Programming in Java Dr. M. T. Wanjari

**Unit – III**

**JDBC:**

Need to develop an application that communicates with a database perform the following task.

a) Store and update the data in the database

b) Retrieve the data stored in the database and present it to users in a proper format. **JDBC Architecture:**

Java application cannot directly communicate with a database to submit and retrieve the results of queries, because a database can interpret only SQL statement and not java language statements. Need a mechanism to translate java statements into SQL statements. The JDBC Architecture can be classified into two layers.

**1. JDBC Application Layer:** It signifies a java application that uses the JDBC API to interact with the JDBC drivers. A JDBC driver is software that a java application uses to access a database. The JDBC driver manager of JDBC API connects the java application to the driver **2. JDBC Driver Layer:** Acts as an interface between a java application and a database. This layer contains a driver, such as a SQL server driver or an oracle driver, with enable connectivity to a database. A driver sends the request of a Java application to the database. After processing the request, the database sends the response back to the driver. The driver translates and sends the response back to the response back to the driver. The driver translates and sends the response to the JDBC API. The JDBC API forwards it to the Java application.

JDBC Driver Layer JDBC Application Layer

Driver

Oracle

Java

Application

JDBC API

Driver

SQL Server

**Fig. 3.1 JDBC Architecture**

Driver

Sybase

M.Sc. Semester – I (Computer Science) 1

Programming in Java Dr. M. T. Wanjari

**JDBC Drivers:**

When you develop JDBC application, need to use JDBC drivers to convert queries into a form that a particular database can interpret. The JDBC Driver also retrieves the result of SQL statements and converts the result into equivalent JDBC API class objects that the Java application uses. As the JDBC driver only takes care of interactions with the database, any change made to the database does not affect the application.

JDBC supports four types of drivers

**1. JDBC-ODBC Bridge Driver**

The JDBC-ODBC Bridge driver is called the Type 1 driver. The JDBC-ODBC Bridge driver converts JDBC calls to open database connectivity (ODBC) calls. ODBC is an open standard APT to communicate with databases. The JDBC-ODBC bridge driver enables a Java application to use any database that supports ODBC driver. A Java application cannot interact directly with the ODBC driver. The application uses the JDBC-ODBC Bridge driver that works as an interface between the application and the ODBC driver. To use the JDBC-ODBC Bridge driver need to have the ODBC driver installed on the client computer. The JDBC-ODBC Bridge driver is usually used in standalone applications.

Java Application

Type 1 JDBC-ODBC Bridge

SQL Result

Statement

ODBC

Driver

Database

**Fig. 3.2 JDBC-ODBC Bridge Driver**

M.Sc. Semester – I (Computer Science) 2

Programming in Java Dr. M. T. Wanjari

**2.The Native-API Partly-Java Driver**

The Native-API partly-Java driver is called the Type 2 driver. It uses the local native libraries provided by the database vendors to access databases. The JDBC driver maps the JDBC calls to the native method calls, which are passed to the local native Call Level Interface (CLI). This interface consists of functions written in C to access databases. CLI needs to be loaded on the client computer. As opposed to the JDBC-ODBC Bridge driver the Native-API Partly-Java driver does not have an ODBC intermediate layer. As a result, this driver has a better performance than the JDBC-ODBC Bridge driver. This driver is usually used for network-based applications.

Java Application

Type 2 JDBC Driver

SQL Result

Statement

Native Database Library

Database

**Fig. 3.3 Native-API Partly-Java Driver**

**3. The JDBC-Net Pure-Java Driver**

The JDBC-Net Pure-Java driver is called the Type 3 driver. You can use the JDBC-Net Pure-Java driver over the web while connecting applets with databases. The JDBC-Net Pure

M.Sc. Semester – I (Computer Science) 3

Programming in Java Dr. M. T. Wanjari

Java driver consists of client and server portions. The client portion contains java functions and the server portion contains java and native methods. The Java application sends JDBC calls to the JDBC-Net Pure-Java driver client portion, which turn, translates JDBC calls into database calls. The database calls are sent to the server portion of the JDBC-Net Pure-Java driver that forwards the request to the database. When you use the JDBC-Net Pure-Java driver, CLI native libraries are loaded on the server.

Type 3 JDBC- Driver

(Client Portion)

Type 3 JDBC- Driver

(Client Portion)

Native Database Library

Database

**Fig. 3.4 JDBC-Net Pure-Java Driver**

**4. The Native-Protocol Pure-Java Driver**

The Native-Protocol Pure-Java driver is called the Type 4 driver. It is a Java driver interacts with the database directly using a vendor-specific network protocols. As opposed to the

M.Sc. Semester – I (Computer Science) 4

Programming in Java Dr. M. T. Wanjari

other JDBC drivers, you do not require to install an vendor specific libraries to use the Type 4 driver. DataDirect Technologies provide Type 4 driver various databases such as MS SQL server, AS/400, and DB2. This driver is usually used for enterprise applications.

Java Application

Type 4 JDBC Driver

Database

**Fig. 3.5 Native-Protocol Pure-Java Driver**

**Using JDBC API:**

Need to use database drivers and the JDBC API while developing a Java application to retrieve or store data in a database. The JDBC API classes and interfaces are available in the java,sql and the javax.sql packages. the classes and interfaces perform a number of tasks, such as establish and close a connection with a database, send a request to a database, retrieve data from a database and update data in a database.

The commonly used classes and interfaces in the JDBC API are:

**1. DriverManager class:** Loads the driver for a database.

**2. Driver interface:** Represents a database driver, all JDBC driver classes must implement the Driver interface.

**3. Connection interface:** Enables you to establish a connection between a Java application and a database.

**4. Statement interface:** Enables you to execute SQL statements.

M.Sc. Semester – I (Computer Science) 5

Programming in Java Dr. M. T. Wanjari

**5. ResultSet interface:** Represents the information retrieved from a database. **6. SQLException class:** Provides information about the exception that occur while interacting with databases. To query a database and display the result using Java applications need to **I. Loading a Driver:**

The first step to develop a JDBC application is to load and register the required driver using the driver manager. You can load and register a driver.

a. Programmatically

Using the forName ( ) method

Using the registerDriver ( ) method

b. Manually

By setting system property

**Using the forName ( ) Method:**

The forName ( ) method is available in the java.lang.class class. the forName ( ) method loads the JDBC driver and register the driver with the driver manager.

The syntax to load a JDBC driver to access a database is

Class.forName(“<driver\_name>”);

You can load the JDBC-ODBC Bridge Driver using the following method call Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);

**Using the registerDriver ( ) Method:**

You can create an instance of the Driver class to load a JDBC driver.

Syntax: Driver d=new<driver name>;

You can use the following statement to create an instance of the Driver class. Driver d=new sun.jdbc.odbc.JdbcOdbcDriver ( );

Once you have created the Driver object, call the registerDriver ( ) method to register it with the DriverManager. You can register the JDBC-ODBC Bridge driver using the following method call to registerDriver ( ) method:

DriverManager.registerDriver (d);

**Setting System Property:**

Driver can also be loaded by setting system property for JDBC drivers. Add the driver name to the jadbc.drivers system property to load a JDBC driver. You use the –D command line

M.Sc. Semester – I (Computer Science) 6

Programming in Java Dr. M. T. Wanjari

option to set the system property on the command line. The command to set the system property is java

-Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver SampleApplication

In the preceding command, jdbc.drivers is the property name and sun.jdbc.odbc.JdbcOdbcDriver is the value that you need to set the property. After you load a driver need to establish the connection with a database.

**II. Connecting to a Database:**

You can create an object of the Connection interface to establish a connection of the Java application with a database. You can create multiple connection object on a Java application to access & retrieves data from multiple databases. The DriverManager class provides the getConnection ( ) method to create a Connection object. The getConnection ( ) method is an overloaded method that has the following three forms.

**1. Connection getConnection (String <url>):**

You accept the JDBC URL of the database which you need to access as a parameter. You can use the following code to connect to a database using the getConnection ( ) method with a single parameter.

String url=”jdbc:odbc:MyData”;

Connection con=DriverManager.getConnection(url);

The syntax for a JDBC URL that is passed as a parameter to the getConnection ( ) method is

<protocol>:<subprotocol>:<subname>

A JDBC URL has the following three components.

**a. Protocol name:** Indicates the name of the protocol that is used to access a database. In JDBC, the name of the access protocol is always jdbc.

**b. Sub-protocol:** Indicates the mechanism to retrieve data from a database. For example if you use the JDBC-ODBC Bridge Driver to access a database, then the name of the sub-protocol is odbc.

**c. Subname:** Indicates the Data Source Name (DSN) that contains database information, such as the name of a database, location of the database server, user name, and password to access a database server.

**2. Connection getConnection(String <url>, String <username>, String <password>):**

M.Sc. Semester – I (Computer Science) 7

Programming in Java Dr. M. T. Wanjari

Accepts the JDBC url of a database. It also accepts the user name and password of the authorized database user. You can use the following code to specify the user name and password information to connect to a database.

String url=”jdbc:odbc:MyData”;

Connection con=DriverManager.getConnection(url. “NewUser”, “NewPassword”); **3. Connection getConnection(String<url>, Properties<properties>):**

You accept the JDBC URL of a database and an object of java.util.properties as parameters. You can specify information, such as User name and Password in the properties object by using the setProperty ( ) method.You can the following code to specify properties to connect to a database.

String url=”jdbc:odbc:MyData”;

Properties p=new properties ( );

p.setProperty(“user”, “NewUser”);

p.setProperty(“password”, “NewPassword”);

Connection con=DriverManager.getConnectio(url, p);

In the preceding code, p is the reference to an object of the properties class. The username and Password properties are set using the setProperty ( ) method. After you create a connection, you need to write JDBC statements that are to be executed. **III. Creating and Executing JDBC Statements:**

Need to create a statement object to send requests to and retrieve results from a database. The connection object provides the createStatement( ) method to create a statement object. Following code to create a statement object.

Connection con=DriverManager.getConnection(“jdbc:odbc:MyData”,“ NewUser”, “NewPassword”);

Statement stmt=con.cretaeStatement ( );

You can use static SQL statements to send request to a database. The SQL statements that do not contain runtime parameters are called static SQL statements. You can send SQL statements to a database using the statement object. the Statement interface contains the following methods to send static SQL statements to a database.

**1. ResultSet executeQuery(String str):**

M.Sc. Semester – I (Computer Science) 8

Programming in Java Dr. M. T. Wanjari

Executes an SQL statement and returns a single object of the type, ResultSet. This objects provides with the methods to access the data from a result set. The executeQuery ( ) method should be used when you need to retrieve data from a database table using the SELECT statement.

The syntax to use the executeQuery ( ) methods is

Statement stmt=con.createStatement ( );

ResultSet rs=stmt.executeQuery(<SQL Statement>);

In this syntax, stmt is a reference to the object of the statement interface. The executeQuery ( ) method executes an SQL statement and the result retrieved from a database is stored in rs, the ResultSet object.

**2. int executeUpdate (String url):**

Executes the SQL statements and returns the no. of data rows that are affected after processing the SQL statement. When you need to modify data in a database table using the Data Manipulation Language (DML) statements, INSERT, DELETE and UPDATE. You can use the executeUpdate ( ) method.

The syntax to use the executeUpadate( ) method is

Statement stmt=con.cretaeStatement ( );

Int count=stmt.executeUpdate(<SQL statement>);

In the syntax, the executeUpadate ( ) method executes an SQL statements and no. of rows affected in a database is stored in count, the int type variable.

**3. boolean execute (String str):**

Executes an SQL statement and returns a boolean value. you can use this method when the type of SQL statement passed as parameter is not known or when the statement being executed returns a result set or an update count. The execute ( ) method returns true if the result of the SQL statement is an object of ResultSet and false if it is an upadate count.

The syntax to use the following execute ( ) methods is

Statement stmt=con.cretaeStatement ( );

Stmt execute(<SQL Statement>);

You can use the DML statements, INSERT, UPDATE and DELETE. In Java application to modify the data stored in the database tables. You can also use the Data Definition Language

M.Sc. Semester – I (Computer Science) 9

Programming in Java Dr. M. T. Wanjari

(DDL) statements, CREATE, ALTER and DROP in Java application to define or change the structure of database objects.

**IV. Handling SQL Exception:**

The java.sql package provides the SQLException class, which derived from the java.lang.Exception class. The SQLException is thrown by various methods in the JDBC API and enables you to determine the reason of the errors that occurs while connecting a Java application to a database. You can catch the SQLException in a Java application using the try and catch exception handling block. The SQLException class provides the following error information.

**a. Error message:** Is a String that describes error

**b. Error code:** Is an integer value that is associated with error. The error code vendor specific and depends upon database in use.

**c. SQL state:** Is an X/OPEN error code that identifies the error.

Various vendors provide different error messages to define same error. As a result, an error give different error messages. The X/OPEN error code is an standard message associated with an error that can identify the error across multiple databases. The SQLException class contains various methods that provide error information. The methods in the SQLException class are.

**a. int getErrorCode ( ):** Returns the error code associated with the error occurred. **b. String getSQLState ():** Returns X/Open error code.

**c. SQLException getNextException ( ):** Returns the next exception in the chain of exception. You can use the following code snippet to catch SQLException.

try

**{**

String str=”DELETE FROM authors WHERE au\_id=’101’ ”;

Statement stmt=con.createStatement ( );

int count=stmt.executeUpdate(str);

}

catch(SQLException sqlExceptionObject)

{

System.out.println(“Display Error Code”);

M.Sc. Semester – I (Computer Science) 10

Programming in Java Dr. M. T. Wanjari

System.out.println(“SQLException ” + sqlExceptionObject.getErrorCode ( ) ); }

In the preceding code, if the DELETE statement at runtime throws an SQLException then it is handled using the try catch block. The sqlException is an object of the SQLException class and is used to invike the getErroCode ( ) method.

