Efficient Support of Java RMI over Heterogeneous Wireless Networks

Cheng-Wei Chen Chung-Kai Chen Jyh-Cheng Chen Chien-Tan Ko
Jenq-Kuen Lee Hong-Wei Lin Wang-Jer Wu

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. For beyond 3G environments, distributed object-oriented platforms can provide the framework and toolkits for application developments with heterogeneous wireless environments. In this paper, we present our support for Java RMI over Bluetooth, GPRS, and WLAN environments. This is an important middleware for component communications. Our support for the Bluetooth layer is done 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 Java RMI socket. Our support for RMI over GPRS/WLAN is done by supporting RMI over IP. We also show how to employ Java proxy and exception handling mechanisms to help perform roaming and resource schedulings among heterogeneous wireless environments. Java Grande benchmarks are used to demonstrate that our RMI implementations over GPRS, WLAN, and Bluetooth environments are effective in supporting parallel and distributed control of Java layers over heterogeneous wireless environments.

Index Terms

Distributed Object-Oriented Computing; Tools for Wireless Computing; Java RMI; Middleware for Communications; Pervasive Computing

Corresponding Author:

Jenq Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 300
Taiwan
Email: jklee@cs.nthu.edu.tw
Tel: +886-3-5715131 ext. 3519
Fax: +886-3-5723694

I. INTRODUCTION

Distributed object-oriented platforms over wireless environments have become important components for distributed computing and collaborative frameworks [1], [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. In addition, for beyond 3G environments, distributed object-oriented platforms can provide the framework and toolkits for application developments with heterogeneous wireless environments. 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.

Efficient supports for Java remote method invocation have been important topics for investigations, as RMI provides a layer of abstractions for communications. Research results are more focused on wired networks and include an open RMI implementation which makes better use of the object-oriented features of Java [18]. ARMI [16] and Manta [13] systems reduce various drawbacks in RMI and to provide new RMI style systems with added functionality. KaRMI [15] is a better way to implement RMI with exploiting Myrinet hardware features to reduce latencies for high performance. Also the broad range of RMI applications was done in [3]. Similar to Java RMI, .NET provides remoting as a way for application in different machines/domains to communicate with each other. All the method calls along with the parameters are passed to the remote object through the HTTP channel or TCP channel. Channels and transport layers can be abstracted for supporting heterogeneous network environments. The issues above are important to optimize Java RMI in terms of better object serializations and communication mechanisms. To support Java RMI over heterogeneous wireless environments, we need to support protocol layers of wireless devices and roaming mechanims to support the seamless roaming and resource schedulings among different wireless environments.

In this paper, we investigate the issues to seamlessly integrate object-oriented layers with heterogeneous wireless environments and schedule resources among heterogeneous wireless environments. Our solution and platform include several key technologies. These include protocol stack supports and the employment of Java proxy and exception handling mechanisms for scheduling resource and roaming. Our support for the Bluetooth layer is done 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 Java RMI socket. Our support for RMI over GPRS/WLAN is done by supporting RMI over IP. The Java proxy and exception handling mechanisms are employed to help perform roaming and resource schedulings among heterogeneous wireless environments. Our support for the roaming of Java RMI over heterogeneous wireless network is based on the concept of direct connection. It avoids the problems caused by forwarding. The difficulty of this strategy is how to handle the existent connection when the mobile node is actually moved to somewhere else. The high-level applications should not been disturbed or broken in this situation. After careful studies, we found this can be solved in Java RMI and Java proxy contexts by supporting of dynamic address, java proxy, and dynamic socket. We give algorithms to handle this handoff process. In addition, we give a transition diagram as our scheduling policy for the RMI layer over heterogeneous wireless environments for roaming cases.

Experimental results with the prototypes of Java RMI support over hetreogeneous wireless environments are given. 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. We perform experiments of numerous benchmarks from RMI Benchmark Suite [15], DHPC Java Benchmarks [14], and the Java Grande Forum MPJ Benchmarks [6]. Experiments are also carried out for supporting Java RMI over GPRS and WLAN. These experiments show that our RMI implementations over GPRS, WLAN, and Bluetooth environments are effective in supporting parallel and distributed control of Java layers over heterogeneous 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 [5], [9], [10], [19].

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 our GPRS supports. Next, Section 4 describes the framework and architecture to support the roaming and scheduling mechanisms for Java RMI over heterogeneous wireless network environments. The experimental results are then given in Section 5. Finally, Section 6 then concludes this paper.

