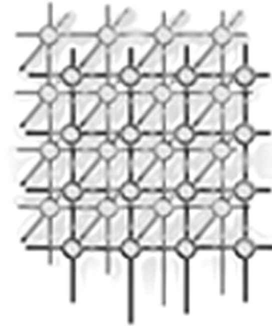
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 00; **00**:1–21 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Support and optimization of Java RMI over a Bluetooth environment



Pu-Chen Wei, Chung-Hsin Chen, Cheng-Wei Chen, Jenq-Kuen Lee*,†

Department of Computer Science, National Tsing Hua University, HsinChu 30055, Taiwan

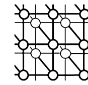
SUMMARY

Distributed object-oriented platforms are increasingly important over wireless environments for providing frameworks for collaborative computations and for managing a large pool of distributed resources. Due to limited bandwidths and heterogeneous architectures of wireless devices, studies are needed into supporting object-oriented frameworks over heterogeneous wireless environments and optimizing system performance. In our research work, we are working towards efficiently supporting object-oriented environments over heterogeneous wireless environments. In this paper, we report the issues and our research results related to the efficient support of Java RMI over a Bluetooth environment. In our work, we first implement support for Java RMI over Bluetooth protocol stacks, by incorporating a set of protocol stack layers for Bluetooth developed by us (which we call *JavaBT*) and by supporting the L2CAP layer with sockets that support the RMI socket. In addition, we model the cost for the access patterns of Java RMI communications. This cost model is used to guide the formation and optimizations of the scatternets of a Java RMI Bluetooth environment. In our approach, we employ the well-known BTCP algorithm to observe initial configurations for the number of piconets. Using the communication-access cost as a criterion, we then employ a spectral-bisection method to cluster the nodes in a piconet and then use a bipartite matching scheme to form the scatternet. Experimental results with the prototypes of Java RMI support over a Bluetooth environment show that our scatternet-formation algorithm incorporating an access-cost model can further optimize the performances of such as system.

KEY WORDS: Distributed Object-Oriented Computing; Wireless Computing; Java RMI; Bluetooth Architectures; Collaborative Computing

---

*Correspondence to: Department of Computer Science, National Tsing Hua University, HsinChu 30055, Taiwan
†E-mail: jklee@pllab.cs.nthu.edu.tw
Contract/grant sponsor: The work was supported in part by NSC-90-2218-E-007-042, NSC-90-2213-E-007-074, NSC-90-2213-E-007-075, MOE research excellent project under grant no. 89-E-FA04-1-4, and MOEA research project under grant no. 91-EC-17-A-03-S1-0002 of Taiwan.



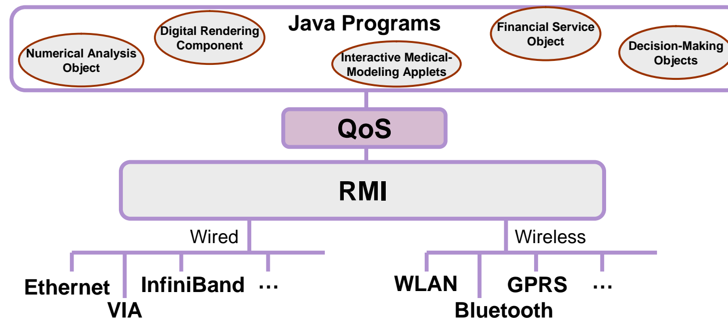


Figure 1. Java RMI over heterogeneous networks.

1. Introduction

Distributed object-oriented platforms operating over wireless environments have become important components for distributed computing and collaborative software frameworks. Among distributed object-oriented software, Java RMI is one of the key methods for performing distributed computing in Java environments. For example, Jini network technology (based on Java RMI) provides an administration-less network environment over which devices can collaborate together via remote service, service discovery, etc. The Java RMI protocols supported 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 use asynchronous thread invocations of RMI to the remote site, including wirelessly. With the increasing number of small devices in computing environments, the software support, scalability, and optimization issues of such environments must be considered. Computation-intensive tasks can be performed on mobile devices, with limited computing power, by communicating with servers via RMI. As illustrated in Figure 1, the RMI services can be implemented on heterogeneous networks, either wired or wireless. Java proxy supports and exception handling can also be employed to support Java RMI over heterogeneous network environments; optimization is then based on the connectivity assignments among networks and the quality of service. We have developed several key technologies in our research efforts to develop a framework for supporting heterogeneous networks. Supporting Java RMI over heterogeneous wireless environments requires components in each wireless environment and the provision of roaming capabilities in the RMI layers over them. In this paper, we report a method for efficiently using Java RMI over a Bluetooth environment.

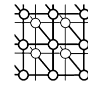
Bluetooth is an emerging technology for short-range, low-power wireless applications. The original purpose of Bluetooth as proposed by Ericsson was to replace cables with RF transmissions, with applications such as cellular phones, PDAs, and digital cameras. Physical limits on the data transmission rate means that it achieves data rates up to 723 kbps, which is sufficient for many applications. Several piconets can be combined into a scatternet. However, interpiconet communications are expensive, and there are physical limits to the number of nodes within the same

piconet. In a specified area, nodes can be configured as connected networks containing masters and bridges. In addition, the communication patterns via RMI of a Java program significantly affect the runtime performance in an ad hoc network, and this should be taken into account when configuring scatternets using Bluetooth.

