

PORTING OF AN EXISTING SOFTWARE  
FROM THE SUN WORKSTATIONS TO A  
PERSONAL COMPUTER ENVIRONMENT

By

LAKSHMANA PAMARTHY

Bachelor of Technology

Andhra University

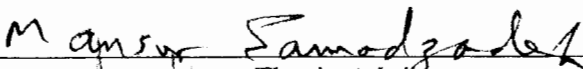
Waltair, India

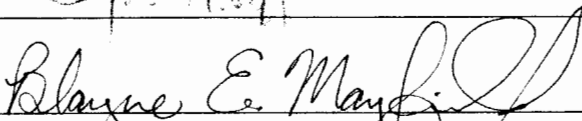
1994


Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the Degree of  
MASTER OF SCIENCE  
December 1996

PORTING OF AN EXISTING SOFTWARE  
FROM THE SUN WORKSTATIONS TO A  
PERSONAL COMPUTER ENVIRONMENT

Thesis Approved:

  
\_\_\_\_\_  
Thesis Adviser

  
\_\_\_\_\_

  
\_\_\_\_\_  
Dean of the Graduate College

## PREFACE

The purpose of this study was to port an existing software component, which has been designed for Sun workstations, to a Personal Computer environment. The software component involves the access of remote objects through Java networking. The ported software was tested on various machines housing different file servers of the Oklahoma State University Computing and Information Services Department. A new user interface was developed for the ported software component and a number of enhanced features were introduced.

The software worked fairly well on all of the test machines. The new feature (i.e., finding out the time taken to look for a remote server) resulted in a more user-informative environment. Other user friendly features such as information about the current server, the reset feature, and the option of returning to the default server were also introduced. The amount of time taken to look for a method located on a specified server is more when the server is searched for the first time compared to the time taken to relocate the same server. This can be attributed to the Registry mechanism of Java, which is like a simple register containing the information of all the successfully located servers. This mechanism makes a record of all the servers that are accessed, and every lookup for a remote server is first searched across the registry. Thus, it reduces the total lookup time for already accessed servers.

## ACKNOWLEDGMENTS

I take this opportunity to express my sincere gratitude to Dr. Mansur Samadzadeh for his guidance and encouragement. He is a constant source of inspiration through his intelligent supervision, constructive criticism, and moral support. I appreciate him greatly for this personal and intellectual relationship. My appreciation extends to my other committee members Drs. Mayfield and Hatcliff for agreeing to be on my thesis committee, and for their comments and advice.

I would also extend my special appreciation to my friends and family for their friendship and support.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION .....	1
II. LITERATURE REVIEW.....	3
2.1 Introduction.....	3
2.2 Software Portability .....	5
2.2.1 History and Past Research Efforts.....	5
2.2.2 Portability Concepts .....	6
2.2.3 Reuse Economy .....	7
2.3 Data Transfer Mechanisms .....	8
2.4 Remote Procedure Call .....	9
2.4.1 The Client-Server Model.....	9
2.4.2 Concept of RPC.....	9
2.4.3 Implementation Of RPC.....	10
2.4.4 Drawbacks Of RPC .....	11
2.4.5 Advances in Using RPC.....	12
III. JAVA REMOTE METHOD INVOCATION.....	14
3.1 Introduction.....	14
3.2 Java Distributed Object Model .....	15
3.2.1 RMI Interfaces and Classes.....	16
3.2.1.1 Remote Interface .....	16
3.2.1.2 Remote Exception Class.....	17
3.2.1.3 Remote Object Class .....	18
3.2.1.4 Remote Server .....	18
3.2.1.5 Unicast Remote Server.....	18
3.2.2 Implementing a Remote Interface .....	18
3.2.3 Parameter Passing in Remote Method Invocation.....	20
3.2.3.1 Passing Non-Remote Objects.....	20
3.2.3.2 Passing Remote Objects .....	20
3.2.4 Exception Handling in Remote Method Invocation.....	21

3.2.5	Locating Remote Objects .....	21
3.3	An Overview of RMI Architecture .....	22
3.3.1	Stub/Skeleton Layer .....	24
3.3.1.1	Dynamic Stub Loading.....	25
3.3.2	Remote Reference Layer.....	26
3.3.2.1	Client-side Component.....	26
3.3.2.2	Server-side Component .....	26
3.3.3	Transport Layer.....	27
IV.	IMPLEMENTATION ISSUES .....	29
4.1	Implementation Platform and Environment.....	29
4.1.1	Present Day Personal Computers.....	29
4.1.2	Windows Environment .....	30
4.2	Java Programming Environment .....	30
4.3	Implementation Details .....	32
4.3.1	Client Interfaces .....	36
4.3.1.1	Remote Interface .....	36
4.3.1.2	RemoteException Class .....	37
4.3.1.3	RemoteObject Class.....	37
4.3.1.4	Naming Class.....	38
4.3.2	Server Interfaces .....	39
4.3.2.1	Remote Server Class .....	39
4.3.2.2	UnicastRemoteServer Class.....	40
4.3.2.3	Creating a New Remote Object .....	41
4.3.2.4	Exporting A Remote Object.....	41
4.3.3	The Registry Interfaces .....	41
4.3.3.1	Registry Interface.....	42
4.3.3.2	Locate Registry Class .....	43
4.3.3.3	RegistryImpl Class.....	43
4.4	Usage Details .....	44
V.	EVALUATION.....	48
5.1	Sample Tests Done with the Program.....	48
5.2	Observations .....	52
5.3	User Appraisal .....	56
VI.	SUMMARY AND FUTURE WORK.....	58
REFERENCES	.....	60
APPENDICES	.....	63

A. GLOSSARY.....	63
B. TRADEMARK INFORMATION.....	65
C. USER GUIDE .....	66
1. Introduction .....	66
2. Setting Up.....	66
2.1 Hardware and Software Requirements .....	67
2.2 Running the Program.....	67
D. PROGRAM LISTINGS.....	83

## LIST OF TABLES

Table	Page
I. Sample Remote Method Lookup Times in Milli Seconds - I	49
II. Sample Remote Method Lookup Times in Milli Seconds - II	51
III. Comparison of the Versions	53



## LIST OF FIGURES

Figure	Page
3-1. Interface and Class Relationships in RMI.....	16
3-2. Overview of the RMI system architecture.....	22
4-1. Overview of RMI Interfaces.....	34
4-2. Overview of RMI Interfaces.....	34
4-3. Overview of RMI Interfaces (continued).....	35
4-4. Overview of RMI Interfaces.....	36
5-1. Sample Remote Method Lookup Data Graph - I.....	50
5-2. Sample Remote Method Lookup Data Graph - II.....	51
C-1. Initial DeskTop Screen.....	68
C-2. Initial Setup Screen.....	69
C-3. Running the Server.....	70
C-4. Applet Loaded on Server Machine.....	71
C-5. Invoking a Method on a Local Host.....	72
C-6. Using the Default Option.....	73
C-7. Using the Reset Option.....	74
C-8. Using Lookup Method with no Active Server.....	75
C-9. Using the Invoke Method with no Active Server.....	76

C-10. Looking for another Method on the same Host.....76

C-11. Invoking another Method on the same Host.....77

C-12. Looking for another Method (Echo5) on the same Host.....77

C-13. Remote Exception during Client Lookup.....78

C-14. Looking for a Remote Client.....79

C-15. Status of the current Client using the Applet.....80

C-16. Looking for another Method on a Remote Host.....81

C-17. Multiple Clients using a Remote Host.....82

## CHAPTER I

### INTRODUCTION

Software reuse is receiving increasing attention [Hooper and Chester 91] [Krueger92] [Wolberg 83]. Reusable software is widely believed to be the key to the general overall increase in productivity in developing new software [Mareddy and Samadzadeh 95]. The development of new software almost always involves the deployment of previously existing software components in addition to developing new code fragments. Portability is one important aspect of reusability. Most software currently being developed is designed in a way to work on almost all the available platforms. This is a rather difficult task as the underlying architecture, hardware, or operating system varies from one platform to another.

Software reuse can be defined as the process of using existing components to create new programs [Karlsson 95]. A component can be defined as any software configuration item, such as a code module or documentation, that can be a candidate for reuse. The major factor that favors this concept of reuse is the efficiency and productivity of the programmers who are involved in the development of the target software [Frakes et al. 91].

Traditionally, the idea of porting of software has been approached with an air of skepticism [Wolberg 83]. Software that is developed for a particular system may contain

components which may not be essential when the same software is made to run on another system. Thus before trying to work on a porting problem, the developer needs to understand the purpose served by the software and determine how best to simulate the same running conditions in the new environment.

As the main objective of this thesis, a software component that was designed for Sun Workstations was modified to run on a personal computer. The software, Java Remote Method Invocation, was developed by Sun Microsystems, Inc., as part of its ongoing effort to develop techniques for accessing remote objects using the Java programming language. The thesis work involved integrating the various relevant executable code modules, and testing and evaluating the modified version on different personal computers.

The rest of the thesis report is organized as follows. Chapter II reviews the current literature on software reuse. It also defines some of the basic terminology, discusses the past and present efforts in software reuse, and provides a brief background on software porting. Chapter III discusses the software under consideration for porting, i.e., Java Remote Method Invocation. It involves giving a brief insight into the various classes that are used in the new version and how to put together all the classes. Chapter IV involves the implementation details. It describes the criterion of implementation and how to use the software on a personal computer. Chapter V summarizes the software testing done and tabulates the observations made. Chapter VI briefly discusses the work done and provides insight into some future work in this area. Appendix D contains the source code used for testing the software program.

## CHAPTER II

### LITERATURE REVIEW

#### 2.1 Introduction

This section contains a number of basic definitions of software porting, software reuse, and various other terminology that is used in the remaining sections of this thesis. Most of the definitions are taken from recent technical papers in the software engineering field and from the reference material that is available on the Internet.

Software reuse can be generally defined as utilizing an existing software artifact in building a new software system [Krueger 92]. A software artifact can be a code fragment, a specification module, a documentation item, a module-level implementation structure, a program design document, etc.

Software porting is the reuse of complete applications on new platforms [Mooney 93]. Portability is concerned with the act of producing an executable version of an existing software unit or system in a new environment. While reuse research has typically concentrated on building and maintaining collections of reusable components and reusing them effectively in new environment, portability is concerned with the reuse of complete applications on new platforms [Krueger 92] [Prieto-Diaz 93].

Abstraction refers to hiding the working details of source code and providing

only the pertinent information to the end user about the functionality of the code [Mareddy and Samadzadeh 95]. Abstraction is an important aspect of software reuse because other user might need to spend more time trying to understand the functionality of each software artifact while trying to correlate different artifacts in reuse.

Selection refers to choosing a particular artifact from a collection of artifacts in accordance to a reuser's need [Mareddy and Samadzadeh 95]. This is a major issue in most types of reuse. In porting there is no selection, the artifact is fully determined at the start [Mooney 93].

Specialization refers to adapting an artifact to suit the requirements of a specific use [Mareddy and Samadzadeh 95]. In many reuse situations, it is expected that the artifacts be parameterized to allow developers to make choices in both functionality and implementation, such as data representation and performance strategies [Mooney 93]. Specialization in such situations becomes limited to selecting the appropriate parameters.

In the case of porting, specialization is the heart of the problem [Mooney 93]. A well designed portable software may still contain some environment dependent variables that need to be changed. Specialization is then achieved by choosing an appropriate set of modules. A program ported at the source level is further specialized by processing it with a language processor compatible with the target environment [Mooney 93].

Integration refers to adding an existing software artifact to the system that is being developed. Module interconnection languages are examples of integration frameworks [Mareddy and Samadzadeh 95]. In traditional reuse strategies, this activity was straightforward as the artifacts were both conceived and constructed with integration in

mind [Mooney 93]. In porting, there is typically only one separate artifact. If specialization is achieved, integration poses no difficulty [Mooney 93].

## 2.2 Software Portability

### 2.2.1 History and Past Research Efforts

The goal of research in software portability is to facilitate reuse of existing applications in new environments [Mooney 93]. Recent reviews of reuse issues have characterized reuse very broadly as any case in which any artifact associated with a software system could be used in more than one situation [Krueger 92] [Prieto-Diaz 93]. Thus, software portability is clearly a form of reusability. However, according to Mooney, "the objectives of enhancing and supporting portability are not often addressed by reuse research" [Mooney 93].

Portability is concerned with the reuse of complete applications on new platforms, while reuse concentrates on building and maintaining collections of reusable components or similar artifacts, and reusing them in entirely new applications.

Portability is increasingly being identified as a desirable attribute of software systems [Mooney 93]. Most software which is intended for a wider utility will face the situation of being ported to new environments over the course of their lifetime [Lewis and Oman 90] [Sommerville 96]. Portability is often cited as a goal even for special purpose software categories [Mooney 93].

Despite the accepted need for better portability, published research on portability issues is meager. A variety of porting experiences have been reported, leading to many

types of anecdotal advice [LeCarme et al. 89]. A few limited areas, such as parallel software conforming to certain models, have been studied more extensively [Skillicorn 94] [Alverson and Notkin 93]. However, a systematic framework to guide developers in maximizing portability in the general software development process is not available [Mooney 93].

The Portability Research Group at West Virginia University is working on a variety of issues related to portability. This includes classification of applications for portability, specifying portability requirements, enhancing portability in the design process, measuring portability, and portability costs [Sitaraman 91] [Eichmann 92].

### 2.2.2 Portability Concepts

This subsection discusses some concepts related to portability, and introduces some commonly used terminology. Most of the ideas that follows are taken from the recent technical papers on software porting [Mooney 90] [Mooney 93] [Eichmann 92].

Porting, as mentioned before [subsection 2.1], is the act of producing a new, modified software based on an existing software version [Mooney 90]. A software unit can be an application program, a system program, or a single component of a program. A software system is a collection of software units.

The term environment refers to the range of elements in an installation that interact with the ported software. This typically includes a processor and an operating system, Input/Output devices, libraries, networks, etc. The term platform can also be used to refer to the context of an environment.



The term portability refers to the ability of a software unit to be ported to a given environment [Wolberg 83]. A program is portable if and to the degree that the cost of porting is less than the cost of redevelopment. A software unit is perfectly portable if the job can be done with zero cost, though this is rarely possible. Instead, a software unit can be characterized by its degree of portability, which is a function of the porting and development costs with respect to a specific environment [Mooney 90].

The principal types of portability usually considered are binary portability, which is porting the executable form, and source portability, which is porting the source language program or the source code [Mooney 93]. Binary portability is desirable but it is possible only for very similar environments. Source portability assumes the availability of the source code and facilitates the flexibility of porting to heterogeneous environments. Most porting projects assume a source portability approach.

### 2.2.3 Reuse Economy

The reason behind the interest in software reuse is the increase in the cost of developing complex software. A simple cost model proposed by John Gaffney (as cited in [Mareddy and Samadzadeh 95] and [Barnes et al. 87]) of the Software Productivity Consortium predicts that

$$C = (1-R) * L + b * R$$

where  $C$  is the cost of developing new software,  $R$  is the percentage of code reused,  $b$  is the cost of reusing a line of code, and  $L$  is the cost of developing a new line of code.

From the above relation, the cost will be less when more code is reused. Reuse is necessarily a characteristic of an entire organization [Abdel-Hamid 93] [Prieto-Diaz 91].

Thus the entire design process in developing a new project must be oriented toward finding and using existing components. If the components are not comprehensive or are not suited to an application, the cost of reuse tends to outweigh the benefits.

### 2.3 Data Transfer Mechanisms

The internet has emerged as an amorphous ocean of data stored in various formats on different hosts [Arthur 96]. During this infant stage, various data storage and transmission protocols have evolved to impose some order into this chaos. Different data transmission methods were developed for speedy access to data lying on some remote host. The fastest growing area of the net is the World Wide Web (WWW), which uses a hypertext-based markup system to navigate across data [Hannah 96].

The concept of hypertext, as described by Vannevar Bush in 1945 and evangelized by Theodore (Ted) Nelson in 1960, involves creating inter-document links across multiple host computers on the network [Java-Spec 96]. The first practical implementation of a network based hypertext system was created by Tim Berners-Lee at CERN, using the NEXTSTEP development environment which later was developed into the Hyper Text Markup Language (HTML), the Hyper Text Transport Protocol (HTTP), and the World Wide Web (WWW or W3) [Java-Spec 96].

Web browsers combine the functions of fetching data with figuring out what the data consists of and displaying the same if possible. One of the most prevalent file formats browsers is the Hyper Text Markup Language or HTML, which embeds simple text formatting commands within text [Hannah 96]. The main advantage of this concept

is the ability to use links to other HTML data either on the same host or elsewhere on the internet.

## 2.4 Remote Procedure Call

Since the Remote Method Invocation is a refinement of the original concept of Remote Procedure Call (RPC), it is apropos to discuss the implementation of RPC.

### 2.4.1 The Client-Server Model

According to Tanenbaum, “a client-server model is a network of diskless personal computers or workstations, referred to as clients, that communicate across a network with a file server which serves as a database for the clients” [Tanenbaum 92]. In this system, a request for accessing data is initiated by a client and is satisfied by the server. Communication is two-way, carried through in the form of request-reply, always initiated by the client and never by the server.

### 2.4.2 Concept of RPC

The RPC mechanism is based on the client server-model though the approach is from a different perspective [Tanenbaum 92]. In this view, a client sending a message to a server and getting a reply is like a program calling a procedure and getting a reply. In either case, the request is initiated by the caller who waits on the message sent by the callee. But the variance in this approach is the case when the called procedure is not on the local machine. In RPC, the calling program has no knowledge about the actual

location where the requested procedure is located. This can be abstracted by implementing RPC in a program and calling a remote procedure.

For instance, consider the case in which a local procedure called `get_info` is used by a client residing on a file server. This call can take as its arguments a file name, a buffer for storing the data, and a count to specify the number of bytes read [Tanenbaum 92]. A call such as

```
get_info (filename, buffer, count)
```

is a call to a local client procedure. This procedure in turn calls the file server for access to the data in the form of sending a message. Thus the client-server interaction is carried in the form of procedures rather than Input/Output or interrupts [Tanenbaum 92]. The details of how the network processes these requests can be hidden from the application program by placing them in local procedures, for instance `get_info` in the above case. These procedures are defined as *Stubs*. A remote procedure is abstracted as a local procedure by using *Stubs*.

### 2.4.3 Implementation of RPC

RPC is typically implemented in ten steps [Tanenbaum 92]. Step1 consists of the client program calling the stub procedure linked in its address space, in which the parameters are passed as in the case of a local procedure call. The client stub collects the parameters and packs them into a message, which is often termed *Parameter Marshaling*. Step2 consists of handling the message from the client stub to the transport entity. The message is transferred from the client side transport entity to the server side entity in Step3. The message is passed to the server side stub in Step4. The

Server stub then invokes the server procedure in Step5. Since this is abstracted as a local procedure call, the server procedure accepts the parameters passed by the server side stub. The result of the server procedure is returned in Step6. This result is packed by the server side stub and is passed to the server side transport entity in Step7. The message is passed to the client side entity in Step8, and the same is handed over to the client side stub in Step9. Finally, the stub returns the result of the server procedure to its caller, the client procedure in Step10.

Thus the main purpose of the whole mechanism is to facilitate a client to call a procedure on a remote server. The mechanism tends to be more transparent if the client has the abstraction that the result of its request is satisfied by the local procedure though the real work is done by the remote server.

#### 2.4.4 Drawbacks of RPC

The principal problem faced by Remote Procedure Call is Parameter Passing [Tanenbaum 92]. Integers, floating point numbers, and character strings pose no problems as the client stub can easily convert the same into a message and pass it to the server. Passing structures, records, or arrays is also direct.

The passing of pointers creates a discrepancy as the remote stub does not have access to the local address space, and thus the concept of RPC fails. In the case of local procedure calls, the local procedure loads the parameter specified at the given address space and any changes that are made are reflected in the address space. But for a remote procedure, updating of a local address space item is not feasible [Tanenbaum 92].

Another important front in which RPC fails is the fact that the utility of RPC is limited to the passing of parameters. In the advanced case of method passing across a network, RPC fails.

#### 2.4.5 Advances in Using RPC

Based on the failures of RPC, the problem of using reference parameters is reduced by eliminating the use of reference parameters, pointers, and procedure or function parameters on remote calls [Tanenbaum 92]. This tends to defeat the purpose as the implementation rules for local and remote procedure calls become different and the transparency is endangered.