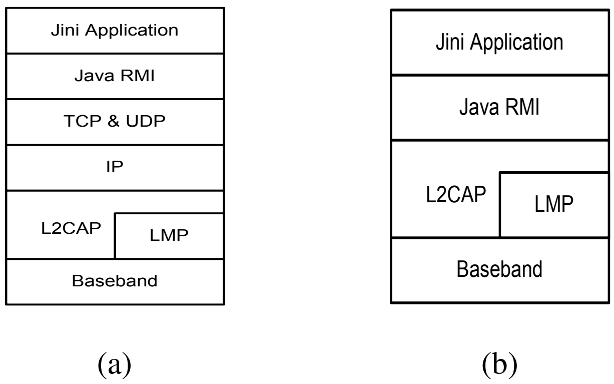


Fig. 1. (a) Run Java RMI over the TCP/IP support in the Bluetooth protocol stack. (b) Run Java RMI over the Bluetooth L2CAP layer directly.

II. Supporting Java RMI over Bluetooth Environments

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 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. It will downgrade the communication performance and spend additional resources. One can run the TCP/IP over L2CAP layer to ease this problem.

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 Fig. 1(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 Fig. 1(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).

We build a family of classes to provide functionalities for the Java Socket over our Bluetooth driver.

They are *BluetoothServerSocket*, *BluetoothSocket*, *BluetoothInputStream*, *BluetoothOututStream*, 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. The *BluetoothSocket* class extends from *java.net.Socket* to provide the same interface as the original Socket but transmits data over the L2CAP layer. 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, *BluetoothRMISocketFactory*, that extends from *java.rmi.server.RMISocketFactory*, 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 *BluetoothRMISocketFactory* class in the RMI, we call the static method, *setSocketFactory* of the *RMISocketFactory* class and create an instance of the *BluetoothRMISocketFactory* for the parameter.

III. Support Java RMI over GPRS Environments

In addition to provide voice service, cellular networks enable packet data service since 2.5G cellular networks have been developed. From GSM (Global System for Mobile Communication) to GPRS (General Packet Radio Service), the core network of the system is based on packet-switch rather than circuit-switch. Hence, GPRS can offer $20 \sim 170$ kbps data rate because packet transmission efficiently shares the bandwidth at the same time in core network. As the radio coverage can exceed $10\ Km$, users can not only conveniently launch a phone call but also surf the Internet anywhere. Comparing with GSM, GPRS mainly extends the function of packet data access that is achieved by adding two nodes, SGSN (Serving GPRS Support Node) and GGSN (Gateway GPRS Support Node). In the core network, SGSN and GGSN play the role of packet transmission. SGSN connects to BSS (Base Station System) and is responsible for mobility management. On the other words, SGSN can track the routing area to transmit packet to the proper destination for MS (Mobile Station). GGSN acts like a router that connects to external PDN

