Support and Optimization of Java RMI over Bluetooth Environments

Pu-Chen Wei Chung-Hsin Chen Cheng-Wei Chen * Jenq-Kuen Lee †
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan 300

ABSTRACT

Distributed object-oriented platforms over wireless environments for high-performance computing and collaborative frameworks have become essential components for pervasive computings. In this paper, we investigate the issues to support Java RMI over Bluetooth environments. First, we develop a set of protocol stack layers for Bluetooth. The system is written in Java and known as *JavaBT*. Next, we provide support in L2CAP layer with sockets to support RMI socket. This enables the support of Java RMI over Bluetooth environments with efficiency. In addition, we model the cost for the access patterns of Java RMI communications. The cost model is used to guide the formation and optimizations of the scatternets of Bluetooth environments associated Java RMI environments. In our approach, we first employ a well-known BTCP algorithm to observe initial configurations for the number of piconets. With the communication access cost, we then employ a spectral-bisection method to cluster the nodes in a piconet. We then give a method to connect the cluster together based on the cost model. We give experimental results with the prototypes of Java RMI support over Bluetooth environments. Experimental results also show that our scatternet formation algorithm incorporating access cost model out-perform the one without considering the access graphs of Java RMI over Bluetooth environments.

Keywords

High-Performance Computing, Wireless Computing, Distributed Object-Oriented Computing, Bluetooth Environments, Graph Partitioning

*Authors' email address are
{pcwey, jschen, cwchen}@pllab.cs.nthu.edu.tw.

†The correspondence author of this work is Jenq-Kuen Lee. His e-mail address is jklee@cs.nthu.edu.tw, FAX umber is 886-3-5723694, and phone number is 886-3-5715131 Ext. 3519.

1. INTRODUCTION

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

Bluetooth is an emerging technology for short-range, low power wireless applications. The original purpose of Bluetooth proposed by Ericsson is used to replace cables with RF transmissions and can be applied on cellular phones, PDAs, digital cameras, etc. Even through natural limits on data transmission rate, it still achieves up to 723k bps data rate and is sufficient to support most of the ad hoc applications. Several piconets can be joined together to form a scatternet. However, inter-piconet communications are expensive, and we can not hold too many nodes in the same piconet due to physical limits. In a specified area, nodes can be configured to various connected network which contains some masters and bridges. In addition, the communication patterns of RMI APIs of a Java program deeply affect the run-time performance in an ad hoc network. This should be taken into account in configuring scatternets of Bluetooth environments.

In this paper, we investigate the issues to support Java RMI over Bluetooth environments. Our supports include several technical items. First, we develop a set of protocol stack layers written in Java for Bluetooth support, called *JavaBT*. In *JavaBT*, the HCI layer provides a uniform interface of accessing the Bluetooth hardware capabilities. The L2CAP provides connection-oriented and connectionless data services to upper layer protocols with protocol multiplexing capability, segmentation and reassembly operation, and group abstractions. These two layers of protocol driver can help programmers to write Bluetooth applications in the Java programming language. Next, we provide supports in L2CAP layer with socket for RMI socket. This enables the support of Java RMI over Bluetooth environments with efficiency. In addition, we model the cost for the access patterns of Java RMI communications. The cost model is used to guide the formation and optimizations of the scatternets of Bluetooth environments associated Java RMI environments. In our approach, we first perform a method for initial configurations for the number of piconets. This is based on well-known BTCP [18] algorithm. However, this does not take the access patterns of RMI communications into considerations. In our method, we provide a cost model for access patterns of RMI communications. With the support of runtime profiling in Java, we can then dynamically re-configure the scatternets to optimize the Java RMI performance over scatternets. This is particularly useful for systems with periodical behaviors or system behaviors with similarity related to historical or profiling data. With the cost model for access patterns of RMI communications, we then perform a two step algorithm. In the first stage, we employ a recursive spectral bisection method and KL-refinement procedure to cluster nodes making the frequent transmissions. After this stage, a post-configuring method is used to determine the role of nodes and form the scatternet topology.

We give experimental results to show that we have a prototype model of Java RMI support over Bluetooth environments. In our experimental testbed, our implementation of Bluetooth protocol stack for Java platform, *JavaBT*, is tested with a pair of Ericsson Bluetooth Develop Kit (EBDK) connected to the personal computer. The EBDK is a Bluetooth hardware and software development board, and we connect the EBDK and the computer in the testbed. The current implementation of *JavaBT* is implemented with JDK 1.1.8 platform with JavaCOMM API 2.0 for Windows. We perform experiments of numerous benchmarks from RMI Benchmark Suite [17], DHPC Java Benchmarks [16], and the Java Grande Forum MPJ Benchmarks [10]. The last benchmarks are based on MPJ, which is an MPI-like message-passing interface for Java. We have implemented an MPJ interface that follows mpiJava 1.2 specification [5]. Besides, it uses RMI as the underlying communication channel to evaluate the performance of our RMI over Bluetooth implementation. In addition, we also experiment for the scalability of scatternet formation by considering the access cost patterns of RMI methods by incorporating our proposed mechanisms. Experimental results show that our scatternet formation algorithm incorporating access cost model outperform the one without considering the access graphs of Java RMI over Bluetooth environments. This work is also a part of research excellence projects of our university and of our research efforts to develop and investigate the research issues to advance the technologies for distributed component architectures [6, 7, 8, 14, 15, 19, 21].

