THE FEASIBILITY OF MULTICASTING IN RMI

Vinay Ujjinihavildar, B.E.

Problem in Lieu of Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2003

APPROVED:

Roy Tom Jacob, Major Professor
Steve Tate, Committee Member
Robert Brazile, Graduate Advisor of the
    Department of Computer Science
Krishna Kavi, Chair of the Department of
    Computer Science
C. Neal Tate, Dean of the Robert B. Toulouse
    School of Graduate Studies
Ujjinhavildar, Vinay, *The feasibility of multicasting in RMI*. Master of Science (Computer Science), May 2003, 25 pp., 1 table, 6 figures, references, 19 titles.

Due to the growing need of the Internet and networking technologies, simple, powerful, easily maintained distributed applications needed to be developed. These kinds of applications can benefit greatly from distributed computing concepts. Despite its powerful mechanisms, Jini has yet to be accepted in mainstream Java development. Until that happens, we need to find better remote method invocation (RMI) solutions. Feasibility of implementation of multicasting in RMI is worked in this paper. Multicasting capability can be used in RMI using Jini-like techniques. Support of multicast over unicast reference layer is also studied. A piece of code explaining how it can be done, is added.
# TABLE OF CONTENTS

LIST OF FIGURES AND TABLES ................................................................. iii

Chapter

1. INTRODUCTION ................................................................................ 1
   
   1.1 Problem Statement
   1.2 An Overview of RMI
      1.2.1 RMI Explained
   1.3 RMI Architecture
      1.3.1 In Detail
      1.3.2 Stubs and Skeletons
      1.3.3 Remote Reference Layer

2. MULTICAST EXPLAINED .................................................................. 8
   
   2.1 Is Multicasting possible in RMI?
   2.2 Implementing a Jini-like Mechanism for RMI
      2.2.1 What is Jini?
   2.3 Implementation
      2.3.1 Implementing server Side classes

3. UNICAST CONNECTION BACK TO DISCOVERER ........................... 18
   
   3.1 Implement Client side Classes
   3.2 Set up a Unicast TCP/IP Listener
   3.3 Set up a Multicast UDP Dispatcher
      3.3.1 Read the Server's Stub

4. LIMITATIONS ............................................................................... 23

5. REFERENCES .................................................................................. 24
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RMI System Overview</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>RMI functionality</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>RMI architecture</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Multicast address</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Jini multicast discovery</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>RMI multicast</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RMI client-server interaction</td>
<td>14</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

The goal of this thesis is to find a way to add a multicasting capability to the existing remote method invocation (RMI) in Java. Several issues need to be investigated in this context:

1.1 Problem Statement

- Find out what the possibilities are to add multicasting capabilities to RMI. Support for multicast over unicast reference layer.
- Investigate the possibilities of using existing code for the task of multicasting.
- Investigate current developments related to multicasting in RMI.
- Investigate the feasibility of an implementation of multicasting in RMI.
- When the feasibility has been proven, implement the multicast capability in the existing RMI layer.

1.2 An Overview to RMI

Due to the growing need of the Internet and networking technologies, simple, powerful, easily maintained distributed applications needed to be developed. These kinds of applications can benefit greatly from distributed computing concepts. Java distributed computing technology has given developers a new solution to solve critical distributed application problems. JDK software, and Java remote method invocation (Java RMI) technology provides a consistent programming platform that significantly simplifies distributed application development. [1]
RMI enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. The Java remote method invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM. RMI provides for remote communication between programs written in Java programming language. [11]

RMI applications are often composed of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.
1.2.1 RMI Explained:

RMI is a mechanism for communicating between two machines running Java Virtual Machines. When Java code on machine A needs a service or a method, respectively, of objB on machine B it starts a remote method invocation. It does this the same way as invoking a local object's method. The whole communication behind the remote invocation is done by the RMI mechanism. [12]

One of the central and unique features of RMI is its ability to download the byte codes of an object’s class if the class is not defined in the receiver’s virtual machine. RMI passes objects by their true type, so the behavior of those objects is not changed when they are sent to another virtual machine. [1]

This study aims at creating a transport layer that supports multicasting functionality which I could do till a little extent. This can be improved further to enhance its capabilities and overcome the limitations and bugs.
1.3 RMI ARCHITECTURE

Distributed object applications need to

- Locate remote objects: Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI’s simple naming facility, the rmiregistry, or the application can pass and return remote object references as part of its normal operation.

