Case Study: Mobile Java RMI Support over Heterogeneous Wireless Networks

Chung-Kai Chen, Cheng-Wei Chen, Chien-Tan Ko,

Jenq-Kuen Lee, Jyh-Cheng Chen

Department of Computer Science

National Tsing Hua University

Hsinchu 30013, Taiwan

The contact author is Prof. Jenq-Kuen Lee. His e-mail address is jklee@cs.nthu.edu.tw. Phone number is +886-3-571-5131 ext. 33519. Fax number is +886-3-572-3694. A preliminary version [4] of this paper was published in Proceedings of IEEE International Conference on Communications (ICC), pp. 1391-1395, Paris, France, June 2004.

Abstract

Distributed object-oriented platforms are increasingly important in wireless environments to provide frameworks for collaborative computation and for managing a large pool of distributed resources. One of the important layer for implementing distributed computing in such environments is via remoting mechanisms. For example, Java uses Remote Method Invocation (RMI) for handling distributed controls. In this paper, we investigate the support for this important layer on wireless environments and address the issues to support Java RMI over heterogeneous wireless environments. Our work present a case study for supporting Java RMI in Bluetooth, GPRS, and WLAN environments, which represents an important middleware for component communications. The Bluetooth layer is supported by incorporating a set of protocol stack layers for Bluetooth, known as *JavaBT* that has been developed by us, and by supporting an L2CAP layer with sockets to support Java RMI sockets. RMI over GPRS/WLAN is achieved by RMI implementation over IP layer. Our support for the roaming of Java RMI over heterogeneous wireless networks is based on the concept of direct connection, which avoids the problems caused by forwarding. The difficulty of this strategy is how to handle the existing connection when the mobile node moves to another location so as to avoid interruption of the high-level applications. We solve this problem in Java RMI by the support of dynamic addresses and dynamic sockets. We also propose algorithms to handle the handoff process. In addition, methods for connect-loss detection and data-integrity maintenance in dealing with roaming scenarios are also presented. Java Grande benchmarks are used to demonstrate that our RMI implementations over GPRS, WLAN, and Bluetooth networks are effective in supporting parallel and distributed control of Java layers in heterogeneous wireless environments.

Index Terms

Java RMI support, heterogeneous wireless computing, component software in overlaying networks, service middleware for wireless environments, parallel and distributed computing.

I. Introduction

Distributed object-oriented platforms in wireless environments have become important components for distributed computing and collaborative frameworks [1], [2]. One of the key methods for implementing distributed computing in Java environments is via Java Remote Method Invocation (RMI). For example, the Jini network technology, based on Java RMI, provides an administration-free network environment. Devices can collaborate via methods of remote service and service discovery. The Java RMI supports in wireless environments, such as Bluetooth, General Packet Radio Service (GPRS), and wireless LAN (WLAN), can provide a software infrastructure for mobile and parallel environments. Parallel computing can be implemented in Java RMI using asynchronous thread invocations of RMI to the remote site, possibly via wireless remote facilities. For beyond-3G environments, distributed object-oriented platforms can provide the framework and toolkits for application development in heterogeneous wireless environments. In addition, the support of remote method invocation also comes to a focus, as this is the layer which looks promising as paradigms to support ubiquitous component communications over heterogeneous network environments. This is also the layer where the well-known software layers such as .NET Remoting, Java RMI, and Common Component Architecture (CCA) remoting are located. This software layer is also close to Open Grid Services Architecture (OGSA) layer where service computing layer is located. Furthermore, Java RMI is also used as a communication primitive for supporting overlaying networks such as peer to peer environments. Therefore, the efficient support of this layer is important as the basic middleware support for distributed computing environments.

Efficient support for Java RMI has been an important topic for investigations. Research on Java RMI support has focused on wired networks and has produced an open RMI implementation that makes better use of the object-oriented features of Java [19]. ARMI [16] and Manta [12] systems reduce various drawbacks in RMI and have provided RMI systems with added functionality. KaRMI [14] exploits Myrinet hardware features to reduce latencies for high performance. A broad range of RMI applications has also been developed [2]. Similar to Java RMI, .NET Remoting provides a way for applications in different machines/domains to communicate with each other. All the method calls along with the parameters are passed to the remote object through an HTTP or TCP channel. Channels and transport layers can be abstracted in order to support heterogeneous network environments. Related research activities were done in [6], where we looked at the stateful properties of components and provided the switching supports for .NET Remoting over network processors for both stateful and stateless component communications. To support Java RMI in heterogeneous wireless environments, we need to support the protocol layers of wireless devices and a roaming mechanism to support roaming and resource scheduling over different wireless networks.

In this paper, we investigate the issues involved in seamlessly integrating object-oriented layers in heterogeneous wireless environments. We investigate the issues to efficiently support Java RMI over Bluetooth, GPRS, and WLAN environments. Our solution can support the mobile

Java RMI over heterogeneous wireless networks without Mobile IP support. This is a likely scenario in a peer to peer environment, as some of the embedded devices might communicate with other devices via wireless protocols, but without Mobile IP support. Although Mobile IP has been proposed for more than a decade, it is still not widely used in the real world. One of the reasons is that it requires the implementation of Mobile IP specific protocol in the networks and mobile stations. Mobile IP might be a reasonable choice if all networks and mobile devices support IP layer. However, for some networks, such as Bluetooth, presenting IP layer would bring extra efforts and costs. This is especially important for hand-held devices with limited hardware resources. Our proposed idea can be used in both IP-based and non-IP-based networks. Another fact about Mobile IP is that Mobile IP works with pre-established network infrastructures. Mobile IP uses home agents (HA) and foreign agents (FA) to forward packets. For ad hoc networks it is not easy to have the infrastructures ready in all cases. Also, HAs and FAs may become the performance bottleneck if they have to serve large number of mobile devices, which is likely to happen in wireless sensor networks. The integration of different mobility management mechanisms in heterogeneous wireless networks could be achieved by using Java RMI in the middleware layer. RMI communication layer is an excellent place to provide such roaming support and optimization over heterogeneous wireless networks for Java-based runtime systems. By careful design, we can change the delegation target in the middle of execution without breaking applications. Unlike Mobile IP and its variants, our approach uses direct connections, thereby avoiding the complexity caused by forwarding.

We demonstrate via a case study to illustrate the idea to support the mobile Java RMI for such a scenario by employing Java proxy support and mobility managements. Our solution includes several key technologies. First, we support Java RMI over Bluetooth layer. The Bluetooth layer is supported by incorporating a set of protocol stack layers for Bluetooth, known as *JavaBT* (developed by us), and by supporting the "logical link control and adaptation protocol" (L2CAP) layer with sockets to support Java RMI sockets. The Java RMI runs over Bluetooth L2CAP layer directly without going through IP layer. Second, the Java RMI over GPRS/WLAN is achieved by RMI implementation over IP layer. Finally, Java proxy and exception-handling mechanisms are employed to support roaming and resource scheduling in heterogeneous wireless environments. Our support for the roaming of Java RMI over heterogeneous wireless networks is based on the concept of direct connection, which avoids the problems caused by forwarding. The difficulty of this strategy is how to handle the existing connection when the mobile node (MN) moves to another location so as to avoid interruption of the high-level applications. We propose to solve this problem in Java RMI by the support of dynamic addresses and dynamic sockets. We develop algorithms to handle the handoff process. In addition, a transition diagram is used to illustrate our scheduling policy for the RMI layer for roaming in heterogeneous wireless environments. Methods for connect-loss detection and data-integrity maintenance in dealing with roaming scenarios are also presented.

Experimental results based on prototype implementation of the Java RMI support over heterogeneous wireless environments are presented. In our experimental testbed, our implementation of Bluetooth protocol stack for Java platform, *JavaBT*, was tested with a pair of Ericsson Bluetooth Development Kits (EBDKs) connected to a personal computer. We performed experiments using various benchmarks from the RMI Benchmark Suite [14], DHPC Java Benchmarks [13], and the Java Grande Forum MPJ Benchmarks [7]. The support for Java RMI over GPRS and WLAN was also tested. This work is also a part of our research efforts to develop and investigate the research issues to advance the technologies for distributed component architectures [5], [21], [20], [11], [10].

The remainder of the paper is organized as follows. Section II gives an overview of Java RMI middleware architecture. Section III mentions the support of Java RMI over L2CAP sockets of Bluetooth environments. This is used as an example of non-IP-based wireless system. Section IV describes the support of Java RMI over IP-based wireless networks. GPRS and WLAN are examples of such wireless systems. Section V describes the framework and architecture to support the roaming mechanisms for Java RMI in heterogeneous wireless network environments. Sections VI discuss the potential scheduling optimization issues and Section VII lists related work. The experimental results are then given in Section VIII. Finally, Section IX concludes this work.