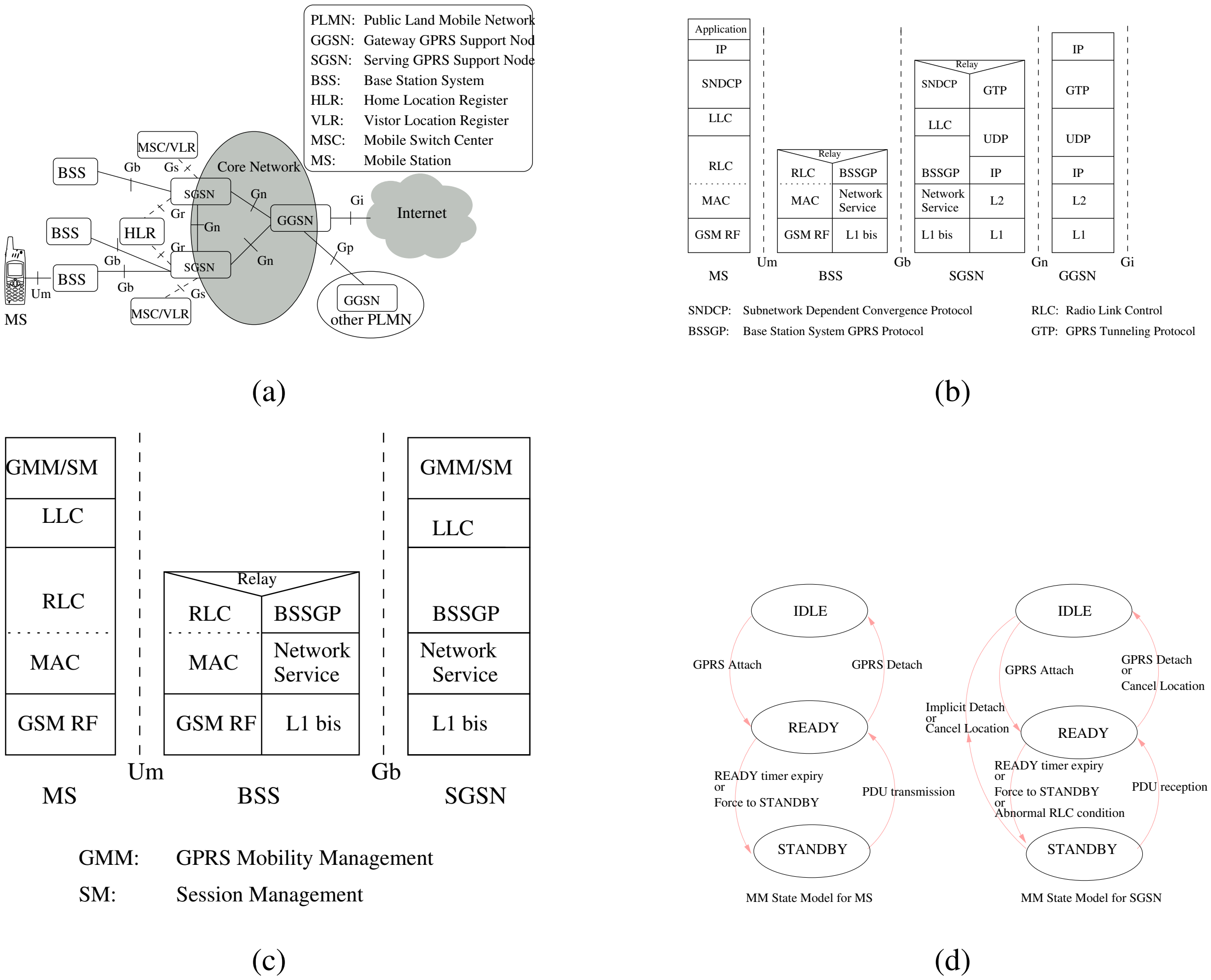


Fig. 2. (a) Architecture of GPRS. (b) User Plane for GPRS. (c) Control Plane between MS and SGSN (d) Mobility Management Transition State

(Packet Data Network). All packets from external PDN can be routed by GGSN to the proper SGSN and finally reach to the destined MS. A GPRS network architecture is shown as Fig. 2(a).

From the aspect of protocol stacks, it is divided into user plane and control plane. The user plane aims to transfer user data and provide flow control, error detection, error correction and error recovery is display as Fig. 2(b). Application data originated from MS will pass through IP layer, then SNDCP layer which is the interface between network-level and underlying network. And a reliable ciphered logical link is provided by LLC. After processing via layer 2 – RLC and MAC, data is transmitted through GSM Radio Frequency finally. BSS can relay LLC PDUs between Um and Gb interface. NS makes the connection between BSS and SGSN aims to transport BSSGP PDUs. Routing and QoS-related information between BSS and SGSN is the major function for BSSGP. SGSN is a network level node differs from BSS just provides relay function. Hence, SNDCP is equipped to map network-level characteristics onto the characteristics of the underlying network. Since core network is based on IP, GSNs follow IP and UDP to transfer user data. Moreover, GTP is leveraged to transmit user data between SGSN and GGSN through Gn interface. On the other hand, the control plane controls packet domain network access connection (Attach, Detach), attributes of an established network access connection (PDP context Activation), routing path (support user mobility) and assignment of network resources. Fig. 2(c) illustrates the control plane between MS and SGSN, others such as SGSN-HLR or SGSN-MSC/VLR are not discussed in this article. In MS and SGSN, GMM/SM offers the functionality of mobility management such as GPRS attach and detach, location and routing area update and PDP context activation.

In terms of mobility management, SGSN maintains and tracks the information of MS such as mobility management state, cell identity and routing area. This kind of information is called mobility management context (MM Context) which is kept by MS, SGSN and HLR for each MS. Mobility management state is divided into IDLE state, STANDBY state and READY state as the state transition is shown as Fig. 2(d). In IDLE state, MS is not attached to GPRS mobility management and it is not possible for paging of MS. Data transmission is not allowed unless MS perform the attach procedure. Soon after MS attaches to GPRS, it enters into READY state. PDP (packet data protocol) PDUs can be sent or received when MS is in READY state. A READY timer is enabled if there is no data transmission. When the timer expires, the MM context turns to STANDBY state in which data transmission and reception are not available. Besides mobility management state, the information of routing and location in SGSN may be updated upon the movement of MS or timer-triggered. There are three types of mobility management timer functions including READY Timer, Periodic RA Update Timer and Mobile Reachable Timer function. MS and SGSN maintain the READY timer, and the timer is enabled when there is no LLC PDU transmission in READY state. Once the timer expires, the MM context turns to STANDBY state. Routing area update is driven when MS detects entering a new RA or the periodic RA update timer expires. According to the RA update request and related parameters of MS, the network could perform the different types of routing area update such as intra SGSN RA update, inter SGSN RA update and inter SGSN combined RA/LA update. In spite of RA update request of MS, SGSN can initiate the request accordingly if no RA update from MS after expiring of Mobile Reachable Timer.