- Communicate with remote objects: Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.

- Load class byte codes for objects that are passed around: Because RMI allows a caller to pass objects to remote objects; RMI provides the necessary mechanisms for loading an object’s code, as well as for transmitting its data.

The following illustration depicts an RMI distributed application that uses the registry to obtain a reference to a remote object. The server calls the registry to associate (or blind) a name with a remote object. The client looks up the remote object by its name in the server’s registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to class byte codes, from server to client and from client to server, for objects when needed. [2]
1.3.1 In Detail:
The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model.

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via Remote Object Activation. [3]
The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.

By using a layered architecture each of the layers could be enhanced or replaced without affecting the rest of the system. For example, the transport layer could be replaced by a UDP/IP layer without affecting the upper layers.

1.3.2 Stubs and Skeletons

The stub and skeleton layer use the proxy design pattern. The proxy knows how to forward method calls between the participating objects. In RMI’s use of the proxy pattern, the stub class plays the role of the proxy.

A skeleton is a helper class that is generated for RMI to use. The skeleton understands how to communicate with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub. [11]
In the java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object.

1.3.3 Remote Reference Layer

The Remote Reference Layers defines and supports the invocation semantics of the RMI connection. This layer provides a RemoteRef object that represents the link to the remote service implementation object.

The stub objects use the invoke () method in RemoteRef to forward the method call. The RemoteRef object understands the invocation semantics for remote services. The JDK 1.1 implementation of RMI provides only one-way for clients to connect to remote service implementations: a unicast, point-to-point connection. Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system.[1]

In Java 2 SDK implementation of RMI, it supports activatable remote objects. When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects. Other types of connection semantics are possible. For example, with multicast, a single proxy could send a method request to multiple implementations simultaneously and accept the first reply (this improves response time and possibly improves availability). In the future, Sun may additional invocation semantics to RMI.[1]
2. MULTICAST EXPLAINED

Multicast can be simply defined as the ability to send a message to either one or more nodes in a single operation. This is different than using replicated unicast which sends messages from one node to a group of nodes by sending to each node individually. Multicasting is a term used for streaming data to multiple destinations at the same time. It’s an efficient way to send high bandwidth data, such as multimedia, when you have a large number of users who want the same data.\[4\]

The usual way of data is sent over the Internet is through unicasting. For example, with unicast each user who wants to view a movie gets the copy transmitted from the server. If one thousand users request to view the same movie, then one thousand copies of the same movie are transmitted over the internet. With multicasting, only one copy of the movie is transmitted, but any user who wants to view it can tune in to the transmission.

Multicast is best in use for synchronization, duplication and coherency of data in various systems. For the implementation of coherency one needs to use atomic operations among different machines. This atomicity can be achieved by using multicasting. [4]
Figure 4: Multicast address

The basic service provided by IP multicast is an unreliable datagram multicast service. With an unreliable multicast service, there is no guarantee that a given packet reached all intended recipients that belong to the multicast group. [9][10]

Every IP multicast group has a group address. IP multicast provides only open groups. That is, it is not necessary to be a member of a group in order to send datagrams to the group. The one, which concerns us, is the “Class D Address”. Every IP datagram whose destination address starts with “1110” is an IP multicast datagram.

The remaining 28 bits identify the multicast “group” the datagram is sent to.
Multicast addresses are like IP addresses used for single hosts, and are written in the same way: A. B. C. D. Multicast addresses will never clash with host addresses because a portion of the IP address space is specifically reserved for multicast. This reserved range consists of addresses from 224.0.0.0 to 239.255.255.255. However, the multicast addresses from 224.0.0.0 to 224.0.0.225 are reserved for multicast routing information. Application programs should use multicast addresses outside this range.[4]

There are some special multicast groups that should not be used in particular applications due to the special purpose they are designed to:

- 224.0.0.1 is the all-hosts group. If you ping that group, all multicast capable hosts on the network should answer, as every multicast capable host must join that group at start-up on all its multicast capable interfaces.
- 224.0.0.2 is the all-routers group. All multicast routers must join that group on all its multicast capable interfaces.
- 224.0.0.4 is the all DVMRP routers, 224.0.0.5 the all OSPF routers, 224.0.0.013 the all PIM routers, etc.