In this paper, we investigate the issues underlying the support of Java RMI over a Bluetooth environment. Our supports include several key technologies. First, a set of protocol stack layers written in Java for Bluetooth – which we call *JavaBT* – was developed, in which the host controller interface (HCI) layer provides a uniform interface for accessing Bluetooth hardware capabilities. The logical link control and adaptation protocol (L2CAP) provides connection-oriented and connection-less data services to upper layer protocols with protocol multiplexing capability, segmentation and reassembly operations, and group abstractions. These two layers of protocol drivers can help programmers to write Bluetooth applications in the Java programming language. Next, we provide a socket in the L2CAP layer for the RMI socket, which provides the support for Java RMI over Bluetooth. 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 a Bluetooth environment associated with Java RMI environments. In our approach, we first initially configure the number of piconets based on the well-known BTCP algorithm [20]. However, this method does not take the access patterns of RMI communications into consideration. In our method, we use a cost model for access patterns of RMI communications. With the support of runtime profiling in Java, we can then dynamically reconfigure the scatternets to optimize the Java RMI performance over them. This is particularly useful for systems with periodic behavior or other system behaviors related to historical or profiling data. With the cost model for access patterns of RMI communications, we then execute 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 postconfiguring method employing a bipartite matching scheme is used to determine the role of nodes and hence form an optimal scatternet topology.

We report experimental results that demonstrate that our prototype model supports Java RMI over a Bluetooth environment. In our experimental test bed, our implementation of the Bluetooth protocol stack for Java platforms, *JavaBT*, is tested with a pair of Ericsson Bluetooth Development Kits (EBDKs) connected to the personal computer. The EBDK is a Bluetooth hardware and software development board, and we connect the EBDK to the computer in the test bed. The current implementation of *JavaBT* is implemented with the JDK 1.1.8 platform with JavaCOMM API 2.0 for Microsoft Windows. We performed experiments using numerous benchmarks: the RMI benchmark suite [17], the DHPC Java benchmarks [15], and the Java Grande Forum MPJ benchmarks [6]. In addition, we tested the scalability of scatternet formation by considering the access-cost patterns of RMI methods by incorporating our proposed mechanisms. The experimental results show that our scatternet-formation algorithm incorporating an access-cost model outperforms the one that does not consider the access graphs of Java RMI over a Bluetooth environment. This work also forms part of research excellence projects of our university and of our research efforts to develop and investigate technologies for distributed-component architectures [4, 12, 13].

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 a Bluetooth environment. Section 3 provides the cost model for the access patterns of RMI communications. Section 4 presents our optimization schemes to form scatternets based on our cost model. The experimental results are then given in Section 5. Section



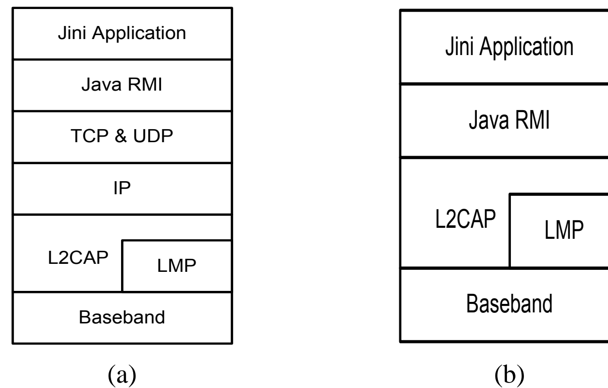


Figure 2. (a) Running Java RMI over TCP/IP in the Bluetooth protocol stack. (b) Running Java RMI over the Bluetooth L2CAP layer directly.

6 describes related work and presents a discussion. Finally, Section 7 concludes this paper. Appendix A presents *JavaBT*, a protocol stack written in Java for the Bluetooth environment.

2. Supporting Java RMI over a Bluetooth Environment

2.1. Support for Java RMI over Bluetooth Layers

In this section, we present our methodology for supporting Java RMI over a Bluetooth environment. In our research framework, we have implemented two sets of software infrastructure: we first implement a set of layers for Bluetooth protocol stacks in Java – the software is called *JavaBT*, and 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 a TCP/IP network. However, the current TCP/IP support in Bluetooth passes through several protocol layers, including baseband, L2CAP, SDP/RFCOMM, PPP, IP, and TCP. This degrades the communication performance and require additional resources. However, this problem can be reduced by running the TCP/IP over an L2CAP layer.

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 2(a)). In our research software framework, we have the Java RMI over the L2CAP layer directly to reduce the protocol stack implementations overhead and to improve the performance (see Figure 2(b)). This reduces the number of layers in the protocol stack as well as the well-known overhead of TCP/IP layers.

We developed a custom Java RMI library that uses the Bluetooth L2CAP layer instead of a TCP/IP socket. A custom RMI socket factory is used to transfer data in the Bluetooth L2CAP layer. Under

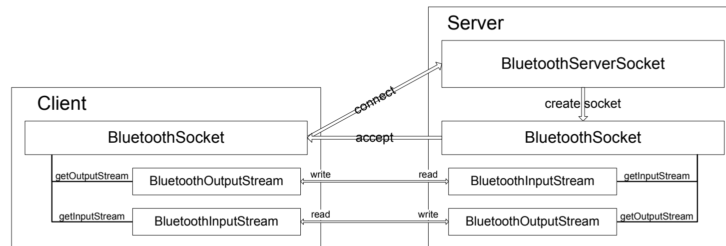


Figure 3. Support for Bluetooth sockets for Java RMI.

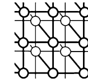
the Java RMI layer, we also develop a Java Bluetooth protocol stack including an L2CAP layer, an HCI layer, and a replaceable transport layer to communicate with the Bluetooth hardware through Java Native Interface.

2.2. The Bluetooth Socket for Java RMI

We build a family of classes to provide functionality for the Java socket over our Bluetooth driver (Figure 3): *BluetoothServerSocket*, *BluetoothSocket*, *BluetoothInputStream*, *BluetoothOututStream*, and *BluetoothInetAddress*. Our *BluetoothServerSocket* class extends the *java.net.ServerSocket* to provide the same interface as the original *ServerSocket* in one that listens to the L2CAP layer. It also implements the *L2CAPEventIndicationInterface* to receive L2CAP events. The functions of the *BluetoothServerSocket* class are as follows:

1. **Listen to the connection request.** *BluetoothServerSocket* overloads the constructor methods for initializing the L2CAP service and waiting for connections. After creating an instance of the *BluetoothServerSocket* class, it listens to a special Bluetooth L2CAP PSM and waits for the *BluetoothSocket*'s connection requests.
2. **Accept the connection request.** *BluetoothServerSocket* overrides the *accept* method of *java.net.ServerSocket* to accept the connection from *BluetoothSocket*. *BluetoothServerSocket* implements *L2CAPEventIndicationInterface* to receive the L2CAP *ConnectInd* event, after which *BluetoothServerSocket* creates a *BluetoothSocket* with the specific L2CAP channel identifier (CID) from that event and returns it.
3. **Retrieve the local Bluetooth address.** *BluetoothServerSocket* overrides the *getInetAddress* method of *java.net.ServerSocket*. In the original *ServerSocket*, this method returns the local address; but in the *BluetoothServerSocket*, it returns a *BD_ADDR* object that represents the Bluetooth address of the local Bluetooth device by calling *getLocalBDADDR* of the L2CAP service.