II. An Overview of Java RMI Extension Framework

An RMI implementation consists of a set of libraries that comply with RMI specification. It simplifies the exporting of RMI remote objects, which includes initiating threads listening to RMI requests from the network and directing RMI requests to corresponding RMI remote objects. It also helps issuing RMI requests to remote objects in a way identical to the function calls on local objects. To improve or extend the functionality of current RMI implementation,

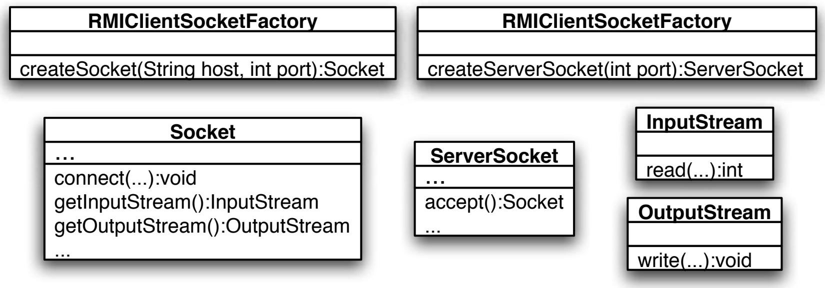


Fig. 1. RMI customization API.

we can either build our own RMI implementation, or we can resort to specific APIs defined in RMI specification, which provides the flexibility of dictating which socket implementation should be used on the data transportation during RMI. Here we describe briefly how these APIs interact with RMI.

Fig 1 shows related class types and functions. The objects of `RMIClientSocketFactory` and `RMIServerSocketFactory`, if presented as parameters in remote object initialization, will be used by RMI to produce necessary `Socket` and `ServerSocket` objects. These `Socket` and `ServerSocket` objects are then used by RMI to construct socket connections. Thus the applications dictate which socket implementation should be used by suppling RMI corresponding socket factory objects. For example, an application may require SSL sockets to be used in RMI for security reasons. In this case it can initialize its remote objects by objects of `SslRMIClientSocketFactory` and `SslRMIServerSocketFactory`, which implement `RMIClientSocketFactory` and `RMIServerSocketFactory` and will produce `SSLSocket` and `SSLServerSocket` objects.

When RMI gets `Socket` and `ServerSocket` objects, either the default implementation or from the factory objects assigned by the applications , RMI use them to construct necessary socket connections for data transportation. To form a connection, the `connect()` function of `Socket` is called to issue a connection request and the `accept()` function of `ServerSocket` is called to wait for connection requests. When the `ServerSocket` object accepts a connection request, a `Socket` object is created. Thus both ends of a connection have their corresponding `Socket` objects representing the connection. Then the `InputStream` and `OutputStream` object fetched from these `socket` objects can be used to transport data.

Although the flexibility provided by these APIs is limited, it extends RMI in a standard and compatible way. In the research of this paper, we successfully implant roaming support into RMI by this approach. This should make this work applicable to any RMI implementation.

III. Supporting Java RMI over Bluetooth Networks

We propose to integrate and manage the network usage in the RMI layer for distributed object-oriented environments. One of the tried networks is Bluetooth. This section presents our methodologies for supporting Java RMI over Bluetooth networks in a model without IP. It is an example of Java RMI over non-IP-based wireless system. Section IV presents Java RMI over IP-based wireless systems. GPRS and WLAN are examples of such wireless systems.

The main idea here is that by skipping the IP layer, which is not necessary for two devices to communicate using Bluetooth, the software drivers/libraries required in the Bluetooth devices can be relatively smaller. This is especially important for hand-held devices. In our researches, we have developed two sets of software: (1) a Java implementation of fundamental protocol stacks, HCI and L2CAP, for Bluetooth , called *JavaBT*, and (2) a Java RMI extension library (see Section II) built on top of *JavaBT* [20]. Notice that the basic functions that *JavaBT* provides can be found in normal Bluetooth native drivers. For performance reasons, the native drivers can be used to work with our RMI Bluetooth extension.

Traditionally, the Java RMI implementation runs over the TCP/IP networks. A straight-forward solution would be using Bluetooth drivers that build up to the TCP/IP layer. We choose not to involve TCP/IP protocols by extending RMI to run over the L2CAP layer of Bluetooth directly. Fig. 2 illustrate our idea. Fig. 2(a) is the traditional designs. Fig. 2(b) is our design as a case study to support the Java RMI directly over the L2CAP layer.

We develop a custom Java RMI library that uses a Bluetooth L2CAP layer instead of TCP/IP sockets. A custom RMI socket factory is used to transfer data in the Bluetooth L2CAP layer. Under the Java RMI layer, we also develop a Java Bluetooth protocol stack including an L2CAP layer, HCI layer, and a replaceable transport layer to communicate with the Bluetooth hardware through Java Native Interface (JNI). The Java library for L2CAP layer is implemented by us and known as *JavaBT*.

We build the following family of classes to provide functionality for the Java socket over our Bluetooth driver: *BluetoothServerSocket*, *BluetoothSocket*, *BluetoothInputStream*, *BluetoothOut-*

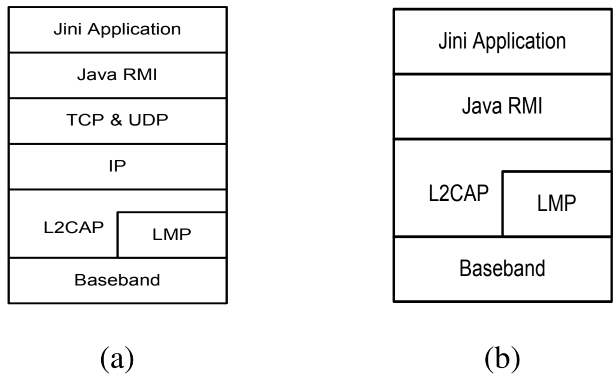


Fig. 2. (a) Run Java RMI over the TCP/IP support in the Bluetooth protocol stack. (b) Run Java RMI over the Bluetooth L2CAP layer directly.

putStream, and *BluetoothInetAddress*. Our *BluetoothServerSocket* class extends *java.net.ServerSocket* to provide the same interface as the original *ServerSocket*, except that it listens to the L2CAP layer. It also implements *L2CAPEventIndicationInterface* to receive L2CAP events. The *BluetoothSocket* class extends *java.net.Socket* to provide the same interface as the original socket, except that it transmits data over the L2CAP layer. A *BluetoothInputStream* class is used as an input stream for reading data from an L2CAPChannel object; it extends *java.io.InputStream*. In addition, a *BluetoothOutputStream* class is implemented as an output stream for writing data to an L2CAPChannel object; it extends *java.io.OutputStream*. Next, we have the *BluetoothInetAddress* class that extends *java.net.InetAddress* to represent the Bluetooth address. The BluetoothInetAddress wraps Bluetooth addresses and provides the same interface as *InetAddress*. Finally, to run the RMI over Bluetooth, we create the class *BluetoothRMISocketFactory* that extends *java.rmi.server.RMISocketFactory*, and implement the *createServerSocket* and *createSocket* methods. 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.

IV. Supporting Java RMI over IP-based Wireless Networks

This section describes related technologies that support Java RMI over IP-based wireless systems. GPRS and WLAN are examples of such wireless systems. In the default implementation

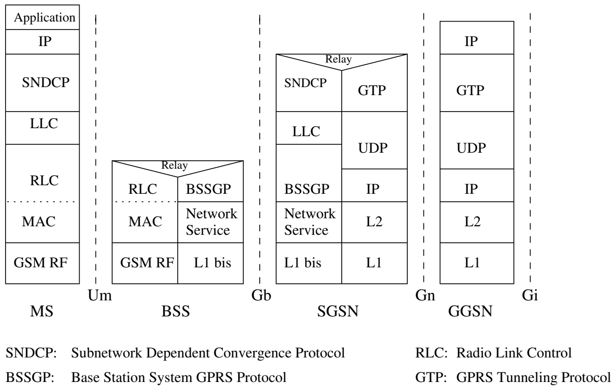


Fig. 3. User plane of GPRS

of Java RMI, sockets are created and used for underlying data communication. Inherently, Java RMI can be run over any wireless network that supports the IP layer.

