Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num. SP03001 Call Num:

General Guidelines:

- Library books should be used with great care.
- Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT

SP03001

DISTRIBUTED LIBRARY SYSTEM

By

SHRUTI MALIK 031215
NEHA SHARMA 031276
TARANDEEP KAUR 031280

1093001

MAY 2007

Submitted in the partial fulfillment of degree of Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY-WAKNAGHAT

CERTIFICATE

This is to certify that the work entitled, "Distributed Library System" submitted by Shruti Malik(031215),Neha Sharma(031276),Tarandeep Kaur(031280) in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science and Engineering of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

25.05.07

Mr. Ajay Kumar Singh

ACKNOWLEDGEMENT

Many people have contributed to this project in a variety of ways over the past few months. We also acknowledge the many helpful comments received from our teachers of the Computer Science department and visualization courses and seminars. We would like to express heartfelt gratitude our friends without whose support and encouragement this project would not have been possible. We are indebted to all those who provided reviews and suggestions for improving the materials and topics covered in our package, and we extend our apologies to anyone we might have failed to mention.

Thanks to our Sr. Lecturer Mr. Ajay Kumar Singh

SHRUTI MALIK (031215)

NEHA SHARMA (031276)

TARANDEEP KAUR (031280)

CONTENTS

CHAPTERS

CHAPTER 1

1.1 INTRODUCTION 9
1.2 MOTIVATION 10
1.3 DISTRIBUTED LIBRARY: OUR PERSPECTIVE 10

CHAPTER 2

2.1 PROJECT WORK 13
2.2 DATABASE CONNECTIVITY
2.2.1 INTRODUCTION 13
2.2.2 MICROSOFT ACCESS 14
2.2.3 ADVANTAGES 14
2.2.4 CREATING A DATABASE 14
2.2.5 REGISTRING A DATABASE 16
2.2.6 JDBC 16
2.2.7 CONNECTONG TO A DATABASE 17

CHAPTER 3

INTRODUCTION TO JAVA RMI

3.1 INTRODUCTION TO RMI 20
3.2 OVERVIEW 22
3.3 HOW DOES RMI WORK 23
3.3.1 REMOTE INTERFACE,OBJECTS AND METHODS 25
3.3.2 CREATING DISTRIBUTED APPLICATION USING RMI 26
3.4 ADVANTAGES OF RMI 36

CHAPTER 4

SOURCE CODE AND TEST RESULTS

4.1 MYCLIENT MODULE 39
4.2 MY SERVER IMPLEMENTATION MODULE 40
4.3 MY SERVER MODULE 42
4.4 QUERY DATABASE MODULE 43

4.5 RUNNING THE SERVER 45
4.6 RUNNING THE CLIENT 48

CHAPTER 5

USER INTERFACE

5.1 MODULES 52
5.2 DATABASE 63
5.3 DIRECTORIES AND STRUCTURES 64

CHAPTER 6

FLOWCHARTS

6.1 LOGIN 65
6.2 USER HOMEPAGE 66
6.3 SEARCH 67
6.4 ADD A BOOK 68
6.5 REMOVE/UPDATE BOOK DETAILS 69
6.6 SNAPSHOTS 70

CONCLUSION 77
BIBLIOGRAPHY 78

LIST OF FIGURES

1. Steps to Develop RMI 21
2. RMI working 25
3. RMI Clients and Servers communication through Stubs and Skeleton 34
4. Test Results
4.1 Experimental Setup and Testing 44
4.2 Compiling files in Java 45
4.3 Compiling the implementation class in RMI 46
4.4 Registering using rmiregistry 47
5. Start the Server 47
6. Run the Server passing policy file as argument 48
7. Running the Client Side
    - Compiling and run the Client files 48
    - Case: When no Server is running 49
    - Case: Data not found on any Server 49
    - Case: Server running,data found and output show 50

6. Flow Charts 65

7. Snapshots of Library System 70

LIST OF ABBREVIATIONS

1. **JDBC** Java Database connectivity
2. **JDK** Java Development Kit
3. **JNI** Java Native Interface
4. **IP** Internet Protocol
5. **ODBC** Open database connectivity
6. **RMI** Remote Method Invocation
7. **RPC** Remote Procedure Call
8. **SQL** Structured Query Language

ABSTRACT

NEED FOR DISTRIBUTION

Presently in such a big world collecting information can be a tiresome and time consuming task and this problem gets aggravated but the physical location of the host from which the information is required. Moreover all the information required may not be present on a single system. And since the reliability of the system also plays its part centralizing the information on one system so it is not a healthy practice. With these problems in mind a shift has been made to distributed systems. In distributed systems all the systems are pooling their resources on one domain and any node can make use of those resources. Hence in a way no system will ever head towards a situation where it finds itself heavily loaded with work or out of resources since it can use the resources from the common pool without the user intervention.

OUR APPROACH

Our main stress is to generate applications in distributed environment making use of RMI(Remote Method Invocation). Our application will take this supposition that there is a library of thousands of books distributed over a large network. The user will address his query for the search of a particular book by some author name. The system would redirect his query to all other systems, initiate the search and return the compiled results to the user.

CHAPTER 1

INTRODUCTION

OVERVIEW

"Man's longing for perfection finding finds theory for optimization. It studies how to describe and attain what is the best, once one knows how to measure and alter what is good or bad."

Nowadays in a world so ever increasing the scaling of any problem is a tough task it becomes a really difficult task for a user far away to find an optimal result for any type of query. When all the people are trying to get the work from far distant away places the present domain just try to get a server client model. But in model like this it becomes difficult to get all the information on one system since in such a case the reliability of the model on one centralized system is low and also the overhead required for storing such a information is high. But since everything is guided by laws of nature and no one can defy it therefore people have to try to device a model. A model that can support information gathering from any place any time any anywhere. With this in mind people are moving to Distributed system. But distributed computing has a lot of complexity. In today's world the economics plays a major role, where time is money and we will develop a distributed library which will minimize the searching time and improve the marginality.

There has been a lot of work that has been going on in this direction and the people are trying to remove scalability from the existing aspect and make people reach each other. A large number of algorithms have been tried and tested in this field but a system in which the machine can itself select the algorithms and depending upon overhead and overload on the system would be very efficient one. Think of a search engine where you would browse a site and put the name of a book you want to select and the machine would itself route upon different algorithms on different machines depending upon earlier results that were collected and the overhead and overload on the other systems.

MOTIVATION TO THE PROJECT

Sometimes when we try to gather details about some information that is related to our work we need to find it in our database system and if that information is not with us and worse when we are not connected with the internet it becomes a problem. But if we have a system in which we are connected to local network in a distributed environment it can be really helpful. But still there are lot of complexities associated with it since in a distributed network where there are a lot of machines on the network if we are trying to gather the information there is no way we can do it till we spend a lot of time searching for that piece of relevant information on each system on the network independently. But it can be done if we create a system that would associate information regarding each node and maintain a database of all that and carry out the searching on its own, returning the complete compiled information to the user. The goal of this project is that if all the systems are joined to each other by the network that has a distributed feature then on the basis of the query results would be generated, compiled and returned to the user. Hence defying all the boundaries and bringing the widespread book database to our doorstep and making each user able to gather information at any time and at any place.

DISTRIBUTED LIBRARY: OUR PERSPECTIVE

According to our approach we are trying to build a digital library based on the following models:

- **SELECTING A PROPER PLATFORM(RMI)**
- **SELECTING PROPER SEARCH ALGORITHM**
- **DESIGNING MODULES**

- **INTEGARTING OUR MODULES IN RMI.**

SELECTING PROPER PLATFORM

In the past, the use of sockets was the primary way for applications to communicate with each other. This of course was not an object-oriented approach to communication and, if you have ever worked with socket code, you realized real fast just how tedious it was to create a client/server architecture that had some complexity to it and performed all the necessary operations you may have needed. Remote Procedure Call (RPC) services were the next attempt at eliminating the complicated communication layer of using sockets and to also make it easier for programmers to call remote procedures, but the parameters that could be passed to these procedures usually weren't very complex. If the need to pass more complex parameters arose, the burden would lie on the programmer to process the types and perform monotonous conversions. Plus, the parameters were usually not very portable between languages.

RMI picks up where RPC services left off by being designed in an object-oriented fashion which allows programmers to communicate using objects and not just predefined data types that are language-centric. These objects can be as complex as you need them to be and values that are returned can be of any type. The communication layer is completely hidden from the programmer, which allows you to concentrate on more important aspects of programming, like the business logic.

RMI makes applet coding a dream since you can now have your applets easily communicate with backend distributed systems. RMI is also very secure and uses security managers to prevent malicious code from attacking your network. If your applications require multithreading, RMI also supports threads flawlessly.

SELECTING PROPER SEARCH ALGORITHM

String matching and searching is a very importánt subject in the wider domain of text processing. String matching algorithms are basic components used in implementations of practical software. Our project also uses string searching and matching algorithms to find the authors and books by the authors.

We studied many string searching and matching algorithms from various sites and found the following algorithms useful for our problem.

Soundex Algorithm

This algorithm was chosen so as to give the user a liberty to the give the name of the author of the author and search for the books written by the author even if he does not remember the correct name of author. The soundex algorithm generates codes for characters on the basis of how they sound. And groups the similar sounding words into one group and gives the various groups an integer code. The string is then minimized to 4 digit code. The similar sounding words have the same soundex code. The search can be made on the basis of generated code.

INTEGRATING OUR MODULES IN RMI

The four main modules in our project are

1. The Interface.
2. The Implementation of the interface.
3. The Client side.
4. A separate module for ODBC connectivity.

After the designing of the individual components, we integrate and implement these modules collectively as a single entity

CHAPTER 2

PROJECT WORK

JAVA DATABASE CONNECTIVITY

Introduction

A ***Distributed Library*** requires access to a database and in such a manner that an individual file can be accessed easily and the queries be executed conveniently. Java provides solid capabilities for two types of file processing. Sequential file processing is appropriate for applications in which most or all of the file's information is to be processed. Random Access file processing is appropriate for applications in which it is crucial to be able to locate and possibly update an individual piece of data quickly, and in which only a small portion of a file's data is to be processed at once. One problem with each of these methods is that they simply provide for accessing data – they do not offer any capabilities for querying the data conveniently. Therefore none of the above mentioned methods is fully successful to be used in our case. This problem is solved using database system.

Database systems not only provide file-processing capabilities, they organize data in a manner that facilitates satisfying sophisticated queries. The most popular style of database system on the kinds of computers that use java is the ***Relational Database***. A language called ***Structured Query Language (SQL)*** is almost universally used among relational database systems to make queries. Java enables programmers to write code that uses SQL queries to access the information in relational database systems. Some popular relational database software packages include Microsoft Access, Sybase, Oracle, Informix, and Microsoft SQL Server.

Having studied all the various database systems we chose MS Access and the Structured Query Language as the database system and query language respectively.

Microsoft Access

A ***database*** is a collection of data of a particular type. It is an organized collection of data viewed as a whole instead of separate unrelated files. A ***Relational Database*** is a multi-table database where the tables in the database have to be related for storing or retrieving data. A ***Relational Database Management System (RDBMS)*** is used to create and maintain relational databases.

Main advantages of using MS Access:

- Redundancy can be reduced.
- Inconsistency can be avoided.
- The data can be shared.
- Standards can be enforced.
- Security restrictions can be applied.
- Integrity can be maintained.
- Conflicting requirements can be balanced.

Creating a database:

Step I: Identify the types of objects to be created.

An access database consists of the following types of objects - Tables, Queries, Forms, Reports, Data Access Pages, Macros, and Modules. In this case we use Tables.

Step II: Identify the names and fields of the table.

The names of the fields are book name, studentname, password, date of issue, date of return. The name of the table is Student.

Step III: Identify the data type of each field in the tables.

An access database allows the following types of data types.Text, memo, number, date/time, currency, autonumber, yes/no, OLE object, Hyperlink, lookup wizard.

Step IV: Identify the fields that contain unique values in the tables.

Primary Key

The primary key ensures that there are no duplicate rows in the table. Every table must have a column that uniquely identifies each row in the table. In our case we use the login as the primary key and the book number.

Foreign Key

When primary key of a table appears as a field in another table, it is called a foreign key.

Composite Key

When a primary key contains a combination of one or more fields, it is called composite key.

Step V: Identify the method to create the database.

Access offers two methods:

- Using the database wizard
- Using a blank database

We chose the second option.

Step VI: Identify the name and location where the database has to be saved.

By default a database is saved in My Documents folder in the C drive unless specified otherwise. We chose the default setting only.

Step VII: Identify the method to create the tables.

Access offers three methods:

- Using datasheet view.
- Using the table wizard.
- Using the design view.

We chose the third option. Besides that we also added entries to the table from the command prompt using the update command of SQL.

Registering the database:

After the database is created, the next important step is to register the database as an ODBC source. The following steps are followed in order to register the database.