In any case, range 224.0.0.0 through 224.0.0.255 is reserved for local purposes (as administrative and maintenance tasks) and datagrams destined to them are never forwarded by multicast routers. Similarly, the range 239.0.0.0 to 239.255.255.255 has been reserved for “administrative scooping”. [5]

There are three basic modes of transmitting packets over an IP network: unicast, broadcast, and multicast.
In unicast mode, packets are transmitted over a point-to-point connection. If there is more than one recipient for the packet, several point-to-point connections are established and a copy of each packet is sent to each recipient.

In broadcast mode, a single copy of a packet is sent to the entire network, including subnets that don’t have recipients for the packet. Thus, extra traffic is created that consumes valuable network bandwidth.

In multicast mode, a single copy of the data is transmitted to subnets that have a recipient host subscribed to the session. Hosts subscribe to the session by joining a multicast group, which they do by propagating (through Internet Group Management Protocol) a pair of addresses — multicast address and recipient’s IP address — to the routers on the network.[9]

2.1 Is Multicasting Possible In RMI??

There is no “official” implementation of a Multicast Remote Object in RMI. However, we can work around its absence by implementing a factory design pattern.


In the latest release of RMI v1.3 support for customizable remote references types are provided. There is no direct support for Multicast Remote Object but the additions to the RMI API allows one to be defined to work in the framework.

Ref: Live chat in JDC forums. Speakers: Ann Wollrath (RMI architect) and Adrian Colley.[18]
2.2 Implementing a Jini-like discovery mechanism for RMI:

During my extensive research in RMI, I found while reading various papers that Jini clients don’t need to know where a service is located. But in RMI we must know the URL of the server we want to use. In this research I tried to show how we can implement a Jini like discovery mechanism for RMI which reduces a few clients’ trouble to look up for the URL. I found the answer to this in www.javaworld.com/javaworld and http://java.sun.com/products/jini/index.html. Many researches are currently working on it and they say that “many RMI based systems still exist, and until Jini is accepted into mainstream Java development, we need to provide more elegant RMI solutions”. The work and research described in my work is based on such a requirement. [19]

2.2.1 What is Jini?

“Jini network technology is an open architecture that enables developers to create network-centric services -- whether implemented in hardware or software -- that are highly adaptive to change. Jini technology can be used to build adaptive networks that are scalable, evolvable and flexible as typically required in dynamic computing environments”. [6][7]

Comparison:

In RMI, a client must know the location of the server it wants to connect. RMI server’s addresses are in the form:

```java
rmi://<host>:<port>/<servername>
```

The port number is the port on which the rmiregistry listens for requests. But in Jini, a client finds a service using a Jini utility class.
2.3 Implementation

In my implementation technique, I learned how Jini does it and tried to extract those concepts that suite RMI server and client. Jini uses a combination of multicast UDP (User Datagram Protocol) and unicast TCP/IP. In this a client sends a multicast request packet, which is then picked up by the look up services looking for it. The lookup services make unicast connections back to the client and serialize their proxy down the stream available over the connection. The client then interacts with the proxy to locate the service it wants.[8]

![Figure 5: Jini Multicast discovery](image)

Here, I am trying to implement a simple multicast listener/unicast dispatcher that an RMI server can use, making each RMI server a lookup service.

![Figure 6: RMI multicast](image)
The Jini client and lookup service here are RMI client and RMI server. The table below describes the interaction between RMI client and RMI server.[8]

<table>
<thead>
<tr>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start listening on multicast address</td>
<td>Start ServerSocket to listen for unicast responses from the server</td>
</tr>
<tr>
<td></td>
<td>Begin sending UDP packets to multicast address</td>
</tr>
<tr>
<td>Parse received UDP packet. If valid, connect back to the client via unicast TCP/IP</td>
<td></td>
</tr>
<tr>
<td>Send remote stub to client</td>
<td>Read remote object from stream</td>
</tr>
<tr>
<td></td>
<td>Close ServerSocket. Stop sending UDP multicast packets</td>
</tr>
</tbody>
</table>