One possible solution to this paradigm is the use of a call-by-copy/restore mechanism [Tanenbaum 92]. With this, the client stub finds the item being pointed to and passes it to the server stub. The server stub puts it in its local memory and passes a pointer to it to the server procedure. The server procedure is able to access the parameter and returns control to the stub. The server stub packs the modified parameter to the client stub, which in turn overwrites the parameter in its address space with the modified parameter. This method has some potential problems of parameter overwriting in the wrong sequence [Tanenbaum 92].

A different scheme has been developed in which the server and clients interact through a database system [Birrell and Nelson 84]. The server sends an initial message to the database system letting its identity be known through a string of ASCII characters, as well as its network address, and a random 32-bit integer. This registration is done by having the server call a procedure `export`, which is handled by the server stub. The

clients make the call to the server by first sending the server name to the database system and obtaining the network address and the unique identifier of the server. This operation is defined as `Binding`. Every transaction with the server is carried through using the unique identifier. This helps the client in finding out whether the server is alive. If the server crashes, it registers a new unique identifier with the database system. Thus the clients can rebind with the database system.

The above features are included in Java Remote Method Invocation (RMI) in which a central registry is maintained and the clients are bound with the servers. A crash on the server side forces a client to rebind. The initial drawback of the inability of passing methods in RPC is overcome in RMI.

## CHAPTER III

### JAVA REMOTE METHOD INVOCATION

#### 3.1 Introduction

Distributed systems require that computations running in distinct address spaces, potentially on different hosts, be able to communicate [Tanenbaum 92]. For a basic communication network, the Java language supports sockets, which are flexible and sufficient for general communication [Rmi-Spec 96]. However, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange. Thus the design of such protocols is cumbersome and error prone.

An alternative to sockets is the Remote Procedure Call (RPC), which abstracts the communication mechanism to procedure level. Instead of working on sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packed and transmitted to a remote target of the call [Tanenbaum 92].

RPC is not suitable for distributed object systems where communication between program level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed systems require Remote Method Invocation (RMI), where a local surrogate object (Stub) manages the invocation on a remote object [Rmi-Spec 96].



### 3.2 Java Distributed Object Model

A Java Distributed Object Model primarily consists of a Remote Object and Remote Interfaces. A Remote Object is one whose methods can be invoked from another Java Virtual Machine, running on a different host. An object of this type is usually defined by Remote Interfaces which are Java interfaces that define the methods of the Remote Object.

Remote Method Invocation (RMI) is the action of invoking a method of a remote interface on a remote object [Rmi-Spec 96]. The important aspect of RMI is that the syntax for method invocation on the remote object has the same syntax as a method invocation on a local object.

The main features of the Java Distributed Object Model can be described as follows [Rmi-Spec 96]:

- (i) A reference to a remote object can be passed as an argument or returned as a computation result in any method invocation (either local or remote).
- (ii) The built-in Java `instance of` operator can be used to verify the remote interfaces supported by a remote object.
- (iii) The clients of remote objects interact with the remote interface and not with the implementation classes of the remote interface.
- (iv) A remote object is passed by reference.
- (v) Arguments to and results from a remote method invocation are passed by copy rather than by reference.

### 3.2.1 RMI Interfaces and Classes

The relation between interfaces and classes is depicted in Figure 3-1.

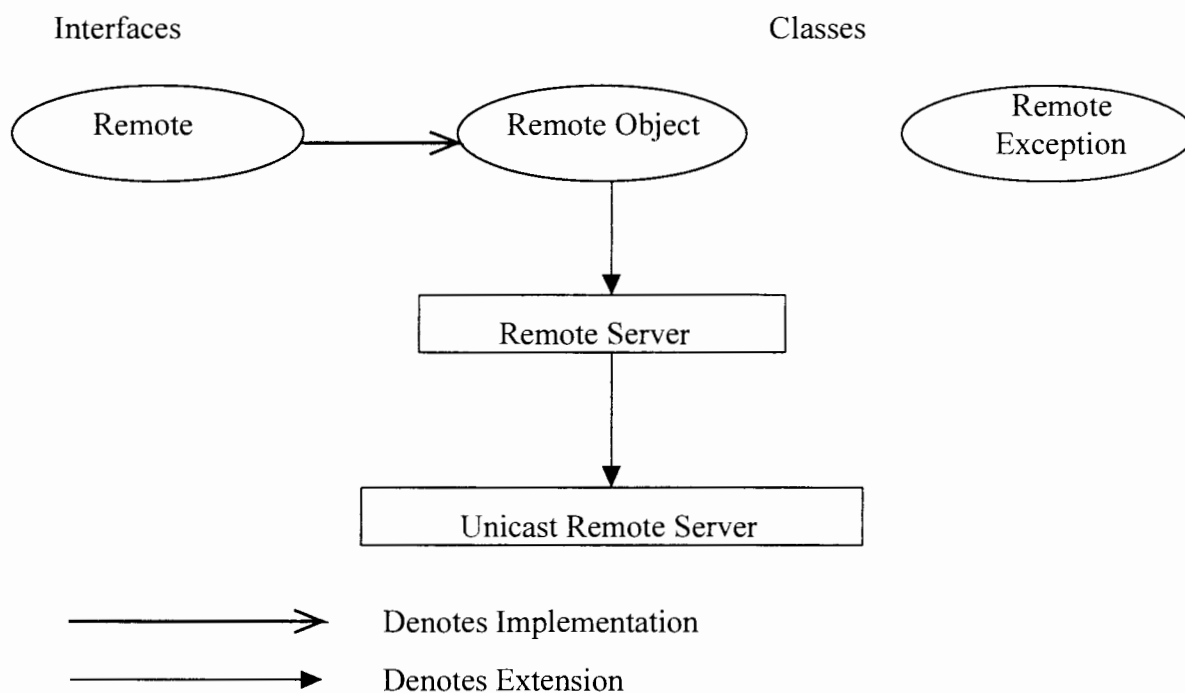


Figure 3-1. Interface and Class Relationships in RMI (source: [Rmi-Spec 96])

In the above figure, the object being pointed to by the blocked arrow is an extension of the object from which the arrow is originating. The object that is being pointed to by the other arrow implements the methods defined in the object from which the arrow originates.

3.2.1.1 Remote Interface All remote interfaces extend, either directly or indirectly, the Java `remote` interface. The `remote` interface defines no methods and is initialized as follows [Rmi-Spec 96]:

```
interface Remote {}
```

For example, the following code fragment defines a remote interface for a common Linked List problem that contains methods for inserting data, deleting data, and copying data [Rmi-Spec 96].

```
public interface linked_list extends remote
{
    public void insert (int data)
        throws RemoteException;
    public void delete (int data)
        throws RemoteException;
    public void copy (int data)
        throws RemoteException;
}
```

Each method defined in the remote interface needs to throw a `RemoteException` as one part of its implementation. It can throw multiple exceptions at the same time like `Null Pointer Exception` and `Input/Output Exception` [Rmi-Spec 96].

3.2.1.2 Remote Exception Class The Remote Exception class can be broadly classified as the super class of all exceptions in the RMI system [Rmi-Spec 96]. Each method declared in the remote interface throws a remote exception in its throws clause. The remote exception occurs when the Remote Method Invocation fails (owing to a network failure or when the server does not responds to a client call or, in the worst case, when the requested method is not found on the specified host). This allows the application making the remote method invocation to determine how to cope with the remote exception [Rmi-Spec 96].

3.2.1.3 Remote Object Class The Remote Object Class provides the semantics of objects for implementing methods for `equals`, `toString`, and `hashCode` [Rmi - Spec 96]. The `equals` method compares whether the references to two objects are equal. The `toString` method returns a string which represents a reference to an object. The information includes the hostname and the port number of the referred object. The `hashCode` method returns the same value for all remote references that refer to the same underlying remote object. It is an extension of the Java language `hashCode`, where references to the same object are considered to be equal [Rmi-Spec 96].

3.2.1.4 Remote Server The routines needed to create objects and export them (make them available remotely) are carried out by the remote server [Rmi-Spec 96]. The subclasses of the remote server identify the remote reference semantics, for instance the server is a single object or a replicated object requiring communication with multiple locations.

3.2.1.5 Unicast Remote Server It defines a non-replicated remote object whose references are valid only when the server is alive. The routine `EchoImpl.java` in Appendix D is an illustration for this type of server.

### 3.2.2 Implementing a Remote Interface

Any class that implements a Remote Interface needs to observe the following set of rules [Rmi-Spec 96]:

- (i) The class must extend the `Unicast RemoteServer` so that it inherits the remote behavior that is provided by the `RemoteObject` and `RemoteServer` classes.

- (ii) The class can implement multiple remote interfaces at the same instance. This facilitates multiple clients from different locations to use different interfaces to look up the methods of the remote object.
- (iii) The class can also extend another remote implementation class that is compatible to the Unicast RemoteServer.
- (iv) The class can define other methods which do not appear in the remote method, but these methods cannot be remotely invoked.

For instance, the same Linked List example can be defined as follows [Rmi-Spec 96]:

```
package linked_list_package;
import java.rmi.*;
import java.rmi.server.UnicastRemoteServer;
public class linked_list_impl extends UnicastRemoteServer
    implements linked_list
{
    public void insert (int data)
        throws RemoteException{
        .....
    }
    public void delete (int data)
        throws RemoteException {
        .....
    }
    public void copy (int data)
        throws RemoteException {
        .....
    }
}
```

The Linked List package might contain other methods such as `print()`, `delete_front()`, and `insert_front()` which are not defined in the remote interface and hence they cannot be invoked remotely.

### 3.2.3 Parameter Passing in Remote Method Invocation

Passing of parameters can be of three basic types, namely Java data types, Non-remote objects, and Remote objects [Rmi-Spec 96]. This may lead to a failure of the remote method invocation with an exception.

3.2.3.1 Passing Non-remote Objects Non-remote objects, either passed as an argument for a remote method invocation, or obtained as a result of a remote method invocation, are passed by `copy` [Rmi-Spec 96]. That is, when a non-remote object is figured out in a remote method invocation, a copy of it is made before the call is invoked. In the same way, when a non-remote object is returned as a result of remote method invocation, a new object is created in the calling machine.

3.2.3.2 Passing Remote Objects When passing a remote object, only the stub for the remote object is passed. Thus on receipt of a remote object, only the remote interfaces that are implemented by the object are available for usage [Rmi-Spec 96]. Any local interfaces that the remote object might have implemented are not usable. Trying to use the local interfaces leads to run-time exception.

### 3.2.4 Exception Handling in Remote Method Invocation

Exceptions in Remote Method Invocation are derived from the `RemoteException` superclass. Hence any exception that might result in the process of invoking a method of a remote object might include the exception messages obtained as a result of the application along with those thrown due to a remote exception [Rmi-Spec 96]. The remote exceptions include failure during the process of invoking, before invoking, or after invoking a remote method of a remote object. To alleviate the problem, system messages are introduced in the remote interfaces and the calling methods to pinpoint the exact exception that is responsible for a failure [Rmi-Spec 96].

### 3.2.5 Locating Remote Objects

For a client to invoke a method on a remote object, the client must obtain a reference to the object. This can be achieved as a return value in a method call. The `java.rmi.Naming` interface provides Uniform Resource Locator (URL) based methods to look up, bind, rebind, unbind, and list the name and object pairings maintained on a specific host and port [Rmi-Spec 96].

For instance, to bind and look up for the Linked List example can be done as

[Rmi-Spec 96]:

```
linked_list list = new linked_list_impl();
String url = "rmi://cishlpdsk3/link";
                                // Create a Reference (URL)
// Bind the Url to the remote object.
java.rmi.Naming.bind(url,list);
    ...
```

```
// Look up for the list (obtain a reference).
list = (linked_list)java.rmi.Naming.lookup(url);
```

### 3.3 An Overview of RMI Architecture

The Remote Method Invocation (RMI) system consists of three layers: the Stub/Skeleton layer, the Remote Reference layer, and the Transport layer. The perimeter of each layer is independent of its corresponding sub-layer/super-layer. This leads to an opportunity of implementing a part of a layer by any other alternative. For example, the transport implementation used in this version is TCP-based (which uses Java sockets), but a transport based on other transmission protocol can also be used without replacing either the stub/skeleton layer or the remote reference layer. Figure 3-2 shows the relationship between the different hierarchies of the RMI system.

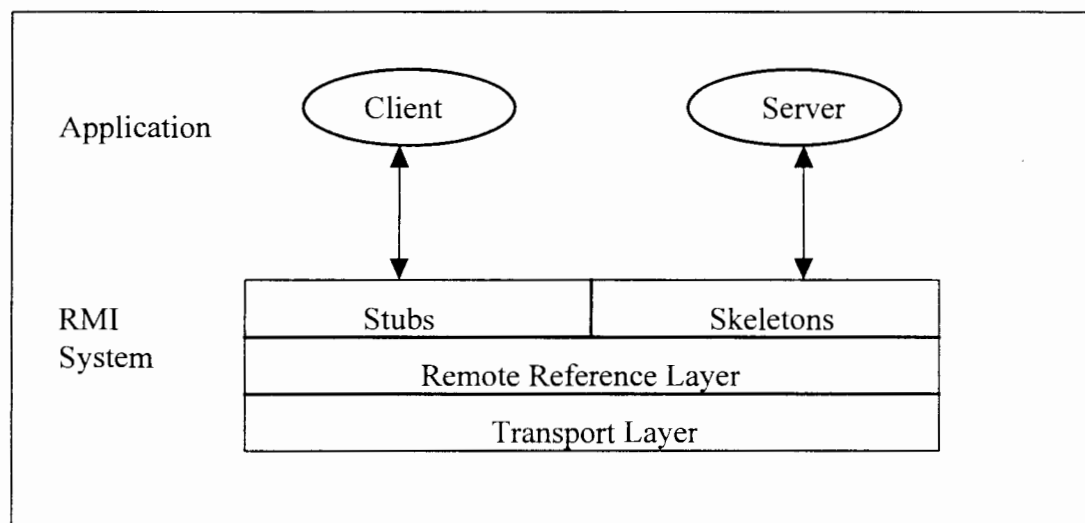


Figure3-2. Overview of the RMI system architecture (source: [Rmi-Spec 96])



The Stub/Skeleton layer essentially consists of client side stubs (proxies) and server side skeletons.

The Remote reference layer deals with the remote reference behavior of objects (e.g., invocation to a single object or to a replicated object).

The Transport Layer is concerned with the connection set-up, and the management and tracking of the remote objects. The application layer sits on top of the RMI system.

A remote method invocation from a client to a remote server object travels down through the layers of the RMI system (which includes the Stub layer and the client side Remote Reference layer) down to the client side, the Transport layer, then up to the server side Transport layer to the server (passing through the server side Remote Reference layer and the Skeleton layer).

A client invoking a method on a remote object, actually creates a local stub or proxy for the remote object and makes use of it as a means of communication with the remote object [Rmi-Spec 96]. A client reference to a remote object is actually a reference to a local stub. This stub is implemented by using the remote interface of the remote server object and forwards all invocation requests to the remote object using the Remote Reference layer.

The semantics of the remote method invocation are carried through the Remote Reference layer [Rmi-Spec 96]. For instance it is the responsibility of the remote reference layer to determine whether the server is a single object or is a replicated object needing communications with multiple locations. In the latter case, each replicated object communicates with the remote reference layer for choosing its own semantics.

The Remote Reference layer is responsible for reference semantics for the server. The remote reference layer, for example, defines the way for referring to objects that are implemented in:

- (i) Servers that are always running on a machine
- (ii) Servers that run only when some method invocation is made on them.

These abstractions are not seen in layers above the remote reference layer.

The Transport layer is concerned with setting up network connection, managing the connection, and keeping track of remote objects residing in the transport's address space.

### 3.3.1 Stub/Skeleton Layer

The Stub/Skeleton layer is the interface between the application and the rest of the RMI system. This layer does not have any information about the specifics of the transport, but communicates the data to the remote reference layer. The data is transferred via the abstraction of `Marshal streams`. `Marshal streams` employ a mechanism, called `Object Serialization`, which enables java objects to be transmitted between address spaces [Rmi-Spec 96]. Objects transmitted using the serialization mechanism are passed by copy to the remote address space, unless they are remote objects, in which case they are passed by reference.

A Stub for a remote object is the client side proxy for the remote object. Such a stub implements all the interfaces that are implemented by the server side remote object. The client side stub is responsible for initiating a call to the remote object (by sending a

request to the remote reference layer), marshaling objects to a marshal stream (obtained from the remote reference layer), informing the remote reference layer that a call needs to be invoked, unmarshaling the return value or exception from a marshal stream, and informing the remote interface layer that the call is complete [Rmi-Spec 96].

A Skeleton is a server side entity for the remote object that contains a method which dispatches calls to the actual remote object. The skeleton carries out the Unmarshaling arguments from the marshal stream, making the call to the actual remote object implementation, and marshaling the output of the call (including a possible exception) to the *marshal stream* [Rmi-Spec 96].

The Stub and Skeleton classes are determined at the run-time and are dynamically loaded as required. This is referred to as Dynamic Stub Loading.

3.3.1.1 Dynamic Stub Loading In Remote Procedure Calls (RPC), a client code needs to be linked either statically or dynamically during run-time via dynamic link libraries (*dll's*). In either case, specific compiled code needs to be available to handle RPC. RMI generalizes the above technique by loading the exact stub code (in the *bytecode* format) at run-time to handle method invocations on a remote object. This is called as Dynamic Stub Loading which uses the Java mechanism for downloading code either from the local file system or from the network [Rmi-Spec 96]. Dynamic Stub Loading is used only when the code for a needed stub is not readily available. When a remote object reference is passed as a parameter, the marshal stream that transmits the reference includes the information as to where the stub class for the remote object can be loaded, provided the Uniform Resource Locator is known.

### 3.3.2 Remote Reference Layer

The remote reference layer deals with the lower level transport interface. This layer carries out specific remote reference protocol which is independent of the client stubs and server skeletons [Rmi-Spec 96].

Various protocols that can be carried using Java in the remote reference layer include, unicast Point-to-Point Protocol (PPP), invocation to replicated object groups, support for persistence reference to the remote object (which enables activation of the remote object), and reconnection strategies (the case when remote object becomes inaccessible) [Rmi-Spec 96].

The remote reference layer essentially consists of two components, Client-side Component and the Server-side Component, which co-operate with each other.

3.3.2.1 Client-side Component This component contains information specific to the remote server (or servers in the case of a replicated object) and communicates via the transport layer to the server-side component. During each method invocation, the client and server-side components exchange specific remote reference semantics. For instance, if a remote object is a part of a replicated object, the client-side can forward a request to each replica rather than to a single remote object.

3.3.2.2 Server-side Component This component implements the specific remote reference semantics prior to delivering a remote method invocation to the skeleton. This component, as an example, can handle ensuring of unique multiple delivery by communicating with other servers in the replica group.

The remote reference layer transmits data to the Transport Layer.

### 3.3.3 Transport Layer

The transport layer takes care of the implementation details of connections. The transport layer of RMI is responsible for [Rmi-Spec 96]:

- (i) Setting up connections to the remote address space.
- (ii) Managing connections.
- (iii) Monitoring connection “liveness”.
- (iv) Listening for incoming calls.
- (v) Maintaining a history of remote objects that reside in the address space.
- (vi) Setting up a connection for an incoming call.
- (vii) Finding the dispatcher for the target of the remote call and passing the connection to this dispatcher.

The transport layer of the RMI system consists of four basic abstractions, Endpoint, Channel, Connection, and Transport [Rmi-Spec 96]. These abstractions are briefly explained below.

EndPoint It is an abstraction used to denote an address space or a Java virtual machine. In the implementation, an endpoint can be mapped to its transport. That implies, given an endpoint, that a specific transport instance can be obtained.

Channel It is an abstraction for a link or conduit between two address spaces. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel.

Connection Is the abstraction for transferring data (input/output).

Transport Manages channels. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces, the local address space and a remote address space. Given an endpoint, the transport sets up a channel to that address space. The transport abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system.

## CHAPTER IV

### IMPLEMENTATION ISSUES

#### 4.1 Implementation Platform and Environment

The testing of the program was done on IBM compatible personal computers running the Microsoft Windows95 operating system. Microsoft Windows95 is the latest version of the windows operating systems developed by Microsoft, Inc., for IBM compatible personal computers. Though Windows95 does not support the MS-DOS operating system, there is a provision in the start menu for the user to switch back and forth between MS-DOS and MSWindows95.