There are existing technology and standards pertaining to constructing the IP layer in wireless networks such as WLAN and GPRS. WLAN standards usually focus on physical layer and Medium Access Control (MAC) layer. It is natural to run IP over WLAN. Thus Java RMI can run over WLAN. Similarly, Java RMI can run on top of GPRS because IP can run on top of GPRS systems. GPRS, an evolution of GSM, mainly extends packet data access by adding two nodes, Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN) in the core network. SGSN and GGSN play the role of packet transmission. SGSN connects to a Base Station System (BSS) which provides radio connections to GPRS terminals. All packets from the external Packet Data Network (PDN) can be routed by GGSN to the appropriate SGSN and finally reach the intended Mobile Station (MS).

The GPRS protocol stack in user plane is shown in Fig. 3. Application data originating from MS will pass through the IP layer and then the Subnetwork Dependent Convergence Protocol (SNDCP) layer, which is the interface between the network layer and the underlying network. A reliable ciphered logical link is provided by Logical Link Control (LLC). After processing by Radio Link Control (RLC) and MAC, data are transmitted through GSM radio. BSS can relay LLC PDUs between U_m and G_b interfaces. Both BSS and SGSN do not understand IP. BSS provides radio connection to MS. SGSN can track the location of MS. As shown in Fig. 3, GGSN supports IP layer. It functions like a router that connects to external networks. All packets from external networks can be routed by GGSN to the appropriate SGSN and finally reach the intended MS. The GPRS Tunneling Protocol (GTP) is used to tunnel IP packets between GGSN and SGSN.

Another aspect of supporting Java RMI over wireless networks is roaming handling, especially among heterogeneous networks. If all networks are based on IP and the IP layer supports mobility, roaming support of Java RMI should be easy. In such case, Java RMI could be simply regarded as an application running on top of the IP protocol stack. Mobile IP, for example, could be used for WLAN. For GPRS, GPRS Mobility Management (GMM) is defined by standards to support mobility for GPRS. Roaming among WLAN and GPRS could be achieved by using a gateway as we proposed earlier [22]. Alternatively, in this paper we propose to support mobility management by using Java RMI. Therefore, the IP layer of different wireless networks does not need to support mobility. The advantage of this approach is that Java RMI can be run over IP-based and non-IP-based wireless networks. Besides, the integration of different mobility management mechanisms in heterogeneous wireless networks could be achieved by using Java RMI in the middleware layer.

V. Roaming Support of Java RMI over Heterogeneous Wireless Networks

Previous sections described related technology on how to support RMI communication over particular wireless networks. Now we consider heterogeneous wireless environments consisting of multiple overlapped networks such as Bluetooth, WLAN, GPRS, or even 3G and Ultra-wideband (UWB). Inevitably these networks will coexist in the future due to their different abilities on coverage, bandwidth, QoS guarantee, and power consumption, etc.. This introduces new optimization issues on how to efficiently utilize available networks to accomplish communications if we can move from one network to another without interrupting applications. We believe that the RMI communication layer is an excellent place to provide such roaming support and optimization over heterogeneous wireless networks for Java-based runtime systems. As Figure 4 shows, we insert a new layer in RMI by creating necessary classes mentioned in Section II. The classes with the name starting with “dynamic” collaborate to implement our roaming mechanism. They will delegate calls to underlying classes which do the real communication jobs for respective networks. By careful design, we can change the delegation target in the middle of execution without breaking applications.

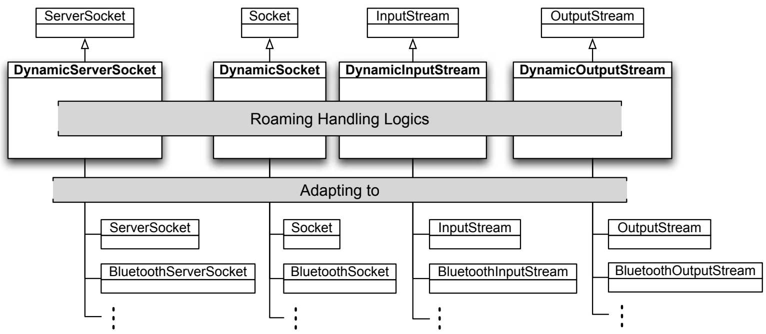


Fig. 4. An inserted layer in RMI to provide the roaming mechanism.

A. Roaming Mechanism

Roaming in a wireless environment has previously been addressed using Mobile IP by providing additional software supports known as home agents, foreign agents, and tunneling protocol to provide the underlying roaming mechanism. The concept of Mobile IP is simply based on packet forwarding. Our support for the roaming of Java RMI in a heterogeneous wireless environment is similar to Mobile IP, but adopts a direct-connection strategy to transfer the packets. In Mobile IP all the communications toward the Mobile Node (MN) are generally passed through the HA and FA; this is the case even if the MN is in fact moving into the same network of the Corresponding Node (CN). This well-known problem in Mobile IP is called the triangular routing problem. Several solutions have been proposed to overcome the problems. To classify mobility management protocols, a well-known approach is to categorize mobility management into *macro-mobility management* and *micro-mobility management*. Macro-mobility management mostly is still based on Mobile IP. Micro-mobility management, however, limits frequent handoffs into *micro-domains*. It intends to localize the location update and location registration to reduce handoff latency in Mobile IP. It potentially could also reduce the end-to-end latency. Normally, micro-mobility management protocols could cooperate with Mobile IP, which serves as the macro-mobility management protocol.

Many micro-mobility management protocols have been proposed. Generally, they can be categorized as *tunnel-based* and *host-specific-routing-based* protocols. Examples of the tunnel-based micro-mobility management protocols include Hierarchical Mobile IP (HMIP) [24], Intra-Domain Mobility Management Protocol (IDMP) [25], Multicast-based mobility [26], and Dynamic Hierarchical Mobile IP (DHMIP) [27]. The tunnel-based protocols usually employ a hierarchical mobility architecture and require a *gateway* to tunnel packets to and from mobile stations. The host-specific-routing-based protocols, on the other hand, adopt new routing schemes to support mobility within the micro-domain. That is, standard IP routing is not used inside the micro-domain. Examples of this category include Cellular IP [28], HAWAII [29], Fast Link-layer and Intra-domain Handoffs (FLIH) [30], and Mobility-aware MPLS [31]. Detailed comparison of various protocols for mobility management can be found in [23].

In contrast to these previous approaches, our approach uses direct connections, thereby avoiding the complexity caused by forwarding. There are two issues in implementing direct-connection roaming: (1) informing the CNs of the new location of the MN to enable connecting to it, and (2) migrating the connections without data loss. These two parts should work without involving high-level applications. We have developed two techniques, *dynamic address* and *dynamic socket*, to handle these issues, as described below.

1) **Dynamic Address:** In order for the CNs to connect to the migrated MN directly without affecting programs, we use an object called `DynamicAddress` that is an extended object of the 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, this object first fetches the correct foreign address of the MN from a registry. Such a registry that holds the latest location of the MN needs to be maintained somewhere reachable by others. This can be achieved by forcing the MN to register its new foreign address with the registry every time this changes. The overheads associated with querying on the registry when constructing a connection is actually small compared to the benefits gained from the direct-connection communications. This is because the RMI tends to reuse 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 migrating existent connections to the direct-connection implementation is not trivial. The Java Dynamic Proxy technique can be employed to adapt socket objects of different wireless networks. It adapts connection-related objects to a new set that forms the new connections. We provide each type of wireless network with a specialized socket support consisting of `Socket` objects, `ServerSocket` objects, and

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.
Step 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. 5. The algorithm for constructing the RMI connection using dynamic IP.

the `InputStream` and `OutputStream` objects. When we are migrating connections to another wireless network, we simply build the necessary new connections, issue a component adaptation, and have the communication forwarded to the new one. The applications using RMI will incorporate the corresponding socket factories of these proxied sockets, called *dynamic sockets*, and need not be aware of the change of underlying networks.

We integrate the above two concepts into Algorithms 1 and 2 to give steps to construct the connection and perform roaming, respectively. Algorithm 1 (shown in Fig. 5) is our proposed new procedure for a Java RMI to make an invocation by employing dynamic sockets (proxied sockets). Algorithm 2 (shown in Fig. 6) gives the procedures for connections in the roaming situation for both RMI client and server. In the case that the wireless network in use is out of range, the communication object in the client catches an exception and then uses Algorithm 2 to try to rebuild the connection and recover the failed RMI operation.