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 presents *JavaBT*, a protocol stack written in Java for Bluetooth environments. The system is developed in our site. Next, Section 4 gives the cost model for the access patterns of RMI communications. Section 5 presents our optimization schemes to forming scatternets based on our cost model. The experimental results are then given in Section 6. Finally, Section 7 gives related work and conclusion.

2. SUPPORTING JAVA RMI OVER BLUETOOTH ENVIRONMENTS

2.1 Support Java RMI over Bluetooth Layers

In this section, we present our methodologies in supporting Java RMI over Bluetooth environments. In our research framework, we have implemented two set of software infrastructures. First, we implement a set of layers for Bluetooth protocol stacks in Java. The software is called *JavaBT*. Second, we then implement the Java RMI layer on top of our software with Bluetooth protocol stacks. Traditionally, the Java RMI implementation in the *Sun*® JDK is running over the TCP/IP network architecture. However, the current TCP/IP support in Bluetooth needs to go through several protocol layers (see Figure 1a). It will downgrade the communication performance and spend additional resources. One can run the TCP/IP over L2CAP layer to ease this problem (see Figure 1b).

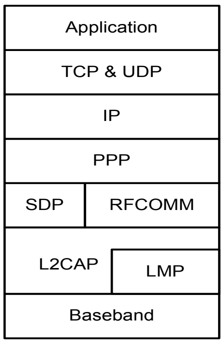

Figure 1a: The current TCP/IP support in the Bluetooth protocol stack.

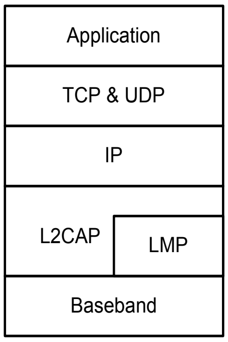

Figure 1b: The more efficient TCP/IP support in the Bluetooth protocol stack.

With the above mechanism, we can make Java environment over the Bluetooth wireless network by running the Java RMI on the TCP/IP over Bluetooth wireless network. This will still need the TCP/IP support of the Bluetooth protocol stack (see Figure 2a). In our research software framework, we have the Java RMI over the L2CAP layer directly to reduce the protocol stack implementations overhead and to improve the performance (see Figure 2b).

We develop a custom Java RMI library that uses Bluetooth L2CAP layer instead of TCP/IP Socket. We create a custom RMI socket factory to transfer data by Bluetooth L2CAP layer. Under the Java RMI layer, we also develop the Java Bluetooth protocol stack included L2CAP, HCI layers, and a replaceable transport layer to communicate with the Bluetooth hardware through Java Native Interface (JNI), Java communications API and other specific interfaces.

2.2 The Bluetooth Socket for Java RMI

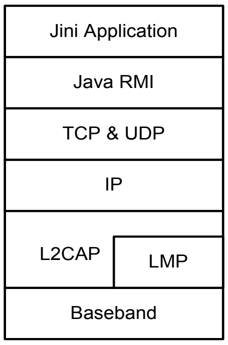

Figure 2a: Run Java RMI over the TCP/IP support in the Bluetooth protocol stack.

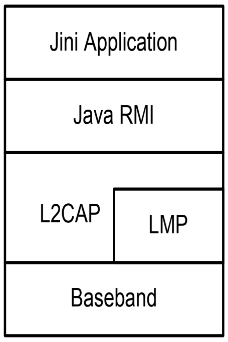

Figure 2b: Run Java RMI over the Bluetooth L2CAP layer directly.

We build a family of classes to provide functionalities for the Java Socket over our Bluetooth driver. These classes are shown in Figure 3. 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* interface to receive the L2CAP events. Detailed functions for *BluetoothServerSocket* class are given below.

- **To listen the connection request**

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

- **To accept the connection request**

  BluetoothServerSocket overrides the *accept* method of *java.net.ServerSocket* to accept the connection from the *BluetoothSocket*. *BluetoothServerSocket* implements the *L2CAPEventIndicationInterface* interface to receive the L2CAP ConnectInd event. After receiving the ConnectInd event, *BluetoothServerSocket* will create a *BluetoothSocket* with the specific L2CAP CID from that event and return it.

- **To retrieve the local Bluetooth address**

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

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

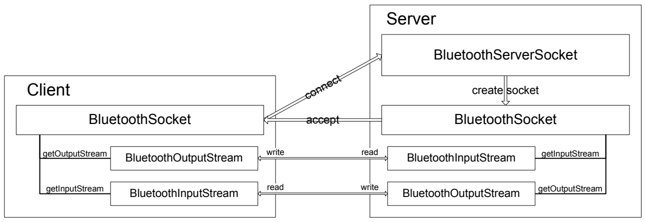

Figure 3: Bluetooth Socket Supports for Java RMI

- **To connect to the remote host**

  BluetoothSocket overrides the constructor methods for initializing the L2CAPService and connecting to the specific Bluetooth address and PSM.