- Go to the *Control Panel* from the Start Menu.
- Open the *ODBC Data Source Administrator* dialog by double-click.
- Make sure the *User DSN* tab is selected and click *Add.*
- *Create New Data Source* dialog appears, select the *Microsoft Access Driver* and click *Finish.*
- The *ODBC Microsoft Access 97 Setup* dialog appears. Here enter the name of the data base.
- Click the *Select* button to display the *Select Database* dialog. Using this dialog locate and select the database. When done click *ok.*
- Click *OK* and dismiss the dialog box.

Now the database is registered to be used as ODBC data source. We can now access the ODBC data source using the JDBC-to-ODBC bridge driver.

JDBC:

The Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases – SQL databases and other tabular data sources, such as spreadsheets or flat files. The JDBC API provides a call-level API for SQL-based database access.

JDBC technology allows you to use the Java programming language to exploit "Write Once, Run Anywhere" capabilities for applications that require access to enterprise data. With a JDBC technology-enabled driver, you can connect all corporate data even in a heterogeneous environment

Connecting to the database:

The main task of connecting to the database is followed after the registration of the database as ODBC data source. This is done as follows. Please note the complete coding is attached in the end.

Import java.sql.*;

Connection conn;

Import the package "java.sql" which contains classes and interfaces for manipulating relational databases in Java. We declare a Connection reference (package java.sql) called conn. This will refer to an object that implements interface Connection. A Connection object manages the connection between the Java Program and the database. It also provides support for executing SQL statements to manipulate the database and transaction processing.

String url = "jdbc:odbc:db3";

The above mentioned line specifies the URL (Uniform Resource Locator) that helps the program locate the database. The URL specifies the protocol for communication (jdbc), the sub protocol for communication (odbc) and the name of the database (db3). The sub protocol odbc indicates that the program will be using jdbc to connect to a Microsoft ODBC data source.

ODBC is a technology developed by the Microsoft to allow generic access to disparate database systems on the Windows platform. The Java 2 Software Development Kit (J2SDK) i.e. the software used comes with the JDBC-to–ODBC bridge database

driver to allow any java program to access any database source. The driver is defined by class JdbcOdbcDriver in package sun.jdbc.odbc.

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

The class definition for the database driver must be loaded before the program can connect to the database. It uses static method forName of class Class (package java.lang) to load the class definition for the database driver. This line throws a java.lang.ClassNotFoundException if the class cannot be located.

Conn = DriverManager.getConnection(url);

This line uses the static method getConnection of class DriverManager (package java.sql) to attempt a connection to the database specified by url. The username and password arguments can also be passed here if the data source is set up to require a username and password. Our database does not use the username and password. If the DriverManager cannot connect to the database, method getConnection throws a java.sql.SQLException. If the connection attempt is successful it proceeds to the following statement.

Statement stat = con.createStatement();

It declares a Statement (package java.sql) reference that will refer to an object that implements interface Statement. This object will submit the query to the database. It invokes Connection method createStatement to obtain an object that implement the Statement interface. We can now use stat to query the database.

ResultSet result1 = stat.executeQuery(query);

This line declares a ResultSet (package java.sql) reference that will refer to an object that implements interface ResultSet. When a query is performed on a database, a ResultSet object is returned containing the results of the query. The methods of interface ResultSet allow the programmer to manipulate the query results. The query passed as the argument is either the actual SQL query or a string containing the query.

Result1.next ();

This moves the ResultSet cursor that keeps track of the current record in the ResultSet to the next record in the ResultSet. This method returns false if there are no more records in the ResultSet. Since initially the pointer points to the beginning of the records and is in an invalid cursor state this method is called.

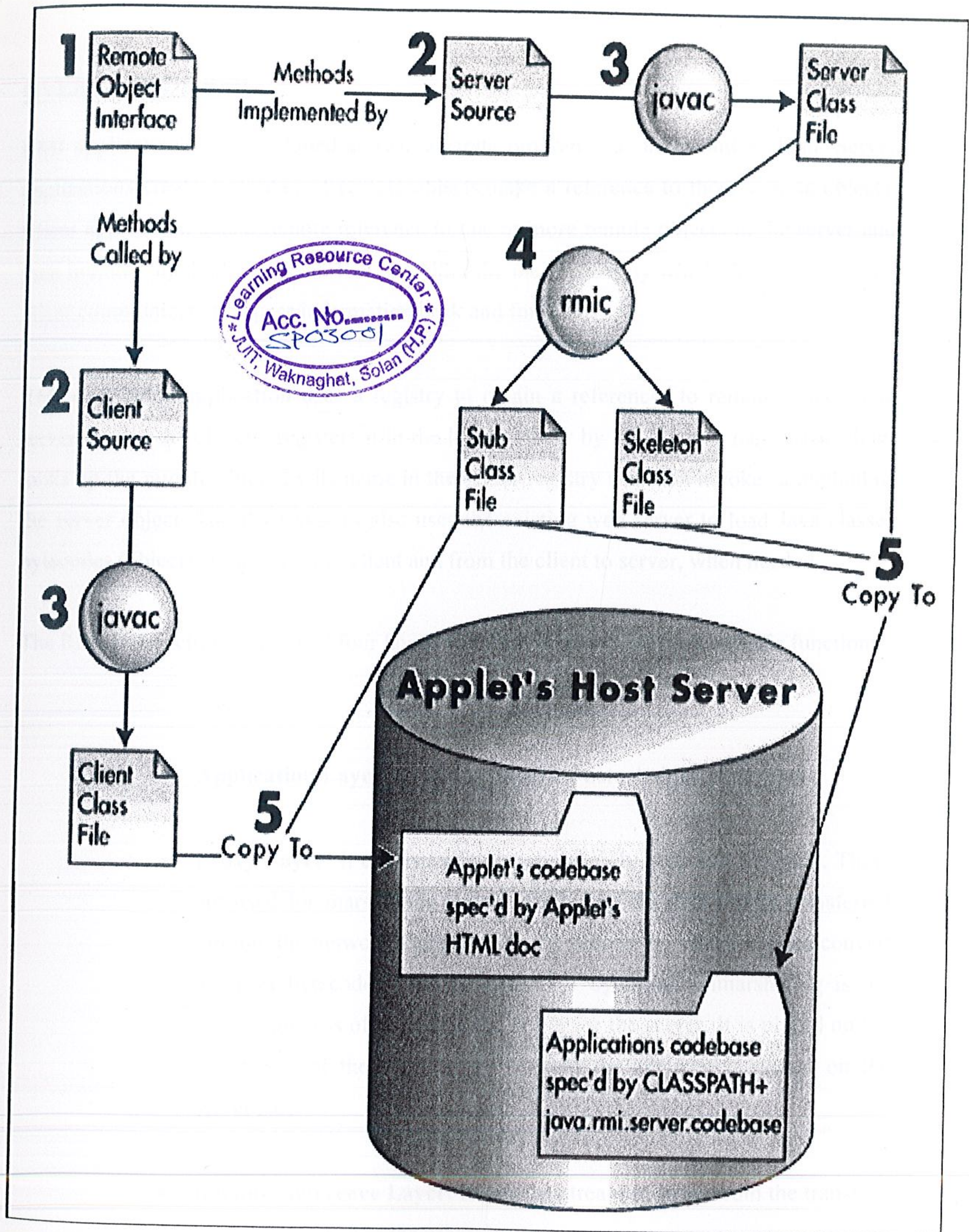
CHAPTER 3

INTRODUCTION TO JAVA RMI

Remote Method Invocation(RMI) is a part of Java Development Kit.It allows us to develop distributed applications.Distributed Systems require computations that are running in different address spaces,practically on different machines,must be able to communicate between one machine to another. Java RMI facilitates such a communication specifically for java applications.The RMI is platform independent because Java is platform independent.The RMI can communicate only from one java virtual machine to another. Using RMI we can write reliable distributed applications that are as simple as possible.

In RMI the application is divided into objects.The objects communicate with each other through an interface.This interface is used to access the remote objects and its methods.RMI passes objects by their true type, as a result the behavior of those objects is not changed when they are sent to another virtual machine. To develop the distributed applications application using RMI we have to follow the steps given below:

- Define the interface
- Implementing these interfaces
- Compile the interfaces and their implementations with the java compliler
- Compile the server implementations with the RMI compiler
- Run the RMI registry
- Run the application



Learning Resource Center
Acc. No. SP03001
JUIT, Waknaghat, Solan (H.P.)

OVERVIEW OF RMI

RMI applications are developed as two separate programs: a server and a client. Server applications create a number of remote objects,make a reference to those remote objects. Client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

RMI distributed application uses a registry to obtain a references to remote object. The server creates the objects, registers it in the local registry by the object's name. The client looks up the remote object by its name in the server registry and then invokes a method of the server object. The RMI system also uses the existing web server to load Java classes bytecodes (objects) from server to client and from the client to server, when needed.

The RMI architecture consists of four layers and each layer can perform specific functions:

- **Application Layer**: It has contained the actual object definition

- **Proxy Layer**: It consists of two parts namely Stub and Skeleton. These are used for marshalling and unmarshaling the data that is transferred through the network.Marshaling is the process by which we can convert the java bytecodes into the stream of bytes,and unmarshaling is the reverse process of it. Stub is the proxy for the server. It is placed on the client side of the applications whereas the skeleton is placed on the server side.

- **Remote Reference Layer**: It gets the stream of bytes from the transport layer and sends it to the proxy layer.

- **Transport Layer**: This layer is responsible for handling the actual machine to machine communication.

HOW DOES RMI WORK?

RMI uses a registry to store information regarding servers that have been bound to it. This article uses the rmiregistry provided in the JDK; however, it's possible to write an RMI-based application without it.

Binding is done by calling the Naming.rebind() method in the server object's constructor (found in the java.rmi package). If the method fails, it'll throw one of the following exceptions RemoteException, MalformedURLException or UnknownHostException. In the case of RemoteException, there was an error with the registry, often occurring because rmiregistry wasn't executed before the server object attempted to bind. Once the server has been bound to the registry, a client can do a Naming.lookup() to get an instance of the RMI server object.

After the client has an instance of the server object, it'll be able to call all the methods defined in the server's list of promised remote methods. These methods are defined in an interface that both the client and server objects implement. By using rmic, we can create a stub and skeleton to use for compiling our client object.

The stub sits in the client's codebase or classpath (the client's .class file usually resides in the same directory). This stub object is what tells the client what methods may be called from the server and handles all the details that allow us to call a remote object's method via the registry.

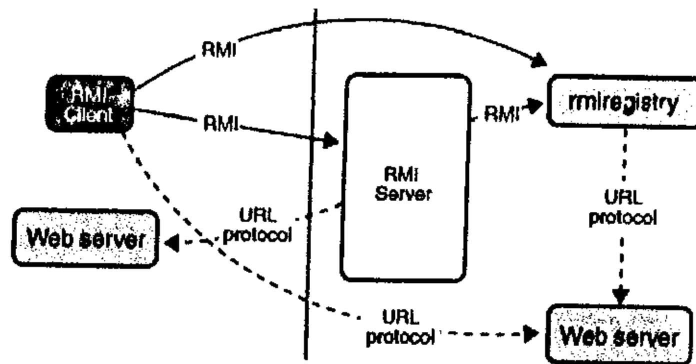
The skeleton is similiar to the stub, except it must be in the server's classpath (like the stub, the skeleton usually resides in the same directory as the .class file for the server). The skeleton handles incoming requests/parameters from clients and returns the results via the registry.

RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application.*

Distributed object applications need to do the following:

- **Locate remote objects:** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- **Communicate with remote objects:** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- **Load class definitions for objects that are passed around:** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



Advantages of Dynamic Code Loading

One of the central and unique features of RMI is its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine. All of the types and behavior of an object, previously available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine. RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine. This capability enables new types and behaviors to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application. The compute engine example in this trail uses this capability to introduce new behavior to a distributed program.

Remote Interfaces, Objects, and Methods

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called *remote objects.*

An object becomes remote by implementing a *remote interface*, which has the following characteristics:

- A remote interface extends the interface java.rmi.Remote.
- Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote *stub* for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, *only* those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Creating Distributed Applications by Using RMI

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.

2. Compiling sources.

3. Making classes network accessible.

4. Starting the application.

Designing and Implementing the Application Components

First, determine your application architecture, including which components are local objects and which components are remotely accessible. This step includes:

Defining the remote interfaces: A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.

Implementing the remote objects: Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.

Implementing the clients: Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Compiling Sources

As with any Java program, you use the javac compiler to compile the source files. The source files contain the declarations of the remote interfaces, their implementations, any other server classes, and the client classes.

Making Classes Network Accessible

In this step, you make certain class definitions network accessible, such as the definitions for the remote interfaces and their associated types, and the definitions for classes that need to be downloaded to the clients or servers. Classes definitions are typically made network accessible through a web server.

DEFINING INTERFACES

Defining interfaces is the first step to writing RMI programs. In RMI programming, any class that exports objects must implement an interface that defines the methods that can be accessed via a remote application(ie client) .The class may have method that are not defined in this interface. Then the client cannot access these methods. It is also possible for the single class to implement many remote interfaces.