Next, our *BluetoothSocket* class extends *java.net.Socket* to provide the same interface as the original socket but whilst transmitting data over the L2CAP layer. The functions of the *BluetoothSocket* class are as follows:



1. **Connect to the remote host.** *BluetoothSocket* overrides the constructor methods for initializing the L2CAP service and connecting to the specific Bluetooth address and PSM.
2. **Read data from *BluetoothSocket*.** *BluetoothSocket* overrides the *getInputStream* method of *java.net.Socket*. This method returns an instance of *BluetoothInputStream* that reads data from the L2CAP channel.
3. **Write data to *BluetoothSocket*.** *BluetoothSocket* overrides the *getOutputStream* method of *java.net.Socket*. This method returns an instance of *BluetoothOutputStream* that writes data to the L2CAP channel.
4. **Retrieve the local and remote Bluetooth address.** *BluetoothSocket* overrides the *getLocalAddress* and *getInetAddress* methods of *java.net.Socket*. In the original socket, the *getLocalAddress* returns the local address; but in *BluetoothSocket*, it returns a *BD_ADDR* object that represents the Bluetooth address of the local Bluetooth device by calling the *getLocalBDADDR* method of the L2CAP service. The *getInetAddress* method of *BluetoothSocket* returns the remote Bluetooth address by calling *getRemoteBDADDR* of the L2CAP service.

Next, we design a *BluetoothInputStream* class as an input stream for reading data from an L2CAP channel – this is an extension of *java.io.InputStream*. In addition, a *BluetoothOutputStream* class is implemented as an output stream for writing data to an L2CAP channel. – this is an extension of *java.io.OutputStream*. Next, we have the *BluetoothInetAddress* class extending *java.net.InetAddress* to represent the Bluetooth address. *BluetoothInetAddress* wraps the *BD_ADDR* and provides the same interface as *InetAddress*.

Finally, to run the RMI over Bluetooth, we create a class, *BluetoothRMISocketFactory*, that extends *java.rmi.server.RMISocketFactory*, and implements the *createServerSocket* and *createSocket* methods. The *createServerSocket* method creates a *BluetoothServerSocket* with a specific port, and returns it; the *createSocket* method creates a *BluetoothSocket* that connects to a specific Bluetooth address with a specific port, and returns it. To use our *BluetoothRMISocketFactory* class in the RMI, we call the static method, *setSocketFactory* of the *RMISocketFactory* class, and create an instance of *BluetoothRMISocketFactory* for the parameter.

For clarity of the main text, the technical details of *JavaBT* – the set of protocol stacks supported by us – are provided in Appendix A.

3. Cost Model for RMI over Scatternets

In the scatternet formation, we have seven nodes forming each piconet which can then be clustered into scatternets. For a Java or Jini system over a Bluetooth environment, objects communicate mainly via RMI and also on top of Bluetooth. The access communication patterns of RMI can be used to help guide the formation of scatternets. Figure 4 shows an example of a packet routing path, which is configured with the BTCP algorithm [20] and is guaranteed to minimize the piconets with complete connections when constructing a scatternet. In Figure 4, filled and open circles denote masters and slaves, respectively. The whole scatternet requires only a single bridge node (S14/S22). Suppose that a remote object placed on S21 is invoked by S11 which is at a different piconet. Assume that the path marked by triangles is the path, and the weights of edges are data flows (in bytes) conveyed from

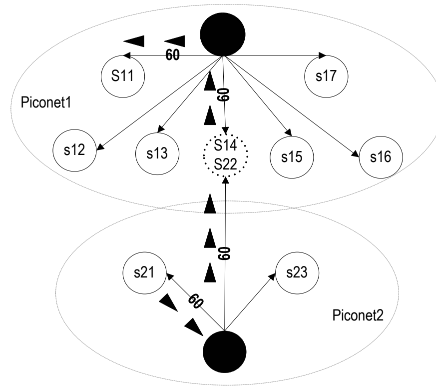


Figure 4. A scatternet with 2 piconets and 11 BT devices.

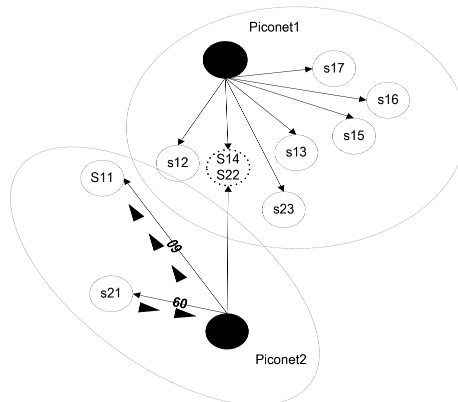
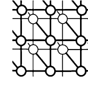


Figure 5. A configuration that takes the communication patterns into consideration.

callers. BTCP is a conventional approach to form scatternets; it does not consider the access graphs of RMI communication patterns. Figure 5 examines another configuration, which considers the access patterns of communications in forming scatternets. Frequently communicated notes S11 and S21 are kept in the same piconet in order to reduce the communication cost. In terms of communication cost, in this case the configuration in Figure 5 is superior to that shown in Figure 4.