**Accessing Result Sets:**

When you execute a query to retrieve data from a table using a Java application, the output of the query is stored in a ResultSet object in a tabular format. A ResultSet object maintains a cursor that enables you to move through the rows stored in a ResultSet object. By default, the ResultSet object maintains a cursor that moves in the forward direction only. As a result, it moves from the first row to the last row in the ResultSet. You cannot update the default ResultSet object. the cursor in the ResultSet object initially point before the first row.

**Types of Result Sets:**

You can create various types of ResultSet objects to store the output returned by a database after executing SQL statements. Various types of ResultSet objects are: Read only: Allows you to only read the rows in a ResultSet object.

Forward only: Allows you to move the result set cursor from first row to last row in forward direction only.

Scrollable: Allows you to move the result set cursor forward or backward. Updatable: Allows you to update the result set rows retrieved from a database table. You can specify the type of a ResultSet object using the createStatement ( ) method of the Connection interface. The createStatement ( ) accepts ResultSet fields as parameters to create different types of the ResultSet object.

**Creating a JDBC Application to Query a Database:**

You can use the following code to create the application.

import java.sql.\*;

public class AuthorsInfo

{

public static void main(String args[])

{

try

{

String str="SELECT \* FROM authors WHERE City LIKE 'N%'";

M.Sc. Semester – I (Computer Science) 11

Programming in Java Dr. M. T. Wanjari

/\* Initialize and load the JDBC-ODBC Bridge Driver \*/

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

/\* Establish a connection with a data source \*/

Connection con = DriverManager.getConnection("jdbc:odbc:Data","",""); /\* Create a statement object to process the SELECT statement \*/

Statement stmt=con.createStatement();

/\* Execute the SELECT SQL statement \*/

ResultSet rs=stmt.executeQuery(str);

System.out.println("Author ID \t First Name\t Last Name\tCity");

/\* Display the result \*/

while(rs.next())

{

String id=rs.getString("Author ID");

String lname=rs.getString("Last Name");

String fname=rs.getString("First Name");

String city=rs.getString("City");

System.out.print(id+"\t");

/\* Use tab to forward the output. If the number equal to 7, the two tabs sre used to specify the position to display the next column in the ResultSet \*/

if(fname.length()<=7)

System.out.print(fname+"\t\t");

else

System.out.print(fname+"\t");

if(lname.length()<=7)

System.out.print(lname+"\t\t");

else

System.out.print(lname+"\t");

System.out.println(city);

}

con.close();

}

catch(Exception ex)

{

System.out.println("Error occured");

System.out.println("Error: "+ex);

}

}

}

M.Sc. Semester – I (Computer Science) 12

Programming in Java Dr. M. T. Wanjari

**Networking**

Java is practically a known for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross-platform, portable code. However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the **java.net** package. They provide an easy-to-use means by which programmers of all skill levels can access network resources.

The **java.net** package, it is important to emphasize that networking is a very large and at times complicated topic. Package focuses on several of its core classes and interfaces. **Socket**

Some key networking concepts and terms, at the core of Java’s networking support is the concept of a *socket*. A socket identifies an endpoint in a network.

A *network socket* is a like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can “plug in” to the socket and communicate. With electrical sockets, it doesn’t matter if plug in a lamp or a toaster; as long as they are expecting 60Hz, 115-volt electricity, the devices will work. Think how electric bill is created. There is a meter somewhere between house and the rest of the network. For each kilowatt of power that goes through that meter, are billed. The bill comes to “address.” So even though the electricity flows freely around the power grid, all of the sockets in house have a particular address. The same idea applies to network sockets, except we talk about TCP/IP packets and IP addresses rather than electrons and street addresses.

Socket communication takes place via a protocol. *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.

*Transmission Control Protocol* (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data.

A third protocol, *User Datagram Protocol (UDP),* sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

**Client/Server**

The term *client/server* mentioned in the context of networking. It seems complicated when read about it in corporate marketing statements, but it is actually quite simple. A *server* is

M.Sc. Semester – I (Computer Science) 13

Programming in Java Dr. M. T. Wanjari

anything that has some resource that can be shared. There are *compute servers,* which provide computing power; *print servers,* which manage a collection of printers; *disk servers,* which provide networked disk space; and *web servers,* which store web pages. A *client* is simply any other entity that wants to gain access to a particular server. The interaction between client and server is just like the interaction between a lamp and an electrical socket. The power grid of the house is the server, and the lamp is a power client. The server is a permanently available resource, while the client is free to “unplug” after it is has been served.

In Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This feat is managed by the introduction of a *port,* which is a numbered socket on a particular machine. A server process is said to “listen” to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

**Reserved Socket**

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will familiar, if it spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for whois; 79 is for finger; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. It works, when a client requests a file from an HTTP server, an action known as a *hit,* it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not the request can be fulfilled.

Here’s an example of a client requesting a single file, **/index.html**, and the server replying that it has successfully found the file and is sending it to the client:

|  |  |
| --- | --- |
| Server | Client |
| Listens to port 80. | Connects to port 80. |

M.Sc. Semester – I (Computer Science) 14

Programming in Java Dr. M. T. Wanjari

|  |  |
| --- | --- |
| Accepts the connection. | Writes “GET /index.html  HTTP/1.0\n\n”. |
| Reads up until the second end-of-line (\n). Sees that GET is a known command and that HTTP/1.0 is a valid protocol version. Reads a local file called /index.html. |  |
| Writes “HTTP/1.0 200 OK\n\n”. | “200” means “here comes the file.” |
| Copies the contents of the file into the socket. | Reads the contents of the file and  displays it. |
| Hangs up. Hangs up. | Hangs up. Hangs up. |

Obviously, the HTTP protocol is much more complicated than this example shows, but this is an actual transaction that could have with any web server.

**Proxy Servers**

A *proxy server* speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network’s connection to the Internet. When a popular web site is being hit by hundreds of users, a proxy server can get the contents of the web server’s popular pages once, saving expensive internetwork transfers while providing faster access to those pages to the clients. We will actually build a complete caching proxy HTTP server. The interesting part about this sample program is that it is both a client and a server. To serve certain pages, it must act as a client to other servers to obtain a copy of the requested content.

**Internet Addressing**

Every computer on the Internet has an *address.* An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32- bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately,

M.Sc. Semester – I (Computer Science) 15

Programming in Java Dr. M. T. Wanjari

IPv6 is downwardly compatible with IPv4. Currently, IPv4 is by far the most widely used scheme, but this situation is likely to change over time. Because of the emerging importance of IPv6, Java 2, version 1.4 has begun to add support for it. However, at the time of this writing, IPv6 is not supported by all environments. Furthermore, for the next few years, IPv4 will continue to be the dominant form of addressing. For these reasons, the form of Internet addresses here, and used the IPv4 form. As mentioned, IPv4 is, loosely, a subset of IPv6, and the largely applicable to both forms of addressing.

There are 32 bits in an IPv4 IP address, and we often refer to them as a sequence of four numbers between 0 and 255 separated by dots (.). This makes them easier to remember, because they are not randomly assigned—they are hierarchically assigned. The first few bits define which class of network, lettered A, B, C, D, or E, the address represents. Most Internet users are on a class C network, since there are over two million networks in class C. The first byte of a class C network is between 192 and 224, with the last byte actually identifying an individual computer among the 256 allowed on a single class C network. This scheme allows for half a billion devices to live on class C networks.

**Domain Naming Service (DNS)**

The Internet wouldn’t be a very friendly place to navigate if everyone had to refer to their addresses as numbers. For example, it is difficult to imagine seeing “http://192.9.9.1/” at the bottom of an advertisement. Thankfully, a clearinghouse exists for a parallel hierarchy of names to go with all these numbers. It is called the *Domain Naming Service (DNS).* Just as the four numbers of an IP address describe a network hierarchy from left to right, the name of an Internet address, called its *domain name,* describes a machine’s location in a name space, from right to left. For example, **www.osborne.com** is in the COM domain (reserved for U.S. commercial sites), it is called osborne (after the company name), and www is the name of the specific computer that is Osborne’s web server. www corresponds to the rightmost number in the equivalent IP address.

**The Networking Classes and Interfaces**

Java supports TCP/IP both by extending the established stream I/O interface and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP

M.Sc. Semester – I (Computer Science) 16

Programming in Java Dr. M. T. Wanjari

supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the **java.net** package are shown here:

|  |  |  |
| --- | --- | --- |
| Authenticator | Inet6Address | SocketImpl |
| ContentHandler | JarURLConnection | SocketPermission |
| DatagramPacket | MulticastSocket | URI |
| DatagramPacket | NetPermission | URL |
| DatagramSocketImpl | NetworkInterface | URLClassLoader |
| HttpURLConnection | PasswordAuthentication | URLConnection |
| InetAddress | ServerSocket | URLDecoder |
| Inet4Address | Socket | URLEncoder |
| Inet6Address | SocketAddress | URLStreamHandler |

Several new classes were added by Java 2, version 1.4. Some of these are to support the new IPv6 addressing scheme. Others provide some added flexibility to the original **java.net** package. Java 2, version 1.4 also added functionality, such as support for the new I/O classes, to several of the preexisting networking classes. Most of the additions made by Java 2, version 1.4, but three new classes, **Inet4Address**, **Inet6Address**, and **URI**. The **java.net** package’s interfaces are listed here:

|  |  |  |
| --- | --- | --- |
| ContentHandlerFactory | SocketImplFactory | URLStreamHandlerFactory |
| FileNameMap | SocketOptions | DatagramSocketImplFactory |

**InetAddress**

Whether making a phone call, sending mail, or establishing a connection across the Internet, addresses are fundamental. The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address, interact with class by using the name of an IP host, which is more convenient and understandable than its IP address. The **InetAddress** class hides the number inside. As of Java 2, version 1.4, **InetAddress** can handle both IPv4 and IPv6 addresses.

**Factory Methods**

The InetAddress class has no visible constructors. To create an InetAddress object, have to use one of the available factory methods. Factory methods are merely a convention whereby

M.Sc. Semester – I (Computer Science) 17

Programming in Java Dr. M. T. Wanjari

static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used InetAddress factory methods are shown here. static InetAddress getLocalHost( )

throws UnknownHostException

static InetAddress getByName(String hostName)

throws UnknownHostException

static InetAddress[ ] getAllByName(String hostName)

throws UnknownHostException

The getLocalHost( ) method simply returns the InetAddress object that represents the local host. The getByName( ) method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The getAllByName( ) factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException if it can’t resolve the name to at least one address.

Java 2, version 1.4 also includes the factory method getByAddress( ), which takes an IP address and returns an InetAddress object. Either an IPv4 or an IPv6 address can be used. The following example prints the addresses and names of the local machine and two well-known Internet web sites:

// Demonstrate InetAddress.

import java.net.\*;

class InetAddressTest

{

public static void main(String args[]) throws UnknownHostException {

InetAddress Address = InetAddress.getLocalHost();

System.out.println(Address);

Address = InetAddress.getByName("osborne.com");

System.out.println(Address);

InetAddress SW[] = InetAddress.getAllByName("www.nba.com");

M.Sc. Semester – I (Computer Science) 18

Programming in Java Dr. M. T. Wanjari

for (int i=0; i<SW.length; i++)

System.out.println(SW[i]);

} }

Here is the output produced by this program. (output will be slightly different.) default/206.148.209.138

osborne.com/198.45.24.162

www.nba.com/64.241.238.153

www.nba.com/64.241.238.142

**Instance Methods**

The InetAddress class also has several other methods, which can be used on the objects returned by the methods. Here are some of the most commonly used.

|  |  |
| --- | --- |
| boolean equals(Object other) | Returns true if this object has the same Internet address as other. |
| byte[ ] getAddress( ) | Returns a byte array that represents the object’s Internet address in network byte order. |
| String getHostAddress( ) | Returns a string that represents the host address associated with the InetAddress object. |
| String getHostName( ) | Returns a string that represents the host name associated with the InetAddress object. |
| boolean isMulticastAddress( ) | Returns true if this Internet address is a multicast address. Otherwise, it returns false. |
| String toString( ) | Returns a string that lists the host name and the IP address for convenience. |

Internet addresses are looked up in a series of hierarchically cached servers. That means local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn’t have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server, called InterNIC (internic.net). This process might take a long time, so it is wise to structure code so that cache IP address information locally rather than look it up repeatedly.

**TCP/IP Client Sockets**

M.Sc. Semester – I (Computer Science) 19

Programming in Java Dr. M. T. Wanjari

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to- point, stream-based connections between hosts on the Internet. A socket can be used to connect Java’s I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a “listener,” which waits for clients to connect before doing anything. The Socket class is designed to connect to server sockets and initiate protocol exchanges. The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

|  |  |
| --- | --- |
| Socket(String hostName, int port) | Creates a socket connecting the local host to the named host and port; can throw an UnknownHostException or an IOException. |
| Socket(InetAddress ipAddress, int port) | Creates a socket using a preexisting InetAddress object and a port; can throw an IOException. |

A socket can be examined at any time for the address and port information associated with it, by use of the following methods:

|  |  |
| --- | --- |
| InetAddress getInetAddress( ) | Returns the InetAddress associated with the Socket object. |
| int getPort( ) | Returns the remote port to which this Socket object is connected. |
| int getLocalPort( ) | Returns the local port to which this Socket object is connected. |

Once the Socket object has been created, it can also be examined to gain access to the input and output streams associated with it. Each of these methods can throw an IOException if the sockets have been invalidated by a loss of connection on the Net. These streams are used exactly like the I/O streams to send and receive data.

M.Sc. Semester – I (Computer Science) 20

Programming in Java Dr. M. T. Wanjari

|  |  |
| --- | --- |
| InputStream getInputStream( ) | Returns the InputStream associated  with the invoking socket. |
| OutputStream getOutputStream( ) | Returns the OutputStream associated  with the invoking socket. |

Java 2, version 1.4 added the getChannel( ) method to Socket. This method returns a channel connected to the Socket object. Channels are used by the new I/O classes contained in java.nio.

**Whois**

The very simple example that follows opens a connection to a “whois” port on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, then send back the IP address and contact information for that site.