Each remote object is identified by its remote interface. The remote interface is an interface that declares a set of methods that may be invoked from a remote Java Remote Machine. A client gets a handle to interface describing the remote method of an object. This interface must extend from **java.rmi.Remote**. The **java.rmi.Remote** serves to identify all remote interfaces. All the remote objects must directly or indirectly implement this interface. Only the remote interfaces can be invoked via RMI. Local interfaces cannot be called in this manner.

IMPLEMENTATION OF INTERFACES

SERVER IMPEMENTATION:

In general, the server implementation class of a remote interface should do the following:

- Declare the remote interfaces being implemented
- Define the constructor of the remote object
- Provide the implementation for each remote method in the remote interface.

The server needs to create a remote object and register it with the local registry. This procedure can be encapsulated in a main() method in the remote object implementation class itself, or it can be included in another class entirely. This procedure should

- Create one or more instance of the remote object
- Register atleast one of the remote objects with the RM registry in order to be located by perspective clients.

Defining constructor for the remote objects

The first method in the MyServerImpl class is the constructor MyServerImpl(). The constructor does not take any arguments. It throws a **Remote Exception**. Each and every remote method in a remote object should throw a Remote Exception so that the client will be informed if there is any problem in passing the parameters or in the connection between the client and the server.

Passing objects in RMI

Objects that are not remote,such as parameters, return values and exceptions, are passed by value in remote method calls. This means that a copy of the objects is created in the receiving virtual machine. Any changes to this object's state at the receiver are reflected only in the receivers copy, not in original instance.

The **main()** method is not a remote method, which means that it cannot be called from different virtual machine. Since the **main()** method is declared **static**, the method is not associated with any object. Instead, the method is associated with the class in which it is declared.

The **main()** method creates an object of the MyServerImpl class, registers it with the registry using **Naming.rebind()** and then gives the output that the server is ready to use. The java.rmi.Naming interface is used as a front-end for binding , or registering and looking up remote objects in the registry. Once a remote object is registered with the RMI registry on the local machine ,callers on the any machine can look up the remote object by name, obtain its reference, and then invoke remote methods on the object. All servers running on a machine may share the registry, or an individual server process may create or use its own registry, if desired.

The following parameters are the arguments to call **Naming.rebind():**

- An URL-formatted name associated with the remote object
- A new remote object to associate with the name

The first parameter, which is an URL formatted java.lang.String, represents the location and the name of the remote object. The location includes the IP address of the server machine. If both of these are omitted from the URL, it defaults to the IP address of the local machine.

The RMI runtime substitutes a refernce to the stub for the remote object reference specified by the argument. Remote implementation objects,such as instances of MyServerImpl mever leaves the virtual machine where they are created. So,when a client performs a lookup in the server's remote object registry, a refernce to the stub is returned.

An application can bind,unbind or rebind remote object references only with a registry on the same machine. This restriction prevents a remote client from removing or overwriting any of the entries in a server's registry. The advantage of rebind method over the bind method is that any existing binding for the same name is replaced by rebind whereas bind gives an error if a binding for the same name exists already.

Once the server has registered with the local RMI registry, it prints out a message indicating that is ready to start handling calls and then the **main** method exists. It is not necessary to have a thread wait to keep the server alive. As long as there is a reference to the server object in another virtual machine,local or remote, the server object will not be shut down, or garbage collected. The server object is reachable from the remote client because the program binds a reference to the server object in the registry.

CLIENT IMPLEMENTATION

After writing the server implementation, we move on to the client implementation. This has to be separate because this part alone should be run in all the branches of Perfect Solution Limited. For the client implementation ,we will use an applet so that the user can interact easily with the applicant. Any method in the client that calls a remote method should have a try and catch clause or throw a **RemoteException**. The applet it should import **java.applet.*** and it extends the Applet class.

The **init()** method adds two labels, the text fields and the text area to the applet and sets the **EchoCharacter** as '*' for the password text field. It looks up the registry to get an interface of the server or the remote objects, using the **Naming.lookup()** method. Suppose client part and server part resides in different machines and server resides in a machine whose **IP** address is 172.16.28.130, then the statement:

```
EK = (EmpInt)Naming.lookup("EmpServer");
```

Can be rewritten as follows:

```
EK = (EmpInt)Naming.lookup("rmi://172.16.28.130/EmpServer");
```

If we wish to specify the IP address during runtime, the statement can still be modified as follows:

```
EK = (EmpInt)Naming.lookup("rmi://" + args[0] + "/EmpServer");
```

Where **args[0]** represents the IP address of the machine.

The **Naming.lookup()** is a static method that searches for the server using the server name registered with the RMI registry. It takes in a single parameter of String type representing the **URL**-formatted name for the remote object and returns a reference for the remote object associated with the specified name. if the name is not currently bound to the registry, then it throws the **NotBoundException**. If the registry could not be contacted then lookup method throws a **RemoteException.**

ARCHITECTURE OF RMI

Architecture of RMI can be stated as:

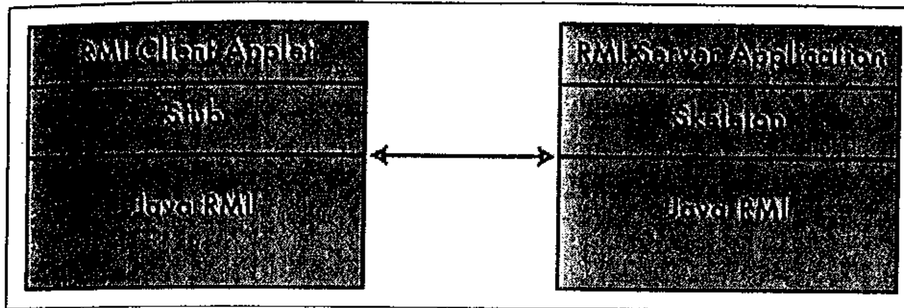
- RMI Layers
- The RMI Registry
- RMI Flow

BUILDING RMI DISTRIBUTED APPLICATION

RMI makes network communications mostly transparent to the developer, but it does require some extra steps in building and running programs.
Shown in are the major pieces of an RMI program that you need to be concerned with for the purpose of this discussion. During this discussion, consider "Java RMI" to be a black box that takes care of making the remote calls happen without you having to worry about it. Also, the client shown is an applet, but it can be any Java executable.
Clients interface with RMI through a chunk of code called a stub. Servers interface with RMI through skeletons. Stubs and skeletons are program specific, user don't have to write them. They are generated with a utility called **rmic** acting upon the class or classes containing the implementation of your remote object's interface methods. During program operation, RMI transparently loads the stubs and skeletons so that the client

and server can communicate. This means that it is very important that the platforms hosting the client and server know where to find the stub and skeleton.

Figure : RMI clients and servers communicate through stubs and skeletons.



Building an RMI program consists of five steps:

1. Write RMI interfaces to describe the methods you want exposed in your remote object.

2. Implement the interfaces in RMI server sources and call the specified methods as necessary in the client sources.

3. Compile the client and server sources into class files.

4. Run rmic on the server class which implements your RMI interface methods to generate the stub and skeleton for the remote object. On Windows, the rmic utility is an .exe. If your system's PATH variable is set up to access the Java Developer Kit's bin directory (as it would have to be if you were using javac), then simply open a DOS window and cd to the location of the server's class file where you can call rmic on it. Remember that you must specify the class file's fully qualified, case-sensitive name (including package names delimited with the '.' character) to rmic.

5. Finally, put all of the classes (stubs and skeletons are classes, too) in the right place.

RUNNNING RMI DISTRIBUTED APPLICATIONS

There are two main steps to running the RMI application.

1. Launch the rmiregistry on each server hosting RMI server objects.
2. Load the server program. If your client is an applet, users access it via their browser, so you don't need to load it. If your client is an application, then you have to load it. Make sure that your stub is in a place that can be found with your CLASSPATH setting.

Accessing the RMI Registry

RMI server applications use the RMI registry on the server to expose their remote objects with the arbitrary service names specified by their programmers.
Once a client has one reference to a remote object, it can be used to obtain others without having to access an RMI registry . However, to get that first remote object, the client will have to look it up from the naming service on the hosting server.
So, RMI servers need to bind their primary remote objects into the registry so that clients can find them.
If your server is not already running an RMI name registry, you must either start one up programmatically with the Java.rmi's **LocateRegistry** class and
its static **createRegistry(int portID)** method or after you have already loaded Java, you can enter a command at the monitor like the one below:

rmiregistry

Note: The default port for the RMI registry is 1099. If your server needs that port for something else, you can specify another port number when you start up the registry. However, then the clients will have to know the unique port number as well.

Loading the Server Program

If your server program performs a registry bind (most will perform at least one), and you attempt to load it before you load the registry, your program will throw a class not found exception as it attempts to find the registry. So, load your server application after you have loaded the registry.

In order to add a remote object to the registry, your server program must get a reference to the registry using one of **java.rmi LocateRegistry's** static **getRegistry( )** methods and then add the remote object and its name to the registry. The **Naming.rebind( )** method handles this automatically.

Clients must already know the domain name or IP address of the server hosting the registry and the registry's port number in order to access the registry to obtain a remote object. An applet already knows the connection information about its server and so can easily obtain the server's name or IP using **java.net's InetAddress** class methods.

ADVANTAGES OF RMI

At the most basic level, RMI is Java's remote procedure call (RPC) mechanism. RMI has several advantages over traditional RPC systems because it is part of Java's object oriented approach. Traditional RPC systems are language-neutral, and therefore are essentially least-common-denominator systems-they cannot provide functionality that is not available on all possible target platforms.

The primary advantages of RMI are:

- **Object Oriented**: RMI can pass full objects as arguments and return values, not just predefined data types. This means that you can pass complex types, such as a standard Java hashtable object, as a single argument. In existing RPC systems

you would have to have the client decompose such an object into primitive data types, ship those data types, and the recreate a hashtable on the server. RMI lets you ship objects directly across the wire with no extra client code.

- **Mobile Behavior**: RMI can move behavior (class implementations) from client to server and server to client. For example, you can define an interface for examining employee expense reports to see whether they conform to current company policy. When an expense report is created, an object that implements that interface can be fetched by the client from the server. When the policies change, the server will start returning a different implementation of that interface that uses the new policies. The constraints will therefore be checked on the client side-providing faster feedback to the user and less load on the server-without installing any new software on user's system. This gives you maximal flexibility, since changing policies requires you to write only one new Java class and install it once on the server host

- **Design Patterns**: Passing objects lets you use the full power of object oriented technology in distributed computing, such as two- and three-tier systems. When you can pass behavior, you can use object oriented design patterns in your solutions. All object oriented design patterns rely upon different behaviors for their power; without passing complete objects-both implementations and type-the benefits provided by the design patterns movement are lost.

- **Safe and Secure**: RMI uses built-in Java security mechanisms that allow your system to be safe when users downloading implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.

- **Easy to Write/Easy to Use**: RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java

interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation. And because RMI programs are easy to write they are also easy to maintain.

- **Connects to Existing/Legacy Systems**: RMI interacts with existing systems through Java's native method interface JNI. Using RMI and JNI you can write your client in Java and use your existing server implementation. When you use RMI/JNI to connect to existing servers you can rewrite any parts of you server in Java when you choose to, and get the full benefits of Java in the new code. Similarly, RMI interacts with existing relational databases using JDBC without modifying existing non-Java source that uses the databases.

- **Write Once, Run Anywhere:** RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine*, as is an RMI/JDBC system. If you use RMI/JNI to interact with an existing system, the code written using JNI will compile and run with any Java virtual machine.

- **Parallel Computing**: RMI is multi-threaded, allowing your servers to exploit Java threads for better concurrent processing of client requests.