Table 1: RMI client-server interaction

For an RMI client to discover a server, it should specify an interface class and a unique name to identify a server instance. This is because multiple servers that implement the same interface may be running concurrently. [13]

We must define the protocol for message passing. For simulation here, we'll use a delimited string containing all the information a RMI server needs to respond to a matching request. We first define a protocol header. This prevents the server classes from attempting to fully parse packets that arrive from other sources. The remainder of
the message packet will contain the unicast response port, the server’s interface class name, and the server instance's unique identifier. [14]

The format of the discovery request message:

<protocol>,<unicast port>,<interface class><unique id>

A sample “hello” message is used to simulate multicasting in RMI. RMI-multicast is the protocol header, and 8080 is the port number on which the client is listening for responses:

RMI-Multicast, 8080, HelloWorld, Hello

I did not include the client's hostname in the request because that information can be obtained from the UDP packet received on the server. Having defined our message format, I can start implementing the discovery classes.[15]

2.3.1. Implementing the Server Side classes:

Now we have to write a utility class that RMI servers can use to create their own personal lookup service.

//instantiate RMI server
Remote server=new HelloWorldServer();

The Remote parameter checks if the server implements the interface the client is trying to discover and whose RMI stub is ultimately serialized back to the client.[15]

//initiates discovery listener
RMILookup.bind (server,"Hello")
The String parameter compares the server name with the name in the request packet.

Server side classes’ responsibilities:

1. Set up a multicast UDP socket to listen for requests.
2. Check the protocol header when a packet arrives.
3. Parse the message packet.
4. Match the unique server name parameter.
5. Match the interface parameter.
6. If server name and interface parameter match, serialize the server’s remote stub to the client via unicast TCP/IP socket.

Set up a multicast UDP listener:

To set up a multicast listener, you must use a known multicast address and port; those in the range 224.0.0.1 to 239.255.255.255, inclusive.

Example:

```java
int port=8080;
String multicastAddress="228.5.6.7";
MulticastSocket socket=new MulticastSocket(port);
InetAddress address=InetAddress.getByName(multicastAddress);
socket.joinGroup(address);
byte[ ] buf = new byte[512];
DatagramPacket packet=new DatagramPacket(buf, buf.length);
socket.receive(packet);
```
socket.leaveGroup(address);

The above piece of code shows how simply we can set up a multicast listener and receive packets on that address/port combination. In the above code, only a single packet can be processed, so we must create a loop around the DatagramPacket creation and socket.receive(); otherwise only one client will be able to discover the server:

While (active) {
    byte[ ] buf=new byte[512];
    DatagramPacket packet=new DatagramPacket(buf,buf.length);
    socket.receive(packet);
}

Check the protocol header:

We check the whether the packet is message we are looking for. We convert the byte array into a String and use the startsWith() method. Although we have hardcoded the protocol header RMI-multicast, it will be accessed as a constant in the actual source code:

String msg=new String(packet.getData()).trim();

boolean validPacket=msg.startsWith("RMI-Multicast");
Parse the message:

Assuming we have a valid packet, we can parse the message. As the message is delimited, we can use StringTokenizer to unpack it:

```java
private String[] parseMessage(String msg, String delim){
    StringTokenizer st=new StringTokenizer(msg,delim);
    st.nextToken();
    String[] str=new String[3];
    str[0]=st.nextToken();
    str[1]=st.nextToken();
    str[2]=st.nextToken();
    return str;
}
```

Once we convert the message packet into its parameters, we can check the interface class name and unique server name against the server's name. To match the server's unique name against the parameter, we simply compare the two String objects.

We can compare the interface names within the interface array implemented by the server:

3. UNICAST CONNECTION BACK TO DISCOVERER

If both the unique server name and interface class match, we can attempt to connect back to the client and serialize the server's stub. Here, repAddress has been obtained
from the incoming DatagramPacket. repPort has been parsed from the message packet. If we simply serialized the Remote object down the stream, a ClassNotFoundException would occur on the client, unless the client had access to the server's stub. The client would experience ClassNotFoundExceptions because, unlike passing objects over RMI where the codebase is annotated into the stream, here we are using serialization over a socket, which doesn't include the codebase.[8]

MarshalledObject was introduced in Java 2 and provides, among other things, a convenient way to pass serialized objects along with their codebases. MarshalledObject serializes an object into a byte array, which means when the MarshalledObject gets deserialized, the underlying object doesn't. This is extremely useful to Jini services, such as lookup services, as they aren't forced to download classes referenced by registered proxies.