//Demonstrate Sockets.

import java.net.\*;

import java.io.\*;

class Whois {

public static void main(String args[]) throws Exception {

int c;

Socket s = new Socket("internic.net", 43);

InputStream in = s.getInputStream();

OutputStream out = s.getOutputStream();

String str = (args.length == 0 ? "osborne.com" : args[0]) + "\n";

byte buf[] = str.getBytes();

out.write(buf);

while ((c = in.read()) != -1) {

System.out.print((char) c);

}

s.close();

}

}

If, for example, obtained information about osborne.com, get something similar to the following:

Whois Server Version 1.3

M.Sc. Semester – I (Computer Science) 21

Programming in Java Dr. M. T. Wanjari

Domain names in the .com, .net, and .org domains can now be registered with many different competing registrars. Go to http://www.internic.net for detailed information.Domain Name: OSBORNE.COM

Registrar: NETWORK SOLUTIONS, INC.

Whois Server: whois.networksolutions.com

Referral URL: http://www.networksolutions.com

Name Server: NS1.EPPG.COM

Name Server: NS2.EPPG.COM

Updated Date: 16-jan-2002

>> Last update of whois database: Thu, 25 Apr 2002 05:05:52 EDT <<

The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains and Registrars.

**URL**

That last example was rather obscure, because the modern Internet is not about the older protocols, like whois, finger, and FTP. It is about WWW, the World Wide Web. The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scalable way to locate all of the resources of the Net. Once can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. In fact, the Web is really just that same old Internet with all of its resources addressed as URLs plus HTML. Within Java’s network class library, the URL class provides a simple, concise API to access information across the Internet using URLs. **Format**

Two examples of URLs are http://www.osborne.com/ and http://www.osborne.com:80/ index.htm. A URL specification is based on four components. The **first** is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are http, ftp, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the “http://” from your URL specification). The **second** component is the host name or IP address of the host to use; this is delimited on the left by

M.Sc. Semester – I (Computer Science) 22

Programming in Java Dr. M. T. Wanjari

double slashes (//) and on the right by a slash (/) or optionally a colon (:). The **third** component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus “:80” is redundant.) The **fourth** part is the actual file path. Most HTTP servers will append a file named index.html or index.htm to URLs that refer directly to a directory resource. Thus,

http://www.osborne.com/ is the same as http://www.osborne.com/index.htm. Java’s URL class has several constructors, and each can throw a MalformedURLException. One commonly used form specifies the URL with a string that is identical to what displayed in a browser:

URL(String *urlSpecifier*)

The next two forms of the constructor allow to break up the URL into its component parts:

URL(String protocolName, String hostName, int port, String path)

URL(String protocolName, String hostName, String path)

Another frequently used constructor allows to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it’s really quite easy and useful.

URL(URL urlObj, String urlSpecifier)

In the following example, we create a URL to Osborne’s download page and then examine its properties:

// Demonstrate URL.

import java.net.\*;

class URLDemo {

public static void main(String args[]) throws MalformedURLException {

URL hp = new URL("http://www.osborne.com/downloads");

System.out.println("Protocol: " + hp.getProtocol());

System.out.println("Port: " + hp.getPort());

System.out.println("Host: " + hp.getHost());

System.out.println("File: " + hp.getFile());

System.out.println("Ext:" + hp.toExternalForm());

}

}

When you run this, you will get the following output:

Protocol: http

M.Sc. Semester – I (Computer Science) 23

Programming in Java Dr. M. T. Wanjari

Port: -1

Host: www.osborne.com

File: /downloads

Ext:http://www.osborne.com/downloads

Notice that the port is –1; this means that one was not explicitly set. Now that we have created a URL object, we want to retrieve the data associated with it. To access the actual bits or content information of a URL, create a URLConnection object from it, using its openConnection( ) method, like this:

url.openConnection()

openConnection( ) has the following general form:

URLConnection openConnection( )

It returns a URLConnection object associated with the invoking URL object. It may throw an IOException.

**URLConnection**

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once make a connection to a remote server, can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for URL objects that are using the HTTP protocol. We’ll examine the most useful elements of URLConnection here.

In the following example, we create a URLConnection using the openConnection( ) method of a URL object and then use it to examine the document’s properties and content:

// Demonstrate URLConnection.

import java.net.\*;

import java.io.\*;

import java.util.Date;

class UCDemo

{

public static void main(String args[]) throws Exception {

int c;

URL hp = new URL("http://www.internic.net");

URLConnection hpCon = hp.openConnection();

// get date

long d = hpCon.getDate();

if(d==0)

M.Sc. Semester – I (Computer Science) 24

Programming in Java Dr. M. T. Wanjari

System.out.println("No date information.");

else

System.out.println("Date: " + new Date(d));

// get content type

System.out.println("Content-Type: " + hpCon.getContentType());

// get expiration date

d = hpCon.getExpiration();

if(d==0)

System.out.println("No expiration information.");

else

System.out.println("Expires: " + new Date(d));

// get last-modified date

d = hpCon.getLastModified();

if(d==0)

System.out.println("No last-modified information.");

else

System.out.println("Last-Modified: " + new Date(d));

// get content length

int len = hpCon.getContentLength();

if(len == -1)

System.out.println("Content length unavailable.");

else

System.out.println("Content-Length: " + len);

if(len != 0) {

System.out.println("=== Content ===");

InputStream input = hpCon.getInputStream();

int i = len;

while (((c = input.read()) != -1)) { // && (--i > 0)) {

System.out.print((char) c);

}

input.close();

} else {

System.out.println("No content available.");

}

}

}

The program establishes an HTTP connection to www.internic.net over port 80. We then list out the header values and retrieve the content. Here are the first lines of the output (the precise output will vary over time).

Date: Sat Apr 27 12:17:32 CDT 2002

M.Sc. Semester – I (Computer Science) 25

Programming in Java Dr. M. T. Wanjari

Content-Type: text/html

No expiration information.

Last-Modified: Tue Mar 19 17:52:42 CST 2002

Content-Length: 5299

=== Content ===

<html>

<head>

<title>InterNIC | The Internet's Network Information Center</title>

<meta name="keywords" content="internic,network information, domain registration"> <style type="text/css">

<!--

p, li, td, ul { font-family: Arial, Helvetica, sans-serif}

-->

</style>

</head>

The URL and URLConnection classes are good enough for simple programs that want to connect to HTTP servers to fetch content. For more complex applications, probably find that are better off studying the specification of the HTTP protocol and implementing own wrappers. **TCP/IP Server Sockets**

As we mentioned earlier, Java has a different socket class that must be used for creating server applications. The ServerSocket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. Since the Web is driving most of the activity on the Internet, this section develops an operational web (http) server.

ServerSockets are quite different from normal Sockets. When create a ServerSocket, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that wish to accept connections on and, optionally, how long want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an IOException under adverse conditions. Here are the constructors:

|  |  |
| --- | --- |
| ServerSocket(int port) | Creates server socket on the specified port with |

M.Sc. Semester – I (Computer Science) 26

Programming in Java Dr. M. T. Wanjari

|  |  |
| --- | --- |
|  | a queue length of 50. |
| ServerSocket(int port, int maxQueue) | Creates a server socket on the specified port with a maximum queue length of maxQueue. |
| ServerSocket(int port, int maxQueue,  InetAddress localAddress) | Creates a server socket on the specified port with a maximum queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds. |

ServerSocket has a method called accept( ), which is a blocking call that will wait for a client to initiate communications, and then return with a normal Socket that is then used for communication with the client. Java 2, version 1.4 added the getChannel( ) method to ServerSocket. This method returns a channel connected to the ServerSocket object. Channels are used by the new I/O classes contained in java.nio.

**A Caching Proxy HTTP Server**

In the remainder of this section, we will develop a simple caching proxy HTTP server, called http, to demonstrate client and server sockets. http supports only GET operations and a very limited range of hard-coded MIME types. (MIME types are the type descriptors for multimedia content.) The proxy HTTP server is single threaded, in that each request is handled in turn while all others wait. It has fairly naive strategies for caching—it keeps everything in RAM forever. When it is acting as a proxy server, http also copies every file it gets to a local cache for which it has no strategy for refreshing or garbage collecting. All of these caveats aside, http represents a productive example of client and server sockets, and it is fun to explore and easy to extend.

**Source Code**

The implementation of this HTTP server is presented here in five classes and one interface. A more complete implementation would likely split many of the methods out of the main class, http, in order to abstract more of the components. For space considerations in this book, most of the functionality is in the single class, and the small support classes are only acting as data structures. We will take a close look at each class and method to examine how this server works, starting with the support classes and ending with the main program. **MimeHeader.java**

M.Sc. Semester – I (Computer Science) 27

Programming in Java Dr. M. T. Wanjari

MIME is an Internet standard for communicating multimedia content over e-mail systems. This standard was created by Nat Borenstein in 1992. The HTTP protocol uses and extends the notion of MIME headers to pass general attribute/value pairs between the HTTP client and server.

**Constructors**

This class is a subclass of Hashtable so that it can conveniently store and retrieve the key/value pairs associated with a MIME header. It has two constructors. One creates a blank MimeHeader with no keys. The other takes a string formatted as a MIME header and parses it for the initial contents of the object.

**parse( )** The parse( ) method is used to take a raw MIME-formatted string and enter its key/value pairs into a given instance of MimeHeader. It uses a StringTokenizer to split the input data into individual lines, marked by the CRLF (\r\n) sequence. It then iterates through each line using the canonical while ... hasMoreTokens( ) ...nextToken( ) sequence.

For each line of the MIME header, the parse( ) method splits the line into two strings separated by a colon (:). The two variables key and val are set by the substring( ) method to extract the characters before the colon, those after the colon, and its following space character. Once these two strings have been extracted, the put( ) method is used to store this association between the key and value in the Hashtable.

**toString( )**

The toString( ) method (used by the String concatenation operator, +) is simply the reverse of parse( ). It takes the current key/value pairs stored in the MimeHeader and returns a string representation of them in the MIME format, where keys are printed followed by a colon and a space, and then the value followed by a CRLF.

**put( ), get( ), AND fix( )**

The put( ) and get( ) methods in Hashtable would work fine for this application if not for one rather odd thing. The MIME specification defined several important keys, such as Content Type and Content-Length. Some early implementors of MIME systems, notably web browsers, took liberties with the capitalization of these fields. Some use Content-type, others content-type. To avoid mishaps, our HTTP server tries to convert all incoming and outgoing MimeHeader keys to be in the canonical form, Content-Type. Thus, we override put( ) and get( ) to convert the

M.Sc. Semester – I (Computer Science) 28

Programming in Java Dr. M. T. Wanjari

values’ capitalization, using the method fix( ), before entering them into the Hashtable and before looking up a given key.

Here is the source code for MimeHeader:

import java.util.\*;

class MimeHeader extends Hashtable {

void parse(String data) {

StringTokenizer st = new StringTokenizer(data, "\r\n");

while (st.hasMoreTokens()) {

String s = st.nextToken();

int colon = s.indexOf(':');

String key = s.substring(0, colon);

String val = s.substring(colon + 2); // skip ": "

put(key, val);

}

}

MimeHeader() {}

MimeHeader(String d) {

parse(d);

}

public String toString() {

String ret = "";

Enumeration e = keys();

while(e.hasMoreElements()) {

String key = (String) e.nextElement();

String val = (String) get(key);

ret += key + ": " + val + "\r\n";

}

return ret;

}

// This simple function converts a mime string from

// any variant of capitalization to a canonical form.

// For example: CONTENT-TYPE or content-type to Content-Type,

// or Content-length or CoNTeNT-LENgth to Content-Length.

private String fix(String ms) {

char chars[] = ms.toLowerCase().toCharArray();

boolean upcaseNext = true;

for (int i = 0; i < chars.length - 1; i++) {

char ch = chars[i];

if (upcaseNext && 'a' <= ch && ch <= 'z') {

chars[i] = (char) (ch - ('a' - 'A'));

M.Sc. Semester – I (Computer Science) 29

Programming in Java Dr. M. T. Wanjari

}

upcaseNext = ch == '-';

}

return new String(chars);

}

public String get(String key) {

return (String) super.get(fix(key));

}

public void put(String key, String val) {

super.put(fix(key), val);

}

}

**HttpResponse.java**

The HttpResponse class is a wrapper around everything associated with a reply from an HTTP server. This is used by the proxy part of our httpd class. When you send a request to an HTTP server, it responds with an integer status code, which we store in status Code, and a textual equivalent, which we store in reason Phrase. This single-line response is followed by a MIME header, which contains further information about the reply. We use the previously explained MimeHeader object to parse this string. The MimeHeader object is stored inside the HttpResponse class in the mh variable. These variables are not made private so that the http class can use them directly.

**CONSTRUCTORS**

If we construct an HttpResponse with a string argument, this is taken to be a raw response from an HTTP server and is passed to parse( ), described next, to initialize the object. Alternatively, can pass in a precomputed status code, reason phrase, and MIME header. **parse( )**

The parse( ) method takes the raw data that was read from the HTTP server, parses the statusCode and reasonPhrase from the first line, and then constructs a MimeHeader out of the remaining lines.

**toString( )**

The toString( ) method is the inverse of parse( ). It takes the current values of the HttpResponse object and returns a string that an HTTP client would expect to read back from a server.

Here is the source code for HttpResponse:

M.Sc. Semester – I (Computer Science) 30

Programming in Java Dr. M. T. Wanjari

import java.io.\*;

/\*

\* HttpResponse

\* Parse a return message and MIME header from a server.

\* HTTP/1.0 302 Found = redirection, check Location for where.

\* HTTP/1.0 200 OK = file data comes after mime header.