A cost model is used to guide the formation of the scatternets for RMI over the Bluetooth environment. We describe 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)∈E. Let $P_{i,j}$ = (v_i,v_{i+1},..,v_j) be a ordered set denoting a path from node v_i to node v_j. The cost model of scatternet traffics over RMI is characterized by the following parameters:

1. A remote method k implemented on node v_j denotes RM^j_k. F^j_k(i) is the number of calls for which the caller at node v_i invokes a remote method k on v_j during a certain time period.
2. $D^i_n(RM^j_k)$ denote the amount of message traffics when node v_i calls node v_j with RMI method k in its nth call. Throughout a call/return pair, the caller passes d^{in} parameters (in byte) during transmission and the callee returns d^{out} processed data (in bytes) whilst invoking RM^j_k. We then have $D^i_n(RM^j_k)$=(d^{in}+d^{out}) bytes transferred during the nth call.
3. Actually, there are six kinds of SCO packets that can be selected in the Bluetooth specification to ensure quality of service, but we merely use DH1 packets here to simplify the cost model.

Our configuration criteria for scatternets under RMI operations are twofold: (1) the cost of interpiconet communication is very expensive and hence the most frequently exchanged data should be placed on the same piconet whenever possible, and (2) the number of members located at each piconet should be similar.

Let A be an adjacent matrix of G, where $a_{i,j}$=1 if a direct edge linking v_i, and v_j exists; otherwise $a_{i,j}$=0, $a_{i,j}$∈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 an n×n cost matrix for the whole cost. hop(i,j) denotes the number of hops in a path P_{ij}. Therefore,

$$c'_{i,j} = \sum_{k \in RM^j} \sum_{n=1}^{F^j_k(i)} D^i_n(RM^j_k).min(hop(i,j)), \tag{1}$$

where RM^j is the set of methods to be invoked at node v_j. In addition, since the coordinator has detailed information about all nodes in scatternet, the shortest path min(hop(i,j)) can be found by Floyd's algorithm [5]. C' finds all the unidirectional traffic between node pairs, but the path workloads are what concerns us. We therefore use the new matrix C, where $c_{i,j} = c'_{i,j} + c'_{j,i}$. This symmetric matrix can be regarded as a complete graph with weighted edges whose values are message costs during a time period between two nodes. This is used as the cost model to guide the formation of scatternets.

4. Algorithms for Optimizing RMI over Scatternets

The strategy and configuration process for employing the Java RMI cost model for optimizations of scatternet formations is as follows:

1. Initialize all Bluetooth devices ready for communication.
2. Form an initial scatternet with a coordinator that knows status of other nodes, using the BTCP algorithm.
3. Coordinate and collect runtime behaviors of programs and access frequency from all nodes.
4. If the scatternet workload is heavy and some nodes at different piconets communicate frequently, determine the new clustering for scatternets.
5. Connect newly formed clusters.

We describe the above steps in the following subsections.

4.1. Initial Configuration of Scatternets

In the first step of our algorithm, we adopt BTCP [20] as an initial scatternet-formation algorithm. BTCP is a state-of-the-art algorithm for forming Bluetooth scatternets, but the standard version does not consider the access patterns of Java RMI. Therefore, once the initial configuration is obtained, we refine the configuration according to the following constraints and properties:

1. All nodes must be located on a transmission power-reachable range.
2. The maximum degree of a bridge node is 2.
3. To prevent interference and to reduce packet delays, the number of piconets forming a scatternet should be minimized.
4. There is at most one bridge between any two piconets.

This algorithm was implemented by finding a central node called the coordinator, allocating minimal piconets, and establishing the whole scatternet. For the sake of completeness, we outline this algorithm in Figure 6. An effective and predefined formula is used in the BTCP algorithm to determine the number of piconets and bridge nodes.

4.2. Scatternet Clustering with Spectral Bisection

By considering the access communication patterns of RMI, we hope to form scatternets that minimize communication costs. We want to determine which nodes should be formed as a piconet so as to minimize interpiconet communication. We use a partitioning algorithm to bisect |V| nodes into two groups of equal size while minimizing the message cost. The partitions can be performed recursively until the piconets are formed. Moreover, a local refinement method can be used to pick the master of a piconet by minimizing intrapiconet traffic. We formulate the equation as follows. To find a piconet of the vertex set in two parts V^+ and V^-, $V^+ \cup V^-$=V, we define a vector X $\in R^n$ called the indicator vector. If $v_i \in V^+$ then x_i=1; otherwise x_i=-1. Therefore, the interpiconet connection costs between two piconets are $f(x) = \sum \frac{1}{4}.c_{ij}(x_i - x_j)^2$. We summarize the constraints and formulate an objective function as follows:

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

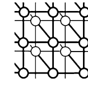
$$Subject\ \ to : \sum_{i=1}^{n} x_i = 0, x_i = \{1, -1\} \tag{3}$$