- **To read data from BluetoothSocket**

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

- **To write data to BluetoothSocket**

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

- **To retrieve the local and remote Bluetooth address**

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

Next, we design a *BluetoothInputStream* class as an input stream for reading data from a L2CAPChannel. It extends from *java.io.InputStream*. In addition, a *BluetoothOutputStream* class is implemented as an output stream for writing data to a L2CAPChannel. It extends from *java.io.OutputStream*. Next, we have the *BluetoothInetAddress* class extends from *java.net.InetAddress* to represent the Bluetooth address. The BluetoothInetAddress wraps the *BD_ADDR* and provides the same interface as *InetAddress*.

Finally, to run the RMI over the Bluetooth, we create a class, *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.

3. JAVABT: BLUETOOTH PROTOCOL DRIVER

We use the Java programming language to build a set of layers of the Bluetooth protocol driver. Our system is called *JavaBT*. In this section, we will focus on the support for the host controller interface (HCI) layer and the logical link control and adaptation protocol (L2CAP) layer, as they provide the low level support for our implementation of Java RMI over Bluetooth environments.

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

3.1 The Host Controller Interface Layer

The host controller interface (HCI) layer provides a uniform command method of accessing the Bluetooth hardware capabilities to the higher layers of Bluetooth protocol stack. The HCI driver sends data and commands to Bluetooth hardware by the HCI packets. There are four types of HCI packet, they are the command packet, the event packet, the ACL data packet, and the SCO data packet. The command packets carry the HCI command 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 data packets and SCO data packets are used to carry the ACL and SCO data, respectively. Figure 4 gives class hierarchies for handling HCI packets in *JavaBT*.

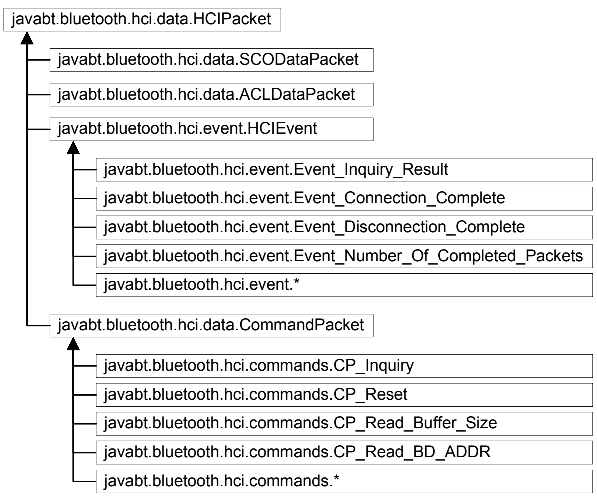

Figure 4: The class hierarchies for handling HCI packets.

When the TransportBaseLayer receives the incoming data, it transports the byte stream to the TransportLayer. The TransportLayer will re-assemble the incoming byte stream, and convert them into the HCIPacket. Then it will identify the message type for dispatching them to the handler. When a program sends out HCI commands by the HCICommandController or send out ACL data packets and SCO data packets, the HCILayer will cast the outgoing packet object into the HCIPacket object and send it to the Transport Layer. The TransportLayer will convert the HCIPackets into a byte stream and transports it out through the TransportBaseLayer. The data flow in HCI layer is shown in Figure 5. Finally, we define a *HCITransportLayer* class to support the replaceable physical bus in HCI layer. To create a custom *HCITransportLayer*, we create a class extended from *HCITransportBaseLayer* and implement the *init* method and *sendData* method. The *init* method is invoked while the TransportLayer starts up. The *sendData* method is invoked by TransportLayer when it wants to send packet to the Bluetooth hardware.

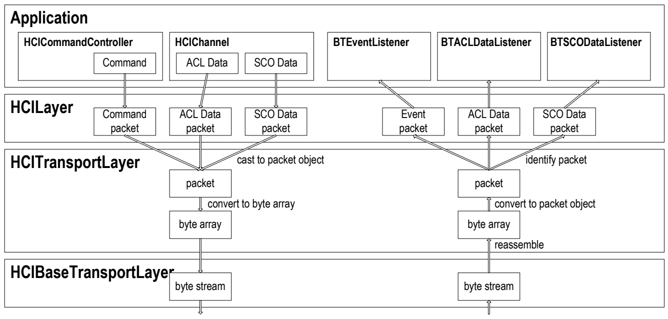

Figure 5: Data flow in a HCI layer.

3.2 The Logical Link Control and Adaptation Protocol Layer

L2CAP layer provides connection-oriented and connectionless data services to upper layer protocols with protocol multiplexing capability. L2CAP can transmit and receive L2CAP data packets up to 64 kilobytes in length. The L2CAP is based around the concept of "channels". Each of the L2CAP channel is referred to by a channel identifier (CID) that is a 16-bits number. CID 0x0001 is used by the signalling channel. CID 0x0002 is used by connection-less reception channel. CID from 0x0003 to 0x003F are reserved for future use and from 0x0040 to 0xFFFF are free to manage.

Figure 6 gives our design of *L2CAP* layers. In Figure 6, the *L2CAPService* class provides the ability to send the L2CAP signalling commands and the protocol multiplexing functionality of L2CAP layer. The signalling commands for L2CAP layer such as L2CA_ConnectReq, L2CA_DataWirte and L2CA_DataRead are packed into the *L2CAPCommandPacket* and sent to the command channel. When the L2CAPService receives the connection request or disconnection request, the L2CAP event will occur. We use a *L2CAPEventIndicationInterface* interface to listen to these L2CAP events.