Comparing with GPRS mobility management, Mobile IP defined by IETF allows terminal mobility that can be widely applied in IP-based network. For cellular networks, Mobile IP can be deployed as a solution of roaming between different PLMNs or PLMNs and other PDNs (Packet Data Network). In Mobile IP architecture, the network that provides permanent reachable IP address is called home network, in which a Home Agent (HA) plays the roles of tracking the location and tunnel packets for MN. Once MN changes to a new subnetwork with different IP routing configuration, it must send a registration message to inform HA the current attachment network. Suppose that MN moves to other network called foreign network, HA will intercept all the datagram destined to MN and forward to its new location after encapsulating the datagram. The packet interception is carried out by gratuitous ARP or proxy ARP for HA. When MN moves to foreign network, it can configure a new IP address or keep the original IP address. If here is the later case, FA-CoA (Foreign Agent Care-of-Address) is the tunnel endpoint for HA. Thus, FA must decapsulate and send the datagram to MN. By registration and packet tunnel, Mobile IP is a simple mechanism offering IP mobility. As our purpose is to support Java RMI layer over GPRS, WLAN, and Bluetooth. We have designed our system to have RMI over IP and GPRS. The mobile management is done in the RMI and socket layer. Alternatively, one can build a Mobile IP over GPRS, WLAN, and bluetooth, and then have RMI over Mobile IP. Our method provides further opportunities to interact object-oriented application layer with wireless environments for resource managements.

IV. Roaming of Java RMI over Heterogeneous Wireless Networks

We now have described supporting RMI over several wireless networks including the Bluetooth, WLAN, and GPRS. We now present the techniques for roaming heterogeneous wireless communication under object-oriented environments. The techniques include the Java proxy mechanism for supporting dynamic sockets among heterogeneous wireless environments, a RMI roaming architecture that uses similar concepts as the Mobile IP technology but with different connecting strategy and addressing RMI layers, and a heuristic approach to schedule resources among heterogeneous networks.

A. Roaming Mechanism

The roaming issue for the wireless environment has also been addressed in the Mobile IP technology. It's done by additional software supports known as home agents, foreign agents, and tunneling protocols to carry out the underlying roaming mechanism. The concept of Mobile IP is based on packet forwarding. A mobile node MN first registers with a home agent residing in its network to get a home IP. Other nodes in the internet, called CN, use this IP to address it. When the MN moves to another wireless network, it will search for advertisement messages from available foreign agents around. Through registration steps, the mobile node gets a foreign IP (FA-CoA) and the traffic regarding to the MN is tunneled from the home agent to the foreign agent.

Our support for the roaming of Java RMI over heterogeneous wireless network is based on the concept of Mobile IP implementation, but manipulations in the RMI layer. In addition, a direct connection method is employed by incorporating Java RMI and proxy mechanisms. As we know all the communications toward the MN are generally passed through the home agent and foreign agent when a mobile node migrates. This is the case even if the MN is in fact moving to the network of the CN. This is a well-known problem in Mobile IP called the triangular routing problem. The home agent and its network also have to stay live and risk as a performance bottlneck. Several solutions are proposed to overcome these issues. For example, A mobile routing table (MRT) is designed in each edge router to provide a seamless handoff approach [20]. Alternatively, the IETF group also proposes certain additional protocols trying to have an optimal routing path. Next, a possible approach is using direct connection. It avoids the problems caused by forwarding. For example, when the physical position of the MN is changed, it might mean that it wants to use or serve those devices in that location. Thus the direct-connect approach is a good solution.

There are two issues on implementing a direct-connect roaming. One is to have the CNs know the new location of the MN to get connected to it. Another is to migrate the existent connections when the MN moves. These two parts should be done without the involvement of high-level applications. We develop two techniques below, *dynamic address* and *dynamic socket*, to handle this issue.

1) **Dynamic Address** In order to have the CNs get connected to the migrated mobile node directly without affecting programs, we use an object called `DynamicAddress`, extended from a standard IP address object called `InetAddress`. The CNs initialize `DynamicAddress` objects with the home address of the MN, and use it to construct a connection with the MN. When the underlying sockets access the `DynamicAddress` object, the `DynamicAddress` object first fetches the correct foreign address of the MN from a registry. We need to maintain such a registry somewhere in the internet to always keep the latest location of the MN. This can be done by forcing the MN to register its new foreign address with the registry everytime it changes its IP. The overheads caused

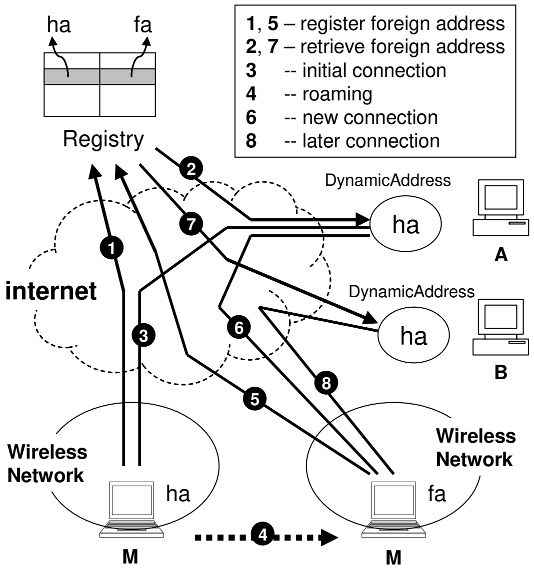


Fig. 3. The underlying architecture of RMI to provide roaming mechanism.

by the need of querys on the registry when constructing a connection is actually small compared to the benefits gained from the direct-connect communications. This is because the RMI tends to reuse the connections instead of building a new one for each operation.