To access the underlying object, we invoke the get() method on MarshalledObject in the client.

3.1 Implement the client-side classes

An RMI client would discover an RMI server by specifying the interface class and the server's unique name:

    Class hello = HelloWorldClient.class;
    String id="Hello";
    HelloWorldClient service=(HelloWorld)RMIDiscovery.lookup(hello,id);
Responsibilities:

1. Listen for unicast responses from the server’s RMILookup
2. Send UDP packets to multicast address
3. Read remote object from stream
4. Stop sending multicast packets
5. Stop listening on unicast socket
6. Use server

3.2 Set up a unicast TCP/IP listener

To set up a unicast TCP/IP socket, we must pick a port on which to listen. We can't define a constant with a fixed port number because another process may be using that port. We therefore need to specify a range of ports to use:[16]

```java
private startListener(int port,int range){
    ServerSocket listener=null;
    for(int i=port;listener==null && i<(port+range+1);i++){
        try{
            listener =new ServerSocket(i);
        }catch(IOException ex){
            }
    }
    return listener;
}
```
The startListener() method attempts to create a ServerSocket on a port within the specified range. The calling method can then check whether the return value is not null and obtain the port being used. Another option would be to throw an exception if the ServerSocket couldn't be created:

```java
ServerSocket listener=startListener(START_PORT,RANGE);
if(listener!=null){
    int port=listener.getLocalPort();
    Socket sock=listener.accept();
}
```

Once we successfully set up the unicast listener, we can then format the message packet and the next step would be to start the multicast dispatcher.

### 3.3 Set up a multicast UDP dispatcher

With the multicast listener, we must use a known multicast address/port combination. We should access this data either via System properties or via a constant:

```java
int port=8080;
String multicastAddress="228.5.6.7";
MulticastSocket socket=new MulticastSocket(port);
InetAddress address=InetAddress.getByName(multicastAddress);
socket.joinGroup(address);
DatagramPacket packet=new DatagramPacket(buf,buf.length,address,multicastPort);
socket.send(packet);
```
socket.leaveGroup(address);
socket.close();

Once we configure the MulticastSocket, the outputMessage String is converted into a byte array ready to be sent down the socket. We then send the message a preconfigured number of times or until the unicast listener receives a response. We have omitted the thread coordination with the unicast listener from the examples to keep them concise.

3.3.1 Read the server's stub:

Earlier we have set up the unicast ServerSocket. The method ServerSocket.accept() is blocking, so it won't return with a Socket object until an incoming connection is made:

Socket sock=listener.accept();
ObjectInputStream ois=new ObjectInputStream(sock.getInputStream());
MarshalledObject mo=(MarshalledObject)ois.readObject();
sock.close();
server=(Remote)mo.get();

Once we have a reference to the server, we can then wake up the thread blocking in the call to RMIDiscovery.lookup(), which will return the Remote object to the client.
4. LIMITATIONS

We saw how we can apply a similar technique to Jini's discovery concept to RMI clients and servers. The RMI discovery mechanism has some limitations that Jini can overcome. For example, multicast UDP has a restricted range often within a subnet. This means that clients using our multicast mechanism cannot discover RMI servers running outside the multicast range. Jini, however, has the concept of federated lookup services to join different subnets and make the discovery process transparent to clients across a WAN (wide area network). Multicast UDP generally won't work on standalone machines not connected to a hub. [13,17,7]
5. REFERENCES


[2]. RMI Documentation and Data Sheet –
http://java.sun.com/marketing/collateral/rmi_ds.html

[3]. Factory patterns to RMI –
http://java.sun.com/j2se/1.3/docs/guide/rmi/Factory.html


[6]. Complete listing of JavaWorld’s Jiniology column –


[12]. Java RMI by William Grosso.


[14]. *Jini in a nutshell*: by Scott Oaks, Henry Wong

[15]. *A Programmer’s guide to Jini technology*.