When we connect to a remote Bluetooth device through the L2CAPService, the L2CAPServer will create a L2CAP

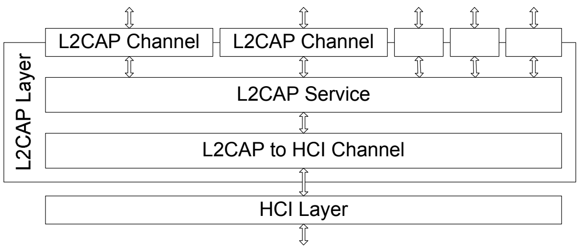

Figure 6: Overview of L2CAP Layers in JavaBT

channel and return a L2CAPChannel object to represent this L2CAP channel. Each *L2CAPChannel* has a unique L2CAP CID. We can send data or receive data by calling the methods of the L2CAPChannel class. The data sent or received by the L2CAPService are through a channel between the HCI layer and L2CAP layer. This channel called L2CAPtoHCIChannel. This channel will merge the communications that come from different L2CAPService and send them to one HCI channel.

4. COST MODEL FOR RMI OVER SCATTERNETS

In the scatternet formation, we have 7 nodes to form a piconet and piconets can then cluster into scatternets. For an object-oriented system over Bluetooth environments, the communications of objects are done via RMI and also on top of Bluetooth environments. The access communication patterns of RMI can be used to help to guide the formation of scatternets. Figure 7 shows an example of packets routing path, which is configured with BTCP Algorithm [18] and is guaranteed to make minimum piconets under completed connections between piconets on constructing a scatternet. In Figure 7, black solid circles denote masters whereas the hollows are slaves. The whole scatternet needs only a bridge node (S14/S22). Supposing that a remote object placed on S21 is invoked by S11 that are at different piconet, the path with triangle marks is the only path, and the weights of edges are data flows (in bytes) conveyed from callers. BTCP is a conventional approach to forming scatternets. BTCP does not consider the access graphs of RMI communication patterns. Figure 8 examines another configuration which considers the access patterns of communications in forming scatternets. Frequently communicated S11 and S21 are kept in the same piconet in reducing communication cost. The one in Figure 8 in this case is better than the one in Figure 7.

In order to forming the scatternets for RMI over Bluetooth environments, we need a cost model to guide the configuration process. We give our cost model below. The Bluetooth scatternet can be represented as a undirected graph G=(V,E). V is a set of nodes of scatternet; E is set of links connecting two nodes, where $e_{i,j}=(v_i,v_j)\in E$. Let $P_{i,j} = (v_i,v_{i+1},..,v_j)$ be a ordered set denoting a path from node v_i to v_j. The cost model of scatternet traffics over RMI is characterized by the following parameters:

(1)A remote method **k** is implemented on node v_j denotes RM_k^j. $F_k^j(i)$ is the number of calls which the caller at node

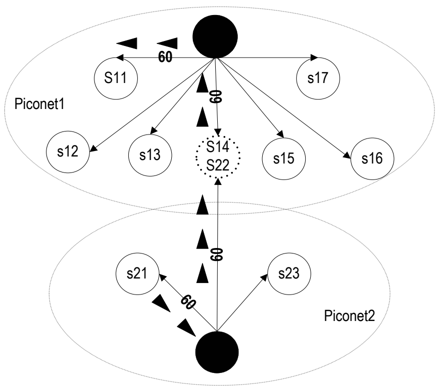

Figure 7: A scatternet with 2 piconets and 11 BT devices

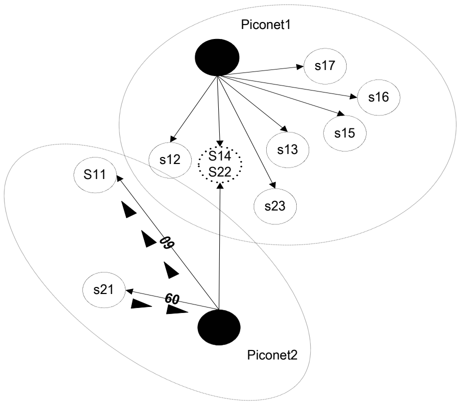

Figure 8: A configuration incorporating the communication patterns.

v_i invokes a remote method on v_j during a specified time periods.

(2)Throughout a call/return pair, caller passed d^{in} parameters (in byte) to transmit and callee returned d^{out} processed data (in byte) during invoking RM_k^j. Node v_i expects the amount of $D_n^i(RM_k^j)=(d^{in}+d^{out})$ bytes via nth calling. $D_n^i(RM_k^j)$ denotes the amount of message traffics when node v_i calls node v_j with RMI method k and in its *nth* call.

(3)Actually, there are six kinds of SCO packets which can be selected in Bluetooth spec to ensure QoS, but we merely use DH1 packet here to simplify the cost model.

Our goal to scatternet traffics over RMI has to satisfy: 1)The cost of inter-piconet communication is very expensive and the most frequently data exchange may be placed on the same piconet whenever possible. 2)The number of members located in each piconet has to be approximately the same.