\*/

class HttpResponse

{

int statusCode; // Status-Code in spec

String reasonPhrase; // Reason-Phrase in spec

MimeHeader mh;

static String CRLF = "\r\n";

void parse(String request) {

int fsp = request.indexOf(' ');

int nsp = request.indexOf(' ', fsp+1);

int eol = request.indexOf('\n');

String protocol = request.substring(0, fsp);

statusCode = Integer.parseInt(request.substring(fsp+1, nsp));

reasonPhrase = request.substring(nsp+1, eol);

String raw\_mime\_header = request.substring(eol + 1);

mh = new MimeHeader(raw\_mime\_header);

}

HttpResponse(String request) {

parse(request);

}

HttpResponse(int code, String reason, MimeHeader m) {

statusCode = code;

reasonPhrase = reason;

mh = m;

}

public String toString() {

return "HTTP/1.0 " + statusCode + " " + reasonPhrase + CRLF +

mh + CRLF;

}

}

**UrlCacheEntry.java**

To cache the contents of a document on a server, we need to make an association between the URL that was used to retrieve the document and the description of the document itself. A document is described by its MimeHeader and the raw data. For example, an image

M.Sc. Semester – I (Computer Science) 31

Programming in Java Dr. M. T. Wanjari

might be described by a MimeHeader with Content-Type: image/gif, and the raw image data is just an array of bytes. Similarly, a web page will likely have a Content-Type: text/html key/value pair in its MimeHeader, while the raw data is the contents of the HTML page. Again, the instance variables are not marked as private so that httpd can have free access to them.

**CONSTRUCTOR**

The constructor for a UrlCacheEntry object requires the URL to use as the key and a MimeHeader to associate with it. If the MimeHeader has a field in it called Content-Length (most do), the data area is preallocated to be large enough to hold such content. **append( )**

The append( ) method is used to add data to a UrlCacheEntry object. The reason this isn’t simply a setData( ) method is that the data might be streaming in over a network and need to be stored a chunk at a time. The append( ) method deals with three cases. In the **first** case, the data buffer has not been allocated at all. In the **second**, the data buffer is too small to accommodate the incoming data, so it is reallocated. In the **last** case, the incoming data fits just fine and is inserted into the buffer. At any time, the length member variable holds the current valid size of the data buffer.

Here is the source code for UrlCacheEntry:

class UrlCacheEntry

{

String url;

MimeHeader mh;

byte data[];

int length = 0;

public UrlCacheEntry(String u, MimeHeader m) {

url = u;

mh = m;

String cl = mh.get("Content-Length");

if (cl != null) {

data = new byte[Integer.parseInt(cl)];

}

}

void append(byte d[], int n) {

if (data == null) {

data = new byte[n];

System.arraycopy(d, 0, data, 0, n);

M.Sc. Semester – I (Computer Science) 32

Programming in Java Dr. M. T. Wanjari

length = n;

} else if (length + n > data.length) {

byte old[] = data;

data = new byte[old.length + n];

System.arraycopy(old, 0, data, 0, old.length);

System.arraycopy(d, 0, data, old.length, n);

} else {

System.arraycopy(d, 0, data, length, n);

length += n;

}

}

}

**LogMessage.java**

LogMessage is a simple interface that declares one method, log( ), which takes a single String parameter. This is used to abstract the output of messages from the httpd. In the application case, this method is implemented to print to the standard output of the console in which the application was started. In the applet case, the data is appended to a windowed text buffer.

Here is the source code for LogMessage:

interface LogMessage {

public void log(String msg);

}

**httpd.java**

This is a really big class that does a lot. We will walk through it method by method. **CONSTRUCTOR**

There are five main instance variables: **port**, **docRoot**, **log**, **cache**, and **stopFlag**, and all of them are private. Three of these can be set by httpd’s lone constructor, shown here: httpd(int p, String dr, LogMessage lm)

It initializes the port to listen on, the directory to retrieve files from, and the interface to send messages to.

The fourth instance variable, cache, is the Hashtable where all of the files are cached in RAM, and is initialized when the object is created. stopFlag controls the execution of the program.

**STATIC SECTION**

M.Sc. Semester – I (Computer Science) 33

Programming in Java Dr. M. T. Wanjari

There are several important static variables in this class. The version reported in the “Server” field of the MIME header is found in the variable version. A few constants are defined next: the MIME type for HTML files, mime\_text\_html; the MIME end-of-line sequence, CRLF; the name of the HTML file to return in place of raw directory requests, indexfile; and the size of the data buffer used in I/O, buffer\_size.

Then mt defines a list of filename extensions and the corresponding MIME types for those files. The types Hashtable is statically initialized in the next block to contain the array mt as alternating keys and values. Then the fnameToMimeType( ) method can be used to return the proper MIME type for each filename passed in. If the filename does not have one of the extensions from the mt table, the method returns the defaultExt, or “text/plain.”

**STATISTICAL COUNTERS**

We declare five more instance variables. These are left without the private modifier so that an external monitor can inspect these values to display them graphically. These variables represent the usage statistics of our web server. The raw number of hits and bytes served is stored in hits\_served and bytes\_served. The number of files and bytes currently stored in the cache is stored in files\_in\_cache and bytes\_in\_cache. Finally, we store the number of hits that were successfully served out of the cache in hits\_to\_cache.

**toBytes( )**

We have a convenience routine, toBytes( ), which converts its string argument to an array of bytes. This is necessary, because Java String objects are stored as Unicode characters, while the lingua franca of Internet protocols such as HTTP is good old 8-bit ASCII. **makeMimeHeader( )**

The makeMimeHeader( ) method is another convenience routine that is used to create a MimeHeader object with a few key values filled in. The MimeHeader that is returned from this method has the current time and date in the Date field, the name and version of our server in the Server field, the type parameter in the Content-Type field, and the length parameter in the Content-Length field.

**error( )**

The error( ) method is used to format an HTML page to send back to web clients who make requests that cannot be completed. The first parameter, code, is the error code to return. Typically, this will be between 400 and 499. Our server sends back 404 and 405 errors. It uses

M.Sc. Semester – I (Computer Science) 34

Programming in Java Dr. M. T. Wanjari

the HttpResponse class to encapsulate the return code with the appropriate MimeHeader. The method returns the string representation of that response concatenated with the HTML page to show the user. The page includes a human-readable version of the error code, msg, and the url request that caused the error.

**getRawRequest( )**

The getRawRequest( ) method is very simple. It reads data from a stream until it gets two consecutive newline characters. It ignores carriage returns and just looks for newlines. Once it has found the second newline, it turns the array of bytes into a String object and returns it. It will return null if the input stream does not produce two consecutive newlines before it ends. This is how messages from HTTP servers and clients are formatted. They begin with one line of status and then are immediately followed by a MIME header. The end of the MIME header is separated from the rest of the content by two newlines.

**logEntry( )**

The logEntry( ) method is used to report on each hit to the HTTP server in a standard format. The format this method produces may seem odd, but it matches the current standard for HTTP log files. This method has several helper variables and methods that are used to format the date stamp on each log entry. The months array is used to convert the month to a string representation. The host variable is set by the main HTTP loop when it accepts a connection from a given host. The fmt02d( ) method formats integers between 0 and 9 as two-digit, leading

zero numbers. The resulting string is then passed through the LogMessage interface variable log. **writeString( )**

Another convenience method, writeString( ), is used to hide the conversion of a String to an array of bytes so that it can be written out to a stream.

**writeUCE( )**

The writeUCE( ) method takes an OutputStream and a UrlCacheEntry. It extracts the information out of the cache entry in order to send a message to a web client containing the appropriate response code, MIME header, and content.

**serveFromCache( )**

This Boolean method attempts to find a particular URL in the cache. If it is successful, then the contents of that cache entry are written to the client, the hits\_to\_cache variable is incremented, and the caller is returned true. Otherwise, it simply returns false.

M.Sc. Semester – I (Computer Science) 35

Programming in Java Dr. M. T. Wanjari

**loadFile( )**

This method takes an InputStream, the url that corresponds to it, and the MimeHeader for that URL. A new UrlCacheEntry is created with the information stored in the MimeHeader. The input stream is read in chunks of buffer\_size bytes and appended to the UrlCacheEntry. The resulting UrlCacheEntry is stored in the cache. The files\_in\_cache and bytes\_in\_cache variables are updated, and the UrlCacheEntry is returned to the caller.

**readFile( )**

The readFile( ) method might seem redundant with the loadFile( ) method. It isn’t. This method is strictly for reading files out of a local file system, where loadFile( ) is used to talk to streams of any sort. If the File object, f, exists, then an InputStream is created for it. The size of the file is determined and the MIME type is derived from the filename. These two variables are used to create the appropriate MimeHeader, then loadFile( ) is called to do the actual reading and caching.

**writeDiskCache( )**

The writeDiskCache( ) method takes a UrlCacheEntry object and writes it persistently into the local disk. It constructs a directory name out of the URL, making sure to replace the slash (/) characters with the system-dependent separatorChar. Then it calls mkdirs( ) to make sure that the local disk path exists for this URL. Lastly, it opens a FileOutputStream, writes all the data into it, and closes it.

**handleProxy( )**

The handleProxy( ) routine is one of the two major modes of this server. The basic idea is this: If set the browser to use this server as a proxy server, then the requests that will be sent to it will include the complete URL, where normal GETs remove the “http://” and host name part. We simply pick apart the complete URL, looking for the “://” sequence, the next slash (/), and optionally another colon (:) for servers using nonstandard port numbers. Once we’ve found these

characters, we know the intended host and port number as well as the URL we need to fetch from there. We can then attempt to load a previously saved version of this document out of our RAM cache. If this fails, we can attempt to load it from the file system into the RAM cache and reattempt loading it from the cache. If that fails, then it gets interesting, because we must read the document from the remote site. To do this, we open a socket to the remote site and port. We send a GET request, asking for the URL that was passed to us. Whatever response header we get back

M.Sc. Semester – I (Computer Science) 36

Programming in Java Dr. M. T. Wanjari

from the remote site, we send on to the client. If that code was 200, for successful file transfer, we also read the ensuing data stream into a new UrlCacheEntry and write it onto the client socket. After that, we call writeDiskCache( ) to save the results of that transfer to the local disk. We log the transaction, close the sockets, and return.

**handleGet( )**

The handleGet( ) method is called when the http daemon is acting like a normal web server. It has a local disk document root out of which it is serving files. The parameters to handleGet( ) tell it where to write the results, the URL to look up, and the MimeHeader from the requesting web browser. This MIME header will include the User-Agent string and other useful attributes. First we attempt to serve the URL out of the RAM cache. If this fails, we look in the file system for the URL. If the file does not exist or is unreadable, we report an error back to the web client. Otherwise, we just use readFile( ) to get the contents of the file and put them in the cache. Then writeUCE( ) is used to send the contents of the file down the client socket.

**doRequest( )**

The doRequest( ) method is called once per connection to the server. It parses the request string and incoming MIME header. It decides to call either handleProxy( ) or handleGet( ), based on whether there is a “://” in the request string. If any methods are used other than GET, such as HEAD or POST, this routine returns a 405 error to the client. Note that the HTTP request is ignored if stopFlag is true.

**run( )**

The run( ) method is called when the server thread is started. It creates a new ServerSocket on the given port, goes into an infinite loop calling accept( ) on the server socket, and then passes the resulting Socket off to doRequest( ) for inspection.

**start( ) AND stop( )**

These are two methods used to start and stop the server process. These methods set the value of stopFlag.

**main( )**

You can use the main( ) method to run this application from a command line. It sets the LogMessage parameter to be the server itself, and then provides a simple console output implementation of log( ).

Here is the source code for httpd:

M.Sc. Semester – I (Computer Science) 37

Programming in Java Dr. M. T. Wanjari

import java.net.\*;

import java.io.\*;

import java.text.\*;

import java.util.\*;

class httpd implements Runnable, LogMessage { private int port;

private String docRoot;

private LogMessage log;

private Hashtable cache = new Hashtable();

private boolean stopFlag;

private static String version = "1.0";

private static String mime\_text\_html = "text/html"; private static String CRLF = "\r\n";

private static String indexfile = "index.html";

private static int buffer\_size = 8192;

static String mt[] = { // mapping from file ext to Mime-Type "txt", "text/plain",

"html", mime\_text\_html,

"htm", "text/html",

"gif", "image/gif",

"jpg", "image/jpg",

"jpeg", "image/jpg",

"class", "application/octet-stream"

};

static String defaultExt = "txt";

static Hashtable types = new Hashtable();

static {

for (int i=0; i<mt.length;i+=2)

types.put(mt[i], mt[i+1]);

}

static String fnameToMimeType(String filename) { if (filename.endsWith("/")) // special for index files. return mime\_text\_html;

int dot = filename.lastIndexOf('.');

String ext = (dot > 0) ? filename.substring(dot + 1) : defaultExt; String ret = (String) types.get(ext);

return ret != null ? ret : (String)types.get(defaultExt); }

int hits\_served = 0;

int bytes\_served = 0;

int files\_in\_cache = 0;

int bytes\_in\_cache = 0;

int hits\_to\_cache = 0;

private final byte toBytes(String s)[] {

byte b[] = s.getBytes();

return b;

M.Sc. Semester – I (Computer Science) 38

Programming in Java Dr. M. T. Wanjari

}

private MimeHeader makeMimeHeader(String type, int length) { MimeHeader mh = new MimeHeader();

Date curDate = new Date();

TimeZone gmtTz = TimeZone.getTimeZone("GMT");

SimpleDateFormat sdf =

new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");

sdf.setTimeZone(gmtTz);

mh.put("Date", sdf.format(curDate));

mh.put("Server", "JavaCompleteReference/" + version);

mh.put("Content-Type", type);

if (length >= 0)

mh.put("Content-Length", String.valueOf(length));

return mh;

}

private String error(int code, String msg, String url) {

String html\_page = "<body>" + CRLF +

"<h1>" + code + " " + msg + "</h1>" + CRLF;

if (url != null)

html\_page += "Error when fetching URL: " + url + CRLF; html\_page += "</body>" + CRLF;

MimeHeader mh = makeMimeHeader(mime\_text\_html, html\_page.length()); HttpResponse hr = new HttpResponse(code, msg, mh);

logEntry("GET", url, code, 0);

return hr + html\_page;

}

// Read 'in' until you get two \n's in a row.

// Return up to that point as a String.