2) **Dynamic Socket** The concept used to direct the following connection to the new place after roaming is straight forward, but to migrate existent connections to the direct-connecting manner, however, is not trivial. The Java proxy method can be employed to adapt socket objects of different wireless networks. It adapts connection-related objects to a new set which forms the new connections. We provide each type of wireless network a specialized socket support which consists of `Socket` objects, `ServerSocket` objects, and the `InputStream`, and `OutputStream` objects. When we are going to migrate connections into another wireless network, we simply build necessary new connections and issue a component adaptation and have the communication forwarded into the new one. The applications using RMI will incorporate the corresponding socket factories of these proxyed sockets, called *dynamic sockets*, and need not be aware of the change of underlying networks.

We integrate the above two concepts into the Algorithm 1 and Algorithm 2 to give steps to construct the connection and perform roaming, respectively. Algorithm 1 is our proposed new procedure for a Java RMI to make an invocation by employing dynamic sockets (proxyed sockets). Algorithm 2 gives the procedures for connections in the roaming situtation for both RMI client and server, respectively. In the

Algorithm 1: **The construction of RMI connections**
Begin
Step 1. The stub is called by a thread for a new RMI invocation.
Step 2. The stub picks up a connected socket, say sk. If sk exists, go to **Step 8**.
Step 3. The stub calls the `DynamicSocketFactory` instance, say dsf.
Step 4. dsf makes a request to the online Registry, fetching the foreign address f_addr of the remote site.
Step 5. dsf creates a `DynamicSocket` object connecting to f_addr and assigns it to sk.
Setp 6. Send an identity the the remote side to indicate that this is a new connection.
Step 7. If an exception happens, go to **Step 4**.
Step 8. The stub uses sk to communicate with the remote object.
End

Fig. 4. The algorithm for constructing the RMI connection using dynamic IP.

case that the wireless network in use is out of range, the communication object in the client catches an exception and then issues Algorithm 2 to try to rebuild the connection and recover the failed RMI operation.

We now take a close look at Fig. 3 again to explain these two algorithms. As we can see in Fig. 3, a mobile node M, hosting a RMI service, first sets its home address `ha` to the current IP, and registers it to the registry with the foreign address `fa` equaling to `ha`. Then a node A in the internet tries to make an connection to the mobile node M to access its service. According to Algorithm 1, the node A invokes remote methods on the stub of the mobile service. The stub then finds no connection is available now. It soon tries to use the `DynamicSocketFactory` instance to get a connected socket to M. The constructor of the socket object is given a `DynamicAddress` object. The dynamic address object queries the registry and fetches the registered foreign address of M when it is used. If the socket fails on constructing this connection, it will repeat this procedure until succeed or try within a given time. After the connection is up, the RMI operation can proceed. Assume after a period of time the mobile node M roams to another network, it applies Algorithm 2 to migrate its RMI service to the new place. First, it detects and acquires a new IP in that network. Then it immediately reports this new IP as its new foreign address to the registry. Now there is an existing connection lost. This needs to be rebuilt. The node A repeats the above actions to try to make a connection to M. Now it will get the new address of M. As the RMI server, the mobile node M uses the `DynamicServerSocketFactory` instances to create a new server socket and restarts its service. When the connection between M and A is up eventually, they can exchange certain identities

Algorithm 2: **The roaming algorithm**
Suppose the `DynamicSocket` object is called ds and the `DynamicServerSocket` instance is called dss.
Begin
Step 1. Detect available networks. Perform necessary setup/configuration/registration to get an available IP.
Step 2. Determine if I am a RMI client or RMI server. If I am a client, go to **Step 8**.
Step 3. Call the `DynamicServerSocketFactory` instance to create a `ServerSocket` object, say ssk.
Step 4. Switch dss to use ssk, set the new IP as the foreign address, and register it to the online Registry.
Step 5. ssk listens to the incoming connections. Suppose the accepted socket is sk.
Step 6. Receive an identity to see if sk is a recovered connection. If so, switch ds to use sk.
Step 7. Go to **Step 12**
Step 8. Make a request to the online Registry and fetch the foreign address f_addr of the remote site.
Step 9. Create a `Socket` object, say sk. Connect sk to f_addr. Switch ds to use sk.
Step 10. Send an identity to indicate that this is a recovered connection.
Step 11. If an exception happens, go to **Step 8**.
Step 12. Recover the communication.
End