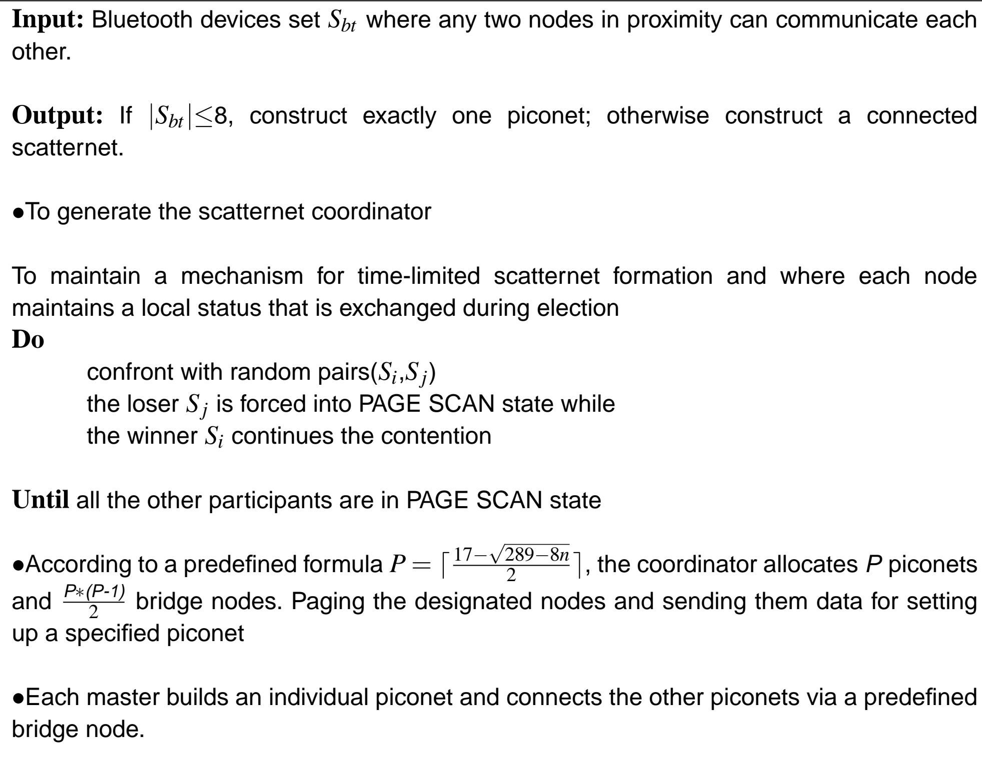
Equation 2 makes these two partitions fully equivalent if |V| is even. We also define a diagonal matrix C^d, where $c^d_{i,i} = \sum_{e_{ij}} c_{i,j}$; otherwise $c^d_{i,i}$=0 for i $\neq$ j. Inspection of our f(x) reveals

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

Following the first theorem of graph theory and formula transformation [22], we obtain

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





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

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

●To generate the scatternet coordinator

To maintain a mechanism for time-limited scatternet formation and where each node maintains a local status that is exchanged during 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 continues the contention

Until all the other participants are in PAGE SCAN state

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

●Each master builds an individual piconet and connects the other piconets via a predefined bridge node.

Figure 6. The BTCP algorithm.

The matrix L= (C^d-C) is a weighted Laplacian matrix, and this type of matrix has proven useful in graph partitioning. Suppose there is a scalar λ such that LY=λY: eigenvector Y is followed by its eigenvalue λ. Combining Equations 2 and 4, the original graph-partitioning problem is converted to a discrete optimal problem. There exists no practical scheme for solving this since the partition problem is NP-hard. Relaxing the constraint in Equation 3 by $\sum_{i=1}^{n} x_i = 0, x^T.x = n$, changes the problem to a continuous bisection problem. We rewrite our cost function and subjection as

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

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

The continuous approximation should be mapped to its primitive problem. If the partition is close to 1, -1, the result is near to an optimal solution. Some important properties of L have been proven, where we

are concerned with the second smallest eigenvalue called λ_2 of L whose eigenvector Y_2 is the minimal solution in Equation 5.

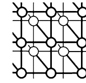
Y_2 is named the "algebra connectivity of graph" or "Fielder vector". The algorithm in this stage has two steps: (1) recursive spectral bisection is used to repartition all nodes in the original scatternet into P clusters, and (2) the KL [8] procedure is applied to these unbalanced partitions to locally refine and produce clusters of approximately the same size. The algorithm in Figure 7 integrates recursive spectral bisection and KL to configure the scatternet according to a cost matrix. This processing produces P clusters of balanced size. The algorithm in Figure 7 first performs a recursive spectral-bisection algorithm and then uses the KL algorithm [8] (also known as the mincut algorithm). The fine tuning of the KL algorithm is used when the number of clusters to be partitioned is not an even number. The last few partitions, which are not well divided, are fine tuned by KL algorithm to produced clusters of balanced size.

4.3. Connecting Scatternets

In this section we focus on postprocessing to connect the clusters together after the piconet partitioning has been performed using the algorithm listed in Figure 7. This phase involves two stages. First we need to determine the role of each node in the clusters and append a feasible edge (link) into each node. A "master" attribute must be assinged 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 8 gives the main steps for a heuristic approach to perform these two steps. In the first step, the node for which the summation of total interior links is maximal 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 is based on it. There are several methods for choosing the bridges. The method presented in Figure 8 follows the concept of BTCP algorithms to produce a complete graph for connecting all the clusters together. In this method, each cluster has its slaves that 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 for choosing the bridges and connections for the partitioned clusters include finding the minimum spanning tree. The minimum-spanning-tree approach gives the connectivity, but reduces the energy consumption and collisions.