##### 4.1.1 Present Day Personal Computers

Currently the software industry has two completely different standards for personal computers: the IBM compatible personal computers that were developed originally by the International Business Machines, Inc., and the Macintosh personal computers developed by the Apple Computers, Inc. The latest additions to the personal computer family include the PowerMacs and PowerPCs based on the Reduced Instruction Set Computer (RISC) processors jointly developed by IBM, Apple, and Motorola. Digital Equipment Corporation (DEC) and Hewlett Packard (HP) are also in the with DEC developing its DEC 3000 Model and with HP developing its HP 9000/755, in response to

IBM 580 [DEC 96].

#### 4.1.2 Windows Environment

Microsoft Windows is a graphical operating environment designed to run on top of the Microsoft Disk Operating System (MS-DOS). The initial versions (MSWindows1.x and MSWindows2.x) of the software failed to achieve the desired success mainly due to lack of supporting hardware during their release time. Version 3.0 followed by Versions 3.1 and 3.1.1 had good reception as they supported multi-media and Object Linking and Embedding (OLE).

Inspired by the success of Versions 3.1 and 3.1.1, Microsoft released WindowsNT, which is a high-end operating system and has MS-DOS as its underlying operating system. The latest release from Microsoft is Windows95. This is like a regular operating system but has MS-DOS running in the background. Windows95 comes with some enhanced features such as custom networking, security, and internet support (through Microsoft Internet Explorer). The user interface is a blend of the previous windows operating environment and that of Apple Macintosh. Applications that use MS-DOS are not supported by Windows95 as Windows95 has its own helper applications.

#### 4.2 Java Programming Environment

The rapid development of Internet and World Wide Web (WWW) has led to a new way of developing and distributing software. Any software that is currently developed is designed in a way to run on almost all available platforms. But the context



of heterogeneous operating systems and underlying architectures tends to defeat the purpose.

Java was developed by Sun Microsystems Inc., as a part of their ongoing research to cater to the need of creating a new language which can satisfy the current market demands for solving some of the above said problems. Java has the following basic features [Arthur 96] [Walrath and Campione 96]:

- (i) It is architecture neutral. Thus any program written in Java can be run on almost all of the existing architectures, though application-specific programs need to be modified.
- (ii) It is best known for its security. Java provides extensive compile time checking followed by a second level of run time checking. This is a useful feature while developing network applications where the invasion of file systems and spread of virus is more prevalent.
- (iii) It is simple and object oriented. Java is considered to be a refinement of the C++ programming language. Hence developing new applications is easier as a programmer needs to manipulate existing libraries which range from Input/Output functions to network and graphical user interface toolkits.
- (iv) Java is dynamic. Though the Java compiler is strict during compile time static checking, the language and run time system are dynamic in their linking stages. Classes are linked only as needed. Code modules can be linked across different sources and even through the network.
- (v) The Java interpreter can execute bytecodes directly on any machine on which the interpreter and the run time system are installed. Java's multithreading capability

provides the means to build applications with many concurrent threads of activity, thus resulting in a high degree of interactivity from the end user.

The most common Java programs are applications and applets. *Applications* are stand alone programs, and run on their own without the assistance of any browser. *Applets* are similar to applications, but they don't run on their own. Instead, applets conform to a set of standard rules that let them run only within a Java compatible browser. Applets can be embedded into an `html` file and can be viewed using an appletviewer. Applications can be compiled using a Java compiler and the compiled `.class` files can be used as a source for an applet, and the execution of the program can be visualized using an appletviewer. The Java Developers Kit is a freeware software from Sun Microsystems Inc., that contains the java compiler, interpreters, and appletviewers. Currently, Java is released for Sparc Solaris, Windows95, WindowsNT, and Macintosh. Since Java runs only on 32-bit instruction set machines, the earlier versions of MSWindows are not compatible with Java.

#### 4.3 Implementation Details

The work was mainly carried out on IBM compatible personal computers. All the computers that were used for testing and running the program were loaded with Microsoft Windows95. The RMI software runs in parallel with the Java Developers Kit (JDK), so JDK Version 1.0.2, which is a freeware was also used. Initially, the user needs to make sure that the hard drive has enough memory to hold JDK and RMI, which together occupy almost three Mega Bytes of memory. The Java Developers Kit is available in a zipped format. After unzipping it creates Java as the parent directory with four sub

directories `bin`, `lib`, `include`, and `demo`. The `bin` directory has the `javac` compiler for personal computers and the interpreters `javaw` and `javap`, as well as the `appletviewer`. One of the network drives (the H drive) of the Oklahoma State University Computing and Information Services Department File server (UCC-FS1) was used for installing all the components of RMI and JDK.

RMI has a parent directory named `RMI`, and has three other subdirectories named `bin`, `lib`, and `examples`. The `bin` directory contains the `rmic` stub compiler and also the `.dll` files for creating applet windows for personal computers. The `lib` directory contains the `rmi` properties. The `examples` directory contains the test example. The `example` subdirectory contains another subdirectory named `echo`, which has all the source files that are used for checking the demo. The compiled classes are kept in a subdirectory named `test` from which all the `.class` files are loaded when running the example. The original example, `Echo`, is written for Sun workstations and the testing was done in windows along with some of the new features that are included.

Any application written using RMI needs to have three basic interfaces, the client interfaces, the server interfaces, and the registry interface. Figure 4-1 shows the relation between the different interfaces.

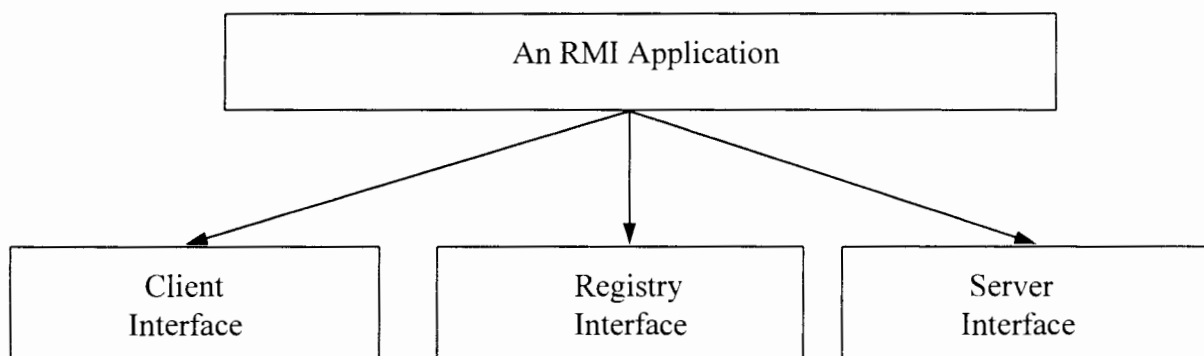


Figure 4-1. Overview of RMI Interfaces (source: [Rmi-Spec 96])

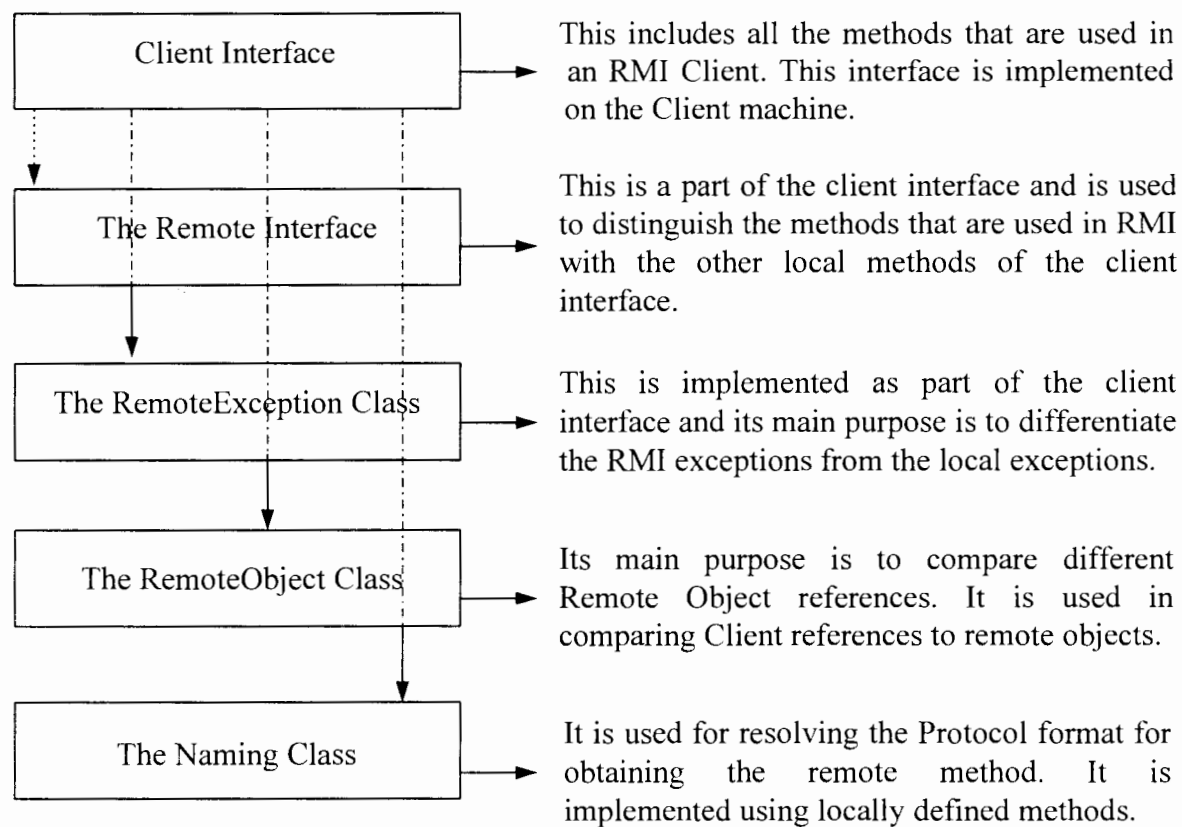


Figure 4-2. Overview of RMI Interfaces (source: [Rmi-Spec 96])

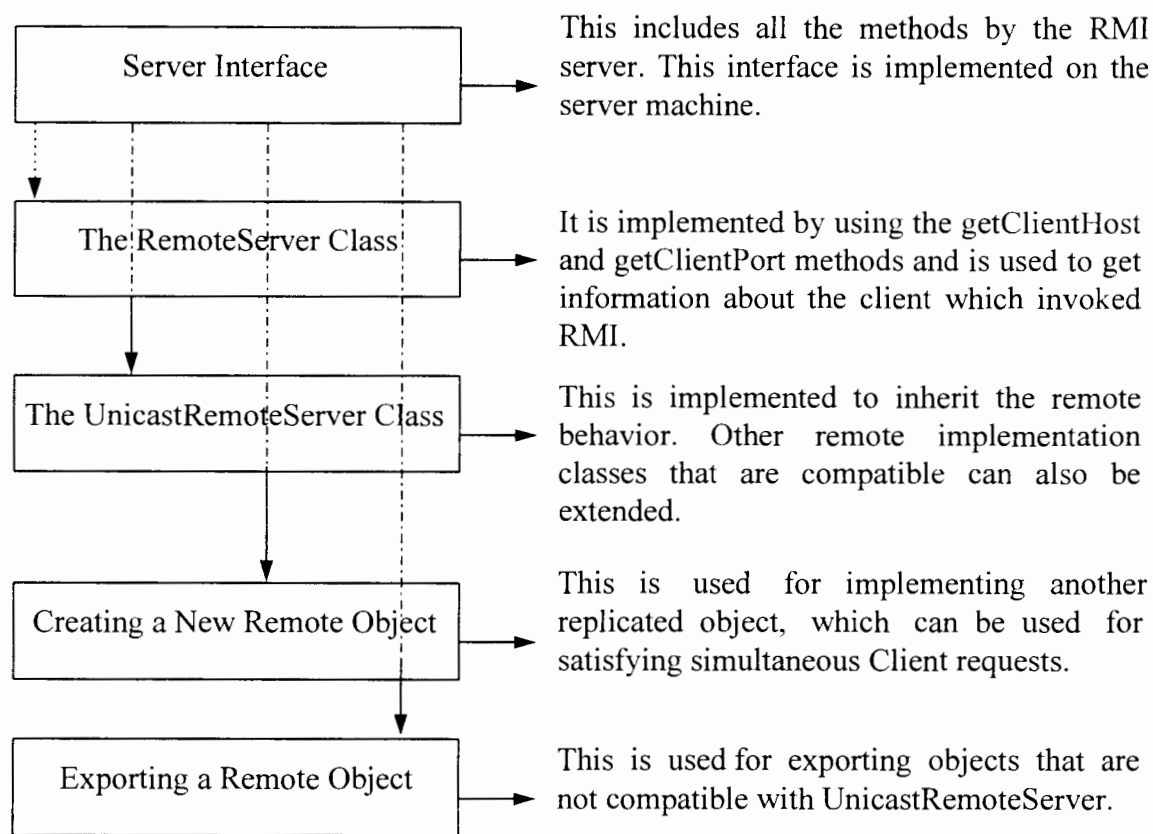


Figure 4-3. Overview of RMI Interfaces (source: [Rmi-Spec 96]) (continued)

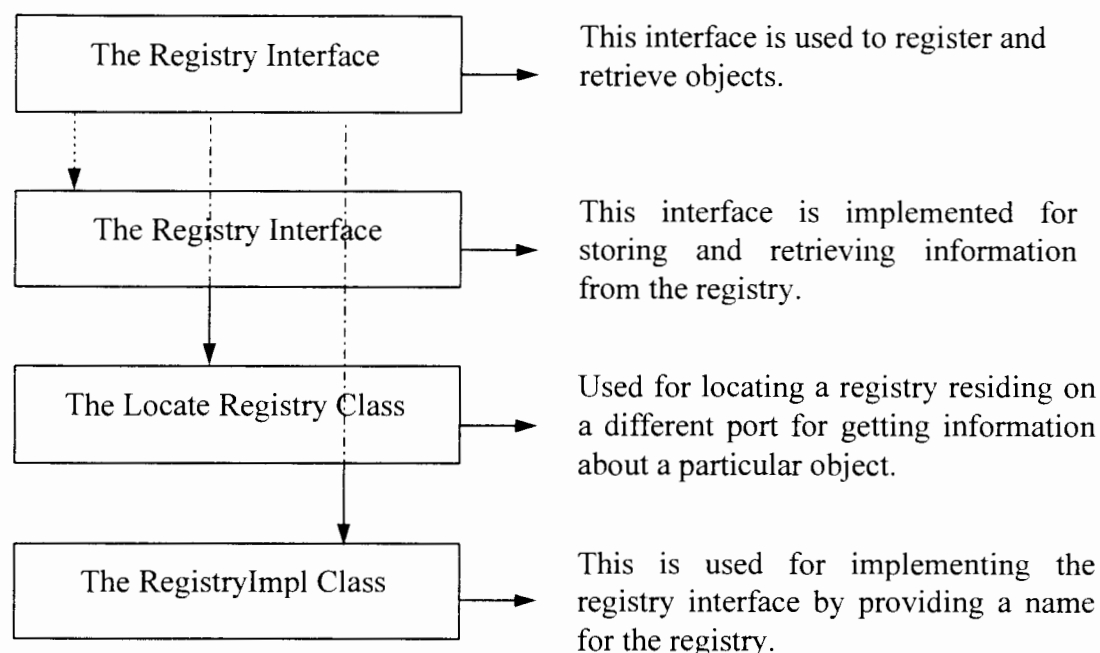


Figure 4-4. Overview of RMI Interfaces (source: [Rmi-Spec 96])

In the above figures, i.e., Figure 4-1 through Figure 4-4, the side text explains the functionality of the different components of each interface. Each component that forms a part of an interface is denoted by a dashed arrow from the corresponding interface. The flow graph is an overview of how each interface component is associated with another, and envisages an overview of the important interfaces and their components of an RMI application.

#### 4.3.1 Client Interfaces

4.3.1.1 Remote Interface The remote interface is used to identify all remote objects. Any object that is a remote object must implement this interface directly or indirectly [Rmi-Spec 96].

```

package java.rmi;
public interface remote {
    ...          // Tags the included methods as
}              // remote

```

All methods that are used in accessing and exchanging information with a remote server should be declared remote in the client interface, so that they would be differentiated from other local methods which are not supported during Remote Method Invocation.

4.3.1.2 RemoteException Class All remote exception classes are a subclass of `java.rmi.RemoteException`. This enables interfaces to distinguish local exceptions from remote exceptions [Rmi-Spec 96].

```

public class RemoteException
    extends java.lang.Exception
// This declaration tags the exceptions that are defined as
// remote, thus distinguishing them from local exceptions.
{
    public RemoteException (String s);
    public RemoteException (String S, Exception e);
}

```

4.3.1.3 RemoteObject Class This class implements the `java.lang.Object` behavior for remote objects. The `hashCode` method is an extension of the Java language `hashCode`, where references to the same object are considered to be equal. The `equals` method is implemented to allow remote object references to be stored in hashtables to be compared. The `equals` method returns true if two references are made to the same remote object. The `toString` method returns a string that defines the remote object [Rmi-Spec 96].

```

package java.rmi.server;

```

```
public abstract class RemoteObject implements Remote
{
    public int hashCode();
    public boolean equals(Object obj);
    public String toString();
}
```

4.3.1.4 NamingClass The Naming interface allows remote objects to be retrieved and defined using the Uniform Resource Locator (URL) syntax. The URL usually consists of a protocol, host, port, and name fields. The Registry service on the specified host and port is used to perform the specified operation. The protocol should be specified like `rmi://host/test_method`, where `test_method` is the object bound with the *remote* interface [Rmi-Spec 96].

```
package java.rmi;
public class Naming {
    public static Remote lookup(String url)
        throws RemoteException, NotBoundException
        AccessException, UnknownHostException;
    public static void bind(String url, Remote obj)
        throws RemoteException, NotBoundException
        AccessException, UnknownHostException;
    public static void rebind(String url, Remote obj)
        throws RemoteException, NotBoundException
        AccessException, UnknownHostException;
    public static void unbind(String url)
        throws RemoteException, NotBoundException
        AccessException, UnknownHostException;
    public static String[] list(String url)
        throws RemoteException, AccessException,
        UnknownHostException;
}
```



The `lookup` method returns the remote interface associated with the file portion of the name, which in the above test protocol is `test_method`. The `NotBoundException` is thrown if the name has not been bound to an object. The `bind` method binds the specified name to the remote object. It throws `AlreadyBoundException` if the name is already bound to another object. The `rebind` method always binds the name to the object even though the name is already bound. The previous binding is lost in this case. The `unbind` method in the same way releases a binding between a name and the remote object. It throws the `NotBoundException` if there was no binding. The `list` method returns an array of strings containing information about the URLs bound in the Registry [Rmi-Spec 96].

#### 4.3.2 Server Interfaces

When implementing the server, the client interfaces need to be available [Rmi-Spec 96]. These interfaces must be able to be extended to allow for the creation, definition, and export of remote objects. The server interfaces consist of four major classes which extend `RemoteObject` and `RemoteServer`. The rule of thumb is that a client interface should be existing when a server interface is started, otherwise a `ServerNotActiveException` is thrown.

4.3.2.1 Remote Server Class The remote server class is the superclass for all types of server implementations. It provides the framework to support a wide range of remote reference semantics [Rmi-Spec 96].

```
package java.rmi.server;
```

```

abstract class RemoteServer extends RemoteObject
{
    static String getClientHost()
        throws ServerNotActiveException;
    // This is a Java method that is used for obtaining
    // information about a specific client which invoked an RMI.
    static int getClientPort()
        throws ServerNotActiveException;
    // This provides the port information about the client which
    // invoked the RMI.
}

```

The two methods `getClientHost` and `getClientPort` allow the active method to find the host and the port that made the remote method active in the current thread. If no remote method is active, the `ServerNotActiveException` is thrown.

4.3.2.2 UnicastRemoteServer Class The `UnicastRemoteServer` class provides point-to-point active object references using TCP streams. Thus, the TCP connection based transport is used and references that are made are valid only for the lifetime of the process that creates the remote object [Rmi-Spec 96].

```

package java.rmi.server;
public class UnicastRemoteServer extends RemoteServer {
    // Create new object whose life time is the process
    // life time.
    public UnicastRemoteServer()
        throws RemoteException;
    // Create a new remote object on the specified port.
    public UnicastRemoteServer(int port)
        throws RemoteException;
    // Export a remote object.
    public static RemoteStub exportObject(Remote obj)
        throws RemoteException;
    // Export a remote object on a specified port.