Fig. 5. The roaming algorithm for moving the connection to the new network.

so the server knows it is a resumed connection and can adapt the network objects of this connection to replace the original ones. All these changes are hidden in the Java `InvocationHandler` of the network-related objects such as sockets and streams. To recover the failed RMI operations, we force the `InvocationHandler` to track the packets they sent and received. The last uncompleted operations will be redone through the new set of network objects. After the roaming, node B will use Algorithm 1 to directly connect M. The original wireless network is no longer involved in the RMI operation on M.

B. Roaming and Scheduling Optimization

In the design architecture of Fig. 3, once the wireless network currently in use is not reachable (possibly due to out of range), the direct-connection object catches an exception and issues Algorithm 2 to try to rebuild the connection and recover the failed RMI operation. More aggressive roaming decisions and schedulings can be made for optimization, as there might be several overlapped wireless networks at the same time and same places with technologies advance. Table I compares different characteristics of Bluetooth, WLAN and GPRS, and each one of them has a certain superiority. For example, GPRS owns

TABLE I

A COMPARISON OF BLUETOOTH, WLAN AND GPRS.

	Bandwidth	Power	Range	Cost	Frequency
Bluetooth	1Mbps	1~10mw	10~100m	None	2.4GHz
WLAN (802.11b)	11Mbps	50~70 mw	100~200m	Low	2.4GHz
GPRS	115~117Kbps	200~800 mw	1x Km	High	900/1800MHz

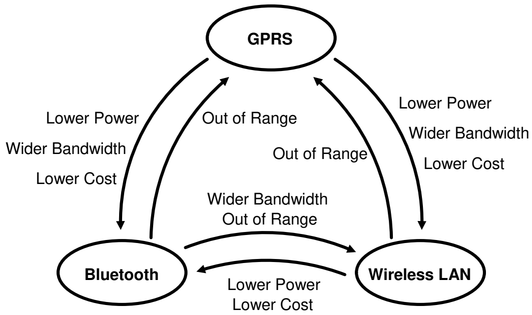


Fig. 6. The state transition graph representing our scheduling policy.

the largest cover range, but is with high power consumption, low bandwidth, and high connection cost. Whereas the WLAN can provide high bandwidth in a small range with low connection costs. As to the Bluetooth, it has lowest power consumption on the connection and higher bandwidth than GPRS. It is suitable to use Bluetooth on small, fast communication on personal small devices. A node having several wireless network available may purposely roam to another network to acquire benefits in bandwidth, power consumption, or connection costs.

Fig. 6 gives a transition diagram to explain our scheduling policy for the RMI layer over heterogeneous wireless environments for roaming cases. Initially, the RMI implementation stays in one state. Later, when other wireless networks detected due to the movement of the mobile node, the transition conditions are evaluated to decide the direction for roaming to another wireless network. Note that other scheduling policies or quality of service controls can also be done in this layer. Our design and architecture presents a layer for scheduling controls between object-oriented applications and heterogeneous wireless network environments. This is certainly important for additional and future research explorations by exploiting application charateristics in RMI accesses on heterogeneous wireless environments. Related research results such as QoS on the wireless networks [2], [7], [12] can also be incorporated.

TABLE II

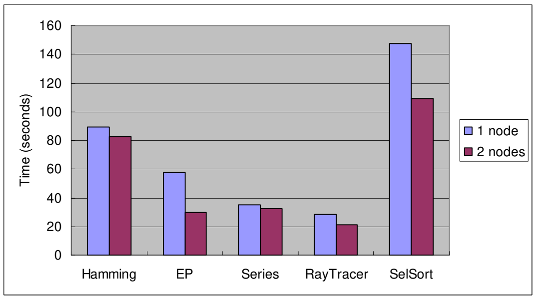
THE DETAILED DESCRIPTIONS OF BENCHMARKS.

Name	Description	Data Size	From
EP	NAS EP generating Gaussian random numbers	16,777,216	[14]
Series	Fourier coefficient analysis	10,000	[6]
LUFact	LU Factorisation	500	[6]
SOR	Successive over-relaxation	1,000	[6]
Crypt	IDEA encryption	3,000,000	[6]
Sparse	Sparse Matrix multiplication	50,000	[6]
MolDyn	Molecular Dynamics simulation	2,048	[6]
MonteCarlo	Monte Carlo simulation	2,000	[6]
RayTracer	3D Ray Tracer	150	[6]
SelSort	Selection Sort	524,288	[8]
Hamming	Given an array of primes, output in increasing order and without duplicates, all integers of the form $a^i * b^j * c^k \ldots \leq n$	5	[15]