We now use the scenario in Fig. 7 to explain these two algorithms. As we can see in Fig. 7, an MN M, hosting an RMI service, first sets its home address `ha` to the current dynamic address, and registers it to the registry with the foreign address `fa` equaling `ha`. Then a node A tries to make an connection to MN M to access its service. According to Algorithm 1, node A invokes remote methods on the stub of the mobile service, and this stub finds that no connection is available. It attempts to use the `DynamicSocketFactory` instance to obtain a connected socket to M. The constructor of the socket object is given a `DynamicAddress` object, which

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 or registration to get an available dynamic address.

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. 6. The roaming algorithm for moving the connection to the new network.

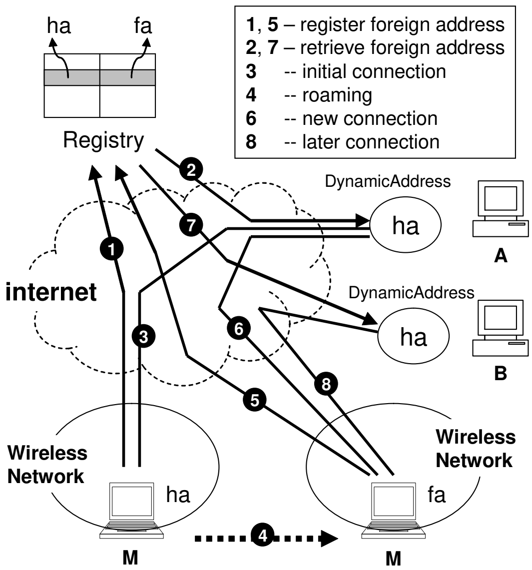


Fig. 7. The underlying architecture of RMI to provide roaming mechanism.

queries the registry and fetches the registered foreign address of M when it is used. If the socket fails to construct this connection, it will repeat this procedure until success or until a certain time period has elapsed. Once the connection is up, the RMI operation can proceed. Assume that MN M subsequently roams to another network; it applies Algorithm 2 to migrate its RMI service to the new location. It first detects and acquires the currently available network services, and then immediately reports the new dynamic address as its new foreign address to the registry. This results in the loss of an existing connection, which needs to be rebuilt. Node A repeats the above actions to try to make a connection to M, and will get the new address of M. As the RMI server, the MN M uses the `DynamicServerSocketFactory` instances to create a new server socket and restarts its service. When the connection between M and A is eventually formed, they can exchange information 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 that they send and receive. 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. Advanced Design Issues and Implementation

There are two issues incurred when we are providing the roaming mechanism for heterogeneous wireless environments. One is the detection of lost connections and the other is the maintenance of data integrity. We describe these two issues in the follow context with proposed solutions and implementation.

1) **Connection-Loss Detection**

Our approach in providing the roaming ability in heterogeneous wireless environments is established in a higher layer, a software layer, rather than the IP layer such as Mobile IP does. The philosophy is that if we can manipulate different kinds of networks to transfer data through corresponding software drivers and libraries, we should be able to insert a software layer between the applications and the network drivers/libraries to manage the data transferring. One of such management is the dynamic change of the used network while not interrupting the transferring, i.e. roaming from one network to another. Therefore, the mechanism of connection-loss detection here should be as generic as possible to be able to be applied on any network.

The simplest solution is to monitor the data transferring. If the receiving of data timed out, the connection is considered lost. Due to the implementation described in the next issue, both sides of a connection will expect messages from each other periodically to maintain their buffers. Thus the lose of the connection will be detected by the both sides through the time-out mechanism even that no data transferring is currently requested by the application. If the implementation described in the next issue is not adopted, using periodic heart-beat messages can achieve the same effect. Although this approach sounds trivial, its concept is applicable on almost every network system. Besides, we can still incorporate network-specific loss-detecting technique in the implementation. This time-out mechanism will serve as the default loss-detecting method if no special detecting mechanism is presented for a kind of network.

The dynamic-sockets middleware is designed to recover the connections as soon as it detects a connection loss. Suppose that t and t' are predefined values, and that t' is larger than t. The `DynamicOutputStream` objects on both sides of a connection are designed to send out a heart-beat message if for t time no data have been requested to be sent out. This infers that if no data arrive for the past t' time, the connection might be lost. There is, of course, the possibility that the packets are simply delayed and that the connection is still alive. It is therefore necessary for the value of t' to be sufficiently higher than t such that network delays will not interfere with the detection of a connection loss. However, a higher t' may also slow down the detection of a connection loss.

Figure 8 shows the object details of the input and output stream pair. Each connection consists of two of such pairs with data flowing in opposite directions. The reading timeout of the network connection is set to t'. Both the `DynamicOutputStream` and `DynamicInputStream` objects are associated with a working thread and a buffer. For `DynamicOutputStream`, the thread labeled "S-Docker" continuously moves data from buffer S to the network. The S-Docker thread checks the network activities and will send out heart-beat messages if necessary. For `DynamicInputStream`, the R-Docker thread continuously moves data from the network to buffer R, and therefore is responsible for detecting reading timeouts. A semantical timeout for high-level data reading

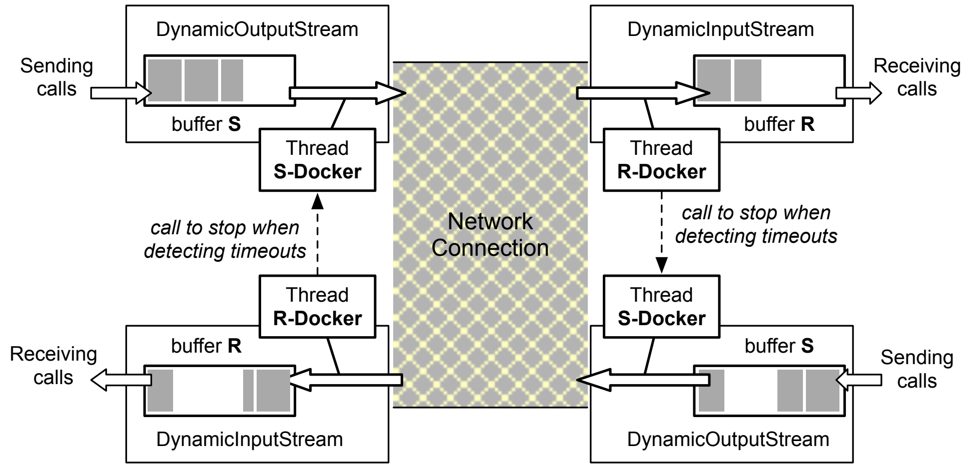


Fig. 8. The objects for the timeout detection mechanism.

is implemented as the timeout of blocking on reading buffer R if no data have arrived. Note that thread R-Docker and buffer R are required because the reading behavior of `DynamicInputStream` is non-controllable. A thread is needed to continuously buffer incoming data so that a connection loss will eventually cause a receiving timeout. Also, when R-Docker detects a timeout event (and thus that the connection is judged as lost), it will stop the operation of S-Docker on the same side. In this way, if the timeout is caused by unexpected long network delays only in one direction, the stopped S-Docker will in turn cause a timeout in the corresponding R-Docker thread. It is for this synchronization aspect that judgment of the connection status should be consistent at both ends.

Detecting connection loss by monitoring the timeout might not be the most elegant solution, but it is applicable to all kinds of network connections. Implementations can use it as a bottom-line solution and then provide additional detection mechanisms for specific networks. For example, the signal strength can be used as the basis to sense the connection status for WLAN.

2) **Maintenance of Data-Integrity**

Most networks provide reliable data transportation, but the data still can be broken during roaming even that reliable data transportation is guaranteed by the used networks. The problem of data-integrity arises because it is impossible to cease the data sending as soon as a connection is lost. The data that the applications assume sent between the time

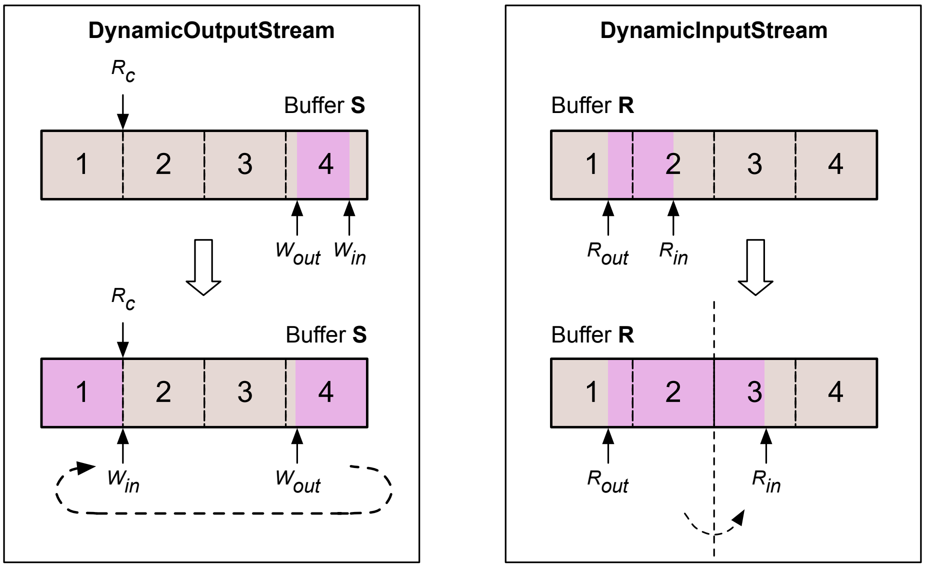


Fig. 9. The design of buffer S and R for the data resending scheme. The pointer R_c is important for the preservation of unconfirmed-received data so we can resend the lost data.

a connection is lost and the time the lost connection is detected don't actually arrive at the receiving side. If this part of data are not resent to the receiving side before resuming the transferring, the data integrity will be broken.