```

```
public static RemoteStub exportObject(Remote obj, int
                                        port)
    throws RemoteException;
}
```

4.3.2.3 Creating a New Remote Object Remote objects are created by the server application. When these remote classes extend the `UnicastRemoteServer`, two constructors are available to create and export the new remote objects. The first constructor creates the remote object on the server port and the second constructor exports the remote object to the specified port.

4.3.2.4 Exporting a Remote Object Two methods, `exportObject(Remote obj)` and `exportObject(Remote obj, int port)`, are available to export remote objects that are not implemented by extending `UnicastRemoteServer`. The `exportObject` is called with the object on the default port. The second form is used to export object on the specified port number. The return value of `exportObject` is a stub object that clients will use to refer to the remote object.

In remote method calls, always the stub object is passed as the return value. If the remote object itself is passed, a lookup is performed to find the stub for the remote object, and is passed as the result of the remote call.

### 4.3.3 The Registry Interfaces

The registry interfaces are used to register and retrieve objects by simple names. Any server can support its own registry or a simple registry can be used for a host. The interface consists of the `LocateRegistry` and `RegistryImpl` classes.

The `java.rmi.Naming` interface uses the registry interface to provide URL based naming.

4.3.3.1 Registry Interface The Registry interface provides methods for lookup, binding, rebinding, unbinding, and listing the contents of a registry. The general syntax for the registry interface is [Rmi-Spec 96]:

```
package java.rmi.registry;
public interface Registry extends Remote {
    public Remote lookup(String name)
        throws RemoteException, NotBoundException,
            AccessException;
    // This method is used for looking up a remote server in the
    // registry by means of the server name provided as a
    // string.
    public void bind(String name, Remote obj)
        throws RemoteException, AlreadyBoundException,
            AccessException;
    // This method is defined to bind a specific remote object
    // with a given string for future references.
    public void rebind(String name, Remote obj)
        throws Remote Exception, AccessException;
    // This method is used for rebinding a remote object with a
    // different string.
    public void unbind(String name)
        throws RemoteException, NotBoundException,
            AccessException;
    // This method unbinds a remote object from its associated
    // name.
    public String[] list()
        throws RemoteException, AccessException;
    // This method returns a list of strings that contain
    // information about the URL bound in the registry.
}
```

4.3.3.2 Locate Registry Class This class contains static methods that retrieve a registry on the current host, current host at a specified port, a specified host, or at a particular port on a specified host [Rmi-Spec 96].

```
package java.rmi.registry;
public class LocateRegistry {
    public static Registry getRegistry()
        throws RemoteException;
    public static Registry getRegistry(int port)
        throws RemoteException;
    public static Registry getRegistry(String host)
        throws RemoteException, UnknownHostException;

    public static Registry getRegistry(String host, int port)
        throws RemoteException, UnknownHostException;
}
```

4.3.3.3 RegistryImpl Class This class implements the Registry interface with a simple naming syntax. The name and remote object bindings are not remembered across server restarts. In other words, the bind, unbind, and rebind methods are allowed only from clients on the same host as the server [Rmi-Spec 96].

```
package java.rmi.registry;
public class RegistryImpl extends
    java.rmi.server.UnicastRemoteServer implements Registry
{
    public RegistryImpl()
        throws RemoteException;
    public RegistryImpl(int port)
        throws RemoteException;
    public static void main(String args[]);
}
```

A `RegistryImpl` can be created using the default port or by using a specified port. For stand-alone applications, `RegistryImpl` defines a main method to which the port number can be passed as an argument. The main method also sets the security manager to check the loading of unnecessary remote objects into the process. Then it creates a `RegistryImpl` on the specified port.

Clients access the registry with the `LocateRegistry` class and the `Registry` interfaces.

#### 4.4 Usage Details

The program requires the following: Microsoft Windows95 installed on the personal computer, Network chord attached to the personal computer, and Java Developers Kit 1.0.2 installed.

The main software program has two directories, `rmi` and `java`. The directory `java` has the Java Developers Kit 1.0.2 installed. It is used for compiling source files and using the `appletviewer` for executing html files. The directory `rmi` has the RMI software loaded. The software was tested using an `Echo` example to test some of the new features. The user interface for the original program was changed to include some enhanced features such as estimating the total time taken to find out a remote host and finding the method desired on the remote host. This is achieved using the `java.lang.Thread` class. The thread runs in parallel with the main program. The precision of the timer is one millisecond. For implementing this feature, the personal computer must support multi-threading. Since installation of Windows95 is set as a basic criterion, the personal computer should be supporting multi-threaded programs (as Windows95 is an example of a multi-threaded program).

There are two batch files, load.bat and start.bat. These files contain the necessary setup procedures needed before attempting to test the example program. All the source files are in the subdirectories included in rmi. The two batch files are listed in Appendix D.

To run the executable file, make changes in the file load.bat as follows:

if the drive used is say the C drive, then the classpath line should something like:

```
SET CLASSPATH=c:\rmi\examples\echo\test;c:\rmi\lib\rmi.zip
```

```
SET RMIHOME=c:\rmi
```

```
PATH=c:\java\bin;c:\rmi\bin;c:\windows\command;PATH%;
```

the rest of the commands are the same. Essentially, change the drive from H:\, as given in the file, to the drive under use.

When running the batch file, if any of the statements gives the error "Out Of Environment Space", this message means that the available environment is insufficient to hold the new variable definition. The suggested solution is to increase the environment space by including the command:

```
shell=c:\windows\command.com /e:1024
```

in the config.sys file of the computer and reboot the machine.

In order to run the echo example on two machines:

- (i) run "javaw java.rmi.registry.RegistryImpl" on hostA
- (ii) run "javaw java.rmi.examples.echo.EchoImpl" also on hostA
- (iii) run "appletviewer rmi/examples/echo/index.html" on hostB

the address to be typed in the URL field of hostB to look up an echo object located on hostA should be:

```
rmi://hostA/Echo1
```

```
rmi://hostA/Echo2
```

When running the echo example on one machine, the URL that needs to be typed is `rmi://hostA/Echo1`, provided the host also runs the server.

There are five buttons, `Look`, `Inv`, `Reset`, `Default`, and `Status`.

`Look` is used to find the remote host given in the `Hostfield`.

`Inv` is used to invoke the `Echo` call on the remote host.

`Reset` is used to disconnect the remote host.

`Default` sets the host to the local host, i.e., the host name of the machine used.

`Status` gives an indication of which host methods are used at a given instance.

When the `Inv` button is used, the `Clock` thread is activated and there is display of the clock counter on the user window. This gives the estimated time taken to connect to the mentioned site.

The `Echo` program implements five echo servers on every host it runs. So the syntax of the hostname is thus: `rmi://hostname/Echo1` through `rmi://hostname/Echo5`.

The new source files for the `Echo` program are in the subdirectory, `rmi/examples/echo`.

The compiled classes are in the subdirectory `rmi/examples/echo/test/java/rmi/examples/echo`.

The original source files were developed by Sun Microsystems, Inc., which has given a free usage permission. The source files for the sample program are listed as:

`Echo.java`: Containing the Standard Interface that defines the remote behavior.

`EchoImpl.java`: Containing the Implementation and sample `Main` that creates several echo servers and registers them with the registry on the local host.



`EchoClient.java`: Containing a client that looks in the local registry for named echo servers and invokes its call method.

`EchoApplet.java`: An applet that finds a named echo server and lets a user invoke the remote interface. This applet uses the java Threads to calculate the lookup time.

`Index.html`: Web page with EchoApplet embedded.

## CHAPTER V

### EVALUATION

#### 5.1 Sample Tests Done with the Program

This chapter gives a brief outline of the testing done with the windows version of the RMI software. As part of testing, a test program was used to test the utility of the software. The program echoes the input string that has been passed as the output. The input string is passed by the client and the method that does the echoing is located on the server machine. The server is looked up by the client interface and the input string is passed. The server has a local method (that has been previously declared remote), which returns the same string back to the client machine. The program was tested on several machines of the Oklahoma State University Computing and Information Services Department.

The different method lookup times taken by the test program for different run conditions were noted and are tabulated in Tables I and II. To simulate the invocation of remote objects across networks, file servers were used as a network medium. Most of the machines mentioned in Table I are connected to the University Computer Center File Sever (UCC-FS1) of Oklahoma State University. To check the performance of the program on different network servers, another file server, HELPDESK-FS1, was used. By connecting the same machines to different file servers, a behavior of how the program

works on invoking methods across networks was determined.

The difference in the times taken to retrieve the same method across different file servers was noted. The initial time to look up any given host was observed to be more when compared to subsequent lookups to the same host, as the `java.rmi.Naming` interface makes a note of the host during the initial lookup and this interface is searched in the registry for any future lookups.

Table I. Sample Remote Method Lookup Times in Milli Seconds - I

	Methods on Remote Server					
	Echo1	Echo2	Echo3	Echo4	Echo5	
Clients	Cishlpdsk1	73	19	14	32	18
Running	Cishlpdsk2	104	37	33	30	18
on	Cishlpdsk3	1	15	9	10	12
Different	Cishlpdsk4	78	20	18	15	19
Hosts	Cismmpc	506	14	15	19	14
	Cisrobin	90	38	38	36	30

In Table I, a remote server was run on machine Cishlpdsk3 and different clients were run on machines Cishlpdsk1, Cishlpdsk2, Cishlpdsk3, Cishlpdsk4, Cismmpc, and Cisrobin.

The first column in the above table specifies the name of the machine on which the client was residing. The server was located on the machine Cishlpdsk3. The remaining columns reflect the amount of time (in milli seconds) taken by the client in the

first column to look up for the method, denoted in the first row of the specified column, that was residing on the server. The amount of time that each client took to look up for a method was noted. Figure 5-1 provides a time/method data graph for Table I.

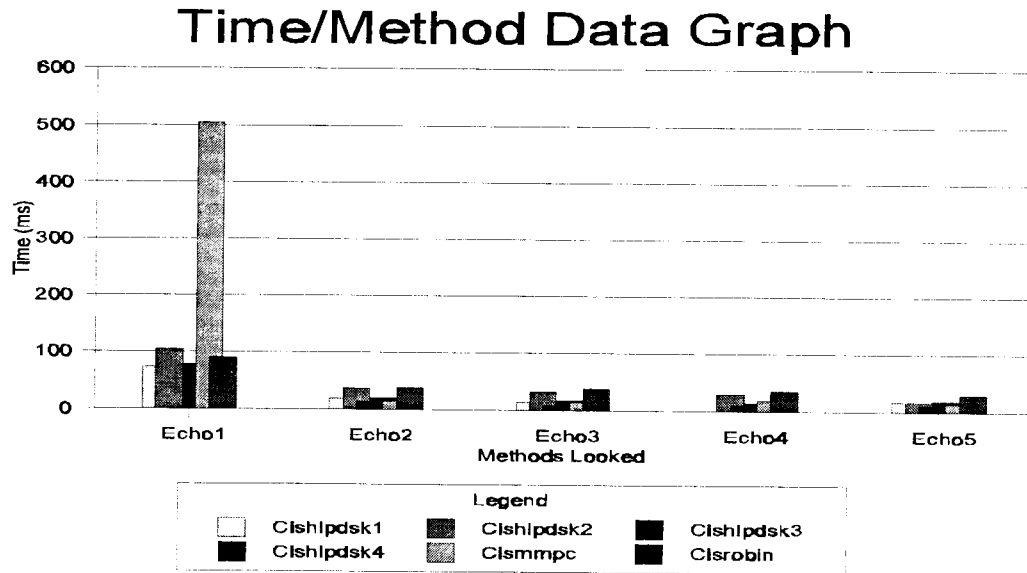


Figure 5-1. Sample Remote Method Lookup Data Graph - I

The sample lookup times that have been obtained when the remote server was run on the file server HELPDESK-FS1, and the clients were running on the file server UCC-FS1, were tabulated and were given in Table II.

Table II. Sample Remote Method Lookup Times in Milli Seconds - II

	Methods on Remote Server					
	Echo1	Echo2	Echo3	Echo4	Echo5	
Clients						
Running						
on						
Different						
Hosts						
	Cishlpdsk1	75	15	13	38	12
	Cishlpdsk2	128	32	39	42	15
	Cishlpdsk3	2	11	7	12	10
	Cishlpdsk4	70	12	14	15	25
	Cismmpc	586	34	31	28	24
	Cisrobin	108	58	45	57	40

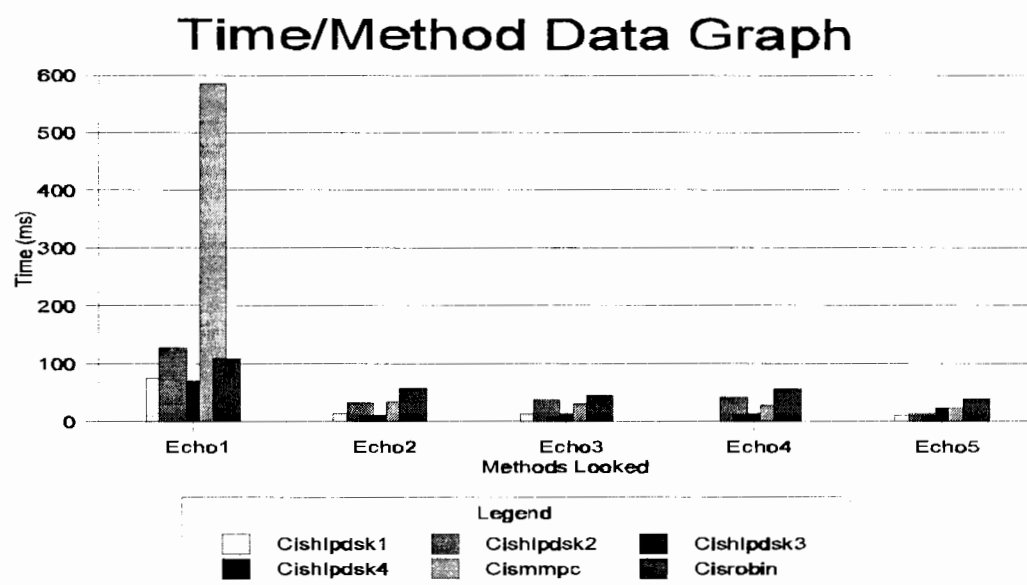


Figure 5-2. Sample Remote Method Lookup Data Graph - II

In Table II, the server was run on machine Cishlpdsk3 running on file server HELPDESK-FS1, and different clients were run on machines Cishlpdsk1, Cishlpdsk2, Cishlpdsk3, Cishlpdsk4, Cismmpc, and Cisrobin, running on the file server UCC-FS1. The amount of time that each client took to look up a method was noted.

## 5.2 Observations

The following observations were made when running the Remote Method Invocation using the Echo interface.

- When the program is run for the first time, the amount of time to look up for any given host (including the server) is more. This can be attributed to the fact that the program needs to implement the registry interface and the server interface. This may take a few seconds.
- The amount of time taken to lookup a remote method lying on the same file server is less when compared to the amount of time taken to look up the same method residing on a different file server. This is due to the amount of time taken to load a stub from the network. Loading a stub from the same file server takes less time than loading from a network server.
- The success rate is very high for this program while looking for a remote object. The program almost always successfully identified all the hosts in various test suits.
- When compared to the sun alpha version, the program is more user friendly and gives useful information regarding the connecting time and network failures. The exception messages are an advantage when trying to debug the source of the exception.

Some of the limitations can be cited as:

- the program runs only under the Windows95 environment.
- the amount of memory used by the program is fairly high, around 2MB.

When compared to the initial Alpha version, this application has more user friendly features. These are tabulated in Table III. The usage of Graphical User Interfaces and the introduction of multi-threaded objects tend to make the present version more informative than the previous version.

Table III. Comparison of the Versions

Properties	Previous Version	Modified Version
Accessibility	The access is limited to Sun Workstations and is not always possible to get access to them.	Compared to the Sun Work Stations, Personal Computers are more easy to access.
User Friendliness	The absence of Graphical User Interfaces (GUIs) makes them less user friendly.	A more user friendly environment due to the use of GUIs.
Means of Accessing the Server	The clients run on mainframe and the server is accessed by using Netscape.	Clients run on the user machine and the server is looked up using appletviewer.
Features	Gives information about the total number of connections.	More information including the amount of time taken to access the remote object, the ability to disconnect from a server, server information, and system errors are introduced.

The test program involved both binary porting and source code porting. Binary porting involved the use of class libraries that were also in use by the existing version.

Most of these libraries were linked dynamically during the execution of the program. Source code porting consisted of using selected procedures for making network connections between clients and remote servers. The code developed for the new user friendly features was not based on the existing version and hence did not come under the rubric of software reuse.

Considering the above aspects, an estimate of the percentage of code that was reused was obtained. The original version consisted of 649 lines of code that were in direct use by the test program. In the new environment, the same application required 290 lines of the original code and an additional 500 lines of code for developing the new user friendly features for the program.

From the above data, the percentage of source code that was reused can be calculated as follows:

Amount of code in the existing version = 649 lines

Amount of code in the new version = 790 lines

Amount of actual code developed = 500 lines

Amount of code reused =  $790 - 500 = 290$

Percentage of code reused =  $\text{Amount of code reused} / \text{Amount of code in the new version} = 290 / 790 = 36.7\%$  (of the existing code)

The above estimate involved only the source code aspect of the porting application. It did not take into account the various class libraries that were used in both versions, as they fall into the category of binary porting. The actual percentage of code reuse would have been more than the amount projected if there was no addition of the



timer process and the new Graphical User Interface. Thus, it can be inferred that instead of developing 100% new code, the original code was reused to the extent of 37%.

Considering the reuse economy, the potential development cost can be calculated using the equation proposed by John Gaffney (as cited in [Mareddy and Samadzadeh 95] and [Barnes et al. 87]) of the Software Productivity Consortium which predicts that

$$C = (1-R) * L + b * R$$

where C is the total cost of developing new software, R is the percentage of code reused, b is the cost of reusing a line of code, and L is the cost of developing a new line of code. For the projected case, where it is assumed that there was no reuse of the existing code, the potential development cost can be calculated as:

$$C_1 = (1 - 0) * L + b * 0 = L \text{ (R=0 for no reuse)}$$

In the actual case, the amount of reuse was 37%, hence the potential development cost can be calculated as:

$$C_2 = (1 - 0.37) * L + b * 0.37 = 0.63L + 0.37b$$

Assuming that the potential cost of reusing a line of code is less than the cost of developing a new line of code,  $C_2$  is less than  $C_1$ . According to Karlsson [Karlsson 95], the current industry standard for software reuse is limited to a maximum of 55% of the source code while the average is around 23% under a 90% confidence level. Hence, by reusing 37% of the source code, there was a significant savings in the potential total development cost.

### 5.3 User Appraisal

The program was shown to eight Computer Science Department Graduate students. Some of the suggestions are listed below:

- The precision of the original timer was one second. Since the time taken to look for a server was rather fast (on the order of one hundredth of a second), it was suggested to increase the precision. The timer currently in use has a precision of one milli second.
- There was a suggestion of mentioning a very detailed report of what the remote exceptions look like. The exception message in the message textfield is a result of different failure paradigms. Provision has been given to display the entire contents of the remote exceptions in the background screen. The messages are usually long with the entire router mentioned along with the point where the exception occurred.
- Suggestion to display the current server that the client is accessing was given. This is provided by means of a Status field which provides the information about the current server in use.
- The time is displayed in the background MS-DOS screen. Suggestion was given to set up a different text field to display the same. This idea is not implemented as it does not hold good when looking for different servers from the same client machine.
- The idea of invoking remote methods on different hosts at a given time was suggested. The applet cloning facility was used to connect each clone to a different host, which resulted in an abstraction of invoking methods on remote hosts.
- There were some ideas of changing the original screen by changing the font to the current size, increasing the window size, and changing the background color. The

initial background color was changed from dark blue to the present color to enhance the appearance of the program.

## CHAPTER VI

### SUMMARY AND FUTURE WORK

This thesis work involved the porting of an application, which has been originally written for Sun Workstations, to a personal computer environment running under Microsoft Windows 95. The work included identifying the necessary building blocks that are required for the program during its execution, writing a new user interface for the program, along with introducing some enhanced and user friendly features.