Let A be an adjacent matrix of G, where $a_{i,j}=1$ if a direct edge linked v_i,v_j exists otherwise $a_{i,j}=0$, $a_{i,j}\in A$. We define the weighted message cost of any two nodes as $c'_{i,j}$ with caller v_i and callee v_j, and C' is a n×n cost matrix for the whole cost matrix. hop(i,j) denotes the number of hops in a path P_{ij}. Therefore,

$$c'_{i,j} = \sum_{k \in RM^j} \sum_{n=1}^{F_k^j(i)} D_n^i(RM_k^j).min(hop(i,j)), \quad (1)$$

where RM^j is the set of methods to be invoked in node v_j. In addition, as the coordinator has the detailed information about all nodes in scatternet, the shortest path min(hop(i,j)) can be found by Floyd's algorithm [9]. C' finds all the unidirectional traffics between node pairs, but the path overloads are what we care about. The new matrix C is made by $c_{i,j} = c'_{i,j} + c'_{j,i}$. This symmetric matrix can be regarded as a completed graph with weighted edges whose values are message costs during specific period between two nodes. This will be used as the cost model to guide the formation of scatternets.

5. ALGORITHMS FOR OPTIMIZING RMI OVER SCATTERNETS

We summarize our scatternet configuration process below.

- Initially, all Bluetooth devices are ready for communication.
- To use BTCP algorithm to form an initial scatternet with a coordinator which knows status of other nodes.
- To coordinate and collect run-time behaviors of programs and access frequency from all nodes.
- If scatternet workload is heavy and some nodes at different piconets communicate frequently, we determine the new clustering for scatternets.
- To connect newly formed clusters.

We now give each step in the sub-sections below.

5.1 Initial Configurations of Scatternets

In the first step of our algorithm, our work adopts BTCP as an initial scatternet formation algorithm. BTCP is a state-of-the-art algorithm in forming Bluetooth scatternets. However, this is a version without considering the access patterns of Java RMI. Once the initial configuration is gotten, we then refine the configuration according to our proposed algorithms in next steps. The BTCP algorithm has the following constraints and properties.

- All nodes must be located on a transmission power-reachable range.
- The maximum degree of a bridge node is 2.
- To prevent from interfering and reduce packet delays, the number of piconets which comprises a resulted scatternet should be minimum.
- There are no other bridge nodes exist on any two piconets.

According to these properties, a Bluetooth topology construction protocol can be done. This algorithm was done by finding a central node called coordinator, allocating minimal piconets, and establishing the whole scatternet. For the sake of completeness, we give the outline of this algorithm in Figure 9.

Input: Bluetooth Devices set S_{bt} where any two nodes with proximity can communicate each other.

Output: If $|S_{bt}| \leq 8$, construct exactly one piconet, a connected scatternet otherwise.

•To generate scatternet coordinator

To maintain a mechanism for time-limited scatternet formation
and each node maintain an local status which is exchanging in election
Do
  confront with random pairs(S_i,S_j)
  the loser S_j is forced into PAGE SCAN state while
  the winner S_i keep continuing contention

Until all the other participants are in PAGE SCAN state

•According to a predefined formula $P = \lceil \frac{17-\sqrt{289-8n}}{2} \rceil$, coordinator allocates P piconets and $\frac{P*(P-1)}{2}$ bridge nodes. Paging the designated nodes and sending them data for specified piconet setting up

•Each master builds individual piconet and connect the other piconets via a predefined bridge node.

Figure 9: The BTCP algorithm

5.2 Scatternet Clustering with Spectral Bisection

With the access communication patterns of RMI, we hope to form scatternets minimizing communication costs. We want to know which nodes should be formed as a piconet and come to a minimal inter-piconet communication. We use a partitioning algorithm which bisects |V| nodes to identical two parts and keep a lower-bound of message cost. The |V|/2 part is treated a piconet. Moreover, using a local refinement method to pick a master of a piconet can minimize intra-piconet traffics. After the two stage, the new scatternet is more efficient and adaptive to complex RMI communications. To find a piconet of the vertex set into two parts V^+ and V^-, $V^+ \cup V^-$=V. We define a vector $X \in R^n$ called indicator vector. If $v_i \in V^+$ then x_i=1 else x_i=-1. Therefore, inter-piconet connection costs between two piconets are function $f(x) = \sum \frac{1}{4}.c_{ij}(x_i - x_j)^2$. We summarized constraints and formulated an objective function as following:

$$Minimize : f(x) = \frac{1}{4}.\sum_{e_{ij}} c_{ij}(x_i - x_j)^2 \quad (2)$$

$$Subject\ \ to : \sum_{i=1}^{n} x_i = 0, x_i = \{1, -1\} \quad (3)$$

Equation 2 makes these two partitions fully equivalent if |V| is even. We also define a diagonal matrix C^d, where $c^d_{i,i} = \sum_{e_{ij}} c_{i,j}$, otherwise $c^d_{i,i}$=0 for i $\neq$ j. Let me inspect our f(x):

$$\sum_{e_{ij}} c_{ij}(x_i - x_j)^2 = \sum_{e_{ij}} c_{ij}.(x_i^2 + x_j^2) - 2.\sum_{e_{ij}} x_i x_j.$$