// Discard all \r's.

private String getRawRequest(InputStream in)

throws IOException {

byte buf[] = new byte[buffer\_size];

int pos=0;

int c;

while ((c = in.read()) != -1) {

switch (c) {

case '\r':

break;

case '\n':

if (buf[pos-1] == c) {

return new String(buf,0,pos);

}

default:

buf[pos++] = (byte) c;

}

}

M.Sc. Semester – I (Computer Science) 39

Programming in Java Dr. M. T. Wanjari

return null;

}

static String months[] = {

"Jan", "Feb", "Mar", "Apr", "May", "Jun",

"Jul", "Aug", "Sep", "Oct", "Nov", "Dec"

};

private String host;

// fmt02d is the same as C's printf("%02d", i)

private final String fmt02d(int i) {

if(i < 0) {

i = -i;

return ((i < 9) ? "-0" : "-") + i;

}

else {

return ((i < 9) ? "0" : "") + i;

}

}

private void logEntry(String cmd, String url, int code, int size) { Calendar calendar = Calendar.getInstance();

int tzmin = calendar.get(Calendar.ZONE\_OFFSET)/(60\*1000); int tzhour = tzmin / 60;

tzmin -= tzhour \* 60;

log.log(host + " - - [" +

fmt02d(calendar.get(Calendar.DATE) ) + "/" +

months[calendar.get(Calendar.MONTH)] + "/" + calendar.get(Calendar.YEAR) + ":" +

fmt02d(calendar.get(Calendar.HOUR) ) + ":" +

fmt02d(calendar.get(Calendar.MINUTE) ) + ":" + fmt02d(calendar.get(Calendar.SECOND)) + " " + fmt02d(tzhour) + fmt02d(tzmin) +

"] \"" +

cmd + " " +

url + " HTTP/1.0\" " +

code + " " +

size + "\n");

hits\_served++;

bytes\_served += size;

}

private void writeString(OutputStream out, String s) throws IOException {

out.write(toBytes(s));

}

private void writeUCE(OutputStream out, UrlCacheEntry uce) throws IOException {

HttpResponse hr = new HttpResponse(200, "OK", uce.mh); writeString(out, hr.toString());

M.Sc. Semester – I (Computer Science) 40

Programming in Java Dr. M. T. Wanjari

out.write(uce.data, 0, uce.length);

logEntry("GET", uce.url, 200, uce.length);

}

private boolean serveFromCache(OutputStream out, String url) throws IOException {

UrlCacheEntry uce;

if ((uce = (UrlCacheEntry)cache.get(url)) != null) { writeUCE(out, uce);

hits\_to\_cache++;

return true;

}

return false;

}

private UrlCacheEntry loadFile(InputStream in, String url, MimeHeader mh)

throws IOException {

UrlCacheEntry uce;

byte file\_buf[] = new byte[buffer\_size];

uce = new UrlCacheEntry(url, mh);

int size = 0;

int n;

while ((n = in.read(file\_buf)) >= 0) {

uce.append(file\_buf, n);

size += n;

}

in.close();

cache.put(url, uce);

files\_in\_cache++;

bytes\_in\_cache += uce.length;

return uce;

}

private UrlCacheEntry readFile(File f, String url) throws IOException {

if (!f.exists())

return null;

InputStream in = new FileInputStream(f);

int file\_length = in.available();

String mime\_type = fnameToMimeType(url);

MimeHeader mh = makeMimeHeader(mime\_type, file\_length); UrlCacheEntry uce = loadFile(in, url, mh);

return uce;

}

private void writeDiskCache(UrlCacheEntry uce) throws IOException {

String path = docRoot + uce.url;

String dir = path.substring(0, path.lastIndexOf("/"));

M.Sc. Semester – I (Computer Science) 41

Programming in Java Dr. M. T. Wanjari

dir.replace('/', File.separatorChar);

new File(dir).mkdirs();

FileOutputStream out = new FileOutputStream(path); out.write(uce.data, 0, uce.length);

out.close();

}

// A client asks us for a url that looks like this: // http://the.internet.site/the/url

// we go get it from the site and return it...

private void handleProxy(OutputStream out, String url, MimeHeader inmh) {

try {

int start = url.indexOf("://") + 3;

int path = url.indexOf('/', start);

String site = url.substring(start, path).toLowerCase(); int port = 80;

String server\_url = url.substring(path);

int colon = site.indexOf(':');

if (colon > 0) {

port = Integer.parseInt(site.substring(colon + 1)); site = site.substring(0, colon);

}

url = "/cache/" + site + ((port != 80) ? (":" + port) : "") + server\_url;

if (url.endsWith("/"))

url += indexfile;

if (!serveFromCache(out, url)) {

if (readFile(new File(docRoot + url), url) != null) { serveFromCache(out, url);

return;

}

// If we haven't already cached this page, open a socket // to the site's port and send a GET command to it. // We modify the user-agent to add ourselves... "via". Socket server = new Socket(site, port);

InputStream server\_in = server.getInputStream(); OutputStream server\_out = server.getOutputStream(); inmh.put("User-Agent", inmh.get("User-Agent") + " via JavaCompleteReferenceProxy/" + version); String req = "GET " + server\_url + " HTTP/1.0" + CRLF + inmh + CRLF;

writeString(server\_out, req);

String raw\_request = getRawRequest(server\_in); HttpResponse server\_response =

new HttpResponse(raw\_request);

writeString(out, server\_response.toString());

M.Sc. Semester – I (Computer Science) 42

Programming in Java Dr. M. T. Wanjari

if (server\_response.statusCode == 200) {

UrlCacheEntry uce = loadFile(server\_in, url,

server\_response.mh);

out.write(uce.data, 0, uce.length);

writeDiskCache(uce);

logEntry("GET", site + server\_url, 200, uce.length); }

server\_out.close();

server.close();

}

} catch (IOException e) {

log.log("Exception: " + e);

}

}

private void handleGet(OutputStream out, String url, MimeHeader inmh) {

byte file\_buf[] = new byte[buffer\_size];

String filename = docRoot + url +

(url.endsWith("/") ? indexfile : "");

try {

if (!serveFromCache(out, url)) {

File f = new File(filename);

if (! f.exists()) {

writeString(out, error(404, "Not Found", filename)); return;

}

if (! f.canRead()) {

writeString(out, error(404, "Permission Denied", filename)); return;

}

UrlCacheEntry uce = readFile(f, url);

writeUCE(out, uce);

}

} catch (IOException e) {

log.log("Exception: " + e);

}

}

private void doRequest(Socket s) throws IOException { if(stopFlag)

return;

InputStream in = s.getInputStream();

OutputStream out = s.getOutputStream();

String request = getRawRequest(in);

int fsp = request.indexOf(' ');

int nsp = request.indexOf(' ', fsp+1);

int eol = request.indexOf('\n');

M.Sc. Semester – I (Computer Science) 43

Programming in Java Dr. M. T. Wanjari

String method = request.substring(0, fsp);

String url = request.substring(fsp+1, nsp);

String raw\_mime\_header = request.substring(eol + 1); MimeHeader inmh = new MimeHeader(raw\_mime\_header); request = request.substring(0, eol);

if (method.equalsIgnoreCase("get")) {

if (url.indexOf("://") >= 0) {

handleProxy(out, url, inmh);

} else {

handleGet(out, url, inmh);

}

} else {

writeString(out, error(405, "Method Not Allowed", method)); }

in.close();

out.close();

}

public void run() {

try {

ServerSocket acceptSocket;

acceptSocket = new ServerSocket(port);

while (true) {

Socket s = acceptSocket.accept();

host = s.getInetAddress().getHostName();

doRequest(s);

s.close();

}

} catch (IOException e) {

log.log("accept loop IOException: " + e + "\n"); } catch (Exception e) {

log.log("Exception: " + e);

}

}

private Thread t;

public synchronized void start() {

stopFlag = false;

if (t == null) {

t = new Thread(this);

t.start();

}

}

public synchronized void stop() {

stopFlag = true;

log.log("Stopped at " + new Date() + "\n");

}

public httpd(int p, String dr, LogMessage lm) {

M.Sc. Semester – I (Computer Science) 44

Programming in Java Dr. M. T. Wanjari

port = p;

docRoot = dr;

log = lm;

}

// This main and log method allow httpd to be run from the console.

public static void main(String args[]) {

httpd h = new httpd(80, "c:\\www", null);

h.log = h;

h.start();

try {

Thread.currentThread().join();

} catch (InterruptedException e) {};

}

public void log(String m) {

System.out.print(m);

}

}

**HTTP.java**

As an added bonus, here is an applet class that gives the HTTP server a functional “front panel.” This applet has two parameters that can be used to configure the server: port and docroot. This is a very simple applet. It makes an instance of the httpd, passing in itself as the LogMessage interface. Then it creates a panel that has a simple label at the top, a TextArea in the middle for displaying the LogMessages, and a panel at the bottom that has two buttons and another label in it. The start( ) and stop( ) methods of the applet call the corresponding methods on the httpd. The buttons labeled “Start” and “Stop” call their corresponding methods in the httpd. Any time a message is logged, the bottom-right Label object is updated to contain the latest statistics from the httpd.

import java.util.\*;

import java.applet.\*;

import java.awt.\*;

import java.awt.event.\*;

public class HTTP extends Applet implements LogMessage,

ActionListener

{

private int m\_port = 80;

private String m\_docroot = "c:\\www";

private httpd m\_httpd;

private TextArea m\_log;

private Label status;

private final String PARAM\_port = "port";

M.Sc. Semester – I (Computer Science) 45

Programming in Java Dr. M. T. Wanjari

private final String PARAM\_docroot = "docroot"; public HTTP() {

}

public void init() {

setBackground(Color.white);

String param;

// port: Port number to listen on

param = getParameter(PARAM\_port);

if (param != null)

m\_port = Integer.parseInt(param);

// docroot: web document root

param = getParameter(PARAM\_docroot);

if (param != null)

m\_docroot = param;

setLayout(new BorderLayout());

Label lab = new Label("Java HTTPD");

lab.setFont(new Font("SansSerif", Font.BOLD, 18)); add("North", lab);

m\_log = new TextArea("", 24, 80);

add("Center", m\_log);

Panel p = new Panel();

p.setLayout(new FlowLayout(FlowLayout.CENTER,1,1)); add("South", p);

Button bstart = new Button("Start");

bstart.addActionListener(this);

p.add(bstart);

Button bstop = new Button("Stop");

bstop.addActionListener(this);

p.add(bstop);

status = new Label("raw");

status.setForeground(Color.green);

status.setFont(new Font("SansSerif", Font.BOLD, 10)); p.add(status);

m\_httpd = new httpd(m\_port, m\_docroot, this); }

public void destroy() {

stop();

}

public void paint(Graphics g) {

}

M.Sc. Semester – I (Computer Science) 46

Programming in Java Dr. M. T. Wanjari

public void start() {

m\_httpd.start();

status.setText("Running ");

clear\_log("Log started on " + new Date() + "\n");

}

public void stop() {

m\_httpd.stop();

status.setText("Stopped ");

}

public void actionPerformed(ActionEvent ae) {

String label = ae.getActionCommand();

if(label.equals("Start")) {

start();

}

else {

stop();

}

}

public void clear\_log(String msg) {

m\_log.setText(msg + "\n");

}

public void log(String msg) {

m\_log.append(msg);

status.setText(m\_httpd.hits\_served + " hits (" +

(m\_httpd.bytes\_served / 1024) + "K), " +

m\_httpd.files\_in\_cache + " cached files (" +

(m\_httpd.bytes\_in\_cache / 1024) + "K), " +

m\_httpd.hits\_to\_cache + " cached hits");

status.setSize(status.getPreferredSize());

}

}

Note:-In the files httpd.java and HTTP.java, the code is built assuming that the document root is “c:\www”. Need to change this value for configuration. Because this applet writes to a log file, it can work only if it is trusted. For example, an applet is trusted if it is accessible from the user’s class path.

**Datagrams**

For most of internetworking needs TCP/IP-style networking. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes

M.Sc. Semester – I (Computer Science) 47

Programming in Java Dr. M. T. Wanjari

many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: The DatagramPacket object is the data container, while the DatagramSocket is the mechanism used to send or receive the DatagramPackets.

**DatagramPacket**

DatagramPacket defines several constructors. Four are described here. The **first** constructor specifies a buffer that will receive data, and the size of a packet. It is used for receiving data over a DatagramSocket. The **second** form allows to specify an offset into the buffer at which data will be stored. The **third** form specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent. The **fourth** form transmits packets beginning at the specified offset into the data. Think of the first two forms as building an “in box,” and the second two forms as stuffing and addressing an envelope. Here are the four constructors:

DatagramPacket(byte data[ ], int size)

DatagramPacket(byte data[ ], int offset, int size)

DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)

DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port) There are several methods for accessing the internal state of a DatagramPacket. They give complete access to the destination address and port number of a packet, as well as the raw data and its length. Here are some of the most commonly used:

InetAddress getAddress( ) Returns the destination InetAddress, typically

used for sending.

int getPort( ) Returns the port number.

byte[ ] getData( ) Returns the byte array of data contained in the

M.Sc. Semester – I (Computer Science) 48

Programming in Java Dr. M. T. Wanjari

datagram. Mostly used to retrieve data from the

datagram after it has been received.

int getLength( ) Returns the length of the valid data contained in

the byte array that would be returned from the

getData( ) method. This typically does not equal

the length of the whole byte array.