The solution we use here is to buffer the data sent to the receiving side. Once the receiving side confirms the arriving of data, the content of this buffer can be discarded. In this way we can keep the data that might need to be resent later if the connection is lost and resent the data transparently without the intervention of user applications.

Out implementation consists of two pairs of buffers and the protocol to synchronize the buffer pointers. Figure 9 gives the design of buffers S and R for the data resending scheme, which are identical cyclic buffers (with the same size and section divisions). When the `DynamicOutputStream` object is requested to send out some data, the data will first been stored in buffer S and pointer W_{in} will be updated; pointer W_{out} is maintained by the S-Docker thread, which will continuously move the data stored between pointers W_{out} and W_{in} to the network. Similarly, when the data-reading methods of the `DynamicInputStream` object are called, the data stored between R_{out} and R_{in} are returned and R_{out} is updated. Pointer R_{in} is updated by the R-Docker thread if data are moved from the network into buffer R. If pointer W_{out} reaches the position of W_{in}, thread S-Docker will send no data. If pointer R_{out} reaches the position of R_{in}, the data reading methods of `DynamicInputStream` will be blocked. Additionally, we force the `DynamicInputStream` object to send out a message whenever pointer R_{in} crosses the section boundaries. These messages can be hidden in the normal data packets or heart-beat messages that come from the `DynamicOutputStream` object on the same side. When the `DynamicInputStream` object on the remote side sees these messages, it will update pointer R_c of buffer S in the `DynamicOutputStream` object in its side. In this scheme, the data stored between pointers R_c and W_{out} will be the data not confirmed as received. When pointer W_{in} reaches the position of R_c, the data-sending methods of `DynamicOutputStream` will be blocked in order to protect those data not confirmed as received. Now, after reconstructing the connection, the position of pointer R_{in} on both sides will first be sent to each other. The data stored between the position of remote R_{in} and pointer W_{out} are then the lost data and should be retransmitted.

VI. Roaming and Scheduling Optimization

In the design architecture of Fig. 7, once the wireless network currently in use is not reachable (possibly due to being out of range), the direct-connection object catches an exception and uses Algorithm 2 to try to rebuild the connection and recover the failed RMI operation. This can be optimized by employing more aggressive roaming decisions and scheduling, because the coverage of several wireless networks might overlap and hence be available simultaneously. Table I compares different characteristics of Bluetooth, WLAN, and GPRS networks, indicating that each has its own advantages. For example, GPRS provides the largest coverage, but exhibits high power consumption, low bandwidth, and a high connection cost. In contrast, WLAN can provide a high bandwidth over a small range with low connection costs. Bluetooth has the lowest power consumption while connected and a higher bandwidth than GPRS. Bluetooth is therefore suited to fast communications between small personal devices. A node with several wireless networks available may purposely roam to another network to acquire benefits in bandwidth, power consumption, or connection cost.

Figure 10 is a transition diagram explaining our scheduling policy for the RMI layer in heterogeneous wireless environments in the case of roaming. Initially, the RMI implementation

	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

TABLE I

A COMPARISON OF BLUETOOTH, WLAN AND GPRS.

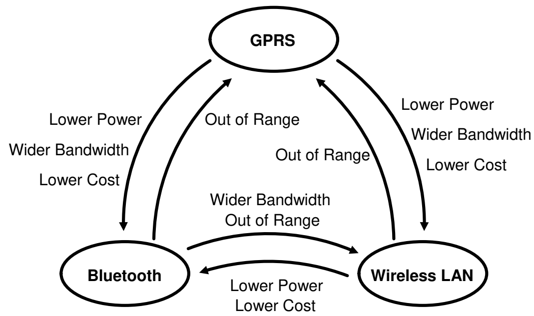


Fig. 10. The state transition graph representing our scheduling policy.

stays in one state. Later, when other wireless networks are detected following movement of the MN, the transition conditions are evaluated to determine whether to roam to another wireless network. Note that other scheduling policies or QoS controls can also be implemented in this layer. Our design and architecture presents a layer for controlling scheduling between object-oriented applications and heterogeneous wireless network environments, which is certainly important for examining how applications perform RMI accesses in heterogeneous wireless environments. Related research results such as on the QoS in the wireless networks [8] can also be incorporated.

VII. EXPERIMENTAL RESULTS

In our experiments we assessed the use of Java RMI in various wireless environments. In the first set of experiments, we evaluated the robustness of our implementation of RMI software over Bluetooth networks, performing experiments with various benchmarks from the RMI Benchmark Suite [14], DHPC Java Benchmarks [13], and the Java Grande Forum MPJ Benchmarks [7]. The last benchmarks are based on MPJ, which is an MPI-like message-passing interface for Java. We

TABLE II

THE DETAILED DESCRIPTIONS OF BENCHMARKS.

Name	Description	Data Size	From
EP	NAS EP generating Gaussian random numbers	16,777,216	[13]
Series	Fourier coefficient analysis	10,000	[7]
LUFact	LU Factorisation	500	[7]
SOR	Successive over-relaxation	1,000	[7]
Crypt	IDEA encryption	3,000,000	[7]
Sparse	Sparse Matrix multiplication	50,000	[7]
MolDyn	Molecular Dynamics simulation	2,048	[7]
MonteCarlo	Monte Carlo simulation	2,000	[7]
RayTracer	3D Ray Tracer	150	[7]
SelSort	Selection Sort	524,288	[9]
Hamming	Given an array of primes, output in increasing order and without duplicates, all integers of the form $a^i * b^j * c^k ... \leq n$	5	[14]

have implemented an MPJ interface that follows the mpiJava 1.2 specification [3]. It uses RMI as the underlying communication channel, and was used to evaluate the performance of our RMI implementation over Bluetooth. Table II lists the applications that our RMI ran successfully in our early tests of the robustness of our software.

In our testbed, the EBDKs are connected to two computers. We tested the performance using one node and two nodes that are, respectively, sequential and parallel versions with RMI over Bluetooth. The RMI server has one 1200-MHz AMD CPU, 768 MB of RAM, and runs Microsoft Windows 2000 Server. The RMI client runs on an IBM ThinkPAD laptop that has a 700-MHz Intel Pentium III CPU, 256 MB of RAM, and runs Microsoft Windows XP Professional. Our *JavaBT* Bluetooth protocol driver and the test applications run on Microsoft JavaVM that supports JDK 1.1.8 with additional RMI support and Sun Java Communications API v2.0 for Windows. The performance is shown in Fig. 11. Despite the limited bandwidth of Bluetooth, we still observe performance gains on Hamming, EP, Series, RayTracer, and SelSort. Note that due to the client

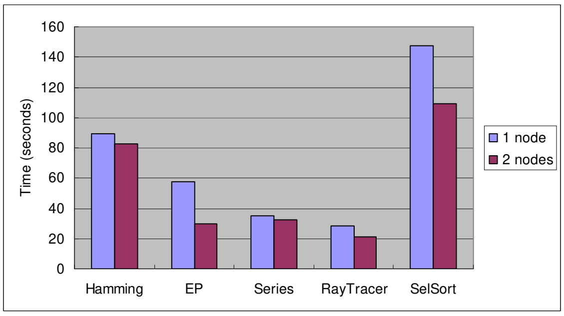


Fig. 11. The performance results of benchmark applications of Java RMI over bluetooth environments.