CHAPTER 4

SOURCE CODE AND TEST RESULTS

MYCLIENT MODULE

```
import java.rmi.*;
import java.io.*;
import java.awt.*;

public class MyClient
{
  static String hostName="";
  static String abc="";
  static String filetoSend = "datafile.dat";
  static String PID;

 public static void main(String args[])
 {
   int tempdata=0;
  //hostName="localhost";
   PID = args[1];
   int i=0;
 }

 public static void searchhost(int i)
 {

  String hostName[]={ "172.16.28.105","172.16.28.130","172.16.28.142"};

try
{
  System.out.println("Looking up machine : "+hostName[i]+"\n\n");

  MyServer server = (MyServer)Naming.lookup("//"+hostName[i]+"/MyServer");

 System.out.println("Machine "+hostName[i]+" found and connection established.\n");
 System.out.println("Sending query to host : "+hostName[i]+"...");
 boolean endofFile = false;
 int ch;
 String buffer=""; // Have to send the query String here.

 abc = server.processData(PID, new String("")); //fileHandle.close();
 System.out.println(abc);
```

```
System.out.println("\nData file sent successfully.\n");
}

catch (Exception ex)
{
System.out.println(ex);
}

} }
```

MYSERVERIMPL MODULE

```
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import javax.swing.*;

public class MyServerImpl extends UnicastRemoteObject implements MyServer
{
  static String hostName = "";
  FileReader fileReadHandle;
  FileWriter fileWriteHandle;
  String filetoStore;

  public void setHostName(String name) throws RemoteException
  {
   hostName=name;
  }

  public MyServerImpl() throws RemoteException
  {
   super();
  }

  public static void main(String args[])
  {
   try
    {
     System.setSecurityManager(new RMISecurityManager());
```

```
        MyServerImpl instance = new MyServerImpl();
        instance.setHostName("localhost");
       //instance.setHostName("172.16.28.142");
        Naming.rebind("//localhost/MyServer", instance);
      //Naming.rebind("//172.16.28.142/MyServer", instance);
        System.out.println("\n"+hostName+" registered with RMI registry!");
      }

   catch (Exception ex2)
      {
       System.out.println("RMI Exception : " +ex2+"\n");
      }
   }

  public void setFileName(String filename) throws RemoteException
   {
    try
     {
        filetoStore = new String(filename);
        fileWriteHandle = new FileWriter(filetoStore);
      }
    catch (IOException E)
     {
      System.out.println(E);
     }
   }

  public String processData(String query,String Result) throws RemoteException
   {
    int n = 0;
   try
     {
       Result = Result.concat(QueryDatabase.getResults(query));
        if(Result.equals("Not Found") && n<3)
        {
         MyClient.searchhost(++n);
        }
        if(Result.equals("Not Found") && n==3)
         {
          Result= Result.concat("Roll no not found");
         }
        else
        {
```

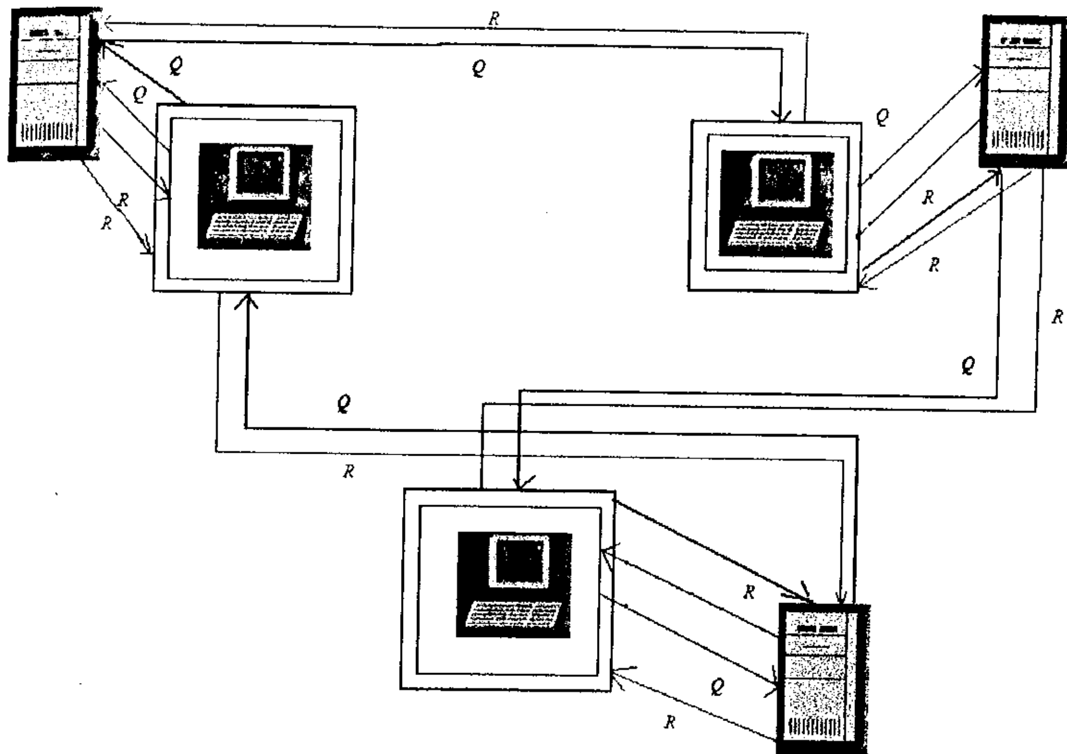
```
            DataOutputStream dout = new DataOutputStream (new                F
FileOutputStream(query+".result"));
              dout.writeBytes(Result);
             dout.flush();
               System.out.println("\nData received and written to File");
             dout.close();
          }
     }

    catch (Exception E)
     {
      System.out.println(E);
     }

   return(Result);
    }
}
```

MYSERVER MODULE

```
import java.rmi.*;
public interface MyServer extends Remote
{
 public void setFileName(String filename) throws RemoteException;
 public String processData(String query, String Result) throws RemoteException;
}
```

QUERY DATABASE MODULE

```
import java.util.*;
import java.net.*;
import java.sql.*;
import java.io.*;
import java.lang.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class QueryDatabase
{
// Contacts the Personalinfo database and returns the name and phone number

 public static String getResults(String PID)
{

String res = " ";

try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con = DriverManager.getConnection("jdbc:odbc:db3");
       // JOptionPane.showMessageDialog(null,"connection successful");
        System.out.println("Connection successful");
        Statement stat = con.createStatement();
 ResultSet result1 = stat.executeQuery( "SELECT studentname FROM student WHERE
rollno = " + Integer.parseInt(PID) + " ");
      // JOptionPane.showMessageDialog(null,"query executed");
          System.out.println("queryexecuted");
        result1.next();
        res = result1.getString(1);
       // JOptionPane.showMessageDialog(null,res);
        System.out.println(res);
}
catch(Exception E)
{ //System.out.println("Exception caught : "+E);
 res = res.concat("Not Found");
//E.printStackTrace();

}
return (res); } }
```



The working of the distributed system can be explained as shown in the Figure above. In the above figure the RED rectangle represents the system acting as Server while the BLACK rectangle represents the Client system. In a distributed system any system can at any point of time act as Server or Client or Both. The red lines show how the server works. It sends the query to the processor, and the result is returned. The black lines represent how the client works. The Client sends the query which is first checked on the same system if the same system is acting as the server also. If the result is not found the query is passed to next active server and checked for result. In the same way the query is passed from one system to another until either the result is found or there are no more servers left. The green lines in the above figure depict how the result of the query sent by the client reaches the client. One server contacts the other server using the IP address and passes the query through the interface method.In the above figure the letter 'R' represents the result while the letter 'Q' represents the query.

TEST RESULTS

RUNNING THE SERVER

1. Compiling files in java:

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\welcome>cd\

C:\>cd c:\java\bin

C:\Java\bin>javac MyServer.java

C:\Java\bin>
```

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\welcome>cd\

C:\>cd c:\java\bin

C:\Java\bin>javac MyServer.java

C:\Java\bin>javac MyServerImpl.java

C:\Java\bin>_
```

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\welcome>cd\

C:\>cd c:\java\bin

C:\Java\bin>javac MyServer.java

C:\Java\bin>javac MyServerImpl.java

C:\Java\bin>javac QueryDatabase.java

C:\Java\bin>_
```

2.Compiling the implementation class in RMI

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\welcome>cd\

C:\>cd c:\java\bin

C:\Java\bin>javac MyServer.java

C:\Java\bin>javac MyServerImpl.java

C:\Java\bin>javac QueryDatabase.java

C:\Java\bin>rmic MyServerImpl

C:\Java\bin>
```

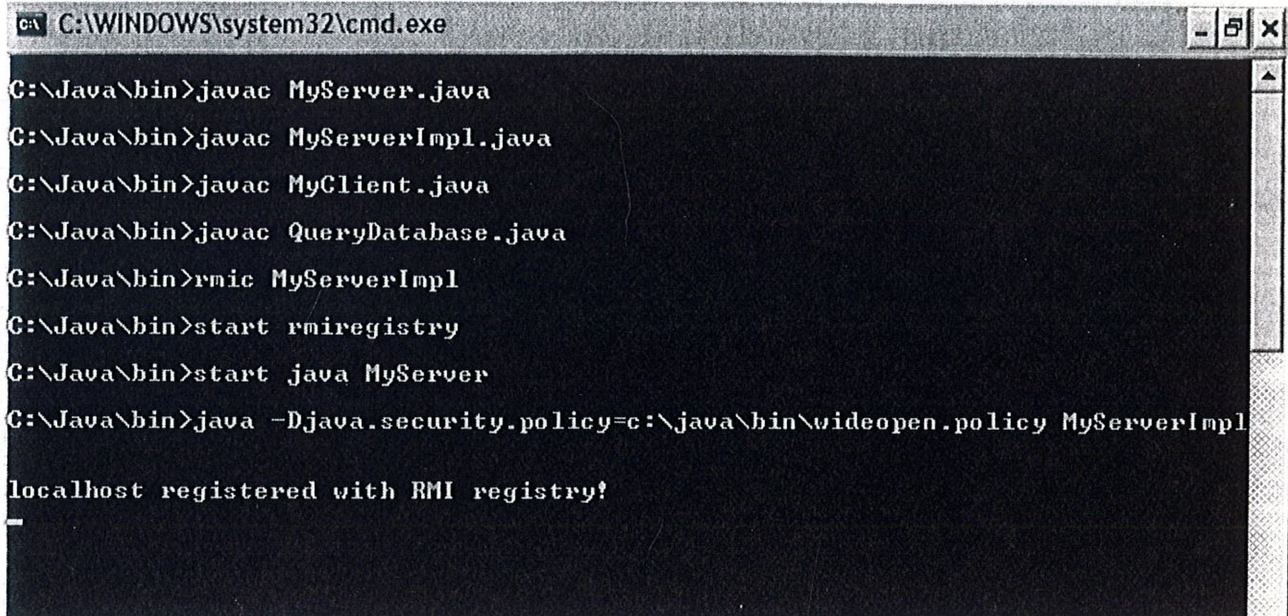
3. Registering using rmiregistry.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\welcome>cd\

C:\>cd c:\java\bin

C:\Java\bin>javac MyServer.java

C:\Java\bin>javac MyServerImpl.java

C:\Java\bin>javac QueryDatabase.java

C:\Java\bin>rmic MyServerImpl

C:\Java\bin>start rmiregistry

C:\Java\bin>
```

1. Start the Server

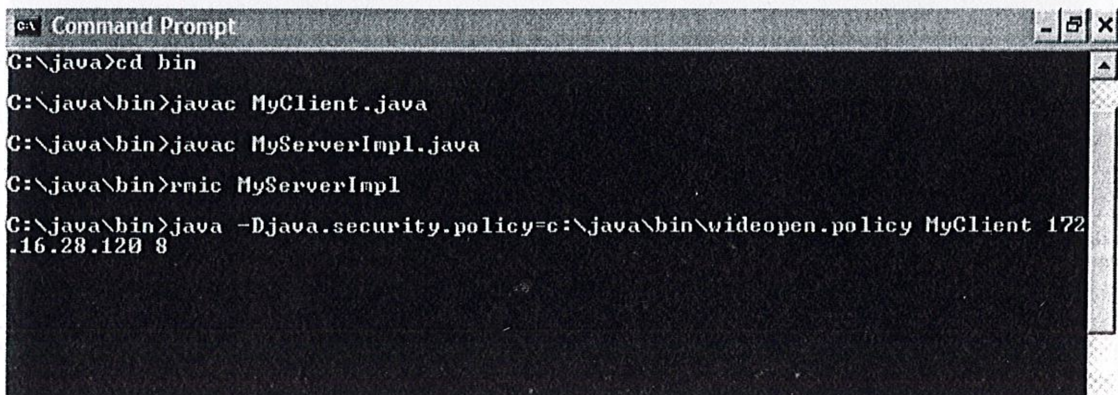
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\welcome>cd\

C:\>cd c:\java\bin

C:\Java\bin>javac MyServer.java

C:\Java\bin>javac MyServerImpl.java

C:\Java\bin>javac QueryDatabase.java

C:\Java\bin>rmic MyServerImpl

C:\Java\bin>start rmiregistry

C:\Java\bin>start java MyServer

C:\Java\bin>
```