Following the first theorem of graph theory and formula transformation [20], we have

$$\sum_{e_{ij}} c_{ij}(x_i - x_j)^2 = X^T(C^d - C)X. \quad (4)$$

The matrix L= (C^d-C) is a weighted *Laplacian* matrix and has been proven useful in graph partitioning. Suppose that a scalar λ such that LY=λY. Eigenvector Y is followed by its eigenvalue λ. Combining equation 2 and 4, the original graph partitioning problem is converted to a discrete optimal problem. It exists no practical scheme for solving this since partition problem's NP-hard property. Relaxing constraint (3) above by $\sum_{i=1}^n x_i = 0, x^T.x = n$ in order to change problem domain to the continuous bisection problem. We rewrite our cost function and subjection

$$Minimize : f(x) = \frac{1}{4}.X^T(C^d - C)X \quad (5)$$

$$Subject\ \ to : \sum_{i=1}^n x_i = 0, x^T.x = n \quad (6)$$

The continuous approximation should be mapped to its primitive problem. If the partition is close to 1, -1, the result is near to optimal solution. Some important properties of L has been proven, where we concerns about the second smallest eigenvalue called λ_2 of L whose eigenvector Y_2, is the minimal solution in equation 5. Y_2 is named "algebra connectivity of graph" or "Fielder vector". The algorithm in this stage accompanied with two steps. First is recursive spectral bisection (RSB) for repartitioning all the nodes in original scatternet into *P* clusters, then use KL [12] procedure applied on these unbalanced partitions to locally refine and make about the same size of clusters. The algorithm in Figure 10 integrates RSB and KL to configure scatternet by cost matrix. After processing, we have *P* balanced-size clusters. The algorithm in Figure 10 first performs a recursive spectral-bisection algorithm and then performs K & L algorithm [12] also known as mincut algorithm. The fine-tuning of KL algorithm is used when the number of clusters to be partitioned is not a power of 2 or is not an even number. The last few partitions which are not well-divided will be fine-tuned by KL algorithm to balance the size of clusters.

5.3 Connecting scatternets

In this sections, we focus on post-processing to connect the cluster together after the partition for piconets done in the algorithm of Figure 10. Two stages are designed in this phase. First, we need to determine the role in nodes of clusters and append feasible edge (link) into each node. What must be done is to assign "master" attribute to designated nodes on all clusters. After this procedure, two bipartite sets are determined, master and slave. In the second step, "bridges" must be selected from slave sets to connect the clusters together. Figure 11 gives the main steps for a heuristic approach to perform these two steps. In the first step, the node of which the summation of total interior links is maximum among a cluster P_i is chosen as the master. In the second step, we calculate the communication cost between two piconets, P_i and P_j by using s_k as a bridge and denote it as $t^k_{i,j}$. Once this cost is calculated, the assignments of bridges will be based on this cost. There are several methods in choosing the bridges. The method presented in Figure 11 is following the concept of BTCP algorithms to have a complete graph to connect all the clusters together. In the method, each cluster has its slaves to serve as bridges for other clusters. In this case, a bipartite matching scheme can be used based on the $t^k_{i,j}$ cost calculated earlier to decide the matching between the slaves of a piconet and other piconets. Other methods in choosing the bridges and connections for the partitioned clusters include the method of finding minimum spanning tree. The minimum spanning

Input:A scatternet topology *S* with *n* Bluetooth devices

let P = the number of piconets and n >8
let C = a n∗n matrix of message cost for peer-to-peer communication
let L = Laplacian of C
Output:*P* clusters, where each cluster has either $\lfloor \frac{n}{P} \rfloor$ or $\lfloor \frac{n}{P} \rfloor$+1 nodes
Do /* Recursive Spectral Bisection */
    Find the Fielder vector Y_2 of L
    Compute $V = X.Y_2$
    Let *m* be a median value of elements of vector V

    if (the *i*th element of $X \leq m$)
        put vertex v_i in cluster V^+
    else
        put vertex v_i in cluster V^-
    end if
Until the number of clusters = *P*

/* Refining and balancing *P* clusters */
Best Cluster = current Cluster
let $k \in \{1..P\}$, S_k = vertices in cluster k

for all v_i, compute positive gains for move v_i from current cluster to another clusters

While (Find a better cluster is possible) /* mincut */
    Do
        pick up the largest gain for moving $v_i \in S_m$ to S_n
        $S_m = S_m \setminus v_i$
        $S_n = S_n \cup v_i$

        if all clusters are balanced and current cluster is superior to Best cluster
            Best cluster = current cluster
        end if
        recompute all the gains for movements
    Until no more movements exist
end While

Figure 10: The recursive spectral partitioning with KL refinement

Input:cluster sets P, where P=$\{P_1, ...P_n\}$

H = $| P_i | * | P_j |$ distance matrix (H is the hopping distance matrix between two clusters)

Output: a connected topology T

Do
for each $P_i \in P$, to find the master m_i and put them into the set M.
To assign $(P_i \setminus m_i)$ into slave set S.
Until all $m_i \in P_i$ are generated

For each $m_i \in M$
To assign slave attribute to $j \in (S \cap P_i)$.
To make links on (m_i, j).
End for

For each $s_k \in S$
To select a bridge $s_k \in (P_i \cup P_j)$; recompute H for P_i and P_j after assuming s_k is the bridge for P_i and P_j.
To find the minimum communication cost $t_{i,j}^k$ between P_i and P_j using the newly computed H.
End for

Do for each $P_i \in P$
To apply weighted bipartite matching algorithm to choose the connection cluster for each slave $s_k \in S$.

Constraints in bipartite matching:
it only permits up to 7 links on the same piconet.