The application was tested with a test suite and was executed on multiple file servers of the Oklahoma State University Computing and Information Services Department, and its performance was analyzed. The ported application worked efficiently on local file servers. The amount of time taken for accessing objects on remote hosts located on network file servers was rather high (of the order of 1000 milli seconds). This can be attributed to the network traffic when a specified object is acquired.

The ported application was able to locate all the valid hosts in the test suite. In the case of invalid hosts (hosts which did not implement the client interface), the application always came up with a pertinent error message as programmed.

The future work includes extending the ported application from different perspectives. Java Soft Inc., has introduced new additions to RMI libraries in which a

user can introduce multi-threaded operations in the existing version. Multi-threading was used in determining the time taken to access a given host by running a timer parallel to the search routine. Similar features such as running various servers in parallel on the same machine by cloning servers using a thread process for each server, and introducing multimedia applications in parallel with the main application can be undertaken as part of the related future work.

## REFERENCES

- [Abdel-Hamid 93] T. K. Abdel-Hamid, "Modeling the Dynamics of Software Reuse", *Proceedings of the Sixth Annual Workshop on Software Reuse*, Owego, NY, pp. 1-5, November 1993.
- [Alverson and Notkin 93] G.A. Alverson and D. Notkin, "Program Structuring for Effective Parallel Portability", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 9, pp. 1041-1059, September 1993.
- [Arthur 96] Vanhoff Arthur, *Hooked on Java, Creating Hot Web Sites with Java Applets*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1996.
- [Barnes et al. 87] B. Barnes, T. Durek, J. Gaffney, and A. Pyster, "A Framework and Economic Foundation for Software Reuse", *Proceedings of the Rocky Mountain Institute of Software Engineering (RMISE) Workshop in Software Reuse*, Rocky Mountain Institute of Software Engineering, Boulder, CO, pp.77-88, October 1987.
- [Birrell and Nelson 84] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, February 1984.
- [Boehm 87] B. Boehm, "Improving Software Productivity", *IEEE Software*, Vol. 4, No. 5, pp. 43-57, September 1987.
- [DEC 96] Internet site of the Digital Equipment Corporation, Inc., <http://www.dec.com>.
- [Eichmann 92] D. Eichmann, "Selecting Reusable Components Using Algebraic Specifications", *Algebraic Methodology And Software Technology (AMAST) 91, Workshops in Computing Series*, Springer-Verlag Ltd., London, UK, 1992.
- [Frakes et al. 91] W. B. Frakes, T. J. Biggerstaff, R. Prieto-Diaz, K. Matsumura, and W. Schaefer, "Software Reuse: Is It Delivering?", *Proceedings of the International Conference on Software Engineering*, Austin, TX, pp. 52-62, May 1991.
- [Hannah 96] Michael Hannah, *HTML Reference Manual*, Sandia National Laboratories, Albuquerque, NM, 1996.
- [Harms et al. 96] David Harms, Barton Fiske, and Jeffrey Rice, *Web Site Programming with Java*, McGraw-Hill, New York, NY, 1996.
- [Hooper and Chester 91] James Hooper and Rowena Chester, *Software Reuse: Guidelines and Methods*, Plenum Press, Inc., New York, NY, 1991.

- [Jackson and McClellan 96] Jerry R. Jackson and Alan L. McClellan, *Java by Example*, Sunsoft Press, Sun Micro Systems, Inc., Mountain View, CA, 1996.
- [Java-Spec 96] ] *Java Programming Language Specifications*, Javasoft Corporation, Mountain View, CA, 1996.
- [Jones 84] T. C. Jones, "Reusability in Programming: A Survey of the State of the Art", *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, pp. 488-494, September 1984.
- [Karlsson 95] Even Karlsson, *Software Reuse: A Holistic Approach*, John Wiley & Sons, New York, NY, 1995.
- [Krueger 92] Charles W. Krueger, "Software Reuse", *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131-179, June 1992.
- [Lanergan and Poynton 79] Robert Lanergan and Brian Poynton, "Reusable Code: The Application Development Technique for the Future", *Proceedings of the IBM Share/Guide Software Symposium*, IBM Users Group, IBM Corporation, Armonk, NY, June 1979.
- [LeCarme et al. 89] O. LeCarme, Pellisier Gart, and M. Gart, *Software Portability with Microcomputer Issues*, McGraw-Hill, New York, NY, 1989.
- [Lewis and Oman 90] T. G. Lewis and P. Oman, "The Challenge of Software Development", *IEEE Software*, Vol. 7, No. 6, pp. 9-12, November 1990.
- [Mareddy and Samadzadeh 95] R. Mareddy and Mansur H. Samadzadeh, "An Implementation of the Faceted Classification System for Software Reuse", in *Intelligent Systems, Volume I*, pp. 181-194, Edited by: E. A. Yfantis, *Theory and Decision Library - Series D: System Theory, Knowledge Engineering, and Problem Solving*, Kluwer Academic Publishers, The Netherlands, 1995.
- [Mooney 90] J. Mooney, "Strategies for Supporting Application Portability", *IEEE Computer*, Vol. 23, No. 11, pp. 59-70, November 1990.
- [Mooney 93] J. Mooney, *Issues in the Specification and Measurement of Software Portability*, Technical Report TR 93-6, Department of Statistics and Computer Science, West Virginia University, Morgantown, WV, 1993.
- [Prieto-Diaz 93] R., Prieto-Diaz, "Status Report: Software Reusability", *IEEE Software*, Vol. 10, No. 3, pp. 61-66, May 1993.
- [RMI-Spec 96] *Remote Method Invocation Specifications*, Javasoft Corporation, Mountain View, CA, 1996.

- [Sitaraman 92] M. Sitaraman, "Towards a Modular Approach for Real-Time Specification and Verification of Reusable Software Components", *Proceedings 9th IEEE Workshop on Real-Time Operating Systems and Software*, May 1992.
- [Skillicorn 94] David Skillicorn, *Foundations of Parallel Programming*, Cambridge University Press, Cambridge, NY, 1994.
- [Sommerville 96] I. Sommerville, *Software Engineering (Fifth Edition)*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1996.
- [Stevens 90] Richard Stevens, *UNIX Network Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [Tanenbaum 92] Andrew Tanenbaum, *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [Walrath and Campione 96] Kathy Walrath and Mary Campione, *The Java Tutorial*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1996.
- [Wolberg 83] John Wolberg, *Conversion of Computer Software*, Prentice-Hall, Englewood Cliffs, NJ, 1983.



## APPENDIX A: GLOSSARY

Application Programming Interface	An API is a library of programming routines which assist in the execution of user defined problems.
Bytecode	An intermediate assembled code that is used by the Java run time system for checking the integrity of Java source code. This is the usual way of transmitting compiled source code in Java.
Classpath	Specifies the path used by the <i>javac</i> compiler for compiling source files.
HTML	Hyper Text Markup Language, used to construct documents which can be viewed by World Wide Web browsers.
Java	An object-oriented programming language for writing applications on the Internet.
<i>Javac</i>	Basic java compiler used for compiling Java source files.
Marshal Stream	An abstraction of how messages are passed between clients and remote servers.
Native Method	A native method is a Java method (either an instance method or a class method) whose implementation is written in another programming language such as C.
Parameter Marshaling	A method of preserving the changes that are made to an object by another function to which

	the object has referenced.
Remote Object	An object whose methods can be invoked from another machine running on a different host.
Remote Interfaces	These are Java Interfaces which defines the methods of the remote object.
Remote Procedure Call	Mechanism for distributing application processing across a distributed computing network.
RMI	Remote Method Invocation, a process of connecting clients running on user machines to a network server.
Rmic	A stub generator tool that produces client and server stubs which handle the details of transmitting calls over the network.
Software Component	A Code fragment meant for a specified function.
Stub File	The stub file contains C code that is responsible for holding the Java class and its parallel C structure together in a <i>native method</i> .
Transport Classes	A set of C++ class routines that are organized as an object-oriented software toolkit for distributed, message-passing based programming.
Transport Layer	The highest of the lower layer protocols in the OSI protocol stack, concerned with the transmission of data between end systems across a communication facility.
URL	Uniform Resource Locator (URL) or Uniform Resource Identifier (URI), a reference (an address) to a resource on the Internet.

## APPENDIX B: TRADEMARK INFORMATION

Hot Java	A registered trademark of Sun Microsystems, Inc.
Java	A registered trademark of Sun Microsystems, Inc.
Java Developers Kit-1.0.2	A registered trademark of Sun Microsystems, Inc.
Java Remote Method Invocation	A registered trademark of Sun Microsystems, Inc.
MS-DOS	A registered trademark of Microsoft Corporation, Inc.
MSWindows95	A registered trademark of Microsoft Corporation, Inc.
MSWindows3.X	A registered trademark of Microsoft Corporation, Inc.
Netscape	A registered trademark of Netscape Corporation, Inc.
Sun Solaris	A registered trademark of Sun Microsystems, Inc.
UNIX	A registered trademark of UNIX System Laboratories, Inc.

## APPENDIX C

### USER GUIDE

#### 1. Introduction

This is a brief description of the various details that can be obtained using the Remote Method Invocation running under Microsoft Windows 95. This software involves accessing remote methods that are implemented by remote objects by using the Java network primitives. The initial version of the software was developed for Sun workstations and later some of the components were modified for running under the Windows 95 environment. The following sections describe the setting up and using the software along with going through the test example that is used to demonstrate the enhanced features that were introduced.

#### 2. Setting up

The main requirements for this program are a properly installed Microsoft Windows 95 operating system on a personal computer. If Windows 95 is not installed properly, the program gives run time exceptions like socketCreation error and ThreadException error. There needs to be a network chord attached to the computer as the program tends to access the network when trying to load some of the files dynamically over the network. Using a modem to run the program is not feasible as modems do not support TCP/IP stacks like network chords.

## 2.1 Hardware and Software Requirements

The computer on which the program can be tested needs to meet the following hardware and software requirements.

- Any IBM-compatible machine with an 80386 or higher processor.
- A hard drive with at least 4MB of free space.
- A VGA monitor or better.
- Two megabytes of memory.
- A network. The network can be an ordinary file server that is connected to the internet at the far end or a direct network chord connecting to the network.
- A mouse.
- Microsoft Windows95 installed.
- MS-DOS running as a background process.

## 2.2 Running the Program

The following steps help the users run the program on their computer. The Java Developers Kit Version1.0.2 is a freeware and hence the user can download it from the internet and install it. The program makes use of the components to compile the source files (using the `javac` compiler), interpret the bytecode format (using the `javaw` interpreter, which simulates running of a background process), and run the applets (using the `appletviewer`). When MicrosoftWindows95 is installed, the desktop looks somewhat like Figure C-1.

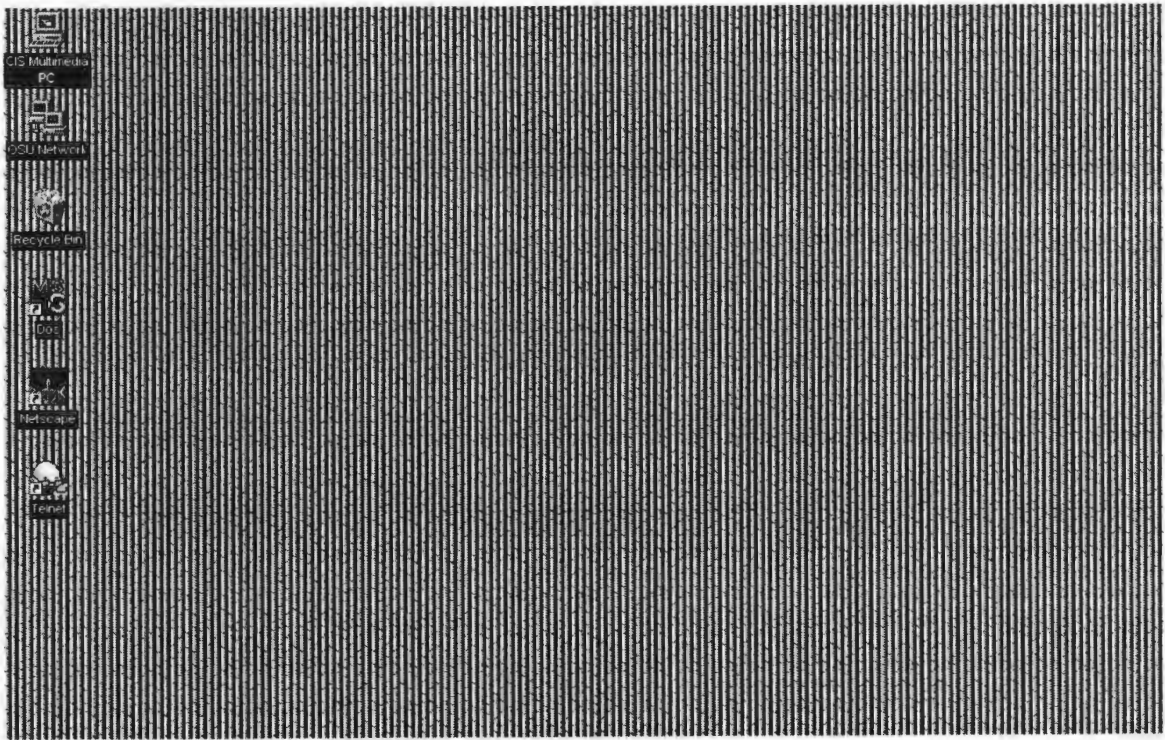


Figure C-1. Initial DeskTop Screen

Before starting the program, the initial setup program needs to be run. I have loaded all the setup commands in a batch file named `run.bat`. Once the user starts the MS-DOS mode, the user can run the batch file by typing the name of the file, i.e., `run` in this case. I have run all the applications on my home drive on the UCC-FS1. If the user wants to run the program in the C drive, the user needs to change the drive from H to C. The initial setup program checks whether the path has been set to all the executable files, like the `javac`, `javaw`, and `appletviewer` that are present in the `java\bin` subdirectory.

Figure C-2 depicts the different commands that run on the MS-DOS screen when the user runs the batch file.



Figure C-2. Initial Setup Screen

Once the first batch file is run, the user needs to run another batch file, i.e., `start.bat`. This file contains the following commands:

```
SET CLASSPATH=h:\rmi\examples\echo\test;h:\rmi\lib\rmi.zip
```

The classpath variable looks for the compiled classes and sets the path to the directory hierarchy mentioned. When the program runs, it looks for the CLASSPATH variable and fetches the required classes from the site mentioned in the classpath. I have compiled the source files and kept the classes in a different subdirectory named `test`. So the program looks for the compiled classes in the subdirectory `test` while it executes.

The server is started by calling the `javaw` interpreter and running the registry on the local file server, which is the remote host. The registry needs to be run on the machine which can be visualized as the remote host, so that the clients can be run from other machines connected through a common network.

Figure C-3 shows how to run the server on a client machine.

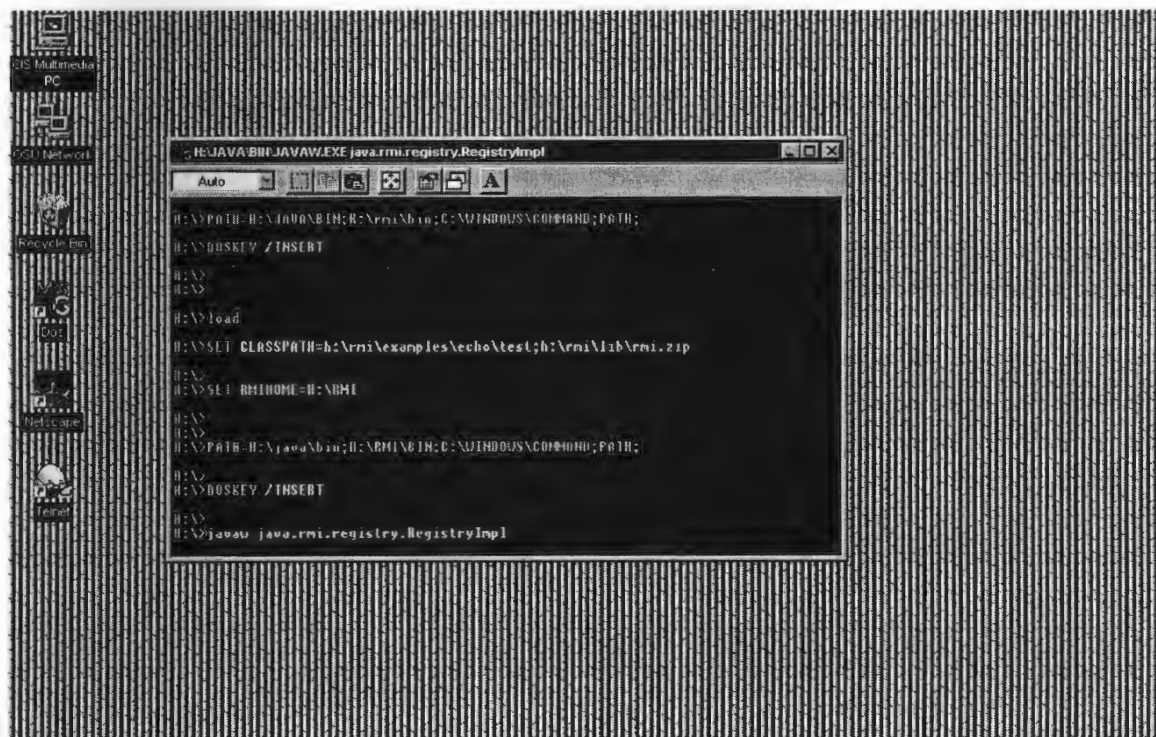


Figure C-3. Running the Server

Once the server is started, the clients can be started by running the client interfaces. The client interface in this example program is named `EchoImpl`. So this is run using the `javaw` interpreter.

Once the client interface is run, the application can be looked up using an appletviewer. I used another batch file, named `start.bat`, which calls the appletviewer to load the application named `EchoApplet`. The appletviewer loads all the classes from the path mentioned in the `codebase` parameter. Since all the compiled classes for this example are located in the subdirectory `test`, the codebase for the applet was named `test` so all the compiled classes are loaded. Once the appletviewer is started, the applet is loaded. The application comes up with the hostname of the local machine on which the



applet is running in the hostfield of the applet. Figure C-4 shows the screen that pops up when the applet is loaded.

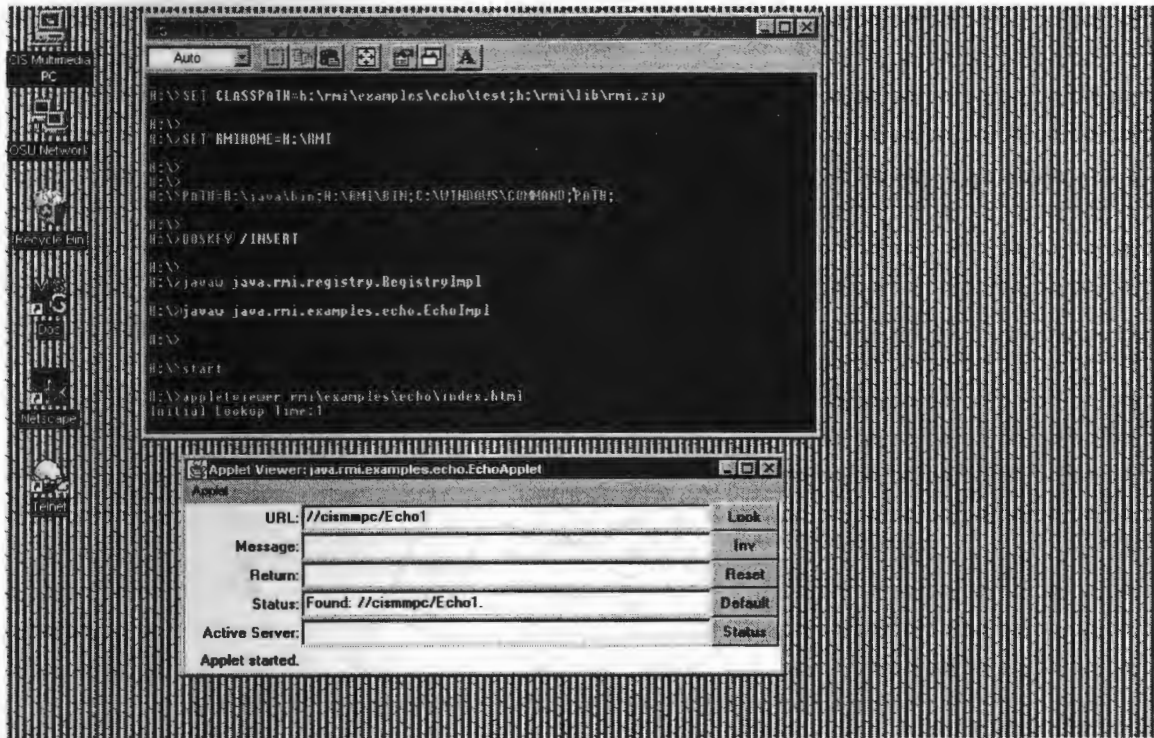


Figure C-4. Applet Loaded on Server Machine

The applet is loaded on the user desktop and the MS-DOS screen runs in the background. Once the applet is loaded, the timer starts automatically and gives the amount of time the application took to search for any host mentioned in the MS-DOS screen. Since the initial host is the same machine, the amount of time taken is mentioned as the initial lookup time. This is a small fraction of time, usually a few milliseconds. The default method that is looked up is named Echo1. So the URL is called HostName/Echo1. In the figure above, cismmpc (which stands for Computing and Information Services Multi Media Personal Computer) is the name of the machine on which the application was running. The Status field shows the present status of the host.

