebind the RMI services to it. After the successful rebinding, the remote reference of the RMI server is replaced with new one in the InvocationHandler and the subsequently remote invocations can be adapted to the new wireless facility.

4. EXPERIMENTAL RESULTS

We perform three experiments. In the first experiment, we evaluate the robustness of our implementation of RMI software over Bluetooth environments. We perform experiments of numerous benchmarks from RMI Benchmark Suite [17], DHPC Java Benchmarks [16], and the Java Grande Forum MPJ Benchmarks [9]. The last benchmarks are based on MPJ, which is an MPI-like message-passing interface for Java. We have implemented an MPJ interface that follows mpiJava 1.2 specification [5]. It uses RMI as the underlying communication channel to evaluate the performance of our RMI over Bluetooth implementation. Table 1 gives the applications that our RMI successfully run through it in our early test for robustness of our software.

In our testbed, the EBDKs (Ericsson Bluetooth Develop Kit) are connected to two computers by the COM port. We test the performance on one node and two nodes, respectively. The one node version is a sequential version and the two node version is a parallel version with RMI over Bluetooth. The computer that the RMI server runs on has one AMD 1200MHz CPU, 768MB of Memory and is running the Microsoft Windows 2000 Server. The RMI client runs on a IBM ThinkPAD notebook that has Intel Pentium III 700MHz CPU, 256MB of SDRAM and is running the Microsoft Windows XP Professional. Our JavaBT Bluetooth protocol driver and the test applications are running on Microsoft JavaVM that supports JDK 1.1.8 with additional RMI support and Sun Java Communications API v2.0 for Windows. Despite of the limitations of Bluetooth bandwidth, we still observe performance gains on Hamming, EP, Series, RayTracer, and SelSort. The performance is shown in Figure 5. Note that due to the client and server reside in different machines, the optimal speedup for this case is around 1.7 times. With the data rate in the peak to be 723k bps, it’s still much slower than fast Ethernet. Therefore, the type of numerical applications run well with RMI over Bluetooth can also be observed from our experimental results. Collaborative software which requires less communication bandwidth will likely run well in this type of environments. Note that our support with RMI over Bluetooth also gives more high-level control with Bluetooth environments. It gives better programming environments. System loads in the OS level can also be distributed to remote site servers with RMIs.

![Figure 5: The performance results of some benchmark applications.](image1)

![Figure 6: SelSort with local computation and a 2-node version based on RMI over Bluetooth communications.](image2)

![Figure 7: The bandwidth comparison for different layers.](image3)
Table 1: The detailed descriptions of benchmarks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Size</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>Fourier coefficient analysis</td>
<td>10,000</td>
<td>[9]</td>
</tr>
<tr>
<td>LUFact</td>
<td>LU Factorisation</td>
<td>500</td>
<td>[9]</td>
</tr>
<tr>
<td>SOR</td>
<td>Successive over-relaxation</td>
<td>1,000</td>
<td>[9]</td>
</tr>
<tr>
<td>Crypt</td>
<td>IDEA encryption</td>
<td>3,000,000</td>
<td>[9]</td>
</tr>
<tr>
<td>Sparse</td>
<td>Sparse Matrix multiplication</td>
<td>50,000</td>
<td>[9]</td>
</tr>
<tr>
<td>MollDyn</td>
<td>Molecular Dynamics simulation</td>
<td>2,048</td>
<td>[9]</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>Monte Carlo simulation</td>
<td>2,000</td>
<td>[9]</td>
</tr>
<tr>
<td>RayTracer</td>
<td>3D Ray Trace</td>
<td>150</td>
<td>[9]</td>
</tr>
<tr>
<td>SelSort</td>
<td>Selection Sort</td>
<td>524,288</td>
<td>[11]</td>
</tr>
<tr>
<td>Hamming</td>
<td>Given an array of primes, output in increasing order and without duplicates, all integers of the form $a^2 + b^2 + c^2 + \ldots \leq n$</td>
<td>5</td>
<td>[17]</td>
</tr>
</tbody>
</table>

We now give more insights for one of our experiment. This is with selection sort. In this experiment, we divide the selection sort task into two parts and send it to remote computer over Bluetooth RMI. Figure 6 illustrates the result. When the array size is increased more than 262,144, the distributed version over the Bluetooth RMI will have better performance that the local version.

In the second experiment, we evaluate the bandwidth and overheads with different layers of JavaRMI and with RMI. The comparison is with HCI, L2CAP, and Socket layers of our JavaRMI and our RMI version. In this experiment, we transmit different sizes of bytes array from one computer to another computer by our Bluetooth protocol driver. Figure 7 gives bandwidth for these four layers. Figure 8 illustrates the overhead for these four layers in our test. We sum up the traffic in our HCITransportBaseLayer that supports the COM port communication between the HCI layer and Bluetooth device. They are the comparisons for the additional data being sent at different layers in our current implementation. Most overheads are observed in the RMI version due to serializations of RMI.

Finally, we have the application scenario for RMI components adapting over heterogeneous wireless environments. In our research site, we have previously built up Java RMI over GPRS, WLAN, and Bluetooth environments under university research excellence project [22, 14]. The GPRS system for our experiment is equipped with HLR, BSS, SGSN and GGSN. In addition, a HA (Home Agent) providing mobile service for GPRS network is connected to GGSN via Gi interface. For Java applications written with RMI components over heterogeneous wireless environments, the dynamic proxy can help interact with applications for choosing proper RMI components. For example, an application can be initially running RMI over GPRS environments, but as the mobile device moved into the coverage of a WLAN, the RMI component over WLAN can be chosen. We demonstrate this idea with experiments. A mobile node is communicating with the PC, and running the MPI benchmarks through the RMI over GPRS. When the mobile node moves into the coverage of one WLAN, the diagnoser of dynamic proxy can decide to adopt the RMI component over WLAN because it has wider bandwidth. Figure 9 shows the potential benefits. The applications are from Java Grande (MPJ) benchmarks. The one note as GPRS is the application always running RMI component over GPRS. The one note WLAN is the one initially running with GPRS, but then the proxy finds out that the application is in the coverage of WLAN in the early stage of the computation. The RMI component over WLAN is quickly adopted. Significant performance are gained in this case.

5. RELATED WORK AND CONCLUSION
Efficient supports for Java remote method invocation have been important topics for investigations, as RMI provides a layer of abstractions for communications. Research results include an open RMI implementation which makes better use of the object-oriented features of Java [21], ARMI [18] and Manta [15] systems to reduce various drawbacks in RMI and to provide new RMI style systems with added functionality, a better way to implement RMI with exploiting Myrinet hardware features to reduce latencies for high performance [17] and the broad range of RMI applications [3]. In the
aspect of Bluetooth environments, there is a Java environment from Zucotto [24]. IBM provides a Bluetooth protocol driver for the Linux operating system, called BlueDrekar [10]. In addition, the OpenBT project [1] that comes from Axis is an open source project to build a Bluetooth protocol driver for the Linux operating system. In this work, we report a first-hand experience with Java implementation of Bluetooth protocol stacks and discuss the research issues to support Java RMI over wireless environments. This work is also a part of research excellence projects of our university and of our research efforts to develop and investigate the research issues to advance the technologies for distributed component architectures [6, 7, 8, 12, 13, 19, 23].

6. REFERENCES