**Datagram Server and Client**

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

// Demonstrate Datagrams.

import java.net.\*;

class WriteServer {

public static int serverPort = 998;

public static int clientPort = 999;

public static int buffer\_size = 1024;

public static DatagramSocket ds;

public static byte buffer[] = new byte[buffer\_size];

public static void TheServer() throws Exception {

int pos=0;

while (true) {

int c = System.in.read();

switch (c) {

case -1:

System.out.println("Server Quits.");

return;

case '\r':

break;

case '\n':

ds.send(new DatagramPacket(buffer,pos,

InetAddress.getLocalHost(),clientPort));

pos=0;

break;

default:

buffer[pos++] = (byte) c;

}

}

M.Sc. Semester – I (Computer Science) 49

Programming in Java Dr. M. T. Wanjari

}

public static void TheClient() throws Exception {

while(true) {

DatagramPacket p = new DatagramPacket(buffer, buffer.length);

ds.receive(p);

System.out.println(new String(p.getData(), 0, p.getLength()));

}

}

public static void main(String args[]) throws Exception {

if(args.length == 1) {

ds = new DatagramSocket(serverPort);

TheServer();

} else {

ds = new DatagramSocket(clientPort);

TheClient();

}

}

}

This sample program is restricted by the DatagramSocket constructor to running between two ports on the local machine. To use the program, run java WriteServer in one window; this will be the client. Then run java WriteServer 1

This will be the server. Anything that is typed in the server window will be sent to the client window after a newline is received.

Note: - This example requires that computer be connected to the Internet.

**Inet4Address and Inet6Address**

As mentioned, Java 2, version 1.4 added support for IPv6 addresses. Because of two new subclasses of InetAddress were created: Inet4Address and Inet6Address. Inet4Address represents a traditional style, IPv4 address. Inet6Address encapsulates a new-style IPv6 address. Because they are subclasses of InetAddress, an InetAddress reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, can simply use InetAddress when working with IP addresses because it can accommodate both styles.

M.Sc. Semester – I (Computer Science) 50

Programming in Java Dr. M. T. Wanjari

**The URI Class**

Java 2, version 1.4 added the URI class, which encapsulates a Uniform Resource Identifier. URIs are similar to URLs. In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource. **Remote Method Invocation (RMI):**

This module explains the basics of object oriented programming System and Java. This module also explains the core technologies that are implemented for developing distributed and enterprise applications in Java.

This includes an introduction to Java Database Connectivity (JDBC) API and moves on to the advanced features of JDBC. This module discusses how to develop JavaBeans reusable components and associate custom events with them. This module also covers Java network and distributed programming.

The module consists of five sub modules:

1. Java Fundamentals

2. Programming in Java

3. Packages and Streams in Java

4. JDBC and JavaBeans

5. Network and Distributed Programming in Java

Distributes applications are based on the multi-tired development architecture where clients remain transparent of the location of the server. Thus distributed applications enable clients and servers to communicate with each other irrespective of their physical locations. In Java, RMI technology provides the platform to develop distributed applications.

**Overview of Distributed Application**

Over the years, there have been three different approaches to application development – the traditional approach, the client/server approach, and the component based approach. In the **traditional approach**, there was a single application that handled the presentation logic, business logic and database interactivity. These applications were also called monolithic applications. With no inherent advantages (being the only known method for application development), the drawback of this approach was that if even a minor change, extension or enhancement was required in an application, the entire application had to be recompiled and integrated again. For example, a minor change in the database structure would require all

M.Sc. Semester – I (Computer Science) 51

Programming in Java Dr. M. T. Wanjari

functions or methods in the application to be changed. This made the cost of updating and redistributing the application very high.

Due to the disadvantage of the traditional approach, the client/server architecture (also called the two-tier architecture) was introduced. In this architecture, data is separated from the client-side and is stored at a centralized location that acts as a server. The business logic is combined with presentation logic either at the client-side or at the server-side that has the database connectivity code. If the business logic is combined with the presentation logic at the client-side, the client is called a fat client. If the business logic is combined with the database server, the server is called a fat server.

However, the client/server architecture also has certain disadvantages:

a) Any change in business policies requires a change in the business logic. To change the business logic, either the presentation logic or the database connectivity code needs to change, depending on the location of the business logic.

b) Applications implemented using two-tier architecture might be difficult to scale up because of the limited number of database connections available to a client. Connection requests beyond a particular limit are simply rejected by the server.

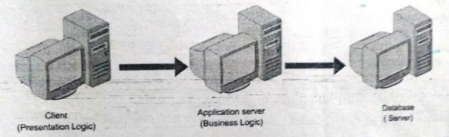
The disadvantages with the client/server architecture led to the development of the three tier architecture. In the three-tier architecture, the presentation logic resides at the client-side, the database access is controlled by the server-side, and the business logic resides between the two layers. This business logic layers is referenced to as application server (also called the middle tier of component based three-tier architecture). This type of architecture is called server-centric, because it enables components on application to run on the middle-tier application servers implementing the business rules, independent of both the presentation interface and the database implementation.

These components can be developed using any programming language that allow the creation of components. The components can be centralized for easy development, maintenance and deployment. Since the middle-tier handles the business logic, the workload is balanced between the client, the database server, and the server handling the business logic. This architecture also provides efficient data access. The problem with database connection limitation is minimized since the database sees only the business logic layer and not all its client. In the case of a two-tier application, a database connection is established early and is maintained,

M.Sc. Semester – I (Computer Science) 52

Programming in Java Dr. M. T. Wanjari

whereas in a three-tier application, a database connection is established only when data access is required and released as soon as the data is retrieved or sent to the server.

**Fig. 3.6 A Distributed Application**

The applications where the presentation logic, the business logic and the database reside on multiple computers are called **distributed applications**.

**Remote Method Invocation**

We already discussed how to cross process boundaries to transfer data from one host to another by using sockets. Another way to achieve the same is by using Remote Method Invocation (RMI). RMI is a specification that enables one Java Virtual Machine (JVM) to invoke methods in an object located in another JVM. These two JVM’s, could be running on different computers or running as separate processes on the same computer. RMI is implemented on the middle-tier of the three-tier architecture framework, thereby facilitating the programmers to invoke distributed components across a networked environment.

Sun introduced RMI as an easy alternative to the complex coding involved in server socket programming. For using RMI, the programmer need not know socket programming or multi-threading and needs to solely concentrate on developing the business logic. **Components of RMI Application**

A distributed RMI application has two components:

RMI Server

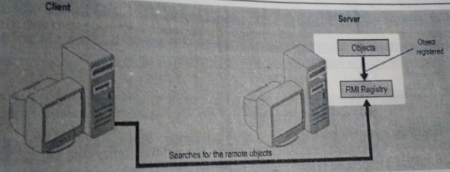
RMI client

RMI server contains the objects whose methods are to be invoked remotely. The server creates several remote objects and makes a reference of these objects in the RMI registry. The RMI registry is a service that runs on the RMI server. The remote objects created by the server are registered by the objects unique name in this registry. The client gets the reference of one or

M.Sc. Semester – I (Computer Science) 53

Programming in Java Dr. M. T. Wanjari

more remote objects from the RMI registry by looking up the object name. The client then invokes the methods on the remote object(s) to access the service of the remote object(s). the following figure illustrates the functionality of applications in RMI.

**Fig. 3.7 Functioning of Applications in RMI**

Once the client gets the reference of the remote object, the methods in the remote object are invoked just like the methods of a local object. The difference cannot be identified in terms of whether the methods are invoked on the remote object or are invoked on the local objects in the client.

**The RMI Architecture**

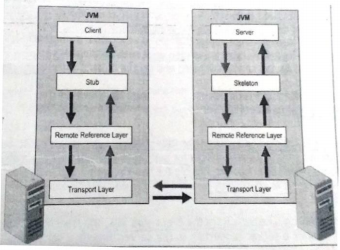
The RMI architecture consists of three layers:

1. Stub/Skeleton layer

2. Remote Reference Layer

3. Transport Layer

M.Sc. Semester – I (Computer Science) 54

Programming in Java Dr. M. T. Wanjari 

**Fig.3.8 RMI Architecture**

**1. Stub/Skeleton Layer**

The stub/skeleton layer listens to remote method calls made by a client and redirects these calls to the remove RMI services on the server. This layer consists of a stub and a skeleton.

To invoke methods of a remote object, the request on the client-side starts with a stub. A stub is a client-side proxy representing the remote object. It is reference by programs as any other local object running on the client and provides methods of a remote object. The stub communicates the method invocations to the remote object through a skeleton that is implemented on the server.

The skeleton is a server-side proxy that continues communication with the stub by:

a) Reading the parameters for the method call

b) Making the call to the remote service implementation object.

c) Accepting the return value.

d) Writing the return value back to the stub.

M.Sc. Semester – I (Computer Science) 55

Programming in Java Dr. M. T. Wanjari

**2. Remote Reference Layer**

The Remote Reference Layer (RRL) interprets and manages the references made by a client to a remote object on a server. This layer is present on the client as well as the server. The RRL on the client-side receives a request for methods from a stub that is transferred as a marshaled stream of data to the RRL on the server-side. Marshalling is a process in which parameters passed by a client are converted to a format that can be transferred across a network. The server-side RRL unmarshals the parameters that are sent to a remote method through the skeleton. Unmarshalling is a process in which the marshaled parameters passed by the client-side RRL through the server-side RRL are converted to a format that the skeleton understands. While returning a value from the skeleton, the data is again marshaled and communicated to the client through the server side RRL.

**3. Transport Layer**

The Transport Layer is a link between the RRL on server-side and RRL on client side. The transport layer is responsible for setting up new connections and managing existing connections. It is also responsible for handling remote objects that are residing in transport layer address space. The following steps explain how the client is connected to the server:

a) On receiving a request from the client-side RRL, the transport layer establishes a socket connection to the server through a server-side RRL.

b) Then, the transport layer passes the established connection to the client-side RRL and adds a remote reference to the connection in its table.

**RMI Packages**

RMI packages consist of various classes and interfaces for creating and running the distributed applications using RMI. Java provides the following packages for remote method invocation:

a) The **java.rmi** package provides the Remote interface, a class for accessing the remote names registered on the server and a security manager for RMI.

b) The **java.rmi.registry** package provides classes and interfaces that are used by the remote registry.

M.Sc. Semester – I (Computer Science) 56

Programming in Java Dr. M. T. Wanjari

c) The **java.rmi.server** package provides classes and interfaces used to implement remote objects, stubs and skeletons and support for RMI communication.

d) The **java.rmi.dgc** package provides classes and interfaces that are used by the RMI distributed garbage collector.

**a) The java.rmi Package**

The *java.rmi* package declares the *Remote* interface and the *Naming* and *RMISecurityManager* classes. It also defines a number of exception classes that are used with RMI.

All remote objects must implement the *Remote* interface. This interface does not have methods and is used for identifying remote objects. The *java.rmi* package consists of the following classes:

**i) Naming** class: Provides static methods for accessing remote objects through URLs. This class supports the following methods:

• **rebind() :** binds a remote object name to a specified URL and is normally used by the server object.

• **unbind() :** Removes the binding between an object name and a URL. • **lookup() :** Returns the remote object specified by a URL and is normally used by the client object.

• **list() :** Returns the list of URLs that are currently known to the RMI registry. **ii) RMISecurityManager** class: defines the default security policy for remote object stubs. In applies only to applications. Applets use the *AppletSecurityManager* class for RMI. The *setSecurityManager()* method of the *System* class is used to set an *RMISecurityManager* object as the security manager to be used for RMI stubs. The *java.rmi* package defines a number of exceptions. The *RemoteException* class is the parent of all the exceptions that are generated during RMI.

**b) The java.rmi.registry Package**

The *java.rmi.registry* package provides the *Registry* and *RegistryHandller* interfaces and locates the registry. These interfaces are used to register and access remote objects by name. *Remote* objects are registered when they are bound to the registry process of a host. The registry process is started when the *start rmiregistry* command is

M.Sc. Semester – I (Computer Science) 57

Programming in Java Dr. M. T. Wanjari

executed. The command defines the *rebind()*, *unbind()*, *list()*, and *lookup()* methods that are used by the *Naming* class to associate the names and RMI URLs.

**c) The java.rmi.server Package**

The *java.rmi.server* package implements the interface and classes that support both the client and server parts of the RMI. The *java.rmi.server* package consists of the following classes and interfaces:

i) *RemoteObject* class: Implements the *Remote* interface and provide remote implementation of the *object* class. All objects that implement remote objects extend the *RemoteObject* class.

ii) *RemoteServer* class: Extends the *RemotObject* class and is a common class that is subclassed by specific types of remote object implementations.

iii) *UnicastRemoteObject* class: Extends the *RemoteServer* class and provides the default *RemoteObject* implementation. Classes that are implement RemoteObject usually subclass the *UnicastRemoteObject*. By extending from the *UnicastRmoteObject*, the subclass can:

• Use RMI’s default socket-based transport for communication.

• Run all the time.

iv) *RemoteStub* class: Provides an abstract implementation of the client-side stubs. The static setRef() method is used to associate a client-side stub with its corresponding remote objects.

v) *RemoteCall* interface: Provides method that are used by all the stubs and skeletons of RMI.

vi) *Skeleton* interface: Provides the method to access methods of the remote object and this interface is implemented by the remote skeletons.

vii) *Unreferenced* interface: Enables to determine when a client no longer references a remote object. This interface is implemented by the Remote class.

**d) The java.rmi.dgc Package**

The *java.rmi.dgc* package contains classes and interfaces that are used by the distributed garbage collector. The server side of the distributed garbage collector implements this interface. This package contains:

i) The *Lease* class that creates objects that are used to keep track of object references.

M.Sc. Semester – I (Computer Science) 58

Programming in Java Dr. M. T. Wanjari

ii) The *dirty()* method that is used to indicate that a client is referencing a remote object. iii) The *clean()* method that is used to indicate that a remote reference has been completed.

**Distributed Garbage Collection**

Java provides a distributed garbage collection mechanism that automatically deletes the remote objects that RMI clients are not using. RMI uses a reference counting garbage collection algorithm to manage the references of the remote objects. The stubs and skeletons layer use the *java.rmi.dgc* interface for implementing the distributed garbage collection mechanism. A remote object that implements the *java.rmi.server.Unreferenced* interface receives notification when no more client references exist. When there is no local or remote reference to the remote object the garbage collection is implemented.