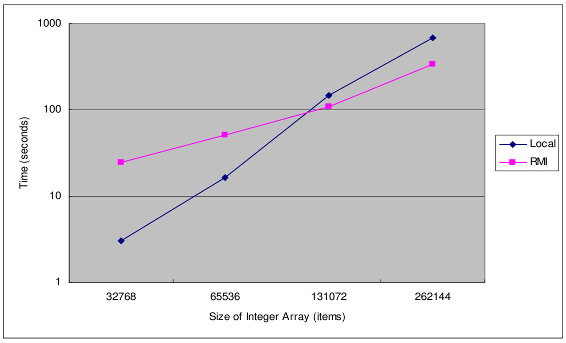
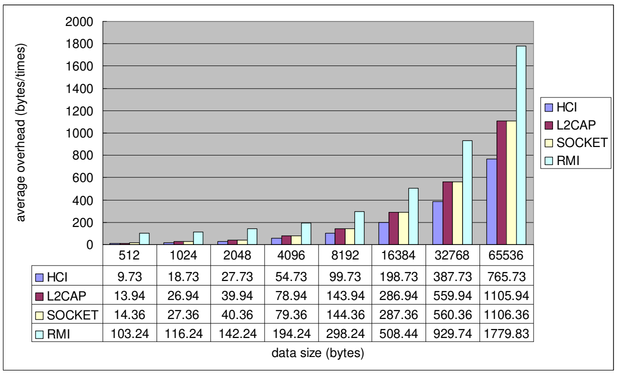


Fig. 12. SelSort with local computation and a 2-node version based on RMI over Bluetooth communications.

and server residing in different machines, the optimal speedup for this case is only around 1.7 times. This is due to the peak data rate being only 723 kbps. However, this result indicates that the numerical applications run well with RMI over Bluetooth can also be observed from our experimental results. Collaborative software that requires less communication among computing nodes will likely run very well in this environment. Note that our support for RMI over Bluetooth also gives more high-level control in Bluetooth networks and a better programming environment. The system load at the OS level can also be distributed to remote servers with RMIs.

We now describe a specific experiment to provide more insight into our results. We chose the selection-sort experiment, in which the selection-sort task is divided into two parts and sent to a remote computer over Bluetooth RMI. Figure 12 illustrates the result, which indicates that when the array size is larger enough, the distributed version running over the Bluetooth RMI exhibits have better performance than the local version.

In the second experiment, we evaluated the overheads associated with different layers of



	512	1024	2048	4096	8192	16384	32768	65536
HCI	9.73	18.73	27.73	54.73	99.73	198.73	387.73	765.73
L2CAP	13.94	26.94	39.94	78.94	143.94	286.94	559.94	1105.94
SOCKET	14.36	27.36	40.36	79.36	144.36	287.36	560.36	1106.36
RMI	103.24	116.24	142.24	194.24	298.24	508.44	929.74	1779.83

Fig. 13. The overhead comparison for different layers of Java communications over Bluetooth. Additional data sent at different layers are observed.

JavaBT and with RMI. The comparison is with HCI, L2CAP, and socket layers of our *JavaBT* and our RMI version. In this experiment, we transmitted arrays of different sizes from one computer to another using our Bluetooth protocol driver. Figure 13 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 the Bluetooth device, which is investigated to find the additional data being sent at different layers in our current implementation. The largest overheads occur in the RMI version due to the data serialization inherent in RMI.

In the third experiment, we tested an application scenario of RMI components adapting and roaming in GPRS and WLAN wireless environments. The GPRS system in our experiment is equipped with Home Location Register (HLR), Base Station System (BSS), Gateway GPRS Support Node (GGSN), and Serving GPRS Support Node (SGSN). The WLAN in our testbed is based on IEEE 802.11b. For Java applications written with RMI components in heterogeneous wireless environments, we employ the dynamic proxy and socket described in Section V to aid applications roaming over different wireless networks. In our experiment, an application is initially running RMI over GPRS, but as the MN moves into the coverage of a WLAN, the RMI component over WLAN can be chosen. We demonstrate this idea with specific experiments. Consider an MN communicating with the PC and running MPJ benchmarks through the RMI over GPRS. When the MN moves into the coverage of one WLAN, the RMI component over WLAN is adopted. All of the experiments are performed such that the second RMI occurrences of each application is set as the time that the device moves within range of the WLAN. Figure 14

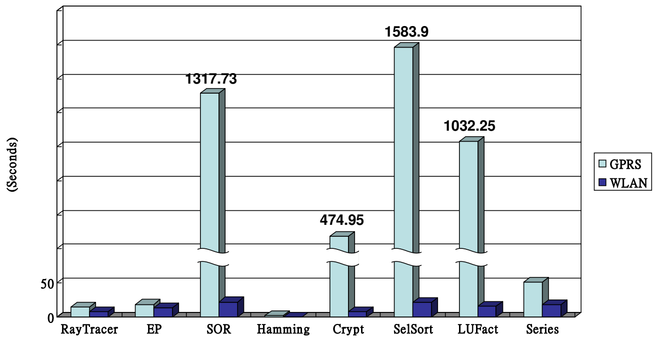


Fig. 14. Adapting RMI component over GPRS/WLAN wireless environments

shows the results. The applications are from Java Grande (MPJ) benchmarks. The line labeled with GPRS denotes that the application always runs RMI components over GPRS. The line labeled with WLAN denotes that the application initially runs RMI components over GPRS, but soon roams into WLAN as we bring the WLAN within range. Significant performance gains are achieved in this case.

Consider now the handoff overhead for RMI over heterogeneous wireless networks. We describe the performance associated with roaming from GPRS into the WLAN. The handoff overheads includes the time to configure and adopt the new IP session from the WLAN and the time to reconstruct connections and adopt new components. In our MN, we use two NICs to access GPRS and WLAN separately. The setting and switching time in average is 52 ms. The reconstruction of the connection within the WLAN takes an additional tens of milliseconds per connection, for communicating with the registry and the remote side (These are performed in parallel). The component adaptation time is generally smaller than 1 ms. In total, the handoff time is about 65 ms. In an alternative solution using RMI over Mobile IP, it takes 57.4 ms to send the registration request, 18.1 ms for the MN to receive an acknowledgment from the HA, and 4.5 ms for the MN to receive packets from the CNs. The average handoff latency for using Mobile IP is thus 77.3 ms. We can see that the handoff overheads of our approach and Mobile IP are about the same. However, our solution has some advantages as discussed earlier. Note that our method uses direct connections, which can further reduce the latency in a large network. In addition, we present a layer for controlling scheduling and roaming between object-oriented applications and heterogeneous wireless network environments.

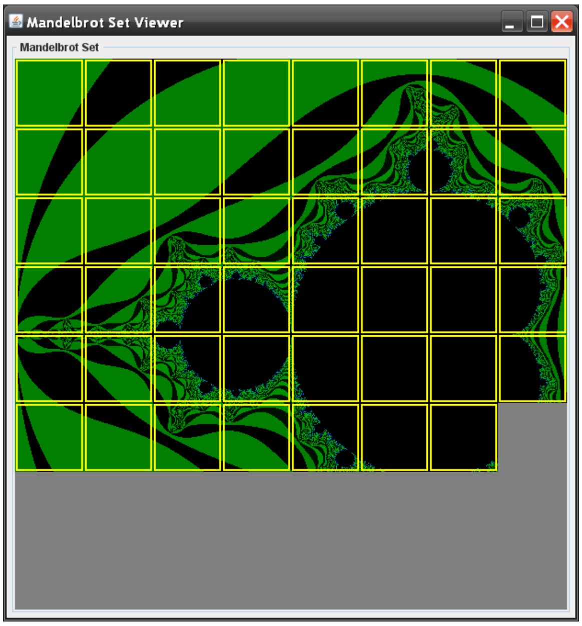


Fig. 15. The mandelbrot viewer application.

Finally, we tested scenario of RMI components adapting and roaming in WLAN and bluetooth wireless environments. We examine the roaming mechanism by implementing an application, Mandelbrot [18], using the RMI components. The Mandelbrot set viewer is composed of a reomte calculating server and a viewer client. A mandelbrot set is divided into 64 tasks, and sequentially issued to remote server for calculating the visual data. After the calculation, client can access the remote data via RMI componets, and draw out the Mandelbrot set picture, shown in Figure 15.