2. Run the server passing policy file as argument.



RUNNING THE CLIENT SIDE

1. Compiling and running the client file.



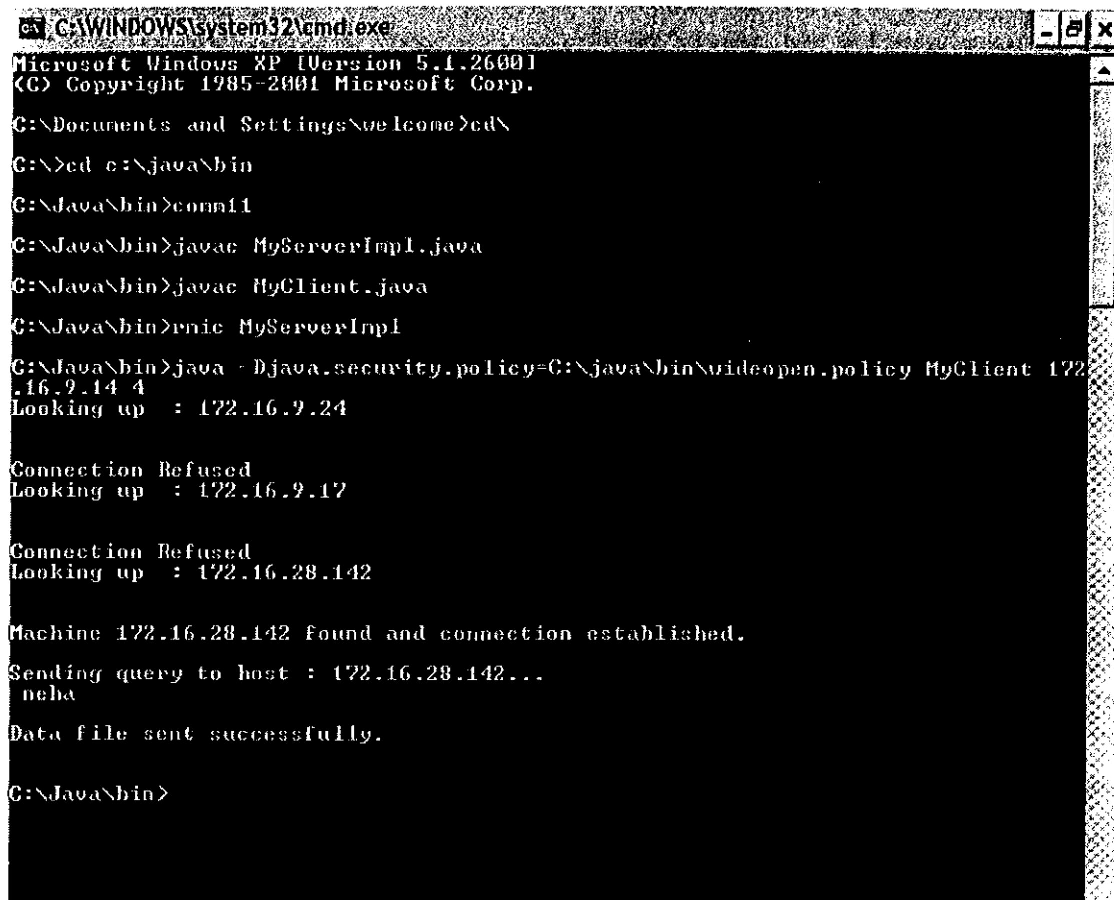
2. **Case I**: When no Server is running.

```
Command Prompt
C:\java\bin>java -Djava.security.policy=c:\java\bin\wideopen.policy MyClient 172
.16.28.120 8
Looking up  : 172.16.28.105

Connection Refused
Looking up  : 172.16.28.130

Connection Refused
Looking up  : 172.16.28.142

Connection Refused
Looking up  : 172.16.28.140

Connection Refused

C:\java\bin>_
```

2. **Case II**: Data not found on any server.

```
C:\java\bin>java -Djava.security.policy=c:\java\bin\wideopen.policy MyClient 172
.16.28.120 18
Looking up  : 172.16.28.105

Connection Refused
Looking up  : 172.16.28.130

Connection Refused
Looking up  : 172.16.28.142

Machine 172.16.28.142 found and connection established.

Sending query to host : 172.16.28.142...
  NotFound

Data file sent successfully.

C:\java\bin>_
```

3. **Case III**: Server running, data found and output shown successfully.



CHAPTER 5

USER INTERFACE

Our interface first takes the login and password of the user. It then matches the password entered by the user with the password already stored in our database. Once the user logs in, it may receive some message if the administrator or the authority at the library has left him some personal message. The administrator can block the account of the user or may leave any message for the user. If his account has been blocked he can proceed no further. The user can check the current status of his account ie how many books has he already issued and their return date. While searching for a book in the distributed library he may spell the book wrong. The Soundex Algorithm help in correcting such mistakes and searching for the desired book. Previous three searches of the user are also stored for future reference of the user. When the user searches for a book the name of the book, author, its location, number of copies available are displayed to the user. The user has the option to add any of the searched book to his favourite list. This will store the details of the book in his account so that in future he does not have to search for the book again. All the particulars of his favourite book are instantly available. Our system takes care that the same user does not get logged in from two different computers since it is a distributed system. Therefore, if a user is already logged in on any of the computers in the network, he won't be allowed to log in from any other computer. It is necessary for the user to logout his account before leaving.

The administrator can also login in our system. He has all the right to add, remove and update the details of a book. He can access the account of any of the users, send message to any of them and logout any of the user if he wants. Provision has been made to add the book distributedly. That is if the database of the computer he is working on is full, the book is added to any of the free database on the computers in the network.

MODULES

When the user logs in:

```
swingFrame = new JFrame("LRC");

mainPanel = new JPanel();
container.setBackground(Color.black);
newacct = new Panel();
 mainPanel.setOpaque(false);
 swingFrame.setSize(600,600);
 tabbedPane = new JTabbedPane();


loginPanel = new Panel();


 tabbedPane.addTab("Login",null,mainPanel,"Student info");
 tabbedPane.addTab("New Account",null,newacct,"Create a new Account");
label = new Label(" Login        ", Label.CENTER);
label.setBackground(Color.black);
label.setForeground(Color.white);
password = new Label(" Password",Label.CENTER);
password.setBackground(Color.black);
password.setForeground(Color.white);
paswd = new TextField(15);
paswd.setEchoChar('*');

rollno = new TextField(15);
bt = new Button("Submit");

loginPanel.add(label);
loginPanel.add(rollno);
loginPanel.add(h26);
loginPanel.add(password);
loginPanel.add(paswd);
 loginPanel.add(h27);
loginPanel.add(bt);

mainPanel.add(loginPanel);
mainPanel.setVisible(true);

}
```

When the user submits his login id and password:

```
private class ButtonHandler implements ActionListener
  {
    public void actionPerformed(ActionEvent evt)
     {
       try
        {
       Button source=(Button)evt.getSource();

       if(source == bt)
        {
          if(rollno.getText().equals( "" ))
           {
            JOptionPane.showMessageDialog(swingFrame,"Enter Username");
           }
          else
           {
            if(paswd.getText().equals( "" ))
             {
              JOptionPane.showMessageDialog(swingFrame,"Enter Pasword");
             }
            else
             {

              query = "SELECT name FROM login WHERE login = '" + rollno.getText()
+ "' AND pasword = '"+paswd.getText()+"'" ;
              abcd = searchhost(i,query,x);
              if(abc.equals("  Not Found"))
               {
                JOptionPane.showMessageDialog(swingFrame,"password and username
do not match");
               }
              else
               {
                loginPanel.setVisible(false);
               tabbedPane.setVisible(false);
               swingFrame.setVisible(false);
                if(Integer.parseInt(rollno.getText())==12000)
                 {
                  adminpage();
                 }
                else
                 {
                  query = "SELECT status FROM login WHERE login = '" +
rollno.getText() + "'";
```

```
                b = searchhost(i,query,x);
               if(b.equals("1"))
               JOptionPane.showMessageDialog(swingFrame,"Multiple login not
allowed");
               else
                {

               query = "SELECT notice FROM notice WHERE login = '" +
rollno.getText() + "'";
               b = searchhost(i,query,x);

               query = "SELECT disp FROM notice WHERE login = '" +
rollno.getText() + "'";
               e = searchhost(i,query,x);

               if(b.equals(""))
                {
                query = "UPDATE login SET status = '1' WHERE login = '" +
rollno.getText() + "'";
                h = searchhost(i,query,x);
                page2();
                }
               else
                {
                 if(e.equals("1"))
                {
                 query = "UPDATE login SET status = '1' WHERE login = '" +
rollno.getText() + "'";
                h = searchhost(i,query,x);
                 page2();

                 query = "SELECT counter FROM notice WHERE login = '" +
rollno.getText() + "'";
                ctr1 = searchhost(i,query,x);
                if(Integer.parseInt(ctr1)<3)
                 {
                JOptionPane.showMessageDialog(swingFrame,"MESSAGE FROM
ADMINISTRATOR" + b);
                int s = Integer.parseInt(ctr1) + 1;

        query = "UPDATE notice SET counter = '" + s + "' WHERE login = '" +
rollno.getText() + "'";
                ctr1 = searchhost(i,query,x);
                }}
                else
```

```
JOptionPane.showMessageDialog(swingFrame,"MESSAGE FROM
ADMINISTRATOR" + b);
                }}
               }
              }
             }
            }

          }
```

To check the current status of user's account

```
public void chkaccount()
{

 frame8 = new JFrame("Check account");
 frame8.setSize(400,200);

 sub = new Panel();
 rnumber = new TextField(30);
 enter = new Label("Enter login to check account",Label.CENTER);
 go = new Button("Submit");

 sub.add(enter);
 sub.add(rnumber);
 sub.add(go);
 sub.setVisible(true);

 ButtonHandler g = new ButtonHandler();
 go.addActionListener(g);


 frame8.getContentPane().add(sub);
 frame8.show();
 frame8.setVisible(true);


}
if(rnumber.getText().equals(""))
        {
         JOptionPane.showMessageDialog(swingFrame,"Enter login");
        }

        else
```

```
{
 query = "SELECT doi1 FROM acct WHERE login = " +
Integer.parseInt(rnumber.getText()) + " ";
 doi1 = searchhost(i,query,x);
 JOptionPane.showMessageDialog(swingFrame,doi1);


 query = "SELECT dor1 FROM acct WHERE login = " +
Integer.parseInt(rnumber.getText()) + " ";
 dor1 = searchhost(i,query,x);
 JOptionPane.showMessageDialog(swingFrame,dor1);


 query = "SELECT book2 FROM acct WHERE login = " +
Integer.parseInt(rnumber.getText()) + " ";
 book2 = searchhost(i,query,x);
 JOptionPane.showMessageDialog(swingFrame,book2);

 query = "SELECT doi2 FROM acct WHERE login = " +
Integer.parseInt(rnumber.getText()) + " ";
 doi2 = searchhost(i,query,x);
 JOptionPane.showMessageDialog(swingFrame,doi2);


 query = "SELECT dor2 FROM acct WHERE login = " +
Integer.parseInt(rnumber.getText()) + " ";
 dor2 = searchhost(i,query,x);
 JOptionPane.showMessageDialog(swingFrame,dor2);
 checkstatus(book1,doi1,dor1,book2,doi2,dor2);
 }
 }

 if(source == status )
 {

 JOptionPane.showMessageDialog(frame,"This is status button");

 query = "SELECT book1 FROM acct WHERE login = " +
Integer.parseInt(rollno.getText()) + " ";
 book1 = searchhost(i,query,x);
 JOptionPane.showMessageDialog(swingFrame,book1);
```

```
        query = "SELECT doi1 FROM acct WHERE login = " +
Integer.parseInt(rollno.getText()) + " ";
        doi1 = searchhost(i,query,x);
        JOptionPane.showMessageDialog(swingFrame,doi1);


        query = "SELECT dor1 FROM acct WHERE login = " +
Integer.parseInt(rollno.getText()) + " ";
        dor1 = searchhost(i,query,x);
        JOptionPane.showMessageDialog(swingFrame,dor1);


        query = "SELECT book2 FROM acct WHERE login = " +
Integer.parseInt(rollno.getText()) + " ";
        book2 = searchhost(i,query,x);
        JOptionPane.showMessageDialog(swingFrame,book2);


        query = "SELECT doi2 FROM acct WHERE login = " +
Integer.parseInt(rollno.getText()) + " ";
        doi2 = searchhost(i,query,x);
        JOptionPane.showMessageDialog(swingFrame,doi2);


        query = "SELECT dor2 FROM acct WHERE login = " +
Integer.parseInt(rollno.getText()) + " ";
        dor2 = searchhost(i,query,x);
        JOptionPane.showMessageDialog(swingFrame,dor2);
        checkstatus(book1,doi1,dor1,book2,doi2,dor2);
        }
```