When any new object enters into the JVM, the RMI runtime object of the Lease class increases the reference counter by one and marks the remote object as dirty. Similarly, when an existing remote object is dropped from the JVM, the object of Lease class decreases the counter by one and marks the object as clean. In the reference counting mechanism, a remote reference can sleep in the JVM for a lease time. The system property, *java.rmi.dgc.leaseValue*, controls the lease time. If the reference does not refresh the connection to the remote object before the lease time expires, RMI deallocates the memory of the remote object using the distributed garbage collection mechanism. The lease time is specified in milliseconds and the default value of lease time is 10 minutes.

**Creating a Distributed Application using RMI**

A distributed application using RMI contains several files, such as interface, server file or client file. According to RMI specifications, you need to follow certain steps to create a distributed RMI application. The steps are as follows:

i) Create a remote interface.

ii) Implement the remote interface.

iii) Create an RMI server.

iv) Create an RMI Client.

v) Run the RMI Application.

**i) Creating Remote Interface**

M.Sc. Semester – I (Computer Science) 59

Programming in Java Dr. M. T. Wanjari

A remote interface specifies the methods that can be invoked remotely by a client. The remote interface needs to declare each of the methods that has to be invoked from other JVMs. Remote interfaces have the following characteristics:

a) A remote interface must be declared *public*, This is because, in most applications, clients are not part of the same package as the remote interface.

b) A remote interface extends the *java.rmi.Remote* interface.

c) Each method must declare *java.rmi.RemoteException* in its throw clause to trap network connection and server problems.

You can use the following code to declare a remote interface that represents a remote object:

import java.rmi.\*;

public interface Hello extends Remote

{

/\* Declare the remote method \*/

public String sayHello() throws RemoteException;

}

In the preceding code, the *Hello* interface declares the *sayHello()* method, which throws the *RemoteException*.

**ii) Implementing the Remote Interface**

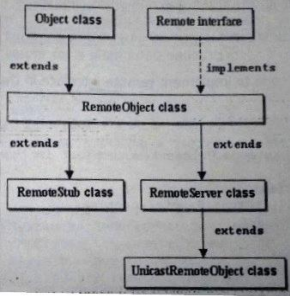
You need to implement the remote interface to create a remote service class that provides information about server objects. The remote service class defines all the methods that are declared in the remote interface. You need to import two packages, java.rmi and java.rmi.server. you can use the following code snippet to import java packages:

import java.rmi.\*;

import java.rmi.server.\*;

The remote service class extends the *UnicastRemoteObject* class for implementing a method of remote interface. The *UnicastRemoteObject* class extends the *RemoteServer* class of *java.rmi* and defines all the methods of the *RemoteServer* class. The following figure shows the hierarchy of the *UnicastRemoteObject* class:

M.Sc. Semester – I (Computer Science) 60

Programming in Java Dr. M. T. Wanjari 

**Fig. 3.9 UnicastRemoteObject Class Hierarchy**

You can create a remote service class, HelloImpl, by implementing the remote interface, Hello after importing the Java packages. You can use the following code snippet to create remote service class:

public class HelloImpl implements Hello extends UnicastRemoteObject

{

**. .**

**. .**

}

The remote object instance needs to be exported. This enables the remote object to accept remote method requests by listening at a particular port. Since the *HelloImpl* class extends the *UnicastRemoteObject* class, it is exported, automatically.

The super() method invokes the constructor of *UnicastRemoteObject* class, which exports the remote object. You can use the following code snippet to define the default constructor of the remote service class:

public HelloImpl() throws RemoteException

M.Sc. Semester – I (Computer Science) 61

Programming in Java Dr. M. T. Wanjari

{

super();

}

You define all the methods of the remote interface in the remote service class after defining the default constructor. You can use the following code snippet to define the remote method in the remote service class:

public String sayHello() throws RemoteException

{

return “Hello! Peter Smith.”;

}

You can use the following code to implement remote interface in the remote service class:

import java.rmi.\*;

import java.rmi.server.\*;

public class HelloImpl extends UnicastRemoteObject implements Hello

{

public HelloImpl() throws RemoteException

{

super();

}

public String sayHello() throws RemoteException

{

return "Hello! Peter Smith.";

}

}

In the preceding code, the *HelloImpl* service class consists of the implementation of *sayHello()* method declare in the *Hello* interface.

**iii) Creating an RMI Server**

The RMI server class contains the server objects that are invoked by the client remotely. You need to create the object of the remote service class in the main() method of the RMI server class to create the server object. You can use the following statement to create a server object: Hello h = new HelloImpl();

You need to register the server object in the bootstrap registry before the server object is ready to accept request from the client. You pass the name and reference of a server object in the RMI registry to register the server objects. The name of server object is used to access the stub object

M.Sc. Semester – I (Computer Science) 62

Programming in Java Dr. M. T. Wanjari

using the lookup mechanism. You can use the following statement to register a server object in the registry:

Naming.rebind(“server”, h);

The rebind() method takes two parameters:

• The first parameter is a URL string that contains the location and name of the remote object. If the port number is not specified, the RMI registry uses the default port, 1099. If the user has to define the port, the URL string has to be changed to “rmi://ipaddress:1234.server”.

• The second parameter is a reference to the object implementation.

The *setSecurityManager()* method of *SecurityManager* class is used in the server class to set the security manager for the RMI application, so any unauthorized client cannot invoked the server object. To authenticate a RMI client, you need to create the security policy file that contains all the required permissions. You can use the following code to create an RMI server for a distributed application:

import java.rmi.\*;

import java.rmi.server.\*;

public class HelloServer extends UnicastRemoteObject implements Hello {

HelloServer() throws RemoteException

{

System.out.println("Server Created");

}

public static void main(String args[])

{

try

{

HelloServer h=new HelloServer();

Naming.rebind("server",h);

System.out.println("object is registered");

System.out.println("now server is waiting for client");

}

catch(Exception e)

{

System.out.println("Error:"+e);

}

}

M.Sc. Semester – I (Computer Science) 63

Programming in Java Dr. M. T. Wanjari

public String sayHello() throws RemoteException

{

return"Hello! Peter Smith.";

}

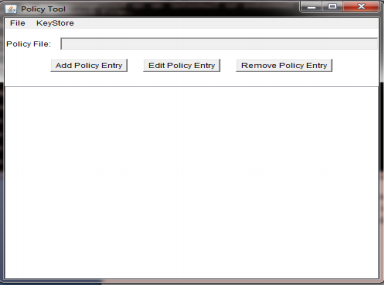
}

In the preceding code, the rebind() method of Naming class throws a remote exception. As a result, you define the rebind() method within the try-catch block.

**Security Policy**

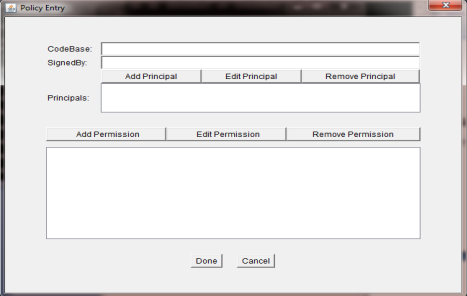
A policy file is a plain text file created by the graphical Policy Tool utility. You will create a policy file named .java.policy under your Home directory that contains the code to grant permission to the required objects. These steps to create a security policy file are:

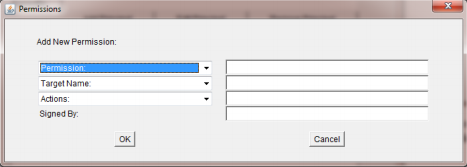
1. Type the command, policytool, on command prompt to display the policy Tool utility of Java. The Policy Tool window is displayed with an Error dialog box. The following figure shows the Policy Tool window:



2. Click the Add Policy Entry button to add a new security policy file. The following figure shows the Policy Entry dialog box:

M.Sc. Semester – I (Computer Science) 64

Programming in Java Dr. M. T. Wanjari 

3. Click the Add Permission button to grant the required permission to access the resources of the server by the client. The following figure shows the Permission dialog box. 

4. Select AllPermission from the Permission list box of the Permissions dialog box. 5. Click the Ok button to close the Permissions dialog box.

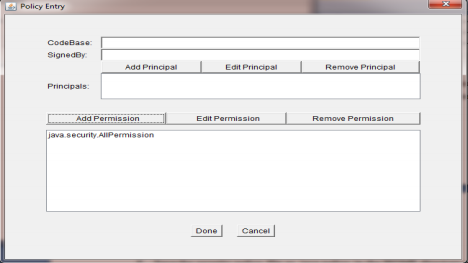
6. Click the Done button of the Policy Entry dialog box.

7. Select the File -> Save as command of the Policy Tool window.

M.Sc. Semester – I (Computer Science) 65

Programming in Java Dr. M. T. Wanjari

8. Save the security policy file as .java,policy in the HOME directory of the operating system. A confirmation message is displayed on the Status dialog box. The following figure shows the Status message box:

9. Click the Done button of the Dialog box

10. Select the File -> Exit command of the Policy Tool window to close the Policy Tool utility of Java.

**iv) Creating an RMI Client**

The client uses a stub object to access the remote object that exists on the server. You specify the name of the server object in the lookup() method of the java.rmi.Naming class to find the stub object. You can use the following statement to access a stub object using the lookup() method:

Hello h = (Hello) Naming.lookup(“rmi://192.168.0.52/server”);

In the preceding statement, the lookup() method throws a remote exception, as a result the lookup() method should be enclosed within the try-catch block. The server represents the name of the server object. When you are running this application on local computer then you use localhost instead of IP address of RMI server. You can use the following code to create an RMI client for the distributed application:

import java.rmi.\*;

public class HelloClient

M.Sc. Semester – I (Computer Science) 66

Programming in Java Dr. M. T. Wanjari

{

public static void main(String args[])

{

try

{

Hello h=(Hello)Naming.lookup("rmi://localhost/server");

System.out.println("Client:Hello!");

System.out.println("Server:"+h.sayHello());

}

catch(Exception e)

{

System.out.println("Error:"+e);

}

}

}

After creating the Java source files, you compile all the source files of the distributed application, Hello, using the standard Java compiler, Javac, to generate class files. The command to compile the Java source files of the distributed application is:

Javac <File\_name>.java

In the preceding command, the File\_Name represents the name of the Java source file and javac is the standard java compiler. After compiling the Java source files, you need to generate stub and skeleton that communicates between client and server.

**v) Running the RMI Application**

You need to register server objects to the RMI registry before you run the RMI application. The client can invoke the registered server objects across the network. The steps to run an RMI application are:

1. Generate the stub and skeleton of the remote service class.

2. Start the RMI registry

3. Run the RMI server of distributed application.

4. Run the RMI client of distributed application.

**1. Generating Stub and Skeleton**

The RMI compiler, rmic, compiles the remote service class that implements the remote interface and generates the stub and skeleton. The stub enables the client to communicate with a particular remote object. The skeleton represents the object of client that is located on the remote host. The command to generate the stub and skeleton is:

M.Sc. Semester – I (Computer Science) 67

Programming in Java Dr. M. T. Wanjari

rmic [option] <ClassFile>

In the preceding command, the ClassFile represents the name of the remote service class. The option represents the parameters that provide the extra facilities to the RMI compiler. The following table lists various options of RMI compiler:

|  |  |
| --- | --- |
| **Options** | **Description** |
| -bootclasspath<path> | Overrides the location of the bootstrap class file. |
| -classpath<path> | Overrides the default classpath environment variables. |
| -d <directory> | Specifies the name of the directory where you want to generate the stub and skeleton. By default, the stub and skeleton are generated in the current drive. |
| -depend | Compiles all the files that are referenced to the remote service class. |
| -extdirs <path> | Overrides the location of the installed extensions. |
| -g | Generates the line numbers and local variables in the form of table. |
| -keep | Stores ‘.java’ file that generates the stub and skeleton. |
| -nowarn | Does not display any warning when you generate the stub and skeleton. |
| -vcompat | Creates stub and skeleton that are compatible with earlier version of RMI protocols. |
| -verbose | Display a message, when the remote service file is compiled by rmic. |
| -v <version> | Creates stub and skeleton for the specified version of the Java Development Kit (JDK). |

The command to generate the stub and skeleton for an RMI application:

rmic HelloImpl

The two files, HelloImpl\_Stub.class and HelloImpl\_Skel.class, are generated in the same folder where you store the remote service classs, HelloImpl. You can test the distributed application on the local computer by storing all the files in the same folder before deploying it on the network. To run the application on the network, you have to create two folders, server folder on server computer and client folder on client computer. The files that are saved in the server folder of the server computer are:

• Hello.class

• HelloImpl.class

• HelloServer.class

• HelloImpl\_Stub.class

The files that are saved in the client folder of the client computer are:

• Hello.class

M.Sc. Semester – I (Computer Science) 68

Programming in Java Dr. M. T. Wanjari

• HelloImpl\_Stub.class

• HelloClient.class

**2. Starting the RMI Registry**

To start the RMI Registry on the server, execute the start rmiregistry command at the command prompt. By default, the registry runs on port 1099.

To start the RMI Registry on a different port, you need to specify the port nimber on the command line as follows:

start rmiregistry 1234

If the registry is running on a port other than 1099, you will have to specify the port number in the URL string specified in the rebind() and the lookup() methods of the Naming class. The command to start the RMI registry at the default port number is as follows: start rmiregistry

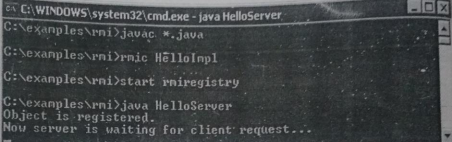
The preceding command opens a new Command Prompt window in which rmiregistry runs.