If the application finds the remote host, then the message shown in Figure C-4 is displayed.

Once the program finds the host, we can run the Remote Method Invocation. In the testing case, the application is checked on the local host, which is `cismmpc` in this case. Since the example is an echo, the program takes a string as the argument for the method of the remote object, which is the method `Echo1` in this case. The result is displayed in the return field of the applet (see Figure C-5).

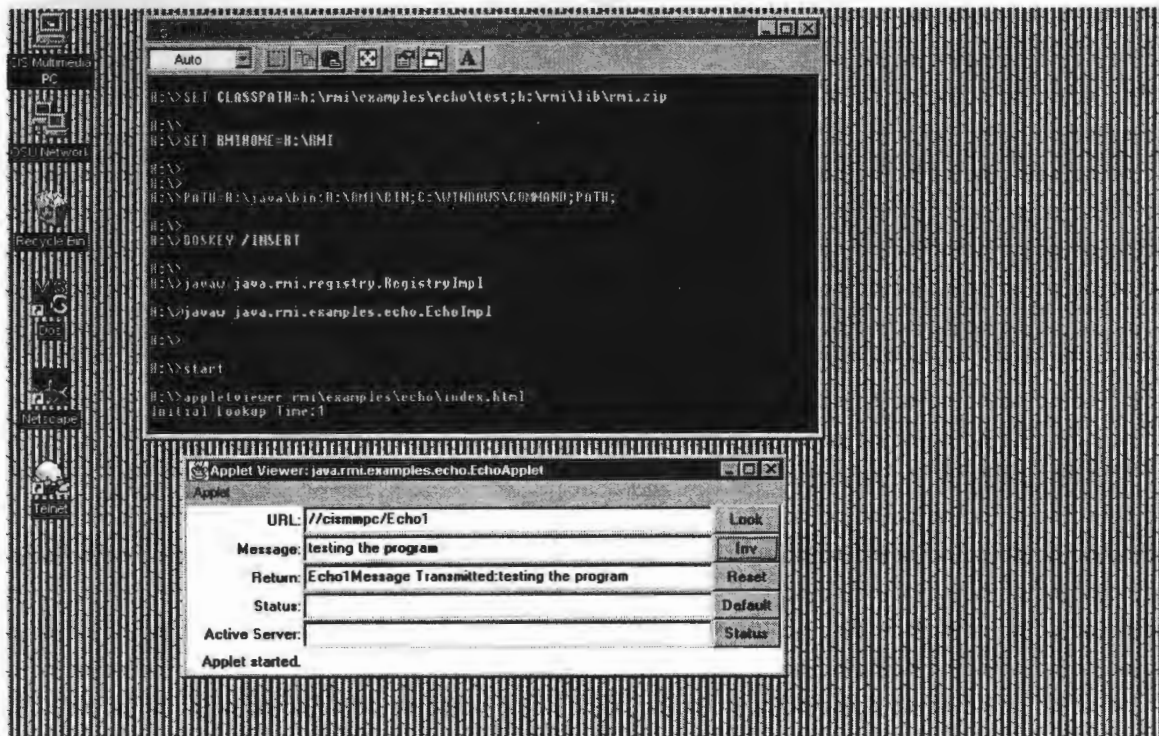


Figure C-5. Invoking a Method on a Local Host

One of the features added to the existing version is the facility of going back to the default host that is implemented in the form of a `pushButton`. The `Default` button is part of the user interface which reconnects to the default host as the user invokes the button. In the example above, the default host is considered as `//cismmpc/Echo1`. So, invoking the default button results in changing the host to the initial host which is

displayed in the status field. Figure C-6 illustrates the result obtained by invoking the default button.

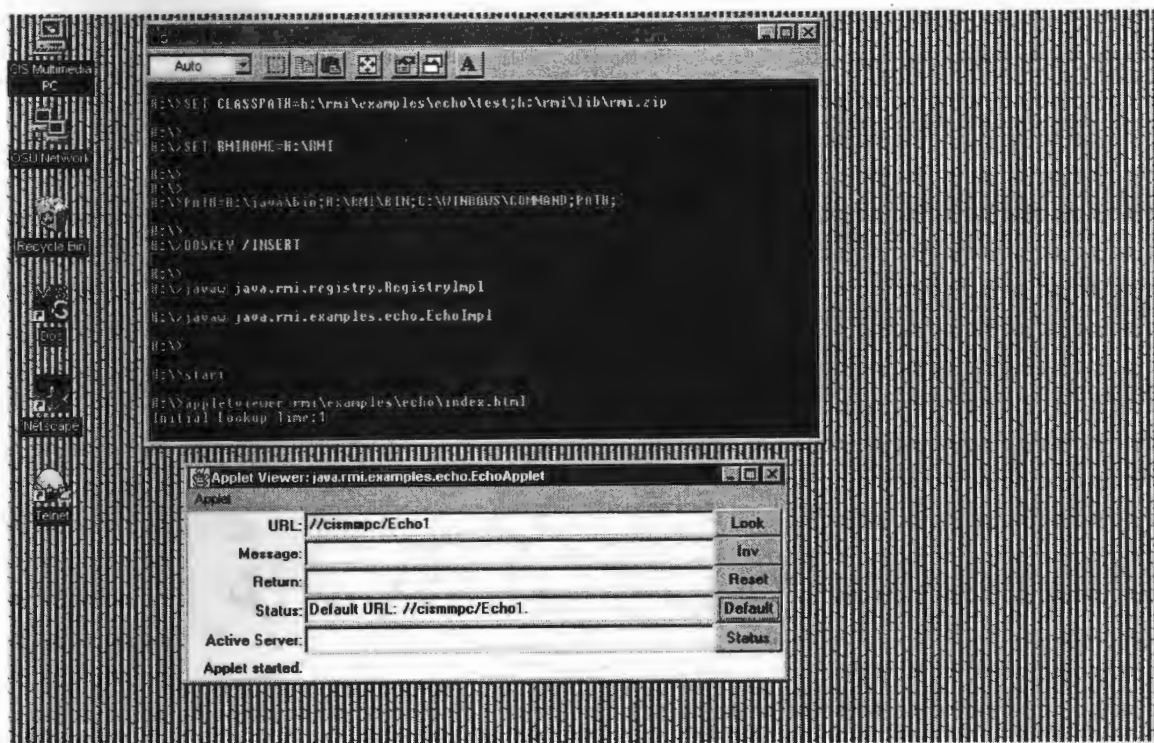


Figure C-6. Using the Default Option

Another feature that was implemented is the Reset option. The Reset button is used to disconnect from a remote host. The network connection that is set up is broken and all the references that are made by the client to the server host are lost. The name of the server host is maintained in the `java.rmi.Naming` interface for a faster lookup in the future. When this button is used, the application waits for a new hostname in the hostfield for searching. Figure C-7 shows the applet when the Reset button is invoked.

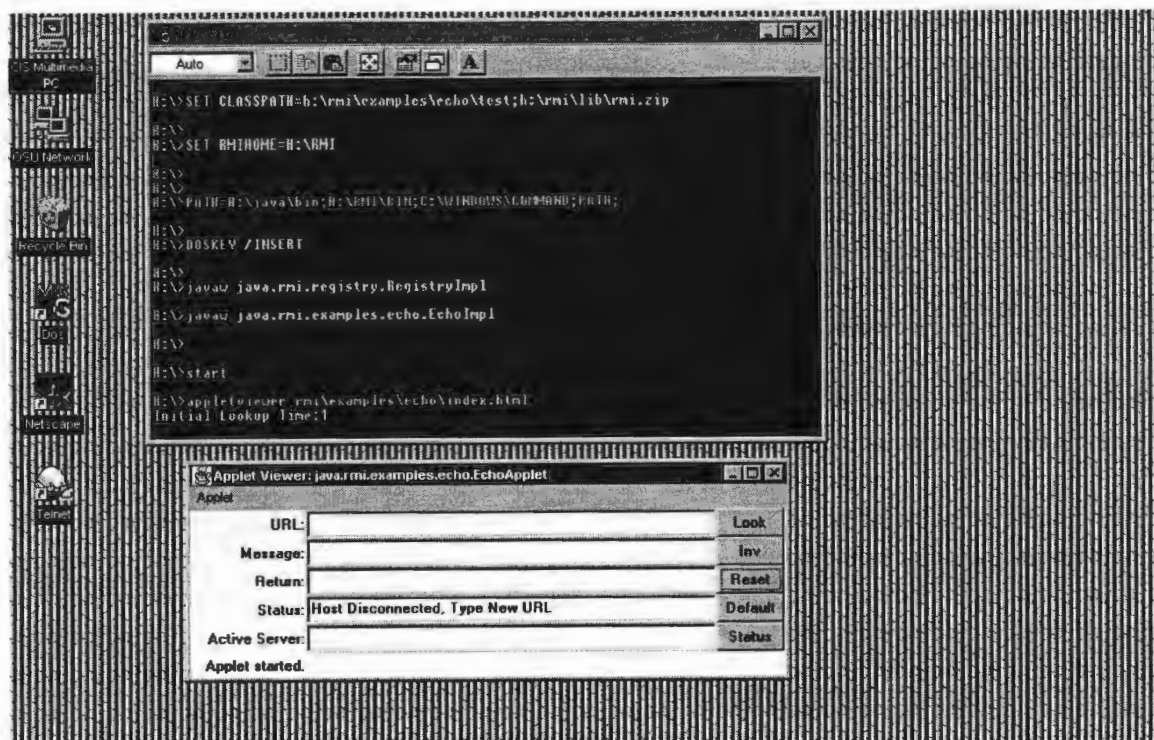


Figure C-7. Using the Reset Option

When the Reset button is used, the client interface functions are disabled. So some of the user interface buttons invoke error messages when they are used. This is displayed in Figure C-8.

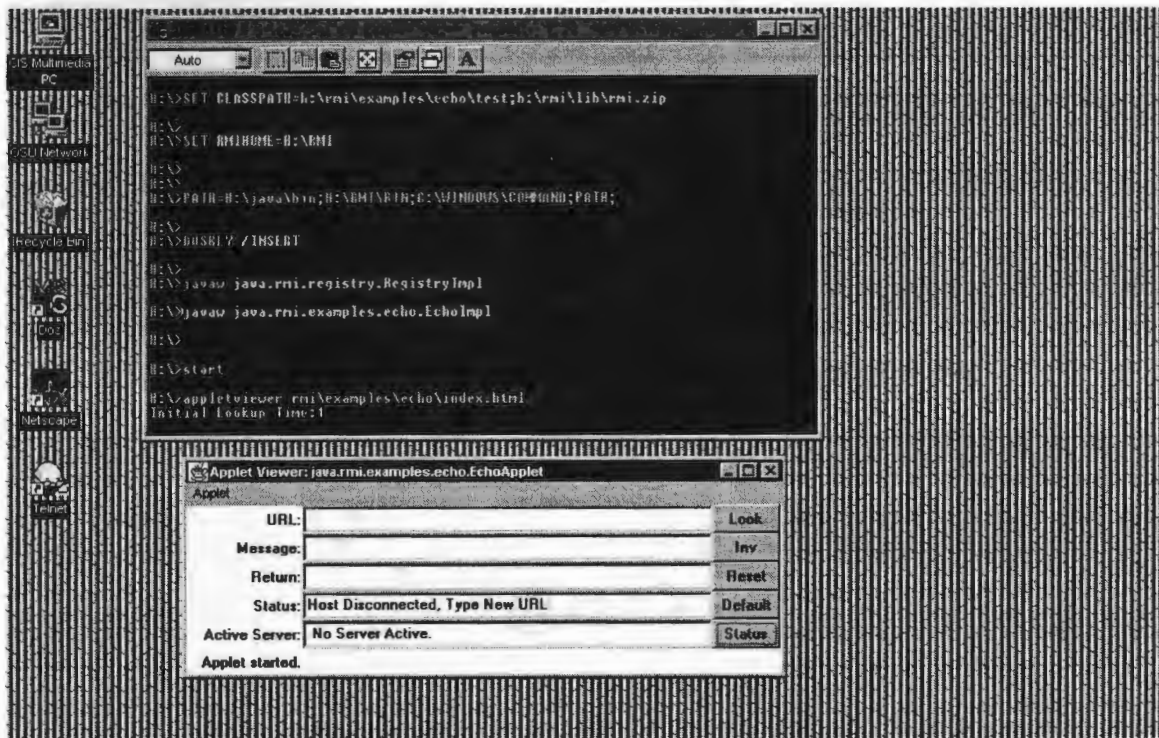


Figure C-8. Using Lookup Method with no Active Server

When the Inv button is used, the application gives an error message as the named object is not an Echo server. Figure C-9 gives the program behavior when the Inv button is used when there is no active server. Figure C-10 shows the program behavior when trying to invoke another method on the same host.