The server and client are connected by the WLAN or Bluetooth ad-hoc network. The RMI components will provide a seamless RMI session. We use two computers as the RMI server and RMI client. The client provides with a WLAN (802.11b) and a Bluetooth dongle for ad-hoc networking, and the server also provides with a Ethernet network and a Bluetooth dongle. In this experiment, the RMI session are designed to roaming between the WLAN and Bluetooth network, every 8 task span. The application is initially running RMI over WLAN. Figure 16 presents the experimental result. The x-axis shows the progress of the amount of tasks being done in the application. The y-axis presents the total execution time accumulated. Because of the bandwidth difference between WLAN and bluetooth, the result forms two different slopes of task

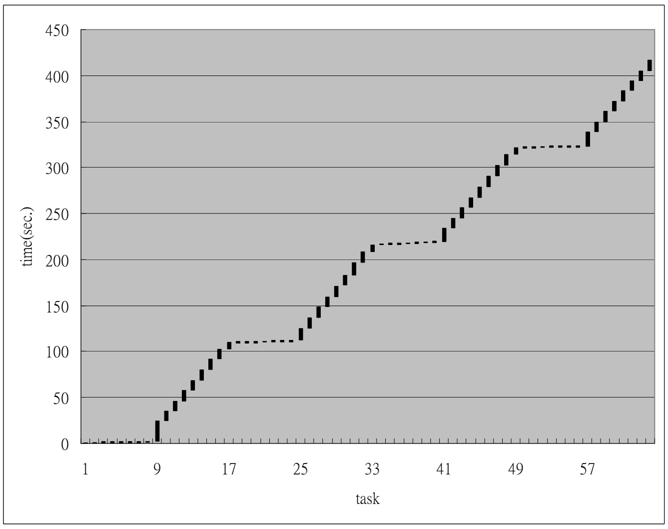


Fig. 16. The performance result of a Mandelbrot application over WLAN/Bluetooth wireless environments.

groups. The WLAN throughput is approximately 50 times faster than the Bluetooth network. Therefore, we can easily to tell which task was done from WLAN or Bluetooth Ad-hoc network. The sharp slope is the one by bluetooth execution, as it takes more time for execution per task unit. Focus is now directed to the execution time of task 9, it costs almost twice time than other tasks in the same group. This is due to bluetooth hardware dongle initialization (it takes 10s in average). After the hardware initialization, when the next roaming event happens, the RMI adapting from WLAN to bluetooth and from bluetooth to WLAN are only takes 18ms and 550ms, respectively.

VIII. Related Work

Another communication paradigm for wireless ad hoc networks would be using loosely-coupled (asynchronous) communication, such as Java Message Service (JMS) or Web Service Reliable Messaging. Loosely-coupled communication usually requires additional supports of the environments to route the messages. Works [35] have been done to try to remove this necessity. Compared to our work in supporting mobiling managements and connections, JMS has also provided abstractions for reliability on top of RMI. However, our scheme can be used to support applications written in RMI but without reliability layer supports or JMS reliability supports. Issues have also been raised that RMI can also be replaced by SOAP and HTTP REST messaging for the considerations of that permanent connections might not scale well. We leave that as future work on how to best coordinate resources for different layers of distributed abstractions.

The most well-known roaming technology is probably Mobile IP (MIP) [32]. MIP provides terminal mobility at the IP layer. Most networks today have the IP layer built on them. For vertical handoff support over IP-based networks, MIP might be a reasonable choice. But for some networks such as Bluetooth, presenting the IP layer could bring extra efforts and costs. This is especially important for hand-held devices with limited hardware resources. Another fact about MIP is that MIP works with pre-established network infrastructures. MIP uses home agents (HA) and foreign agents (FA) to forward packets. For ad hoc networks it is not easy to have these infrastructures ready in all cases. Also, HAs and FAs may become the performance bottleneck if they have to serve large number of mobile hosts. Some routing issues such as the triangular-routing problem should be taken care as well to avoid unnecessary traveling of packets. The BARWAN architecture proposed in [34] hosts heterogeneous mobile computing for diverse types of networks. It use MIP-based handoff mechanism. The Snoop protocol used in BARWAN has similar goal as the data integrity maintenance mechanism proposed in this paper. It caches unacknowledged TCP segments and performs local retransmissions of lost segments.

Another way to provide vertical handoff is using end-to-end mobility management. The method we proposed in this paper belongs to this catalog. In [33] Guo *et al.* presented a architecture and some key elements for end-to-end roaming schemes, which includes the connection manager (CM), the virtual connectivity (VC), and the local connection translation (VCT). The roaming decision algorithm and metrics mentioned in [33] considers the roaming from WLAN to WWAN as well as WWAN to WLAN. They also have a specific implementation which handles roaming in the kernel level.

Efficient support for Java RMI has been an important topic for investigations [19], [16], [12], [14], [2]. Previous work addressed the issues of efficiency and for grid environments. We address the issues to support the mobile Java RMI over heterogeneous wireless networks without Mobile IP support. This is a likely scenario in a peer to peer environment, as some of the embedded devices might communicate with other devices via wireless protocols, but without Mobile IP support.

IX. CONCLUSION

Methods for efficiently supporting Java RMI have received considerable attention because RMI provides a layer of abstraction for communications. In this paper, we have reported our first-hand experience with the design and implementation of Java RMI over heterogeneous wireless networks that supports roaming. Our architecture incorporates a layer for controlling scheduling between object-oriented applications and heterogeneous wireless network environments. The experiments show that our implementation can effectively support parallel and distributed control of Java layers in heterogeneous wireless environments.

X. ACKNOWLEDGMENTS

We thank Mr. Hong-Wei Lin and Mr. Wang-Jer Wu for their help in constructing the testbed.

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] 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):941–955, 1998.

[3] 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.

[4] Cheng-Wei Chen, Chung-Kai Chen, Jyh-Cheng Chen, Chien-Tan Ko, Jenq-Kuen Lee, Hong-Wei Lin, and Wang-Jer Wu. Efficient Support of Java RMI over Heterogeneous Wireless Networks. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pp. 1391-1395, June 2004.

[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] Chung-Kai Chen, Yu-Hao Chang, Cheng-Wei Chen, Yu-Tin Chen, Chih-Chieh Yang, Jenq-Kuen Lee. Efficient Switching Supports of Distributed .NET Remoting with Network Processors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pp. 350-357, June 2005.

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

[8] P.J.M. Havinga and G.J.M Smit. QoS Scheduling for Energy-Efficient Wireless Communication. In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC2001), special session on recent advances in wireless networking and communications (invited), Las Vegas, NV, April 2–4, 2001.

[9] 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.

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

[11] 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(2):227–241, 1997.

[12] 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 Symposium on Principles and Practice of Parallel Programming*, pp. 173–182, Atlanta, GA, May 1999.

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

[14] 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.

[15] C. Perkins. IP Mobility Support for IPv4. *IETF RFC 3344*, Aug. 2002.

[16] R. Raje, J. Williams, and M. Boyles. Asynchronous Remote Method Invocation Mechanism for Java, *Concurrency: Practice and Experience*, 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] Peter Alfeld, *The Mandelbrot Set*, Dep. of Mathematics, University of Utah,
http://www.math.utah.edu/ pa/math/mandelbrot/mandelbrot.html.

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

[20] 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*, November 2002.

[21] Chung-Kai Chen, Cheng-Wei Chen, and Jenq Kuen Lee. Specification and Architecture Supports for Component Adaptations on Distributed Environments. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2004.

[22] J.-C. Chen and H.-W. Lin. A Gateway Approach to Mobility Integration of GPRS and Wireless LANs. *IEEE Wireless Communications*, pp. 86–95, April 2005.

[23] J.-C. Chen and T. Zhang, *IP-Based Next-Generation Wireless Networks*. Wiley, Jan. 2004.

[24] E. Gustafsson, A. Jonsson, and C. Perkins, "Mobile IPv4 Regional Registration." IETF Internet Draft, <draft-ietf-mip4-reg-tunnel-00>, Nov. 2004.

[25] S. Das, A. McAuley, A. Dutta, A. Misra, K. Chakraborty, and S. K. Das, "IDMP: An Intra-domain Mobility Management Protocol for Next-generation Wireless Networks," *IEEE Wireless Communications*, pp. 38–45, June 2002.

[26] A. A.-G. Helmy, M. Jaseemuddin, and G. Bhaskara, "Multicast-based Mobility: A Novel Architecture for Efficient Micromobility," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 677–690, May 2004.

[27] W. Ma and Y. Fang, "Dynamic Hierarchical Mobility Management Strategy for Mobile IP Networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 664–676, May 2004.

[28] A. Campbell, J. Gomez, S. Kim, A. G. Valkó, C.-Y. Wang, and Z. R. Turányi, "Design, Implementation and Evaluation of Cellular IP," *IEEE Personal Communications*, pp. 42–49, Aug. 2000.