When the user wants to search a book

```
       {
        JOptionPane.showMessageDialog(frame,"find button");
       if(list1.getSelectedIndex() == 0)
        {
        query = "SELECT bookname FROM book WHERE authorname = '" + soundx +
" '";

          bname = searchhost(i,query,x);

         if(bname.length()>=2)
            {
            JOptionPane.showMessageDialog(frame,"no array only result");
```

```
x=1;
r[0] = new String(bname);
}
else
{
 if(Integer.parseInt(bname) >0)
 {
  for(x = 0;x<Integer.parseInt(bname);x++)
  {
   query = "";
   r[x] = searchhost(i,query,x);
   JOptionPane.showMessageDialog(frame,r[x]);
  }
 }
}


 query = "SELECT location FROM book WHERE authorname = '" + soundx +
" '";
loc = searchhost(i,query,x);
if(loc.length()>2)
 {
  l[0] = new String(loc);
 }
 else
 {
if(Integer.parseInt(loc) >0)
 {
  for(x = 0;x<Integer.parseInt(bname);x++)
  {
   query = "";
   l[x] = searchhost(i,query,x);
   JOptionPane.showMessageDialog(frame,l[x]);
  }
 }
}
JOptionPane.showMessageDialog(frame,loc);


query = "SELECT referencecopy FROM book WHERE authorname = '" +
soundx + "'";
rcopy = searchhost(i,query,x);
JOptionPane.showMessageDialog(frame,"rcpy length:"+rcopy.length());
if(bname.length() >2)
 {
  rc[0] = new String(rcopy);
```

```
}
else
{

  for(x = 0;x<Integer.parseInt(bname);x++)
  {
   query = "";
   rc[x] = searchhost(i,query,x);
   JOptionPane.showMessageDialog(frame,rc[x]);
  }
 }

JOptionPane.showMessageDialog(frame,rcopy);


query = "SELECT issuecopy FROM book WHERE authorname = '" + soundx
+ " '";
icopy = searchhost(i,query,x);
 if(bname.length() >2)
 {
  x = 1;
 ic[0] = new String(icopy);

 }
else
{
if(Integer.parseInt(icopy) >0)
 {
  for(x = 0;x<Integer.parseInt(bname);x++)
  {
   query = "";
   ic[x] = searchhost(i,query,x);
   JOptionPane.showMessageDialog(frame,ic[x]);
  }
 }
}
JOptionPane.showMessageDialog(frame,icopy);


query = "SELECT isbn FROM book WHERE authorname = '" + soundx + " '";
 isbn = searchhost(i,query,x);
 if(bname.length() >2)
 {
  x = 1;
 isb[0] = new String(icopy);
```

```
}
else
{
if(Integer.parseInt(icopy) >0)
{
  for(x = 0;x<Integer.parseInt(bname);x++)
  {
   query = "";
   isb[x] = searchhost(i,query,x);
   JOptionPane.showMessageDialog(frame,ic[x]);
  }
 }
}
JOptionPane.showMessageDialog(frame,isbn);


display(soundx, x, r, l, rc, ic, isb);
}

else
 {
  if(list1.getSelectedIndex() == 1)
  {
   JOptionPane.showMessageDialog(frame,"ddddddddddddddddddddd");
   query = "SELECT authorname FROM book WHERE bookname = '" +
soundx + " '";
   aname = searchhost(i,query,x);

   if(aname.length()>2)
   {
    x=1;
    JOptionPane.showMessageDialog(frame,"This is the new part");
    r[0] = new String(aname);

   }
  else
   {
   if(Integer.parseInt(aname) >0)
    {
    for(x = 0;x<Integer.parseInt(aname);x++)
    {
    query = "";
    r[x] = searchhost(i,query,x);
    JOptionPane.showMessageDialog(frame,r[x]);
    }
   }
```

```
        }

        query = "SELECT location FROM book WHERE bookname = '" + soundx +
" '";
        loc = searchhost(i,query,x);
        if(loc.length()>2)
         {
         l1[0] = new String(loc);
         }
        else
         {
         if(Integer.parseInt(loc) >0)
          {
          for(x = 0;x<Integer.parseInt(loc);x++)
           {
           query = "";
           l1[x] = searchhost(i,query,x);
           JOptionPane.showMessageDialog(frame,l1[x]);
           }
          }
         }
       JOptionPane.showMessageDialog(frame,loc);

       query = "SELECT referencecopy FROM book WHERE bookname = '" +
soundx + " '";
       rcopy = searchhost(i,query,x);
        if(aname.length()>2)
        {
         rc1[0] = new String(rcopy);
        }
       else
        {
       if(Integer.parseInt(aname) >0)
         {
          for(x = 0;x<Integer.parseInt(aname);x++)
           {
            query = "";
            rc1[x] = searchhost(i,query,x);
            JOptionPane.showMessageDialog(frame,rc1[x]);
           }
         }
        }
      JOptionPane.showMessageDialog(frame,rcopy);
```

```
query = "SELECT issuecopy FROM book WHERE bookname = '" + soundx
+ "'";
icopy = searchhost(i,query,x);
 if(aname.length()>2)
 {
  ic1[0] = new String(icopy);
 }
 else
 {
 if(Integer.parseInt(aname) >0)
   {
    for(x = 0;x<Integer.parseInt(aname);x++)
     {
      query = "";
      ic1[x] = searchhost(i,query,x);
      JOptionPane.showMessageDialog(frame,ic1[x]);
     }
   }
 }
JOptionPane.showMessageDialog(frame,icopy);

query = "SELECT ISBN FROM book WHERE bookname = '" + soundx + "'";
 isbn = searchhost(i,query,x);
 if(aname.length()>2)
 {
  isb[0] = new String(isbn);
 }
 else
 {
 if(Integer.parseInt(aname) >0)
   {
    for(x = 0;x<Integer.parseInt(aname);x++)
     {
      query = "";
      isb[x] = searchhost(i,query,x);
      JOptionPane.showMessageDialog(frame,isb[x]);
     }
   }
 }
JOptionPane.showMessageDialog(frame,isbn);

display(soundx,x, r, l1, rc1, ic1, isb);
}
```

DATABASE

Table 1: Login

Attributes: login,password,name,status

Login: user id

Password:user password

Name: user name

Status: sets 1 when the user logs in and 0 when he logs out

Table 2: Account

Attributes: login,book1,book2,date of issue, date of return(for both of them)

Book: name of the book the user has issued

Table 3: Book

Attributes: bookname, authorname, location, reference copy, issue copy, book code, author code, ISBN

ISBN: ISBN number of the book

Location: floor and the rack the book is kept

Reference copy: Number of reference copies available

Issue copy: Number of issue copies available

Table 4: Notice

Attributes: login,notice,display,counter

Notice: the message the administrator wants to send

Display: Sets the display priority

Counter: The number of times the message is displayed to the user

Table 5: Previous search

Attributes: login,ISBN

Table6: Favourite

Attributes: login,ISBN,counter

DIRECTORIES AND STRUCTURES

We have used j2sdk (version 7.5) designed by Sun Microsystems. After installing this software a folder named "java" is created in our C drive. Inside "java" is the folder named "bin" where all the files being created is stored. All the files that we have created are stored with an extension " .java ".

While compiling and running such code the user first has to move to the directory java and then to bin. This can be done by providing classpath as shown below in the command prompt:

c:\cd java

c:\java\cd bin

c:\java\bin javac filename.java (for compiling)

c:\java\bin java filename.java (for running)

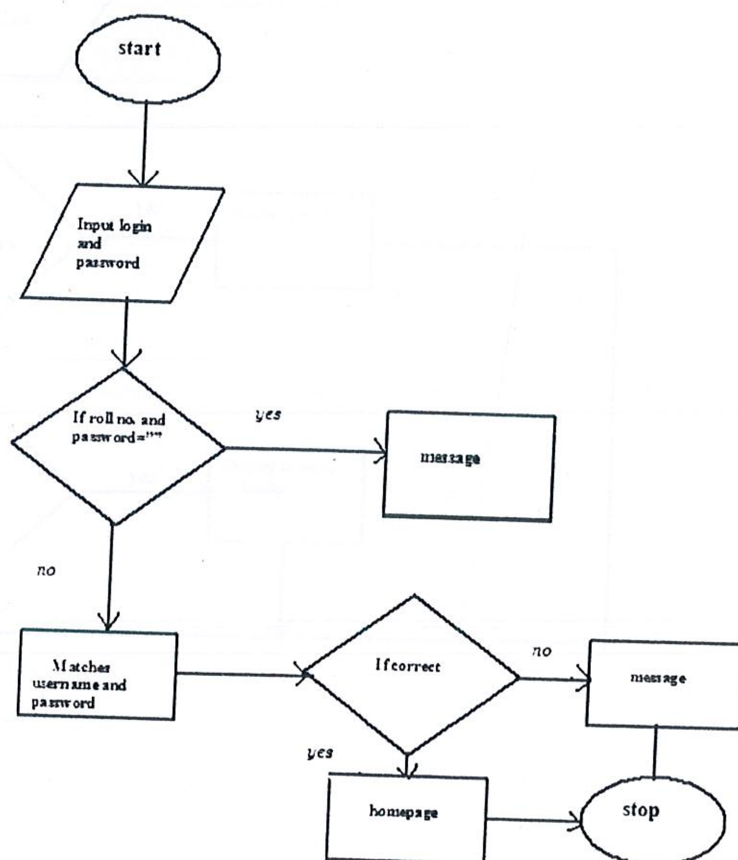
All the files should be located in the bin.

We have a "security.policy" file. This file is also located in the bin folder. This file is runned while running the client in order to get permission to access the remote computers properly.

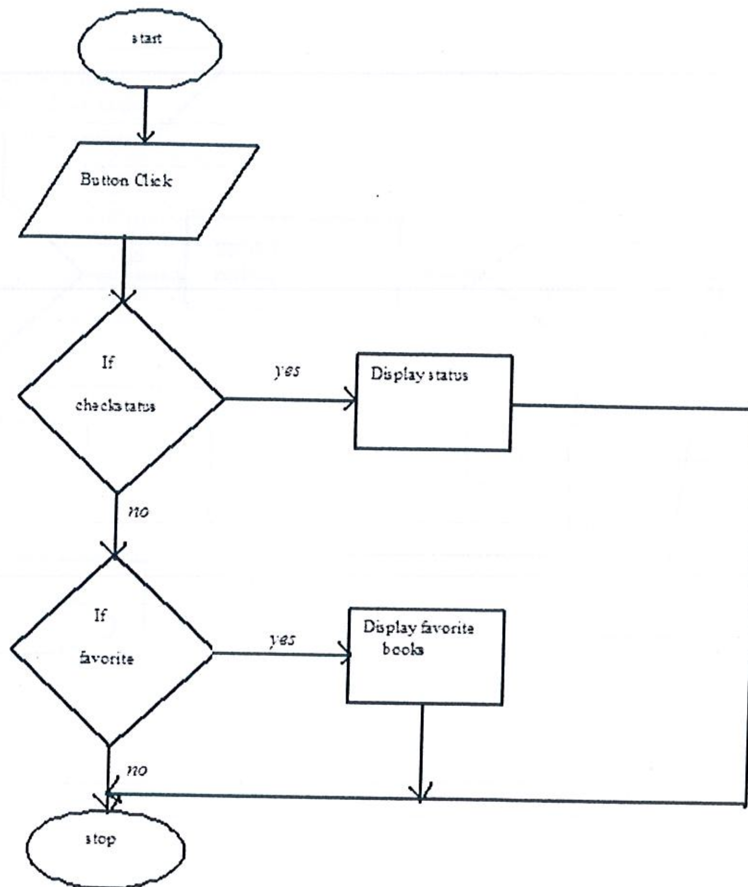
CHAPTER 6

FLOW CHARTS

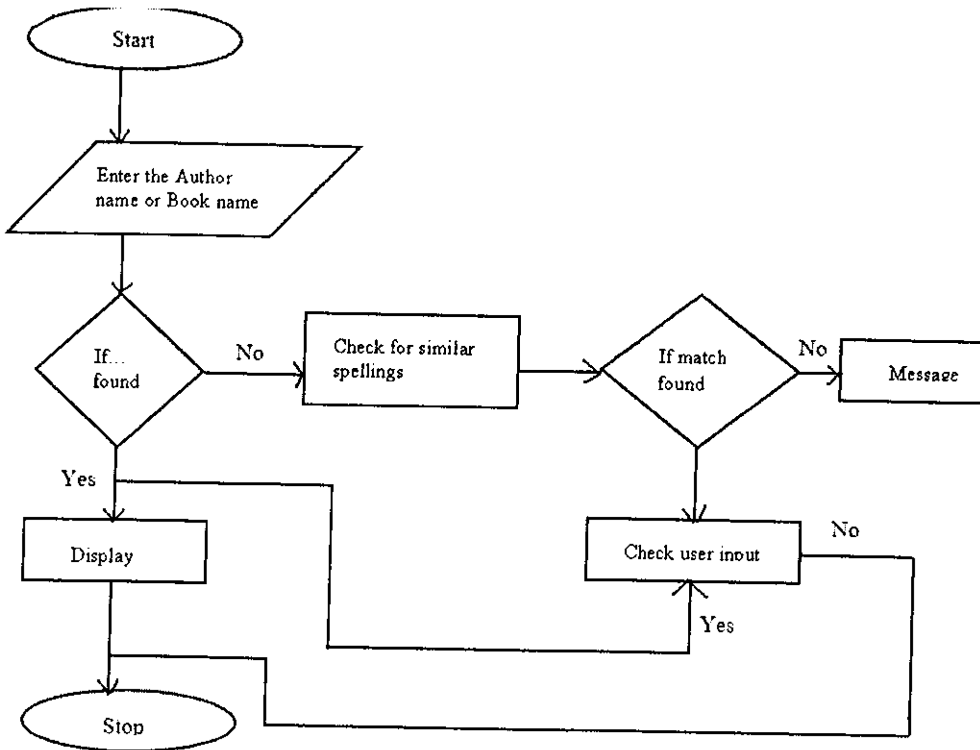
When the user or the administrator enters login and password