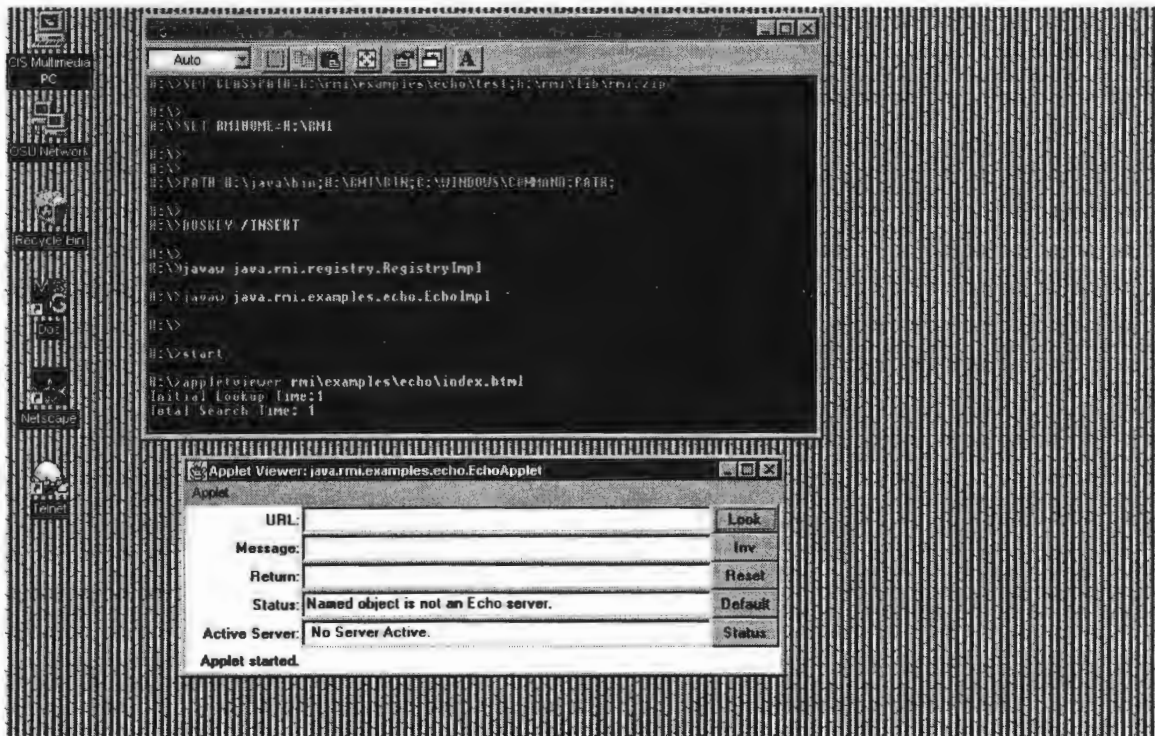


Figure C-9. Using the Invoke Method with no Active Server

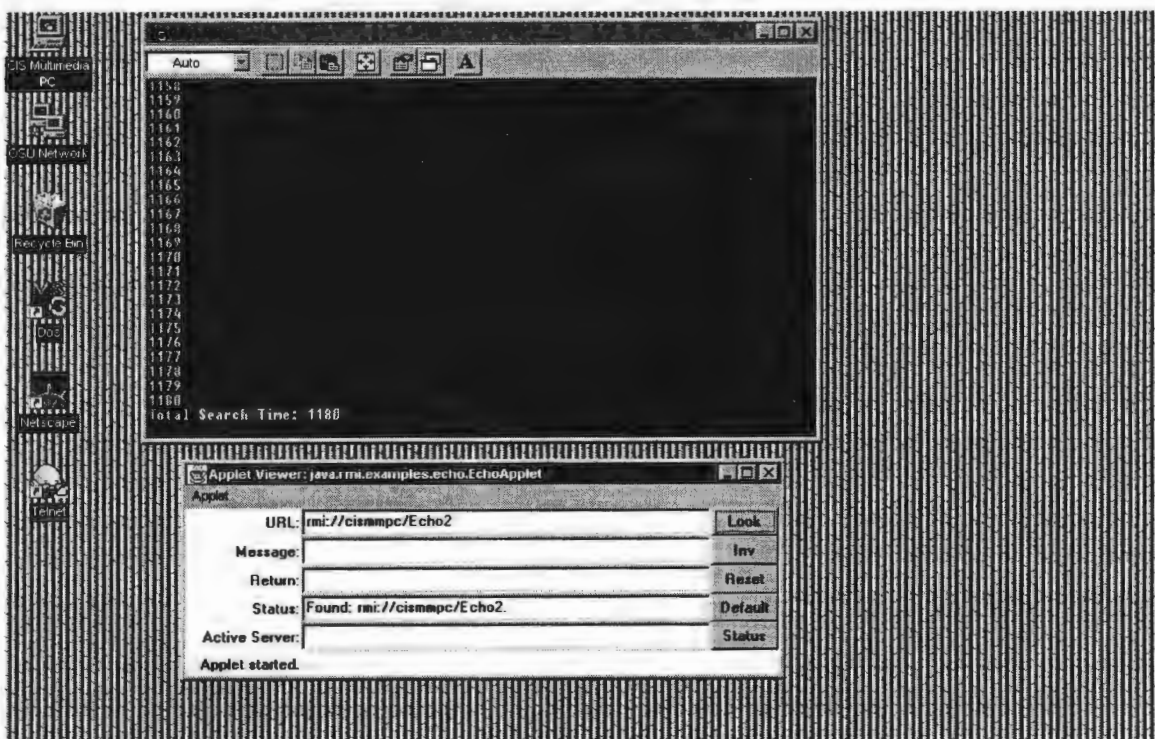


Figure C-10. Looking for another Method on the same Host

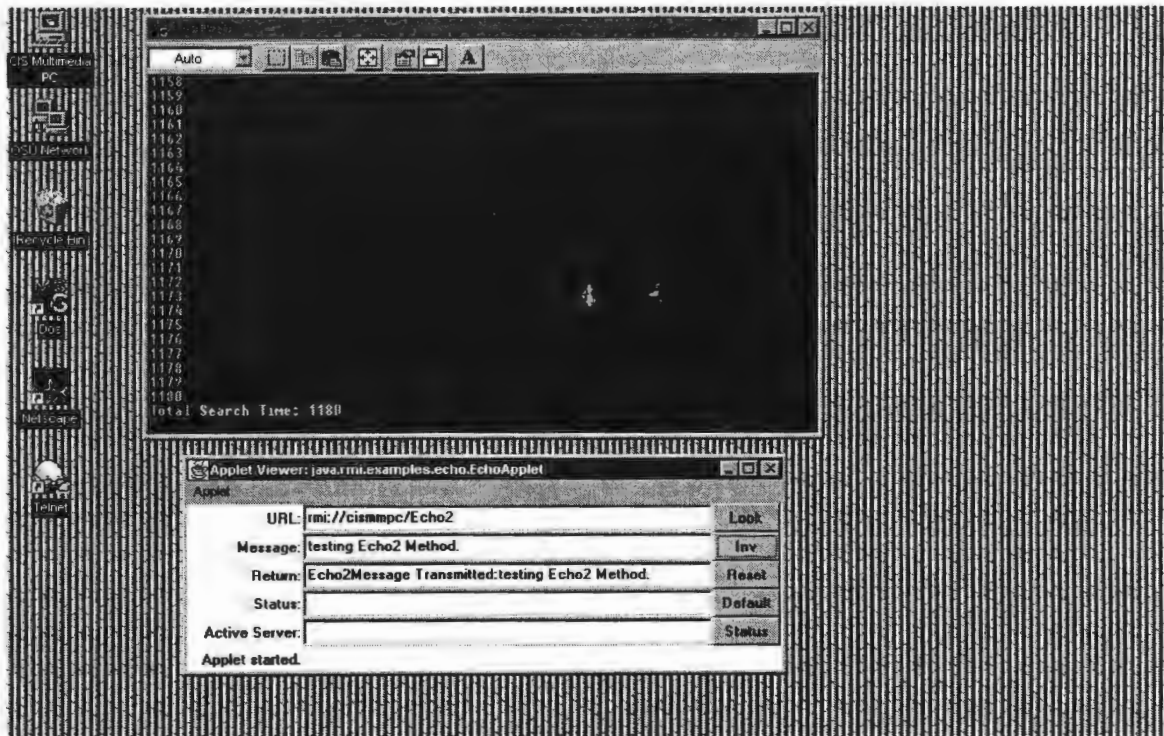


Figure C-11. Invoking another Method on the same Host

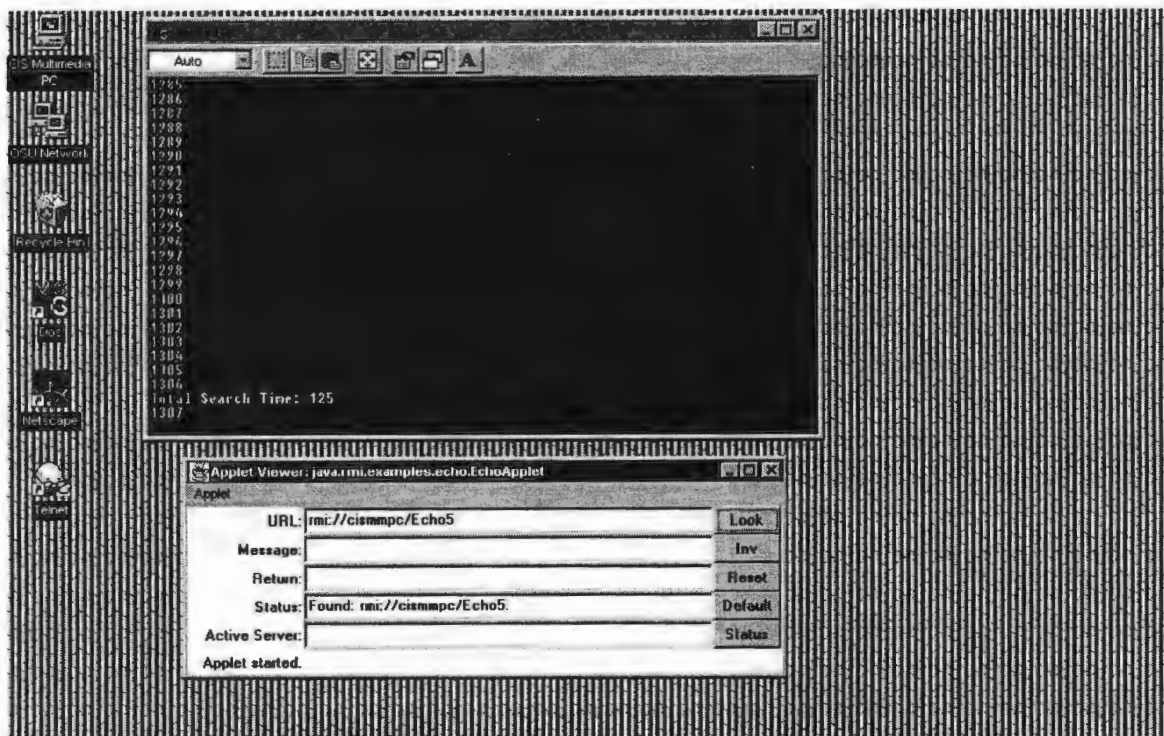


Figure C-12. Looking for another Method (Echo5) on the same Host

Figures C-11 and C-12 illustrate the different method invocations on the same host.

The application creates five different objects, each having a method which implements the remote interface. So, when the client invokes any of the five methods, they are displayed on the screen in the status field.

Before a client can look up for any server or a server can look up for a client, the client needs to implement the client interface. This is the `EchoImpl` interface in the example program. If the client does not implement this interface, it leads to a `remoteException`.

When a server looks up for a client that implements a client interface, the hostname of the client is noted in the registry, which uses the `java.rmi.Naming` interface. Thus, in the future, lookups for the same host are done faster as the hostname is first searched in the registry.

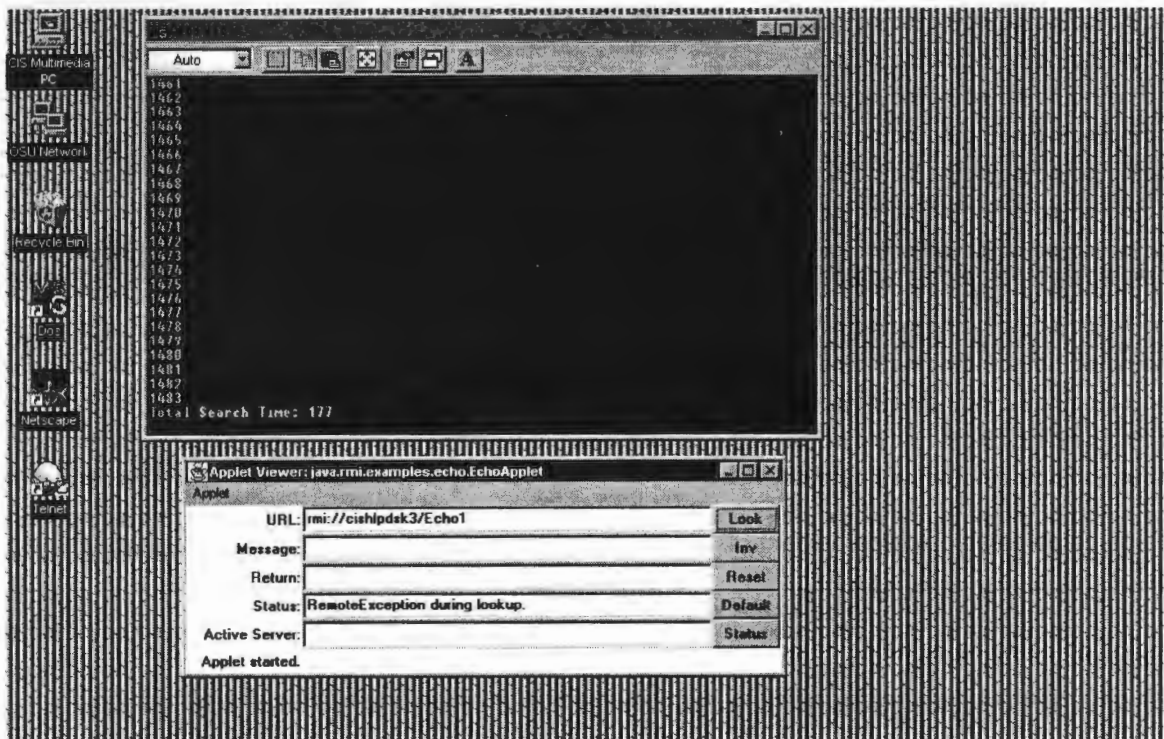


Figure C-13. Remote Exception during Client Lookup



Figure C-13 shows an exception that occurs when a server looks up for a client that does not implements a client interface.

But when the client implements the interface, the application finds the client and comes up with the hostname. Figure C-14 displays the applet when it finds a host mentioned in the hostfield.

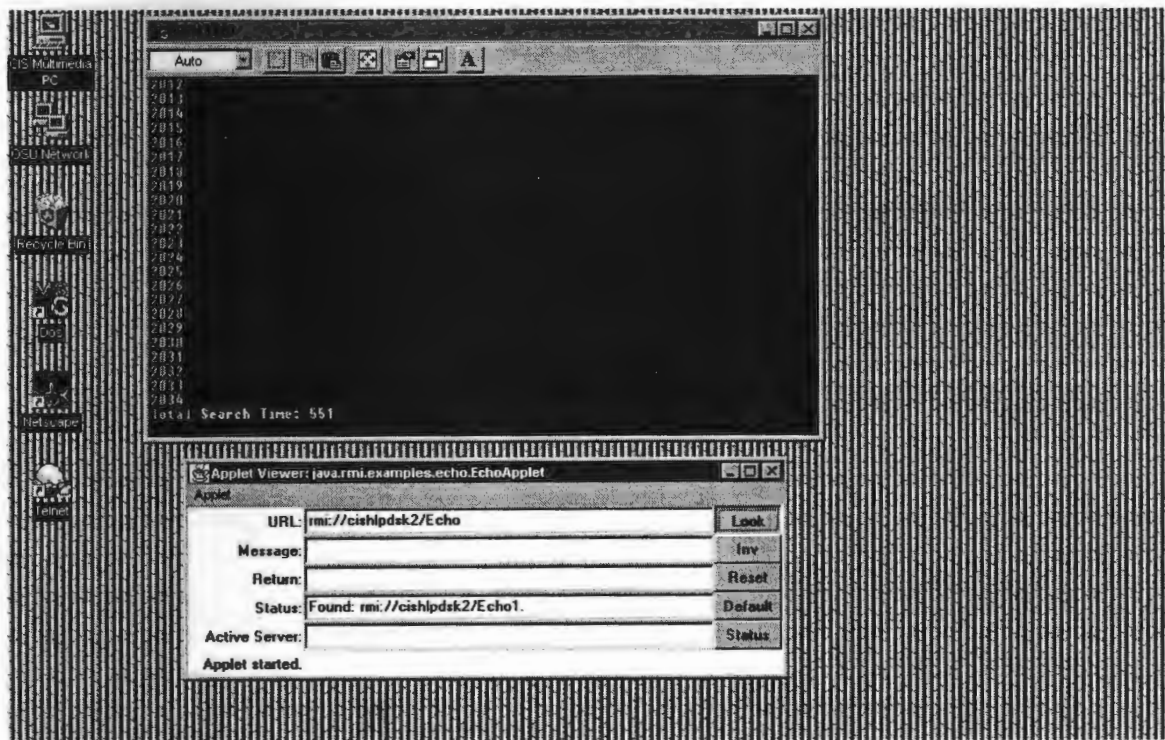


Figure C-14. Looking for a Remote Client

All the methods that are declared remote by the remote object and which implement the remote interface can be accessed by the client. A remote object can implement other methods but these cannot be accessed as long as they are not declared remote.

Once the client is looked up, the Status button displays the current client that is using the application through the applet. Figure C-15 shows that the client on machine `cishlpdsk3` is using the server methods on machine `cismmpc`. Looking for another

host changes the status to the latest user. Thus, at any particular moment, an idea as to who is using the remote application can be obtained.

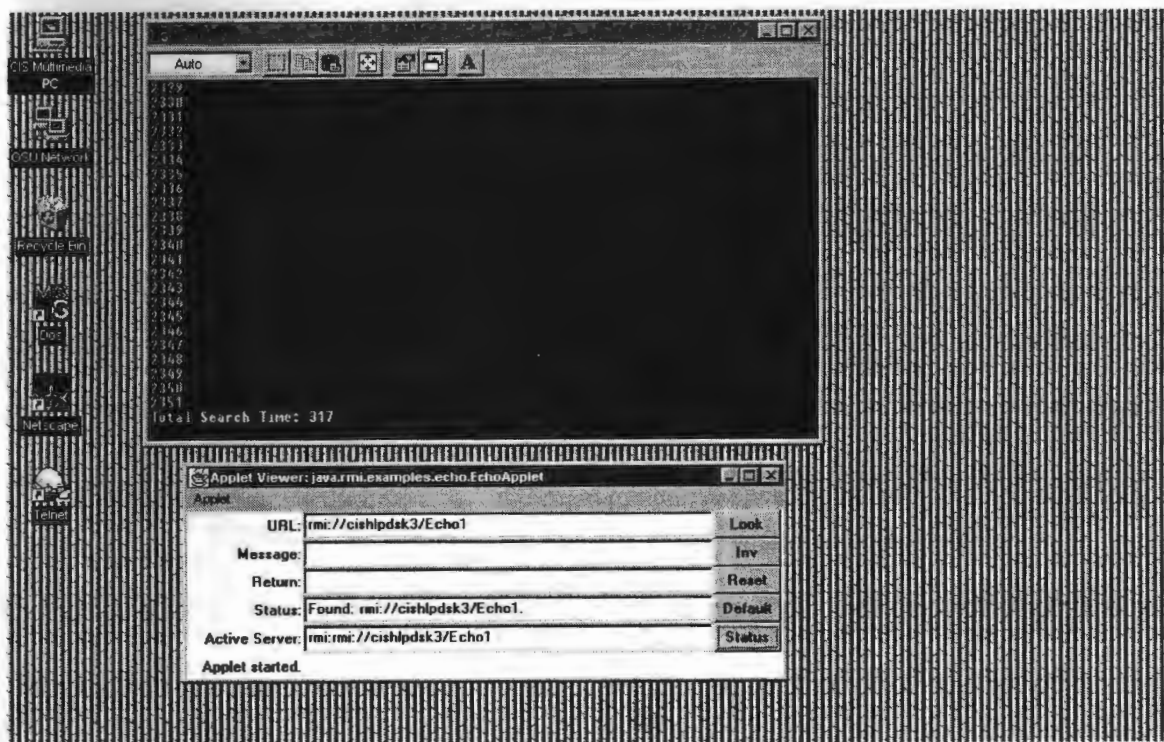


Figure C-15. Status of the current Client using the Applet

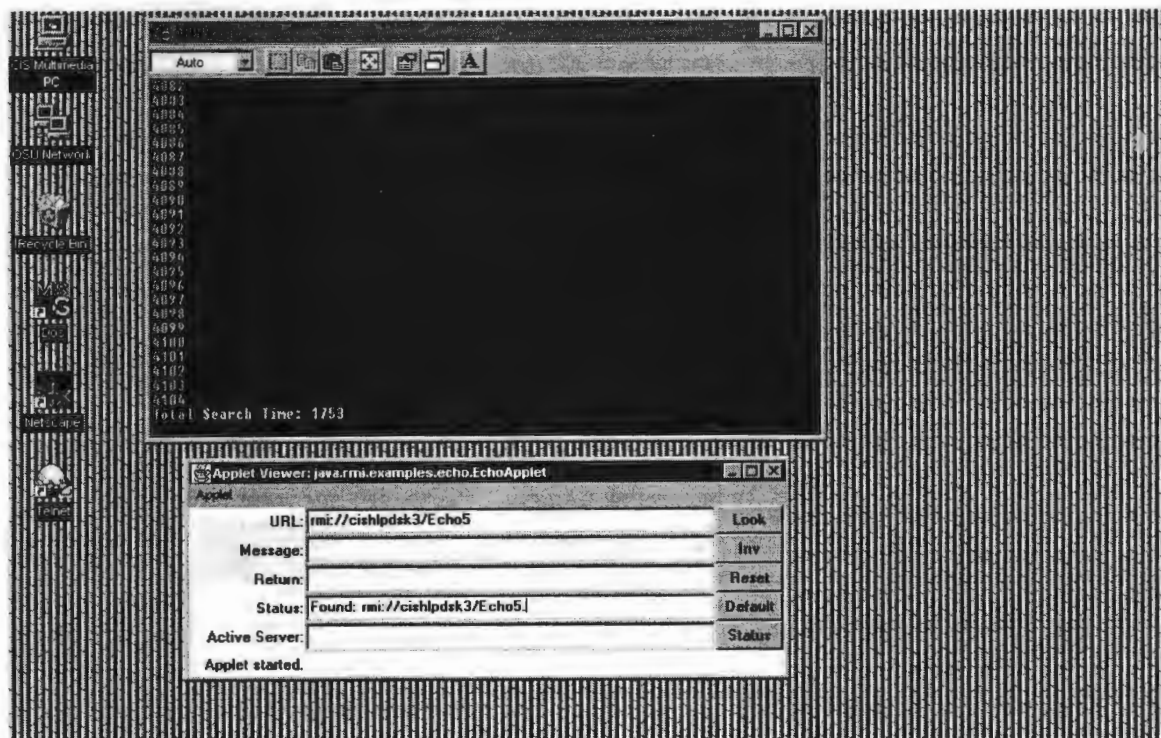


Figure C-16. Looking for another Method on a Remote Host

Figure C-16 displays an access to another method of a remote object. The server in the example has five methods declared remote. So any of the five can be invoked successfully by a client.

Multiple clients can use a remote object at the same time. This is possible by having the remote object create a Skeleton object for each of the client that tried to invoke the methods. Figure C-17 depicts how multiple clients can be abstracted by the `clone` property of an applet and how each applet runs as an independent application.

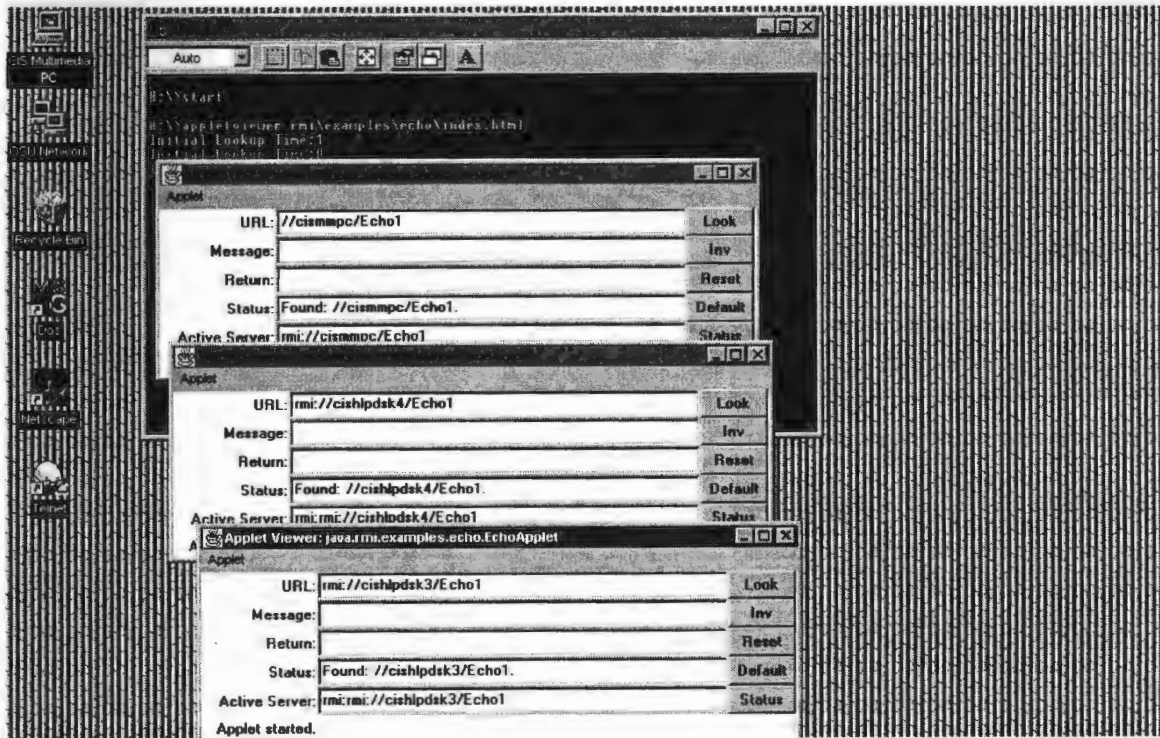


Figure C-17. Multiple Clients using a Remote Host

In the above figure, each applet is independent and runs as a single entity. The Active Server label shows that each of the methods have been remotely accessed and are used by the client machine.