4.4. Complexity Analysis and Discussion

We now give the complexity of our proposed algorithm. The complexity of our initial configuration follows that of the BTCP algorithm [20]. The complexity of the BTCP algorithm has time complexity $\Omega(n/k)$, where n is the number of nodes and k is the maximum number of slaves in a scatternet. In our second stage, we partition the communication traffic in the Bluetooth environment using spectral bisection followed by tuning with the mincut algorithm. The complexity of the spectral bisection is $O(mn)$ when the Lanczos algorithm is employed. This algorithm normally converges in m iterations, each requiring O(n) operations [18]. Next, the fine tuning of the KL algorithm is used when the number of clusters to be partitioned is not an even number. Each iteration of KL algorithm requires $O(|E|)$ iterations, where E is the set of edges. Finally, we perform postprocessing to connect the



```
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 ⌊n/P⌋ or ⌊n/P⌋+1 nodes
Do /* Recursive spectral bisection */
        Find the Fielder vector Y_2 of L
        Compute V = X.Y_2
        Let m be the median value of elements of vector V

        if (the ith element of X ≤ 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 ∈ {1..P}, S_k = vertices in cluster k

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

While (Find a better cluster is possible) /* mincut */
       Do
          pick up the largest gain for moving v_i ∈ S_m to
          S_n
          S_m = S_m \ v_i
          S_n = S_n ∪ 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 7. Recursive spectral partitioning with KL refinement.

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

H = $\mid P_i \mid * \mid P_j \mid$ 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 the 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 $t^k_{i,j}$ 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:
permit no more than seven links on the same piconet.

Until all the bridges are generated

Figure 8. Algorithm for designating the role of nodes and the connecting scatternet.

cluster together after the piconets have been partitioned. This phase uses a bipartite matching scheme. The complexity of bipartite matching [11] is $O(P^3)$, where P is the number of piconets; since this is performed for each picotnet, $O(P^4)$ is the upper bound.

We discuss some of the design and application issues related to our algorithms below. First, the information for traffic loads can be collected into one server node. The algorithm then performs graph partitioning as a background process without interfering with the main computations. Once the partitioning is complete, we can initiate the reformation of scatternets.

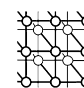


Table I. Detailed descriptions of benchmarks used.

Name	Description	Data size	Reference
EP	NAS EP generating Gaussian random numbers	16,777,216	[15]
Series	Fourier coefficient analysis	10,000	[6]
LUFact	LU Factorisation	500	[6]
SOR	Successive over-relaxation	1,000	[6]
Crypt	IDEA encryption	3,000,000	[6]
Sparse	Sparse Matrix multiplication	50,000	[6]
MolDyn	Molecular Dynamics simulation	2,048	[6]
MonteCarlo	Monte Carlo simulation	2,000	[6]
RayTracer	3D Ray Tracer	150	[6]
SelSort	Selection Sort	524,288	[9]
Hamming	Given an array of primes, output in numerical order and without duplicates all the integers of the form $a^i * b^j * c^k \ldots \leq n$	5	[17]

Second, our cost model assumes static nodes in the network and assumes nodes can be reached by each other. Taking into consideration the distances between nodes and the interference effects on the new arrangements of scatternets will be interesting for future explorations. However, our algorithm can deal with nodes joining or leaving. Nodes joining or leaving the network can be handled by conventional Bluetooth-formation algorithms (algorithms that do not consider RMI traffic). Once our system collects sufficient summary information about communication traffic, we can reconfigure the system according to our proposed mechanisms – this allows our method to deal with nodes joining or leaving. Our work also considers the interference effects of Bluetooth devices. The initial configuration and the amount of piconets in our algorithm are decided by BTCP (a conventional algorithm for Bluetooth scatternet formations). BTCP is known to minimize the number of piconets so as to reduce interference effects. In the second step of our algorithm, we use spectral bisection to determine which group nodes will be in the same piconets. After that, in the third step we connect the piconets together to form scatternets. In this phase of computations, we have some choices regarding the formation of connections between different piconets. To reduce the interference effects, a minimum spanning tree might be used instead of connecting the piconets together by a complete graph.

5. Experimental Results

We performed three experiments. In the first experiment, we evaluated the robustness of our implementation of RMI software over a Bluetooth environment using numerous benchmarks from the RMI benchmark suite [17], the DHPC Java benchmarks [15], and the Java Grande Forum MPJ benchmarks [6]. The last benchmarks are based on MPJ, which is an MPI-like message-passing interface for Java. We have implemented an MPJ interface that follows the mpiJava 1.2 specification [3]. It uses RMI as the underlying communication channel to evaluate the performance of our implementation of RMI over Bluetooth. Table I lists the applications that our RMI successfully ran in our early testing of the robustness of our software.

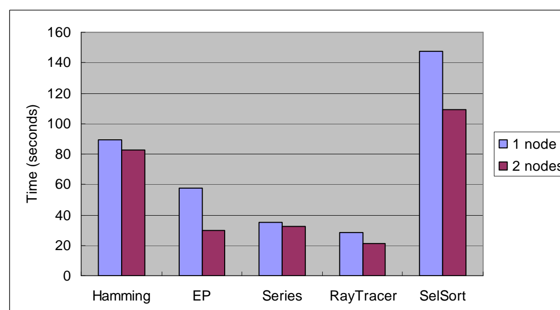