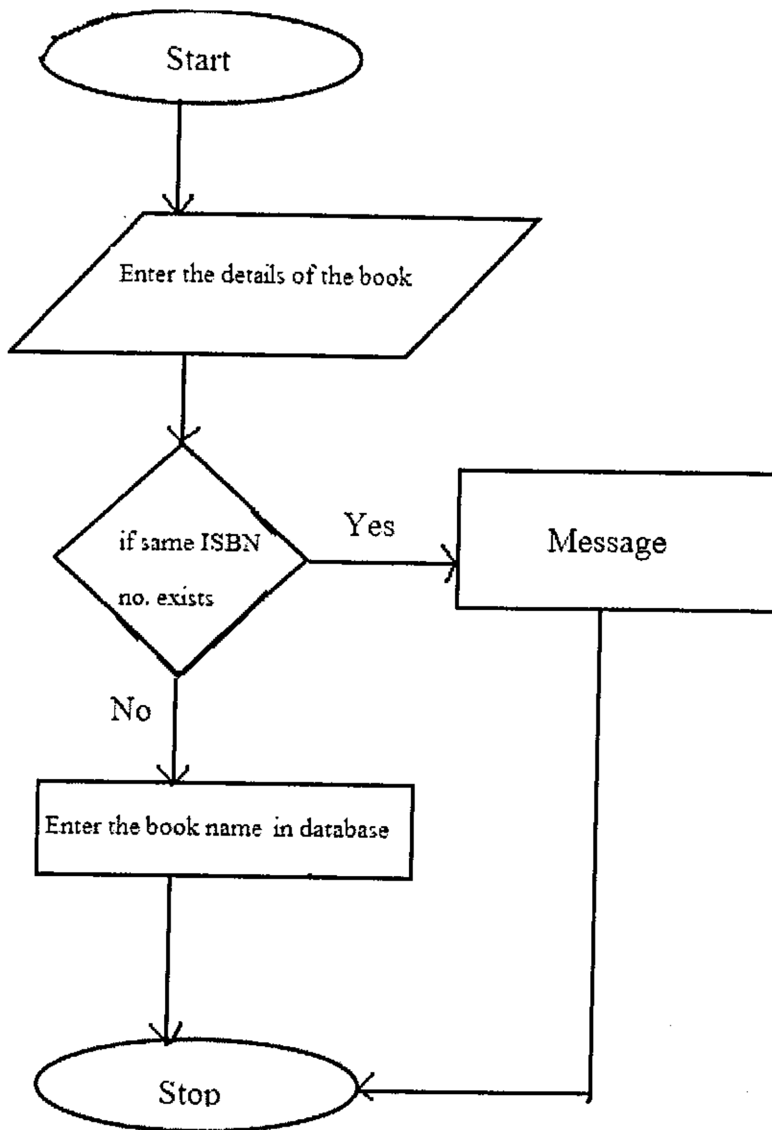
When the user enters his homepage



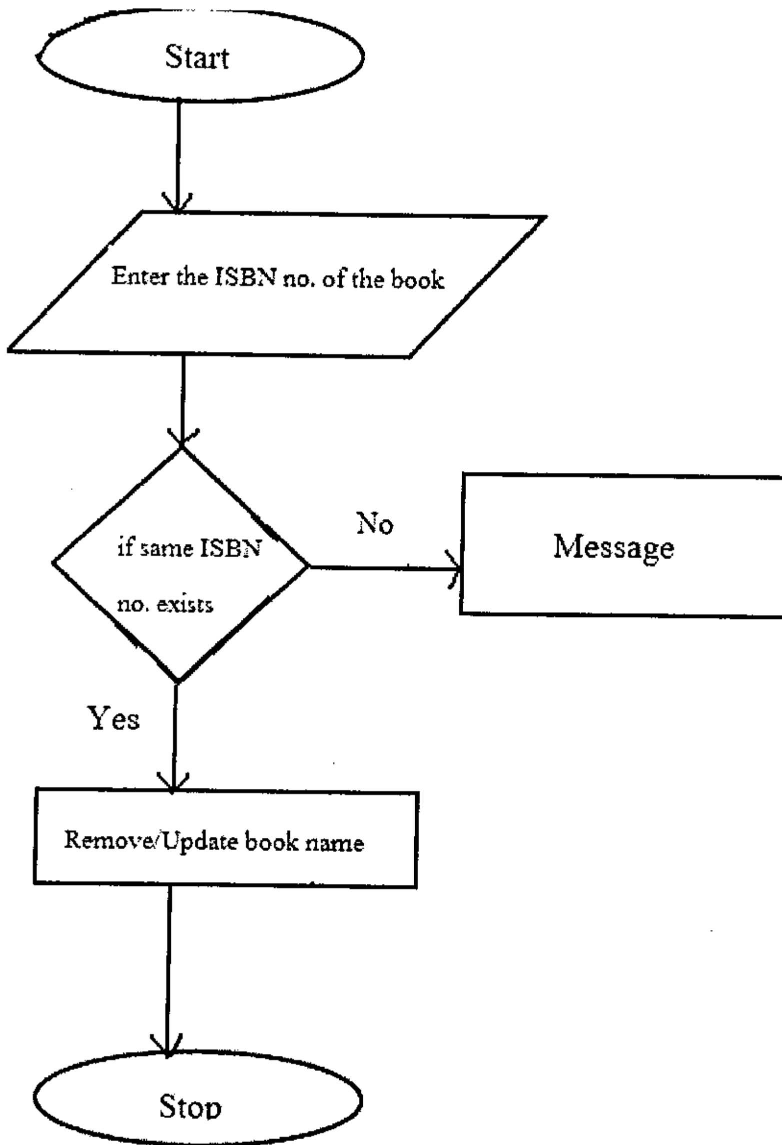
Searching for a book



Adding a book

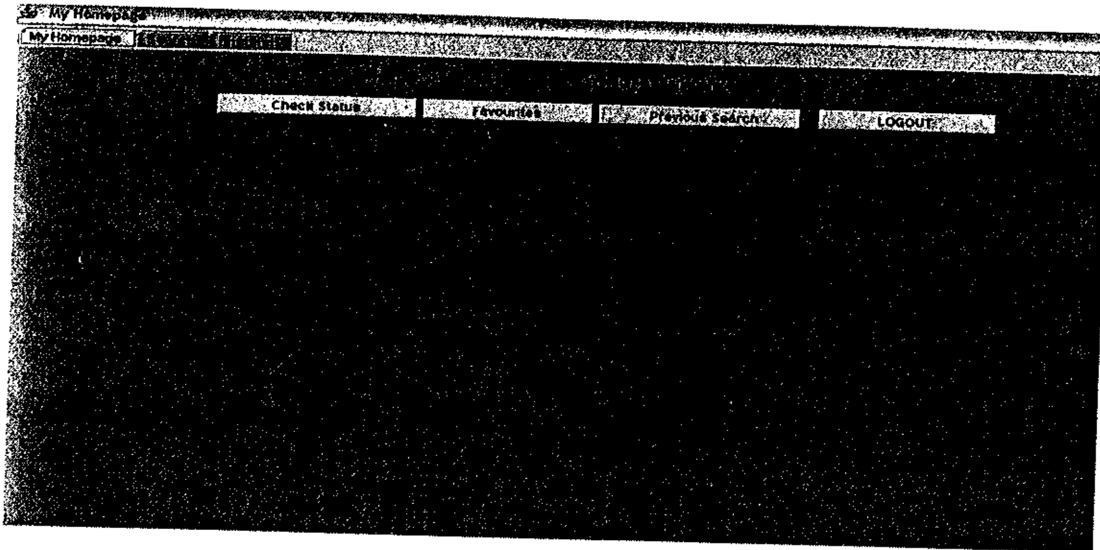


Removing or Updating a book

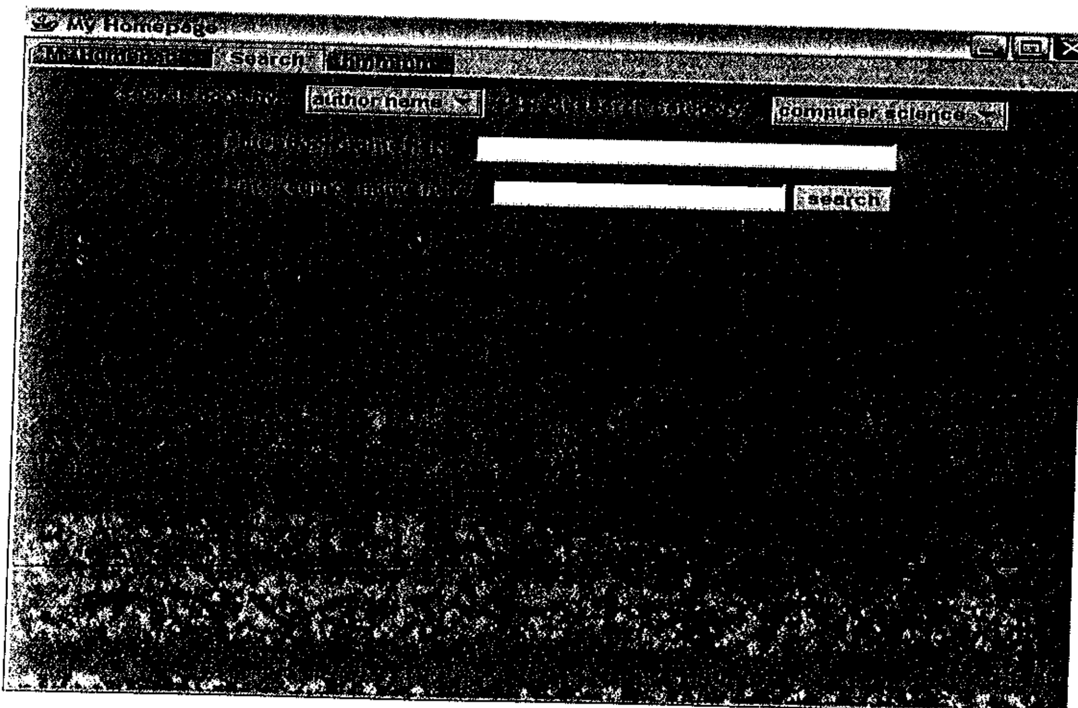


SNAPSHOTS OF OUR LIBRARY SYSTEM

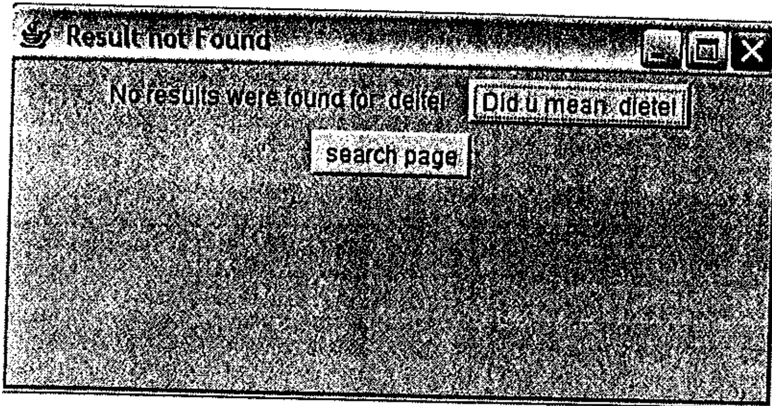
When the user logs in:



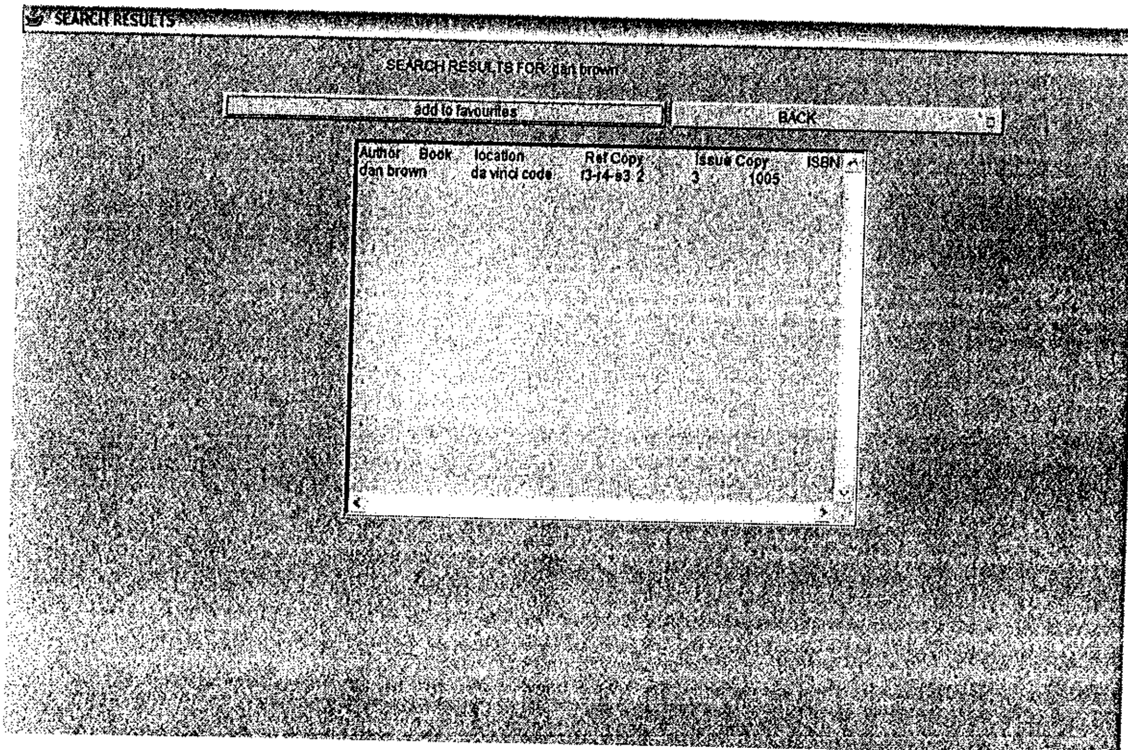
While searching:



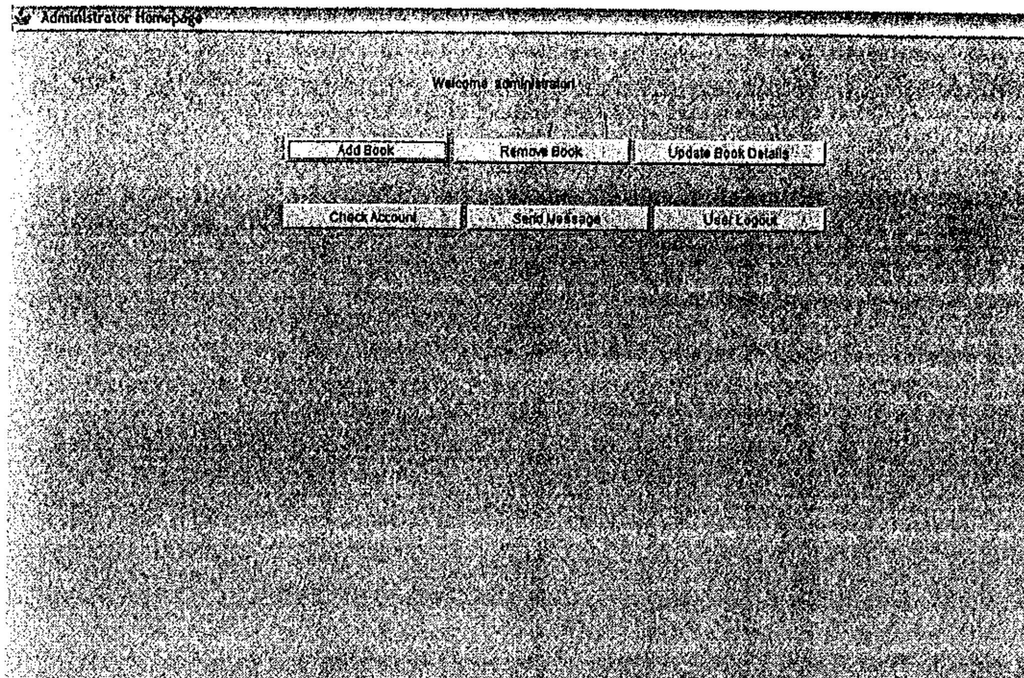
When the user incorrectly spells the bookname (Soundex Implemented)



Search Result



Administrator's Homepage



Adding a book

Add book

Enter book name

Enter author name

Enter location

Enter the number of Reference copies available

Enter the number of Issue copies available

Enter the category

Enter ISBN

ADD

If the book being added already exists

Add book

Enter book name: Fundamentals of Data Warehouse

Enter author name: navathe

Enter location: f1-r3-s5

Enter the number of Reference copies available: 6

Enter the number of Issue copies available: 4

Enter the category: cs

Enter ISBN: 1006

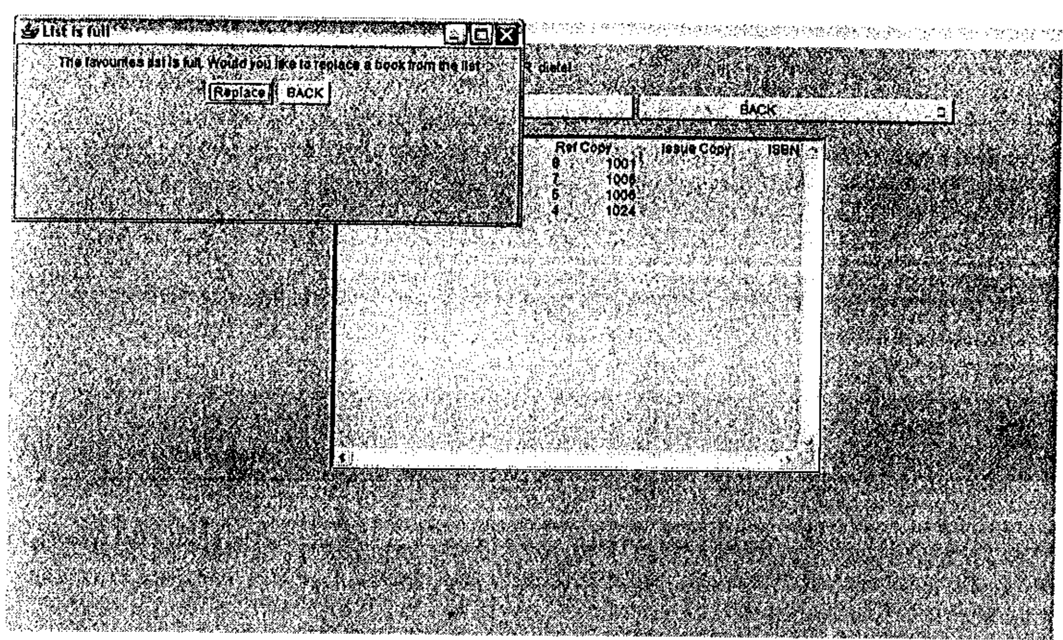
ADD

Message

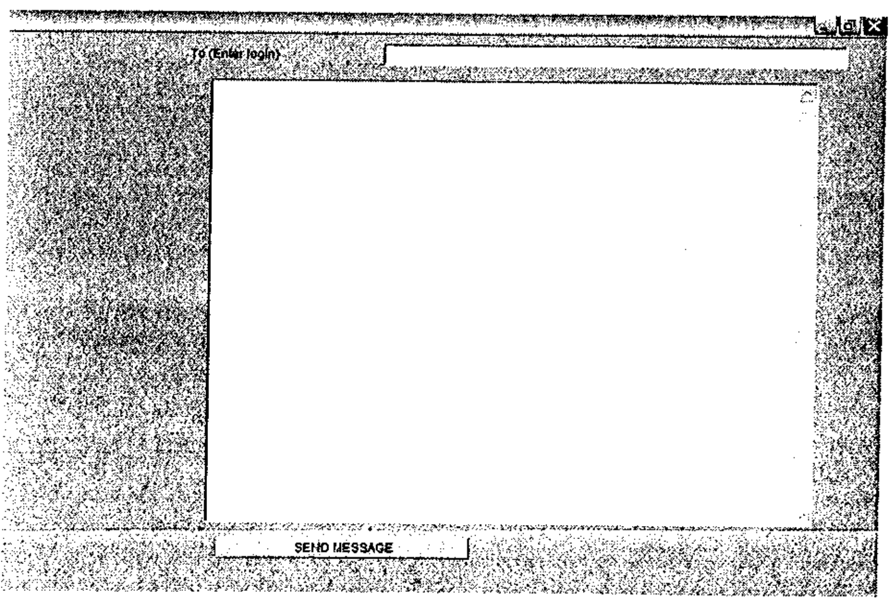
Book already exists in the database

OK

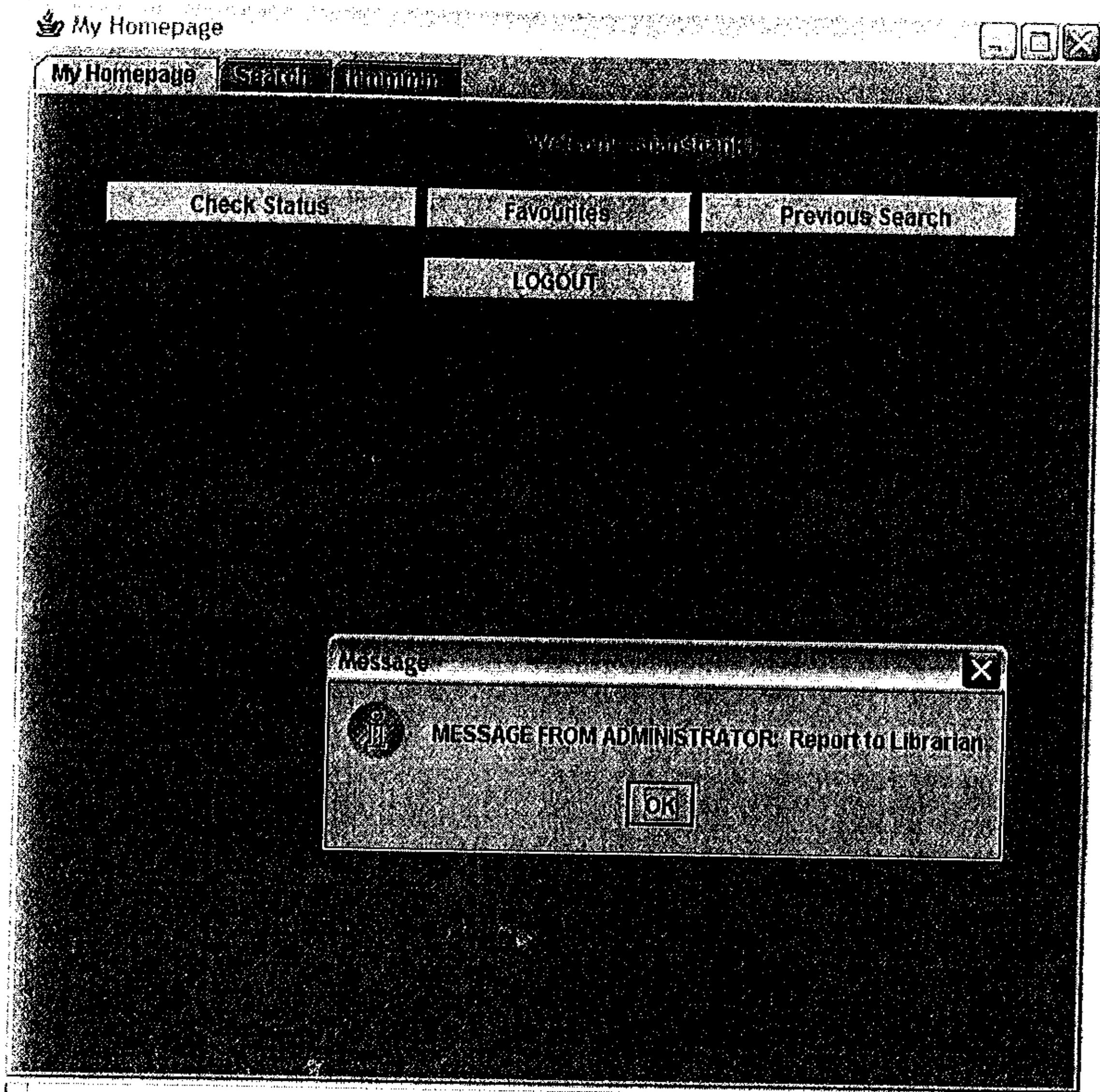
When the number of books in favorite list is full, the user can replace one of the books



The administrator can send message to the user

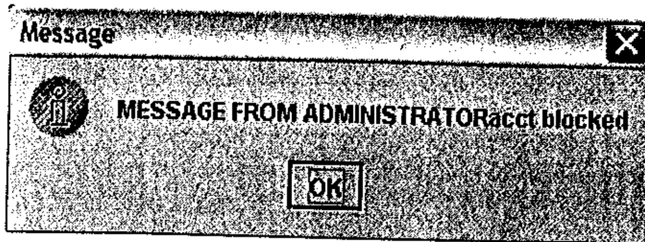


When the user logs in he gets the message as shown




```
Data file sent successfully.
```

The account of the user can also be blocked by the administrator giving the user the message that his account has been blocked



CONCLUSION

On basis of our study of the java environment we found out that RMI can be a robust and efficient platform for developing applications in distributed environment. Since RMI makes use of object based application can be developed and then integrated or can be developed as stand alone systems.

In our study of the various searching algorithms we found that the soundex algorithm is the best one in case the user is not sure about the complete and correct information about the book and the author.

BIBLIOGRAPHY

The information was obtained from the following websites and books

- www.devshed.com/c/b/MySQL
- www.arenives.gov/research_room/genealogy/census/soundex.html
- www.avotaynu.com/soundex.html
- A distributed library information system by Wolfram Schneider
- Computer Algorithms by Coreman
- A complete Reference to Java by Herbert Schield