Until all the bridges are generated

Figure 11: The algorithm for designating the role of nodes and connecting scatternet

tree approach gives the connectivity, but reduces the energy consumptions and collisions.

6. EXPERIMENTAL RESULTS

We perform three experiments. In the first experiment, we evaluate the robustness of our implementation of RMI software over Bluetooth environments. We perform experiments of numerous benchmarks from RMI Benchmark Suite [17], DHPC Java Benchmarks [16], and the Java Grande Forum MPJ Benchmarks [10]. The last benchmarks are based on MPJ, which is an MPI-like message-passing interface for Java. We have implemented an MPJ interface that follows mpiJava 1.2 specification [5]. It uses RMI as the underlying communication channel to evaluate the performance of our RMI over Bluetooth implementation. Table 1 gives the applications that our RMI successfully run through it in our early test for robustness of our software.

In our testbed, the EBDKs (Ericsson Bluetooth Develop Kit) are connected to two computers by the COM port. We test the performance on one node and two nodes, respectively. The one node version is a sequential version and the two node version is a parallel version with RMI over Bluetooth. The computer that the RMI server runs on has one AMD 1200MHz CPU, 768MB of Memory and is running the Microsoft Windows 2000 Server. The RMI client runs on a IBM ThinkPAD notebook that has Intel Pentium III 700MHz CPU, 256MB of SDRAM and is running the Microsoft Windows XP Professional. Our *JavaBT* Bluetooth protocol driver and the test applications are running on Microsoft JavaVM that supports JDK 1.1.8 with additional RMI support and Sun Java Communications API v2.0 for Windows. As we will show later in our next experiments, our testbed with EBDKs are limited by the COM port performance. Note that our software of protocol stacks can be used for other Bluetooth environments as well. Despite of the limitations of COM port in our experiments, we still observe performance gains on Hamming, EP, Series, RayTracer, and SelSort. The performance is shown in Figure 12. Note that due to the server and client are different machines, the optimal speedup for this case is around 1.7 times. With better connections between host and Bluetooth devices, we expect the performance to be improved. 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. Note that our support with RMI over Bluetooth also gives more high-level control with Bluetooth 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. Figure 13 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. If the limitation of COM port performance in our testbed can be removed, the performance will be better.

In the second experiment, we evaluate the bandwidth and 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. Figure 14 gives bandwidth for these four layers. As mentioned earlier that the EBDK boards are limited with the COM port between Bluetooth device and the host computer. This bandwidth experiment does give the needed information. As we are deploying our *JavaBT* on Penbex PDA and OS (a local firm in Taiwan) with the CF connections and Bluetooth devices, we hope to get more accurate bandwidth information at that time. We still manage to get useful information during our experiments. Figure 15 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. There are the comparisons for the additional data being sent for different layers in our current implementation. Most overheads are observed in the RMI version due to serializations of RMI.

Finally, we experiment the scatternet formation algorithms exploiting the communication patterns of RMI over Bluetooth environments. The experiments are done by simulations with our cost model. Normal distributions are used

Table 1: The detailed descriptions of benchmarks.

Name	Description	Data Size	From
EP	NAS EP generating Gaussian random numbers	16,777,216	[16]
Series	Fourier coefficient analysis	10,000	[10]
LUFact	LU Factorisation	500	[10]
SOR	Successive over-relaxation	1,000	[10]
Crypt	IDEA encryption	3,000,000	[10]
Sparse	Sparse Matrix multiplication	50,000	[10]
MolDyn	Molecular Dynamics simulation	2,048	[10]
MonteCarlo	Monte Carlo simulation	2,000	[10]
RayTracer	3D Ray Tracer	150	[10]
SelSort	Selection Sort	524,288	[13]
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	[17]

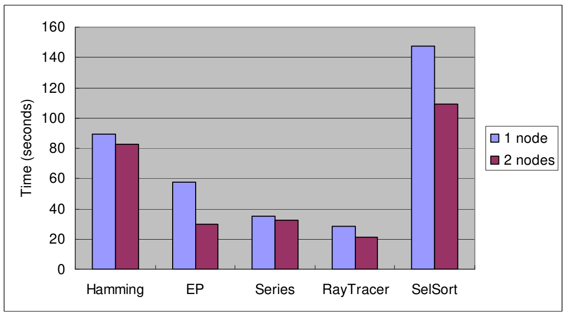

Figure 12: The performance results of some benchmark applications.

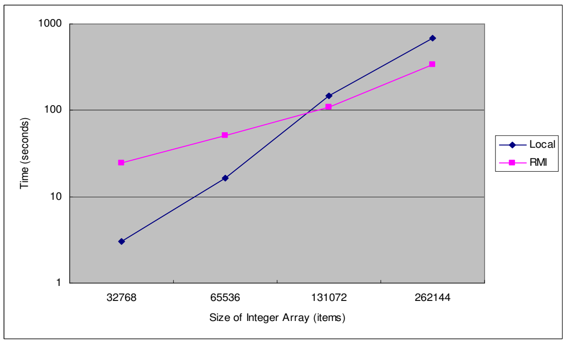

Figure 13: SelSort with local computation and a 2-node version based on RMI over Bluetooth communications.