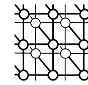


Figure 9. The performance results of some benchmark applications.

In our test bed, the EBDKs are connected to two computers via their COM ports. We tested the performance with one node and with two nodes : the one- and the two-node versions are sequential and parallel versions for RMI over Bluetooth, respectively. The RMI server has one 1200-MHz AMD CPU and 768 MB of RAM, and runs Microsoft Windows 2000 Server. The RMI client runs on an IBM ThinkPad notebook containing a 700-MHz Intel Pentium III CPU and 256 MB of SDRAM, 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 Microsoft Windows. Despite the limited bandwidth of the Bluetooth specification, we observed performance gains on Hamming, EP, Series, RayTracer, and SelSort benchmarks. The performance is shown in Figure 9. Note that due to the client and server residing in different machines, the optimal speed-up for this case is around 1.7 times. However, with a peak data rate of 723 kbps it is still much slower than fast Ethernet. Therefore, our experimental results show that these types of numerical applications run well on our implementation of RMI over Bluetooth. Collaborative software that requires a low communication bandwidth will probably run well in this type of environment. Note that our support of RMI over Bluetooth also gives more high-level control and a better programming environment than other Bluetooth environments. System loads at the operating-system level can also be distributed to remote site servers with RMIs.

We now provide more insight into one of our experiments – that 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 10 illustrates that when the array size is increased to more than 262,144 bytes, the distributed version running over the Bluetooth RMI has better performance that the local version.

In the second experiment, we evaluated the bandwidth and overheads with different layers of *JavaBT* and with RMI. The comparison was performed with HCI, L2CAP, and socket layers of our *JavaBT* and our RMI version. In this experiment, we transmitted byte arrays of different sizes between two computers using our Bluetooth protocol driver. Figure 11 shows the bandwidth for these four layers, and Figure 12 illustrates the overhead for these four layers in our test. Our *HCITransportBaseLayer* supports the COM port communication between the HCI layer and the Bluetooth device. These figures



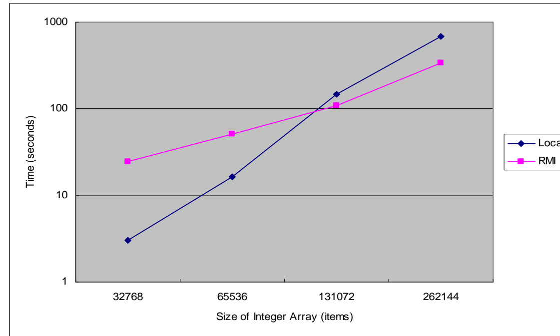
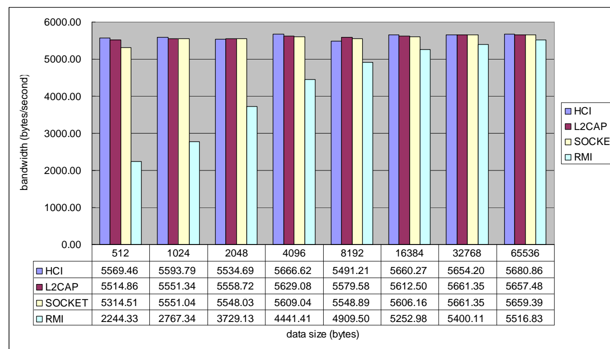


Figure 10. SelSort with local computation and a two-node version based on RMI over Bluetooth.

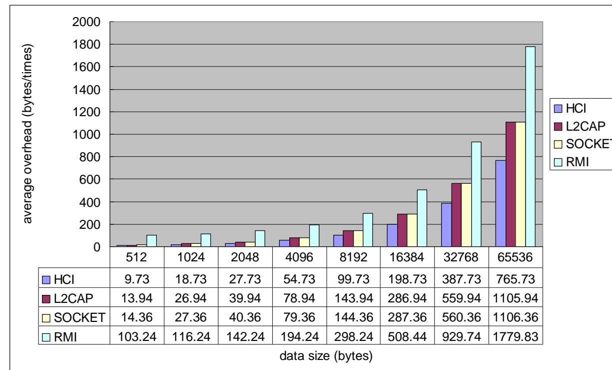


	512	1024	2048	4096	8192	16384	32768	65536
HCI	5569.46	5593.79	5534.69	5666.62	5491.21	5660.27	5654.20	5680.86
L2CAP	5514.86	5551.34	5558.72	5629.08	5579.58	5612.50	5661.35	5657.48
SOCKET	5314.51	5551.04	5548.03	5609.04	5548.89	5606.16	5661.35	5659.39
RMI	2244.33	2767.34	3729.13	4441.41	4909.50	5252.98	5400.11	5516.83

Figure 11. A bandwidth comparison for different layers.

provide comparisons of additional data being sent at different layers in our current implementation; most overheads are observed in the RMI version due to the sequential running of RMI.

Finally, we experimented with the scatternet-formation algorithms by exploiting the communication patterns of RMI over a Bluetooth environment. The experiments involved simulations with our cost model, with normal distributions being used for the necesary parameters. The cost in the experiment is defined in Equation 1. Although matrix C is symmetric, the total cost can be only the summation of the upper-right part of this matrix. To compute the total cost, we need a hop cost H, an end-to-end call frequency F, and the amount D of data being transferred. H is dependent on the initial topology, and we use a random-number generator to produce values for F and D. Figure 13 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 – improvements of around 40% are observed in this case.



	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

Figure 12. Overhead comparison for different layers. Additional data sent at different layers are presented.

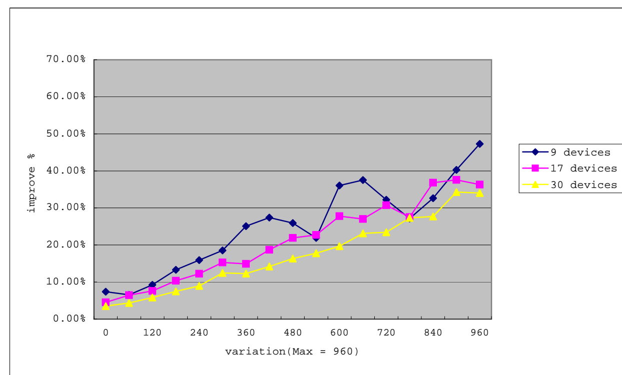
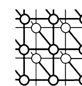


Figure 13. Communication patterns using normal distribution with $\mu = 300, \sigma = 0 - 960$.

6. Related Work

The efficient support for remote Java invocation is an important topic, since RMI provides a layer of abstraction for communications. Previous research results have included an open RMI implementation that makes better use of the object-oriented features of Java [21], ARMI [19], and Manta [14] systems to reduce various drawbacks in RMI and to provide new types of RMI systems with added functionality, and a better way of implementing RMI by exploiting Myrinet hardware features to reduce latencies [17] and provide support for a broader range of RMI applications [2].

Zucotto has provided a Java environment for the Bluetooth environment [23], and IBM provides a Bluetooth protocol driver for the Linux operating system, called BlueDrekar [7]. In addition, the