Before a client interface is started, the server interface must be executed. That implies an active server should exist before a client can make a reference to it. Otherwise, the reference leads to a `remoteException`.

The details of the client interfaces and the server interfaces, and the troubleshooting procedures about the program are mentioned in a file `README`.

APPENDIX D  
PROGRAM LISTINGS

```
-----
File Name:   Run.bat
Type:       Batch File
Parameters:  None
Description: Initializes the path for the various executable files that are later
            used by the program.
Author:     Lakshmana Pamarthy
-----
```

```
SET CLASSPATH=h:\rmi\examples\echo\test;h:\rmi\lib\rmi.zip
```

```
SET RMIHOME=H:\RMI
```

```
PATH=H:\java\bin;H:\RMI\BIN;C:\WINDOWS\COMMAND;PATH%;
```

```
DOSKEY /INSERT
```

```
-----
Name:       Start.bat
Type:       Batch File
Parameters:  None
Description: Runs the server in the background and starts the client on the
            host machine, along with starting the applet.
Source:     Sun Micro Systems, Inc.,
-----
```

```
javaw java.rmi.registry.RegistryImpl
javaw java.rmi.examples.echo.EchoImpl
appletviewer index.html
```

```
-----
File:       Echo.java
Type:       Method Declaration File
Parameters:  None
Description: This package defines the interface. The method which is defined as remote can
            be accessed remotely by any other client running on a different Java Virtual
            Machine.
Source:     Sun Micro Systems, Inc.,
-----
```

```
-----
package java.rmi.examples.echo;
```

```
// This is the standard way of mentioning the interface. It should extend the
java.rmi.Remote and should have a throw remote
// exception in its methods.
```

```
public interface Echo extends java.rmi.Remote {
    String call(String message) throws java.rmi.RemoteException;
}
```

```
-----
File:       EchoApplet.java
Type:       Interface Description File
Parameters:  None
Description: This package defines the user interface. It contains the code for the
            different graphical user interfaces along with the code for calculating the
            lookup time for any server mentioned.
Author:     Lakshmana Pamarthy
-----
```

```
-----
package java.rmi.examples.echo;
```

```
/**
This is the main part of the program which defines the interface of one of the software
examples. This code fragment implements the applet interface. Along with extending the
java.applet.Applet interface, the program extends runnable ( which is used for calculating
```

the lookup time by implementing threads). The code consists in implementing the user interface along with calling the main routines from the other source files. The compiled class file is in the sub directory named "test". The code imports various rmi object files for its compiling.

```

*/

import java.io.*;
import java.awt.*;

import java.util.Date;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.RegistryImpl;
import java.rmi.registry.RegistryImpl_Stub;
import java.rmi.registry.RegistryImpl_Skel;
import java.rmi.NotBoundException;
import java.rmi.UnexpectedException;
import java.rmi.UnknownHostException;
import java.rmi.AlreadyBoundException;
import java.rmi.AccessException;
import java.rmi.NoSuchObjectException;
import java.rmi.RemoteRuntimeException;
import java.rmi.RemoteException;
import java.rmi.Remote;
import java.rmi.StubSecurityException;
import java.rmi.Naming;
import java.rmi.UnknownServiceException;
import java.rmi.StubNotFoundException;
import java.rmi.examples.util.FlexGridLayout;
import java.net.*;

/**
The following routine is similar to the main routine. It declares all the Alternative
window Toolkits, like the TextFields, Buttons, and Layouts.
*/

public class EchoApplet extends java.applet.Applet implements Runnable {
    TextField hostfield;
    TextField argfield;
    TextField resultfield;
    TextField statusfield;
    TextField serverfield;
    Thread clockThread;    // Initiates the Thread class.
    int search_time = 0;
    int initial_time = 0;
    Button lookupbutton;
    Button invokebutton;
    Button Reset;
    Button Default;
    Button Status;
    Echo echo;

    public synchronized void init() {

        start(); // This starts the thread for initial lookup time.

        String url = getParameter("url"); // gets the text from the hostfield
        if (url == null) {
            try {
                SecurityManager mgr = System.getSecurityManager();
                // Get information about the Installed Security Manager.
                Object context;
                String host;
                if (mgr != null &&
                    (context = mgr.getSecurityContext()) != null &&
                    context instanceof URL) {
                    host = ((URL)context).getHost(); // Case where given URL already in

```

```

    } else {
        // The Echo context.
        String prot = getCodeBase().getProtocol(); // Get the hostname.
        if (prot.equals("file"))
            host = InetAddress.getLocalHost().getHostName();
        else
            // Get the local host name.
            host = getCodeBase().getHost();
    }
    url = "://" + host + "/Echo1"; //Standard rmi lookup format.

    clockThread.suspend(); // Stop the thread since lookup is over
    System.out.println("Initial Lookup Time:"+(search_time-initial_time));
    // Compute the time for the initial lookup.
} catch (java.net.UnknownHostException ex) {
    showStatus("Default host is unknown."); // Unexisting Server name
}
}

// Create the url label and textfield as panel using the default flowlayout.

setLayout(new FlexGridLayout(0,3)); // Layout required for aligning all the
// toolkits

add(new Label("URL:", Label.RIGHT)); // Label for entering host address
hostfield = new TextField(30); // Text area for entering host address
hostfield.setText(url);
add(hostfield); // add the text area to the panel
add(lookupbutton = new Button("Look")); // create the Lookup button

// Create the values group
add(new Label("Message:", Label.RIGHT)); // Label for entering message
argfield = new TextField(30); // Text area for entering message
add(argfield); // add the text area to the panel
add(invokebutton = new Button("Inv")); // create the Invoke button

add(new Label("Return:", Label.RIGHT)); // Label for getting result
resultfield = new TextField(40); // Text area for getting message
resultfield.setEditable(false);
add(resultfield); // add the text area to the panel
add(Reset = new Button("Reset")); // create the Reset button

add(new Label("Status:", Label.RIGHT)); // Label for displaying the default host
statusfield = new TextField(40); // Text area for displaying the default host
statusfield.setEditable(true);
add(statusfield); // add the text area to the panel
add(Default = new Button("Default")); // create the Default button

add(new Label("Active Server:", Label.RIGHT)); // Label for getting status
serverfield = new TextField(40); // Text area for getting Server status
serverfield.setEditable(true);
add(serverfield); // add the text area to the

panel
add(Status = new Button("Status")); // create the Status button
getEchoServer(hostfield.getText());
}

// this is a routine written for all layout programs for removing all the window toolkits

public void destroy() {
    removeAll();
}

// this is the action routine which invokes different actions for different user
// responses. There are four buttons and two text fields which are sensitive to user
// actions. Each action in turn calls different routine.

```



```

public boolean action(Event ev, Object obj) {
    if (ev.target == hostfield) {
        getEchoServer((String)ev.arg); //If user types something and hits return,
                                        // it calls the getEchoServer routine.
    } else if (ev.target == lookupbutton) {
        start(); // If the user wants to lookup a host,
                // the timer thread starts.
        getEchoServer(hostfield.getText()); // Searches for the host
        stop(); // this indicates the end time for looking
                // for the host.
    } else if (ev.target == argfield) {
        doEcho((String)ev.arg); // Perform the echo function
    } else if (ev.target == invokebutton) {
        doEcho(argfield.getText()); // do the same thing as above
    } else if (ev.target == Reset) {
        showCommand(); // Disconnect the host
    } else if (ev.target == Default) {
        setHost();
    } else if (ev.target == Status) {
        showServer(hostfield.getText()); // Brief description of the current server
    }

    return true;
}

// Common window routine for handling extreme conditions where the control is returned
// to the operating system.

public boolean handleEvent(Event ev) {
    boolean rc = super.handleEvent(ev);
    return rc;
}

// This routine does the function of getting the mentioned server. the host mentioned
// needs to run the rmi server. If the host is already looked once, the program checks its
// context for getting it, else it throws the remote exceptions.

public void getEchoServer(String where) {
    try {
        Remote obj = Naming.lookup(where); //search the name server
        if (obj instanceof Echo) {
            echo = (Echo)obj;
            showStatus("Found: " + where + ".");
        } else {
            showStatus("Named object is not an Echo server.");
        }
    } catch (NotBoundException ex) {
        showStatus("Name not Bound.");
    } catch (java.net.UnknownHostException ex) {
        showStatus("Host " +where+ " unknown.");
    } catch (RemoteException ex) {
        // ex.printStackTrace();
        showStatus("RemoteException during lookup.");
    } catch (java.net.MalformedURLException ex) {
        showStatus("Malformed URL " + where);
    }
}

// this routine performs the echo. It gets the message from the host. If no host is
// specified, it comes up with an error message else it calls the echo instance at the
// host and displays the result.

public void doEcho(String arg) {
    String test = hostfield.getText();

```

```

    if(test.equals(""))
        showStatus("No Host Specified");

    else {
        try {
            String result = echo.call(arg);
            resultfield.setText(result);
            showStatus("");
        } catch (RemoteException ex) {
            // ex.printStackTrace();
            showStatus("RemoteException during echo.");
        }
    }
}

// This is a small routine which displays the server status at any given instance.

public void showStatus(String arg) {
    statusfield.setText(arg);
}

// It gives the information about the active server in use.

public void showServer(String arg) {
    if (arg.equals(""))
        serverfield.setText(" No Server Active.");
    else
        serverfield.setText("rmi:" + arg);
}

// This routine sets the different textfields when a reset button is used.

public void showCommand() {
    hostfield.setText("");
    resultfield.setText("");
    argfield.setText("");
    statusfield.setText("Host Disconnected, Type New URL");
    serverfield.setText("");
}

// This routine disconnects the host when the reset button is used.

public void setHost() {
    String url = getParameter("url");
    hostfield.setText(url);

    if (url == null) {
        try {
            SecurityManager mgr = System.getSecurityManager();
            Object context;
            String host;
            if (mgr != null &&
                (context = mgr.getSecurityContext()) != null &&
                context instanceof URL) {
                host = ((URL)context).getHost();
            } else {
                String prot = getCodeBase().getProtocol();
                if (prot.equals("file"))
                    host = InetAddress.getLocalHost().getHostName();
                else
                    host = getCodeBase().getHost();
            }
        }
        url = "://" + host + "/Echo1";
        hostfield.setText(url);
        getEchoServer(url);
        showStatus("Default URL: " + url + ".");
    }
}

```

```

        } catch (java.net.UnknownHostException ex) {
            showStatus("Default host is unknown.");
        }
    }
    else
        getEchoServer(hostfield.getText());
}

// This starts the thread and initializes the thread. The thread once started runs until
// it is either suspended or stopped. The thread is still in the start state and is
// changed to the running stage in the run() routine.

public void start() {

    if(clockThread == null) {
        clockThread = new Thread(this, "EchoApplet"); // constructor
        clockThread.start();
    }
}

// In this run method we overload the threads run() and write our own code. Here i have
// used the thread to sleep for a milli second so that it acts like a counter whose
// precision is one milli second.

public void run() {
    clockThread.resume();
    while (clockThread != null) {
        try {
            search_time++; // Increment the search time
            clockThread.sleep(1); // sleep for one milli second
            System.out.println(search_time);
        } catch( InterruptedException e) {
        }
    }
}

// This routine stops the thread and cleans the garbage. It also gives the total search
// time.

public void stop() {

    if( clockThread != null && clockThread.isAlive() )
        System.out.println("Total Search Time: "+(search_time-initial_time));
        initial_time = search_time;

    clockThread.stop();
    clockThread = null;
}
}

-----
File:      EchoClient.java
Type:      Client Program
Parameters: None
Description: Looks up for the remote object in the Naming.lookup() class and comes up with
            an exception if the host is not found.
Source:    Sun Micro Systems, Inc.,
-----

package java.rmi.examples.echo;

/* This is the client program which looks up the remote object using the naming class and
   invokes the methods on the remote object. It include looking up for the named server in
   the Naming.lookup() class and catching any exceptions.
*/

```

```
// The following are the library classes.

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.RegistryImpl;
import java.rmi.registry.RegistryImpl_Stub;
import java.rmi.registry.RegistryImpl_Skel;
import java.rmi.NotBoundException;
import java.rmi.UnexpectedException;
import java.rmi.UnknownHostException;
import java.rmi.AlreadyBoundException;
import java.rmi.AccessException;
import java.rmi.NoSuchObjectException;
import java.rmi.RemoteRuntimeException;
import java.rmi.RemoteException;
import java.rmi.Remote;
import java.rmi.StubSecurityException;
import java.rmi.Naming;
import java.rmi.UnknownServiceException;
import java.rmi.StubNotFoundException;
import java.rmi.examples.util.FlexGridLayout;
import java.rmi.server.StubSecurityManager;

public class EchoClient {

    public static void main(String args[])
    {
        // Create and install the security manager.
        System.setSecurityManager(new StubSecurityManager());

        try { // Search for the given hostname in the Naming.lookup().

            for(int i=1; i<=5; i++) {
                String name = "Echo" + i;
                System.out.println("EchoClient: lookup " + name);
                Echo echo = (Echo)Naming.lookup(name);

                String message = echo.call("Hi #" + i);
                System.out.println("EchoClient: message from " + name + ":");
                System.out.println("\t" + message + "\n");
            }
        } catch (Exception e) { // Exception occurred.
            System.out.println("EchoClient: an exception occurred: " +
                e.getMessage());
            e.printStackTrace();
        }
        System.exit(0);
    }
}

```

---

```
File:      EchoImpl.java
Type:      Interface File
Parameters: None
Description: Defines the standard interface for the methods which are declared remote.
            Specifies the necessary syntax for declaring a method remote.
Source:    Sun Micro Systems, Inc.

```

---

```
package java.rmi.examples.echo;

// This is used to create the echoImpl interface. This must extend the standard
// UnicastRemoteServer and should implement the standard echo. The following are the
// library classes.

```

```

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.RegistryImpl;
import java.rmi.registry.RegistryImpl_Stub;
import java.rmi.registry.RegistryImpl_Skel;
import java.rmi.NotBoundException;
import java.rmi.UnexpectedException;
import java.rmi.UnknownHostException;
import java.rmi.AlreadyBoundException;
import java.rmi.AccessException;
import java.rmi.NoSuchObjectException;
import java.rmi.RemoteRuntimeException;
import java.rmi.RemoteException;
import java.rmi.Remote;
import java.rmi.StubSecurityException;
import java.rmi.Naming;
import java.rmi.UnknownServiceException;
import java.rmi.StubNotFoundException;
import java.rmi.examples.util.FlexGridLayout;
import java.rmi.server.UnicastRemoteServer;
import java.rmi.server.StubSecurityManager;

public class EchoImpl
    extends UnicastRemoteServer
    implements Echo
{
    private String name;

    // The standard form of the Implementation interface is to have a no arg constructor which
    // throws a RemoteException or a constructor which calls super() and throws Remote
    // Exception.

    public EchoImpl(String s) throws RemoteException {
        super();
        name = s;
    }

    public String call(String message) throws RemoteException
    {
        return name + "Message Transmitted:" + message;
    }

    public static void main(String args[])
    {
        // Create and install the security manager
        System.setSecurityManager(new StubSecurityManager());

        try {
            for(int i=1; i<=5; i++) // create five echo implements and name them Echo1
            // through Echo5

            {

                System.out.println("EchoImpl.main: create an EchoImpl");
                String name = "Echo" + i;
                EchoImpl echo = new EchoImpl(name);

                System.out.println("EchoImpl.main: bind it to name: " + name);
                Naming.rebind(name, echo);
            }
            System.out.println("Echo Server ready.");

        } catch (Exception e) {
            System.out.println("EchoImpl.main: an exception occurred: " +
                e.getMessage());
            e.printStackTrace();
        }
    }
}

```

}

```
-----
File:      Index.html
Type:      Interface File
Parameters: EchoApplet.class
Description: This file is used as the input to the appletviewer. The appletviewer looks
            for the file named in the code part mentioned, which in this case is
            java.rmi.examples.echo.EchoApplet. The file is looked in the sub directory
            mentioned in the codebase part of the file.
Author:    Lakshmana Pamarthy
-----
```

```
<applet codebase="test"
        code="java.rmi.examples.echo.EchoApplet"
        width=500 height=120>
</applet>
```

```
-----
File:      Readme.txt
Type:      Documentation File
Description: Provides information about setting up the software and trouble shooting the
            problems faced while running the program.,
Author:    Lakshmana Pamarthy
-----
```

The program requires the following:

- Microsoft Windows95 installed in the computer
- Network chord attached
- Java Developers Kit 1.0.2 installed( optional )

The program has two directories:

- rmi.
- java

There are two batch files:

- load.bat
- start.bat

All the source files are in the sub directories included in rmi.

The java directory consists of JDK1.0.2. It is used for compiling source files and using the appletviewer for executing html files.

To run the executable file make changes in the file load.bat as follows:

if the drive used is, say C drive, then the classpath line should be similar to:

```
SET CLASSPATH=c:\rmi\examples\echo\test;c:\rmi\lib\rmi.zip
SET RMIHOME=c:\rmi
```

```
PATH=c:\java\bin;c:\rmi\bin;c:\windows\command;PATH%;
```

the rest of the commands are same. Essentially, change the drive from H:\, as given in the file, to the drive under use.

When running the batch file, if any of the statements gives an error "Out Of Environment Space", then there would be a problem while loading the applet. The suggested solution is to restart the machine and try again.

In order to run the echo example on two machines:

- run "javaw java.rmi.registry.RegistryImpl" on hostA
- run "javaw java.rmi.examples.echo.EchoImpl" also on hostA
- run "appletviewer rmi/examples/echo/index.html" on hostB

the URL that to type into the URL field to look up an echo object is the following:

```
rmi://hostA/Echo1  
rmi://hostA/Echo2
```

When running the echo example on one machine, the URL that needs to be typed is `rmi://hostA/Echo1`, provided the host also runs the server.

There are five buttons, Look, Inv, Reset, Default, Status.

Look is used to find the remote host given in the Hostfield.

Inv is used to invoke the Echo call on the remote host.

Reset is used to disconnect the remote host.

Default sets the host to the local host, i.e., the host name of the machine used.

Status gives an indication of which host methods are used at the instant of time.

When the Inv button is used, the Clock thread is activated and there is display of the clock counter on the user window. This gives the estimated time taken to connect to the mentioned site.

Echo program implements five echo servers on every host it runs. So the syntax of the hostname is thus: `rmi://hostname/Echo1` through `rmi://hostname/Echo5`.

The new source files for the Echo program are in the sub directory, `rmi/examples/echo`. The compiled classes are in the subdirectory `rmi/examples/echo/test/java/rmi/examples/echo`.

The source files for the example are listed as follows:

Echo.java:

Standard Interface that defines the remote behavior.

EchoImpl.java:

Implementation and sample Main that creates several echo servers and registers them with the registry on the local host.

EchoClient.java:

A client that looks in the local registry for named echo servers and invokes its call method.

EchoApplet.java:

An applet that finds a named echo server and lets a user invoke the remote interface. This applet uses the java Threads to calculate the lookup time.

index.html:

Web page with EchoApplet embedded.

VITA

Lakshmana Pamarthy

Candidate for the Degree of

Master of Science

Thesis: PORTING OF AN EXISTING SOFTWARE FROM THE SUN  
WORKSTATIONS TO A PERSONAL COMPUTER ENVIRONMENT

Major Field: Computer Science

Biographical:

Personal Data: Born in Bhadrachalam, Andhra Pradesh, India, on May 23, 1974,  
son of Lakshmi Narayana and Varalakshmi Pamarthy.

Education: Graduated from St. Mary's Junior College, Hyderabad, India in August  
1990; received Bachelor of Science in Mechanical Engineering, Andhra  
University, Waltair, India in August 1994. Completed the requirements for the  
Master of Science Degree in Computer Science at the Computer Science  
Department at Oklahoma State University in December 1996.

Experience: Client Services Support Group, Computing and Information Services,  
Oklahoma State University, September 1995 - November 1996.