[29] R. Ramjee, K. Varadhan, L. Salgarelli, S. R. Thuel, S.-Y. Wang, and T. L. Porta, "HAWAII: A Domain-based Approach for Supporting Mobility in Wide-area Wireless Networks," *IEEE/ACM Transactions on Networking*, vol. 10, pp. 396–410, June 2002.

[30] J.-C. Chen and P. Agrawal, "Fast Link Layer and Intra-domain Handoffs for Mobile Internet," in *Proc. of IEEE International Computer Software and Applications Conference (COMPSAC)*, (Taipei, Taiwan), pp. 325–330, Oct. 2000.

[31] H. Kim, K.-S. D. Wong, W. Chen, and C. L. Lau, "Mobility-aware MPLS in IP-based Wireless Access Networks," in *Proc. of IEEE GLOBECOM*, (San Antonio, TX), pp. 3444–3448, 2001.

[32] Charles E. Perkins, "Mobile Networking Through Mobile IP," *IEEE Internet Computing*, vol. 2, pp. 58–69, Jan/Feb 1998.

[33] Chuanxiong Guo, Zihua Guo, Qian Zhang, and Wenwu Zhu, "A Seamless and Proactive End-to-End Mobility Solution for Roaming Across Heterogeneous Wireless Networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, issue 5, pp. 834–848, June 2004.

[34] E. Amir, H. Balakrishnan, Y. Chawathe, A. Fox, S. D. Gribble, T. Hodes, G. Nguyen, V. N. Padmanabhan, M. Stemm, S. Seshan, and T. Henderson, "A Network Architecture for Heterogeneous Mobile Computing," *IEEE Personal Communications*, vol. 5, issue 5, pp. 8–24, Oct 1998.

[35] Mirco Musolesi, Cecilia Mascolo, Stephen Hailes, "Adapting Asynchronous Messaging Middleware to Ad Hoc Networking," *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, October 2004.

APPENDIX

A. Acronyms

The following table list the frequently used acronyms in this paper for the help of reading.

CCA	Common Component Architecture
CN	Corresponding node
FA	Foreign agent
GPRS	General Packet Radio Service
HA	Home agent
HCI	Host Controller Interface
L2CAP	Logical Link Control and Adaptation Protocol
MN	Moving node
OGSA	Open Grid Service Architecture
RMI	Remote Method Invocation
UWB	Ultra-wideband
WLAN	Wireless local area network

(The following context provides detailed technique information about JavaBT. *It won't be presented in the final version of this paper.)*

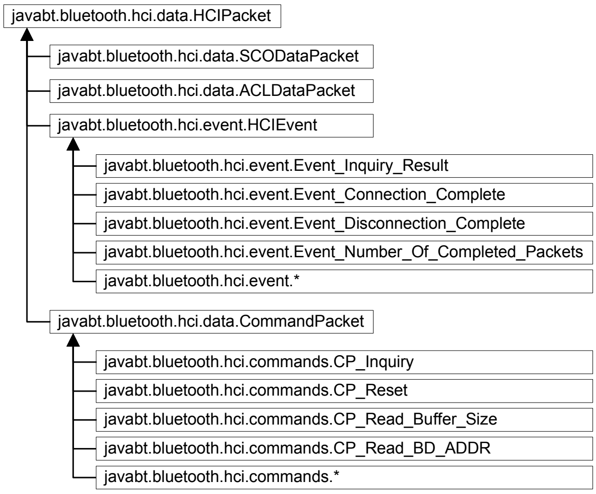


Fig. 17. The class hierarchies for handling HCI packets.

B. JavaBT: Bluetooth Protocol Driver

We use the Java programming language to build a set of layers in a Bluetooth protocol driver. We call our system *JavaBT*. In this section we focus on the support for the HCI and L2CAP layers, as they provide the low-level support for our implementation of Java RMI over a Bluetooth environment.

Each Bluetooth device is allocated a unique 48-bit Bluetooth device address (BD_ADDR), which is derived from the IEEE 802 standard. In *JavaBT*, we define a class, BD_ADDR, to represent a Bluetooth address. In our protocol driver, we use the BD_ADDR objects to indicate which Bluetooth device to connect with.

1) The HCI Layer: The HCI layer provides a uniform command method for accessing the Bluetooth hardware capabilities of the higher layers of Bluetooth protocol stack. The HCI driver sends data and commands to the Bluetooth hardware via HCI packets. There are four types of HCI packet: the command packet, the event packet, the ACL data packet, and the SCO data packet. The command packets carry the HCI commands and their parameters from HCI drivers to Bluetooth hardware. The event packets are used by the Bluetooth hardware to notify the HCI driver when the events occur. The ACL and SCO data packets are used to carry the ACL and SCO data, respectively. Figure 17 shows class hierarchies for handling HCI packets in *JavaBT*.

When the transport base layer receives the incoming data, it transports the byte streams to the

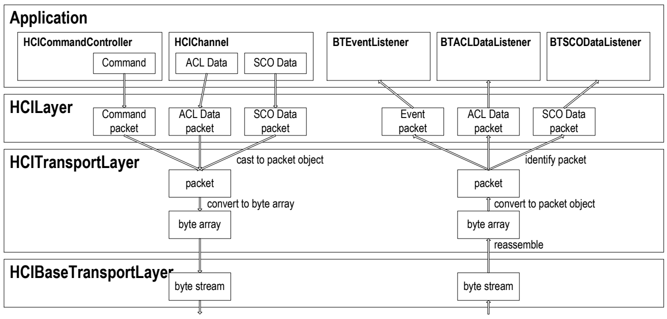


Fig. 18. Data flow in an HCI layer.

transport layer. The transport layer reassembles the incoming byte streams, and converts them into an HCI packet, after which it identifies the message type for dispatching them to the handler. When a program sends out HCI commands using the HCI command controller or sends out ACL and SCO data packets, the HCI layer casts the outgoing packet object into the HCIPacket object and sends it to the transport layer. The transport layer converts the HCI packets into a byte stream and transports it out through the transport base layer. The data flow in the HCI layer is shown in Figure 18. Finally, we define a *HCITransportLayer* class to support the replaceable physical bus in the HCI layer. To create a custom *HCITransportLayer* object, we create a class extended from *HCITransportBaseLayer* and implement the *init* and *sendData* methods. The *init* method is invoked while the transport layer starts up, and the *sendData* method is invoked by the transport layer when it wants to send a packet to the Bluetooth hardware.

2) The L2CAP Layer: The L2CAP layer provides connection-oriented and connection-less data services to upper-layer protocols with a protocol-multiplexing capability. The L2CAP can transmit and receive L2CAP data packets up to 64 kilobytes in length. The L2CAP is based around the concept of "channels". Each L2CAP channel is referred to by a CID, which is a 16-bit number: CID 0x0001 is used by the signaling channel, CID 0x0002 is used by the connection-less reception channel, CIDs 0x0003-0x003F are reserved for future use, and CIDs 0x0040-0xFFFF are available.

Figure 19 shows our design of the L2CAP layers. In this figure, the L2CAP service class provides the ability to send the L2CAP signaling commands and the protocol-multiplexing functionality of the L2CAP layer. The signaling commands for the L2CAP layer, such as L2CA_ConnectReq, L2CA_DataWrite, and L2CA_DataRead, are packed into the *L2CAPCommandPacket*

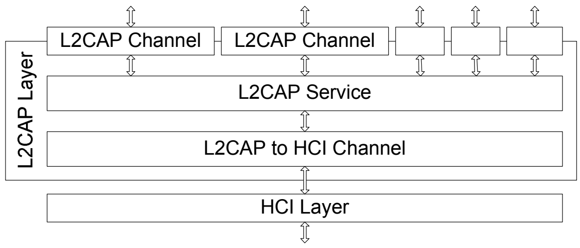


Fig. 19. Overview of L2CAP layers in *JavaBT*.

object and sent to the command channel. When the L2CAP service receives a connection or disconnection request, the L2CAP event occurs. We use an *L2CAPEventIndicationInterface* interface to listen for these L2CAP events.

When we connect to a remote Bluetooth device through the L2CAP service, the L2CAP server creates an L2CAP channel and returns an object representing this channel. Each such object has a unique L2CAP CID. We can send or receive data by calling the methods of the L2CAP channel class. The data are sent or received by the L2CAP service through a channel between the HCI and L2CAP layers. This channel is called the L2CAP-to-HCI channel, and it merges the communications that come from different L2CAP services and send them to a single HCI channel.