OpenBT project [1] from Axis is an open-source project aimed at building a Bluetooth protocol driver for the Linux operating system. Recently, the JSR-82 specification of the Java Community Process [10] has standardized a set of Java APIs to allow these Java-enabled devices to integrate into a Bluetooth environment. It would be interesting to further revise our *JavaBT* protocol stack to conform with JSR-82 specifications. Rococo Software has provided a reference implementation [16] for the specification defined by the JSR-82 Expert Group. Our implementation with Java RMI over the L2CAP layer also provides a performance baseline for comparisons with other designs in the future. In addition, we provide optimization methods for RMI programs over a Bluetooth environment to optimize the formation of scatternets. We use spectral bisection methods [18] for graph partitioning. Spectral bisection provides accurate mathematic modeling of the problem, but it takes longer. When execution time is important, it can be reduced using multilevel schemes.

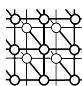
7. Conclusions

In this paper, we have reported the issues and our research results related to the efficient support of Java RMI over a Bluetooth environment. In our work, we first enable the support of Java RMI over Bluetooth protocol stacks, by incorporating a set of protocol stack layers for Bluetooth developed by us, called *JavaBT*, and by supporting the L2CAP layer with sockets that support the RMI socket. In addition, we have presented a cost model for the access patterns of Java RMI communications. The cost model was used to optimize the formation of scatternets in a Bluetooth environment associated with Java RMI environments. Experimental results show that our scatternet-formation algorithm incorporating an access-cost model can be used to optimize the performance of Java RMI over a Bluetooth environment.

There are several research items left for future exploration. First, the JSR-82 specification of the Java Community Process [10] recently standardized a set of Java APIs to allow these Java-enabled devices to be integrated into a Bluetooth environment. It would be interesting to further revise our proposed *JavaBT* to conform with the JSR-82 specification. Second, implicit in our scatternet-optimization algorithm is the assumption that all nodes can be reached by other node, and hence a more flexible algorithm is needed. Finally, supporting distributed computations over wireless devices with heterogeneous architectures is an important direction for future explorations.

REFERENCES

1. Axis Communications. *An open source Bluetooth protocol stack for Linux*. http://java.sun.com/products/javacomm/.
2. F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, 1998.
3. B. Carpenter, G. Fox, S.-H. Ko, and S. Lim. *mpiJava 1.2: API Specification*. http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html, October 1999.
4. 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.
5. T. B. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
6. The Java Grande Forum MPJ benchmarks. Edinburgh Parallel Computing Centre. http://www.epcc.ed.ac.uk/javagrande/mpj.html.
7. *A Bluetooth protocol stack for Linux*. IBM. http://www.alphaworks.ibm.com/tech/bluedrekar/.



8. B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, Feb., 1970.
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. Bala Kumar. Java APIs for Bluetooth Wireless Technology Specification Version 1.0a, (JSR-82). http://jcp.org/en/jsr/detail?id=082, March 2002.
11. H. W. Kuhn. Variants of the Hungarian method for the assignment problems. Naval Res Logist Quart, 3:253–258, 1956.
12. J. K. Lee and D. Gannon. Object-oriented parallel programming: experiments and results. In *Proceedings of Supercomputing'91*, pp. 273–282, New Mexico, November 1991.
13. J. K. Lee, I.-K. Tsaur, and S.-Y. Hwang. Parallel Array Object I/O Support on Distributed Environments. *Journal of Parallel and Distributed Computing*, 40:227–241, 1997.
14. J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, pp. 173–182, May 1999.
15. J. A. Mathew, P. D. Coddington, and K. A. Hawick. DHPC Java benchmarks. http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks.
16. K. McCabe. Rococo Bluetooth Technology Licensing Kit (TLK) technical overview. http://www.rococosoft.com/downloads/pdf/tlk_technical_overview.pdf.
17. C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *Proceedings of ACM 1999 Java Grande Conference*, pp. 152–157, San Francisco, CA, June 1999.
18. A. Pothen, D. H. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. In *SIAM Journal of Matrix Analysis and Applications*, 11:430–452, 1990.
19. R. Raje, J. Williams, and M. Boyles. An asynchronous remote method invocation mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, 1997.
20. T. Salonidis, P. Bhagwat, L. Tassiulas, and R. LaMaire. Distributed topology construction of Bluetooth personal area networks. In *Proceedings of the IEEE INFOCOM*, Vol. 3, pp. 1577–1586, Anchorage, AK, April 2001.
21. G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, 1998.
22. D. B. West. *Introduction to graph theory*. Prentice Hall, 2001.
23. *XJB 100 Bluetooth Host Stack*. Zucotto Wireless Corp. http://www.zucotto.com/.

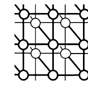
A. JavaBT: Bluetooth Protocol Driver

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.

A.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



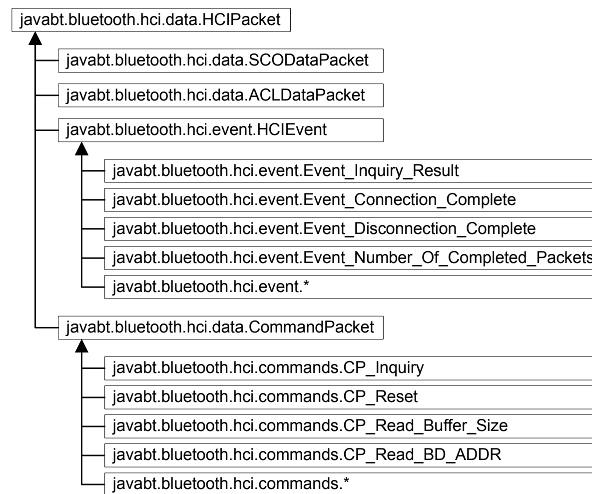


Figure 14. The class hierarchies for handling HCI packets.

packets are used to carry the ACL and SCO data, respectively. Figure 14 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 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 15. 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.

A.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

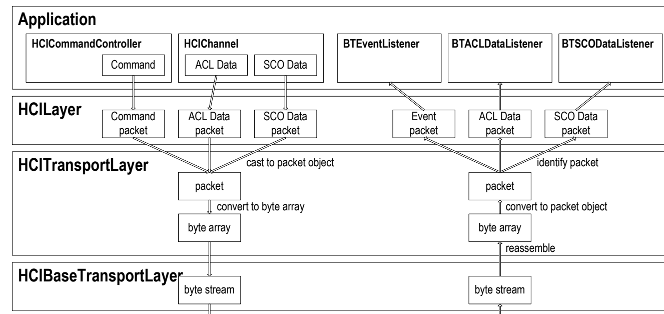


Figure 15. Data flow in an HCI layer.

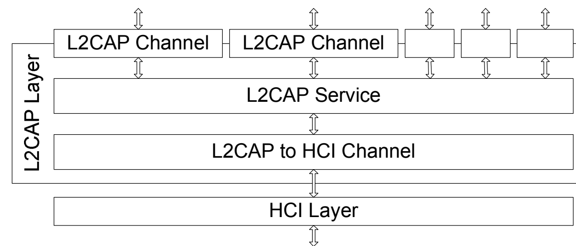


Figure 16. Overview of L2CAP layers in *JavaBT*.

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 16 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* 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.