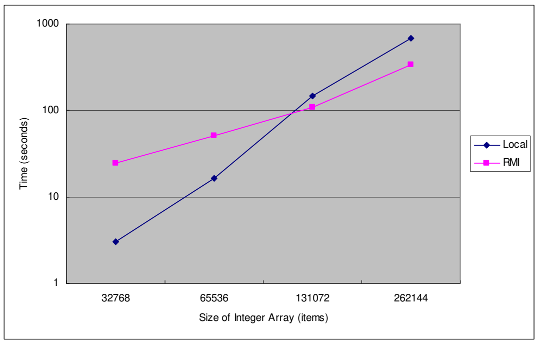
V. Experimental Results

In our experiments, we perform Java RMI over misc. wireless environments to show that we have an effective environment to perform Java RMI over wireless environments. In the first set of experiments, we evaluate the robustness of our implementation of RMI software over Bluetooth environments. We perform experiments of numerous benchmarks from RMI Benchmark Suite [15], DHPC Java Benchmarks [14], and the Java Grande Forum MPJ Benchmarks [6]. 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 [4]. It uses RMI as the underlying communication channel to evaluate the performance of our RMI over Bluetooth implementation. Table II gives the applications that our RMI successfully run through it in our early test for robustness of our software.

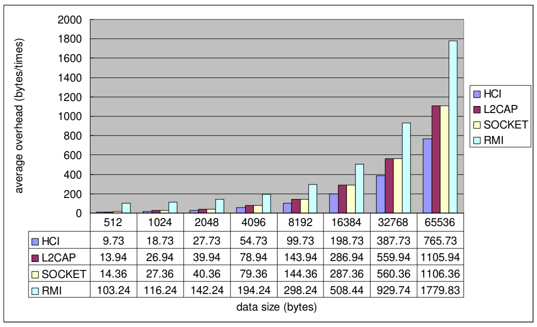
In our testbed, the EBDKs (Ericsson Bluetooth Develop Kit) are connected to two computers. We test the performance on one node and two nodes, respectively. The one node version is a sequential version and the two node version is a parallel version with RMI over Bluetooth. The computer that the RMI server 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



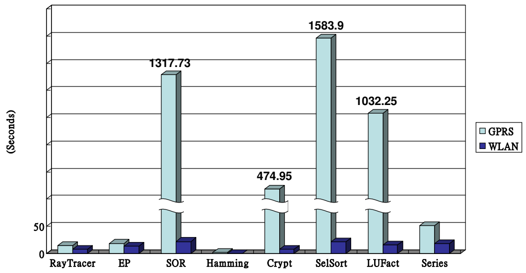
(a)



(b)



(c)



(d)

Fig. 7. (a) The performance results of some benchmark applications. (b) SelSort with local computation and a 2-node version based on RMI over Bluetooth communications. (c) The overhead comparison for different layers. Additional data sent at different layers are observed. (d) Adapting RMI component over wireless environments.

of Bluetooth bandwidth, we still observe performance gains on Hamming, EP, Series, RayTracer, and SelSort. The performance is shown in Fig. 7(a). 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.

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. Fig. 7(b) 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 overheads with different layers of *JavaBT* and with RMI. The comparison is with HCI, L2CAP, and Socket layers of our *JavaBT* 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. Fig. 7(c) 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 and roaming over GPRS and WLAN wireless environments. The GPRS system for our experiment is equipped with HLR, BSS, SGSN and GGSN. For Java applications written with RMI components over heterogeneous wireless environments, we employ the dynamic proxy and socket described in Section 4 to help interact with applications for roaming among different wireless environments. In our experiment, an application is 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 MPJ benchmarks through the RMI over GPRS. When the mobile node moves into the coverage of one WLAN, the RMI component over WLAN is adopted. All of the experiments are done that the second RMI occurrences of each application is set as the time that the device moves into range of WLAN. Fig. 7(d) shows the results. 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 in the second occurrences of RMI in each application, we arrange the WLAN into the range and the proxy finds out that the application is in the coverage of WLAN and schedules the socket into WLAN. Significant performance are gained in this case.

Focus is now directed to the handoff overhead for RMI over different wireless networks. We give the performance aspect from GPRS roaming into WLAN. The handoff overheads includes the time to configure and adopt the new IP from WLAN and the time to reconstruct connections and adopting new components. In our mobile node, we use two NICs to access GPRS and WLAN separately. The setting and switching time is about 52 ms in average. The connection reconstruction is done within WLAN taking additional dozens of milliseconds communicating with the registry and the remote side per connection (This is done in parallel). As to the component adaptation time, it is smaller than one millisecond roughly. To add everything together, the handoff time is 65 ms. Compared to the alternative solution that we can have Mobile IP and have RMI over Mobile IP, it takes of 57.4 ms around to send registration request, 18.1 ms for the mobile node to receive acknowledge from the HA. and 4.5 ms for it to receive packets from the CNs again. The average handoff latency for mobile IP thus is about 77.3 ms [11]. We can see the handoff overheads of our approach and the Mobile IP is about the same with ours have small advantages. Note that Mobile IPv4 implementation follows the standard so that it gives slightly overheads. Our method is with direct connection which can further reduce bottlenecks. In addition, we present a layer for scheduling and roaming controls between object-oriented applications and heterogeneous wireless network environments.