**3. Running an RMI Server**

You need to start the server to service the client requests. The command to run the HelloServer RMI server is:

Java HelloServer

The following figure shows the output of the RMI Server application:

**Fig. 3.10 Running an RMI Server Application**

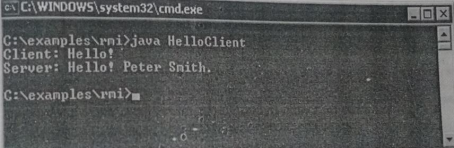
**4. Running an RMI Client**

The command to run the RMI client is:

Java HelloClient

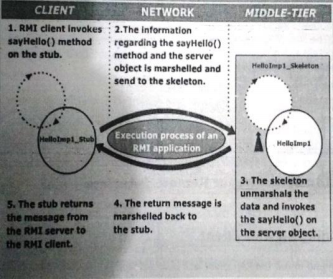
The following figure shows the output of the RMI Client application:

M.Sc. Semester – I (Computer Science) 69

Programming in Java Dr. M. T. Wanjari 

**Fig. 3.11 Running an RMI Client Application**

The RMI client invokes the sayHello() method to display the message received from the RMI server. The following figure shows the communication between the RMI client and server:



**Fig. 3.12 Working of a Distributed Application Using RMI**

**Implementing an RMI Application**

**Problem Statement**

Earnest publishing house has branches in all major cities across the globe. They have a central database in Chicago. The publishing house wants to create an application that will allow

M.Sc. Semester – I (Computer Science) 70

Programming in Java Dr. M. T. Wanjari

its authors to register with the publishing house. The details submitted by the authors needs to be updated in the database server.

**Solution**

For the solution this problem, you need to create a distributed application using RMI that contains two modules, server and client. The client is a GUI-based application that accepts an author’s details and sends these details to the server by invoking a remote method. The server application creates a connection with the database server using RMI; you need to perform the following tasks:

1. Create the AuthorServer remote interface.

2. Create the AutorServerImpl class

3. Create the Client class.

4. Run the Application.

**1. Creating the AuthorServer Remote Interface**

The AuthorServer interface declares the remote objects that will be invoked by the RMI client. The AuthorServer interface declares an insertDetails() method that stores the details of an author in the database server.

The code to create the AuthorServer interface is as follows:

import java.rmi.\*;

public interface AuthorServer extends Remote

{

/\* Declare the remote method\*/

String insertDetails(String authorID,String lastName,String firstName,String phone,String address,String city,String state,String zip) throws RemoteException;

}

Save the preceding code as AuthorServer.java.

**2. Creating the AuthorServerImpl Class**

The AuthorServerImpl class stores the details of an author in the database server. To create an AuthorServerImpl class you need to perform the following tasks: • Implement the ServerRemote interface.

• Define the constructor for a remote object.

• Provide implementation for a remote method.

M.Sc. Semester – I (Computer Science) 71

Programming in Java Dr. M. T. Wanjari

• Create and install the security manager.

• Create an instance of a remote object.

• Register a remote object with the RMI registry.

The code to create the AuthorServerImpl class is as follows:

import java.rmi.\*;

import java.rmi.server.UnicastRemoteObject;

import java.sql.\*;

public class AuthorServerImpl extends UnicastRemoteObject implements AuthorServer {

static ResultSet result;

static Connection con;

static PreparedStatement stat;

/\*Define the default constructor of the Impl class\*/

public AuthorServerImpl() throws RemoteException

{

super();

}

/\* Define the remote object\*/

public String insertDetails(String authorID,String lastName,String firstName,String phone,String address,String city,String state,String zip)throws RemoteException {

int rowsAffected=0;

String sReturn="fail";

try

{

/\*Register the database driver\*/

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

/\* Create the connection\*/

con=DriverManager.getConnection("jdbc:odbc:mydatasource","administrator"," " );

/\* Create the prepareStatement to insert the value in the database\*/

stat=con.prepareStatement("insert into

authors(au\_id,au\_lname,au\_fname,phone,address,city,state,zip)values(?,?,?,?,?,?,?,?)"); stat.setString(1,authorID);

stat.setString(2,lastName);

stat.setString(3,firstName);

stat.setString(4,phone);

stat.setString(5,address);

stat.setString(6,city);

stat.setString(7,state);

stat.setString(8,zip);

rowsAffected=stat.executeUpdate();

if(rowsAffected>0)

M.Sc. Semester – I (Computer Science) 72

Programming in Java Dr. M. T. Wanjari

{

sReturn="success";

}

}

catch(Exception v)

{

System.out.println("Error at value insert"+v);

}

return sReturn;

}

/\*Define the main method\*/

public static void main(String args[])

{

/\* set the security manager\*/

System.setSecurityManager(new RMISecurityManager());

try

{

/\* Create the instance of the Impl class\*/

AuthorServerImpl instance=new AuthorServerImpl();

/\* Bind the server object to the RMI registry \*/

Naming.rebind("AuthorServer",instance);

System.out.println("Server Registered");

}

catch(Exception e)

{

System.out.println(e);

}

}

Save the preceding code as AuthorServerImpl.java.

**3. Creating the Client Class**

The Client class obtains the reference of the remote object and invokes the remote methods. To create a Client class, you need to perform the following tasks: • Create an instance of the Client class to connect it to the RMI registry. • Locate the server object that is registered with the RMI server.

The code to create the Client class is as follows:

import javax.swing.\*;

import java.rmi.\*;

import java.awt.event.\*;

import java.awt.\*;

public class Client

{

/\* declare the variables\*/

static JFrame frame;

M.Sc. Semester – I (Computer Science) 73

Programming in Java Dr. M. T. Wanjari

static JPanel panel;

static JPanel panel1;

JLabel labelAuthorID;

JLabel labelLastName;

JLabel labelFirstName;

JLabel labelPhone;

JLabel labelAddress;

JLabel labelCity;

JLabel labelState;

JLabel labelZip;

JTextField textAuthorID;

JTextField textLastName;

JTextField textFirstName;

JTextField textPhone;

JTextField textAddress;

JTextField textCity;

JTextField textState;

JTextField textZip;

JButton submit;

static String authorID;

static String lastName;

static String firstName;

static String phone;

static String address;

static String city;

static String state;

static String zip;

/\* define the default constructor \*/

public Client()

{

/\* create the JFrame \*/

frame=new JFrame("Earnest publishing house"); panel=new JPanel();

panel1=new JPanel();

/\* set the layout managers \*/

panel.setLayout(new GridLayout(8,2));

panel1.setLayout(new GridLayout(1,1));

frame.setVisible(true);

frame.setSize(400,400);

frame.getContentPane().setLayout(new BorderLayout()); /\*Define the swing components on the jframe\*/

labelAuthorID=new JLabel("Author ID");

labelLastName=new JLabel("Last Name");

labelFirstName=new JLabel("First Name");

labelAddress=new JLabel("Phone");

labelPhone=new JLabel("Address");

M.Sc. Semester – I (Computer Science) 74

Programming in Java Dr. M. T. Wanjari

labelCity=new JLabel("City");

labelState=new JLabel("State");

labelZip=new JLabel("Zip");

textAuthorID=new JTextField(5);

textLastName=new JTextField(15);

textFirstName=new JTextField(15);

textPhone=new JTextField(10);

textAddress=new JTextField(50);

textCity=new JTextField(10);

textState=new JTextField(10);

textZip=new JTextField(6);

submit=new JButton("Submit");

/\*Add the swing components on the panel\*/

panel.add(labelAuthorID);

panel.add(textAuthorID);

panel.add(labelLastName);

panel.add(textLastName);

panel.add(labelFirstName);

panel.add(textFirstName);

panel.add(labelPhone);

panel.add(textPhone);

panel.add(labelAddress);

panel.add(textAddress);

panel.add(labelCity);

panel.add(textCity);

panel.add(labelState);

panel.add(textState);

panel.add(labelZip);

panel.add(textZip);

panel1.add(submit);

ButtonListener blisten=new ButtonListener();

submit.addActionListener(blisten);

frame.getContentPane().add(new JPanel(),BorderLayout.WEST); frame.getContentPane().add(new JPanel(),BorderLayout.EAST); frame.getContentPane().add(panel,BorderLayout.CENTER); frame.getContentPane().add(panel1,BorderLayout.SOUTH); frame.setVisible(true);

}

/\*Create a ButtonListener class\*/

class ButtonListener implements ActionListener

{

/\*Define the ActionPerformed() method\*/

public void actionPerformed(ActionEvent evt)

M.Sc. Semester – I (Computer Science) 75

Programming in Java Dr. M. T. Wanjari

{

JButton source=(JButton)evt.getSource();

MyDialog myDialog;

try

{

/\*Lookup the server object\*/

AuthorServer

server=(AuthorServer)Naming.lookup("rmi:/localHost/AuthorServer");

//\*Retrieve the details of author from the client.frame\*/

authorID=textAuthorID.getText();

lastName=textLastName.getText();

firstName=textFirstName.getText();

phone=textPhone.getText();

address=textAddress.getText();

city=textCity.getText();

state=textState.getText();

zip=textZip.getText();

/\*Invoke the remote method\*/

String

str=server.insertDetails(authorID,lastName,firstName,phone,address,city,state,zip); System.out.println(str);

if(str.equals("success"))

{

myDialog=new MyDialog(frame,"Inserted Successfully");

}

else

{

/\*Create the object of myDialog class\*/

myDialog=new MyDialog(frame,"no record inserted");

}

myDialog.setVisible(true);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

/\*Define the main() method\*/

public static void main(String args[])

{

/\*Create the object of client class\*/

new Client();

}

M.Sc. Semester – I (Computer Science) 76

Programming in Java Dr. M. T. Wanjari

}

/\*Create the MyDialog class\*/

class MyDialog extends Dialog implements ActionListener

{

/\*Define the default constructor of the MyDialog class\*/

MyDialog(Frame parent,String title)

{

super(parent,title ,false);

setLayout(new FlowLayout());

setSize(200,80);

add(new JLabel(title));

JButton btn\_OK=new JButton("OK");

add(btn\_OK);

btn\_OK.addActionListener(this);

}

/\*Define the actionPerformed() method\*/

public void actionPerformed(ActionEvent ae)

{

dispose();

}

}

Save the preceding code as Client.java.

**4. Running the Application**

The steps to run the distributed application created in the previous step are: 1. Select the Start -> Program -> Accessories -> Command Prompt command to display the Command Prompt window.

2. Change the directory to the directory that contains the files, AuthorServer.java, AuthorServerImpl.java and Clent.java.

3. Compile the java classes using the javac compiler of Java. The command to compile all the files is:

javac \*.java

4. Generate the stub and skeleton using the following command:

rmic AuthorServerImpl

5. Start the RMI registry using the following command:

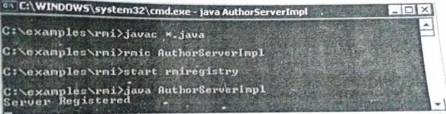
start rmiregistry

6. Run the RMI server using the following command:

java AuthorServerImpl

The following figure shows the output of the RMI server application:

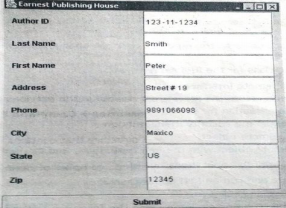
M.Sc. Semester – I (Computer Science) 77

Programming in Java Dr. M. T. Wanjari 

**Fig. 3.13 Console Window of the RMI Server Application**

7. Open a new command window and change to the directory where the AuthorServer.java and Client.java files are saved. Execute the client class using the following command: java Client

The following figure shows the output of the RMI client:



**Fig. 3.14 RMI Client**

**Transmitting Files using RMI**

**Problem Statement**

Create a distributed application using RMI, where client can download a file (plain text or binary) from the RMI server.

**Solution**

To send and receive files across JVM, you use serialization in RMI application. Serialization enables you to send data files and pictures from one JVM to another JVM in the

M.Sc. Semester – I (Computer Science) 78

Programming in Java Dr. M. T. Wanjari

form of objects. Serilization is the process of converting a set of object instances into a linear stream of bytes. You can serialize a remove object into a byte array and send byte array to the client.

To create the distributed application discussed above using RMI, you need to perform the following tasks:

1. Create the FileRemote remote interface for the RMI Server.

2. Create the FileRemoteImpl class that implements the FileRemote interface. 3. Create the RMIFileServer class

4. Create the RMIFileClient class

5. Run the Application.

**1. Creating the FileRemote Remote Interface**

The FileRemote interface declares the loadFile() method that will be invoked by the RMU client to download the specified file from the RMI server. The loadFile() method takes a string argument, as filename and returns a byte array.

The code to create the FileRemote interface is as follows:

import java.rmi.Remote;

import java.rmi.RemoteException;

public interface FileRemote extends Remote

{

/\*declare the remote method\*/

public byte[]loadFile(String filename)throws RemoteException;

}

Save the preceding code as FileRmote.java.

**2. Creating the FileRemoteImpl Class**

The FileRemoteImpl class implements the remote method declared in the remote interface. To create the FileRemoteImpl class, you need to perform the following tasks: • Implement the FileRemote interface

• Extend the UnicastRemoteObject class

• Create the constructor.

• Implement the remote method.

• In the remote method create an object of the File class.

• In the remote method convert the file object to a byte array.

The code to create the FileRemoteImpl class is as follow:

M.Sc. Semester – I (Computer Science) 79

Programming in Java Dr. M. T. Wanjari

import java.io.\*;

import java.rmi.\*;

import java.rmi.server.UnicastRemoteObject;

public class FileRemoteImpl extends UnicastRemoteObject implements FileRemote

{

private String name;

/\*Define the default constructor \*/

public FileRemoteImpl(String str) throws RemoteException

{

super();

name=str;

}

/\*Define the remote method\*/

public byte[]loadFile(String filename)

{

try

{

/\*Create the object of file class\*/

File file=new File(filename);

/\*Create the bytearray same as the size of the file\*/

byte bufferFile[]=new byte[(int)file.length()];

/\*Create the object of BufferedInputStream class to read the file\*/ BufferedInputStream inputFile=new

BufferedInputStream(new FileInputStream(filename));

/\*Read the file\*/

inputFile.read(bufferFile,0,bufferFile.length);

/\*Close the file\*/

inputFile.close();

return(bufferFile);

}

catch(Exception e)

{

System.out.println("Error:"+e);

}

return(null);

}

}

Save the preceding code as FileRemoteImpl.java.

M.Sc. Semester – I (Computer Science) 80