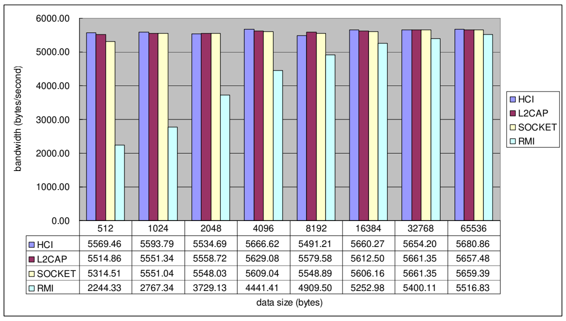

Figure 14: The bandwidth comparison for different layers.

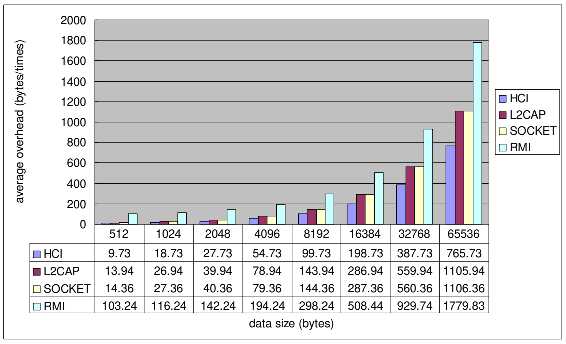

Figure 15: The overhead comparison for different layers. Additional data sent at different layers are observed.

for the needed parameters. The cost in the experiment is defined from equation 1. Although matrix C is symmetric, the total cost can be only the summation of the upper right part on matrix C. To compute total cost, we need a hop cost H, end-to-end call frequency F, and D for data size of passing in and returning. H is depend on initial topology. We use random number generator to produce values for F and D. Figure 16 shows the performance improvements over the initial configurations (BTCP algorithm) without considering the access patterns of RMI executions for the specified parameter distribution in the figure. Around 40% improvements are observed in this case.

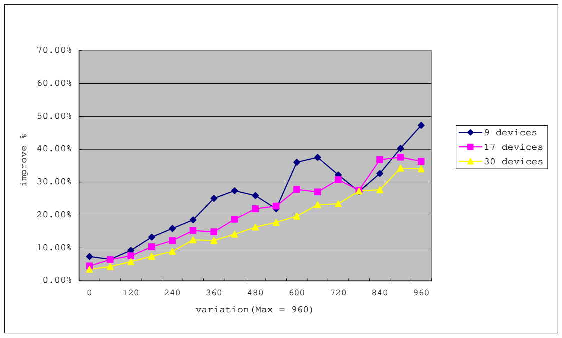

Figure 16: Communication patterns using normal distribution with $\mu = 300, \sigma = 0 - 960$

7. RELATED WORK AND CONCLUSION

Efficient supports for Java remote method invocation have been important topics for investigations. Research results include a better way to implement RMI [17] and the broad range of RMI application [3]. Other Bluetooth protocol stack are described as follows. There is a Java environment from Zucotto [22]. IBM provides a Bluetooth protocol driver for the Linux operating system, called BlueDrekar [11]. And the OpenBT project [1] that comes from Axis is an open source project to build a Bluetooth protocol driver for the Linux operating system. In this work, we report a first-hand experience with Java implementation of Bluetooth protocol stacks, the optimization issues of RMI, and experimental results with several key benchmarks. This work is also a part of research excellence projects of our university and of our research efforts to develop and investigate the research issues to advance the technologies for distributed component architectures [6, 7, 8, 14, 15, 19, 21].

8. REFERENCES

[1] Axis Communications. *An open source Bluetooth protocol stack for Linux.* http://java.sun.com/products/javacomm/.

[2] 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.

[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] B. S. I. Group. *Specification of the Bluetooth System version 1.1*, volume 1, 2, and profiles. February 2001.

[5] 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.

[6] R. G. Chang, C. W. Chen, T. R. Chuang, and J. K. Lee. Toward Automatic Support of Parallel Sparse Computation in Java with Continuous Compilations. *Concurrency: Practice and Experience*, 1997.

[7] C. W. Chen, C. K. Chen, and J. K. Lee. Runtime Compositions and Optimizations of Parallel JavaBean Programs on Clustering Environments. In *Proceedings of PDPTA'00*, June 2000.

[8] 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.

[9] T. B. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

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

[11] IBM. *A Bluetooth protocol stack for Linux.* http://www.alphaworks.ibm.com/tech/bluedrekar/.

[12] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, pages 291–307, 1970.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] T. Salonidis, P. Bhagwat, L. Tassiulas, and R. LaMaire. Distributed topology construction of Bluetooth personal area networks. In *Proceedings of the 20th annual Joint Conference of the IEEE Computer and Communications Societies*, 2001.

[19] Y. L. Shieh and Jenq Kuen Lee. Implementation of Vortex Filament Methods on Parallel Machines with Distributed Adaptive Data Structure. *International Journal of Numerical Methods in Fluids* VOL. 24, pp. 939-951, 1997.

[20] D. B. West. *Introduction to graph theory.* Prentice Hall, 2001.

[21] Jian-Zhi Wu and Jenq Kuen Lee. A bytecode optimizer to engineer bytecodes for performances. In *Proceedings of the 13th International Workshop on Parallel Computing (LCPC)*, August 2000.

[22] Zucotto Wireless Corp. *XJB 100 Bluetooth Host Stack.* http://www.zucotto.com/.