VI. Conclusion

Efficient supports for Java remote method invocation have been important topics for investigations, as RMI provides a layer of abstractions for communications. In this work, we reported a first-hand experience with design and implementation of Java RMI over heterogeneous wireless environments with the roaming supports. Our design and architecture presented a layer for scheduling controls between object-oriented applications and heterogeneous wireless network environments. This is certainly important for additional and future research explorations by exploiting application charateristics in RMI accesses on heterogeneous wireless environments. Experiments show that that our supports are effective in supporting parallel and distributed control of Java layers over heterogeneous wireless environments.

References

[1] Milind Bhandarkar and L. V. Kale. An Interface Model for Parallel Components. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 2001.

[2] Vaduvur Bharghavan, Songwu Lu and Thyagarajan Nandagopal, Fair Queueing in Wireless Networks: Issues and Approaches. in *IEEE Personal Communications Magazine*, February 1999.

[3] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, September–November 1998.

[4] Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko, and Sang Lim. *mpiJava 1.2: API Specification*. http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html, October 1999.

[5] C. W. Chen, C. K. Chen, and J. K. Lee. Building Ontology for Composition and Optimization of Parallel JavaBean Programs. In *Proceedings of IEEE I-SPAN*, May 2002.

[6] Edinburgh Parallel Computing Centre. The Java Grande Forum MPJ Benchmarks. http://www.epcc.ed.ac.uk/javagrande/mpj.html.

[7] P.J.M. Havinga, G.J.M Smit, QoS scheduling for energy-efficient wireless communication. International Conference on Information Technology: Coding and Computing (ITCC2001), special session on recent advances in wireless networking and communications (invited), Las Vegas, USA, April 2-4, 2001.

[8] U. Kremer, J. Hicks, and J. Rehg. A Compilation Framework for Power and Energy Management on Mobile Computers. In *Proceedings of the 14th International Workshop on Parallel Computing (LCPC)*, August 2001.

[9] Jenq Kuen Lee and Dennis Gannon. Object-Oriented Parallel Programming: Experiments and Results. In *Proceedings of Supercomputing'91*, New Mexico, November 1991.

[10] Jenq Kuen Lee, Ing-Kuen Tsaur, and San-Yih Hwang. Parallel Array Object I/O Support on Distributed Environments. *Journal of Parallel and Distributed Computing*, 40, 227-241, 1997.

[11] Hong-Wei Lin, Jyh-Cheng Chen, Ming-Chia Jiang, Ching-Yang Huang. Integration of GPRS and Wireless LANs with Multimedia Applications. *Proceedings of IEEE Pacific Rim Conference on Multimedia*, 704–711, 2002.

[12] Haiyun Luo, Songwu Lu, Vaduvur Bharghavan, Jerry Cheng and Gary Zhong, A Packet Scheduling Approach to QoS Support in Multihop Wireless Networks. in *ACM Journal of Mobile Networks and Applications (MONET), Special Issue on QoS in Heterogeneous Wireless Networks*, 2002.

[13] J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java remote method invocation. In *Proceedings of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[14] J. A. Mathew, P. D. Coddington, and K. A. Hawick. DHPC Java Benchmarks. http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks.

[15] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A More Efficient RMI for Java. In *Proceedings of ACM 1999 Java Grande Conference*, pp. 152–157, June 1999.

[16] R. Raje, J. Williams, and M. Boyles. An asynchronous remote method invocation mechanism for Java, *Concurrency: Practice and Experience*, Vol. 9(11), 1207-1211, 1997.

[17] Sun Microsystems, Inc. *Dynamic Proxy Class API*, Java 2 SDK Std. Documentation, http://java.sun.com/j2se/1.4/docs/guide/reflection/proxy.html.

[18] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective Remote Method Invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, September–November 1998.

[19] Pu-Chen Wei, Chung-Hsin Chen, Cheng-Wei Chen, Jenq Kuen Lee. Support and Optimization of Java RMI over Bluetooth Environments. *ACM Java Grande Conference*, Nov. 2002.

[20] I-Wei Wu, Wen-Shiung Chen, Ho-En Liao and Fongray Frank Young, A Seamless Handoff Approach of Mobile IP Protocol for Mobile Wireless Data Networks. In *IEEE Transaction on Consumer Electronics*, Volume 48, Issue 2, pages 335-344, May 2002