Chapter 13

Socket Programming

This chapter presents key concepts of intercommunication between programs running on different computers in the network. It introduces elements of network programming and concepts involved in creating network applications using sockets. The chapter introduces the `java.net` package containing various classes required for creating sockets and message communication using two different protocols. It provides several example programs demonstrating various capabilities supported by Java for creating network applications.

Objectives

After learning the contents of this chapter, the reader will be able to:

- understand fundamental concepts of computer communication
- understand sockets and ports
- understand java.net package features
- program Java Sockets
- create comprehensive network applications using sockets

13.1 Introduction

Internet and WWW have emerged as global ubiquitous media for communication and changed the way we conduct science, engineering, and commerce. They are also changing the way we learn, live, enjoy, communicate, interact, engage, etc. The modern life activities are getting completely centered around or driven by the Internet.

To take advantage of opportunities presented by the Internet, businesses are continuously seeking new and innovative ways and means for offering their services via the Internet. This created a huge demand for software designers and engineers with skills in creating new Internet-enabled applications or porting existing/legacy applications to the Internet platform. The key elements for developing Internet enabled applications are a good understanding of the issues involved in implementing distributed applications and sound knowledge of the fundamental network programming models.

13.1.1 Client/Server Communication

At a basic level, network based systems consist of a server, client, and a media for communication as shown in Figure 13.1. A computer running a program that makes request for services is called client machine. A computer running a program that offers requested services from one or more clients is called server machine. The media for communication can be wired or wireless network.

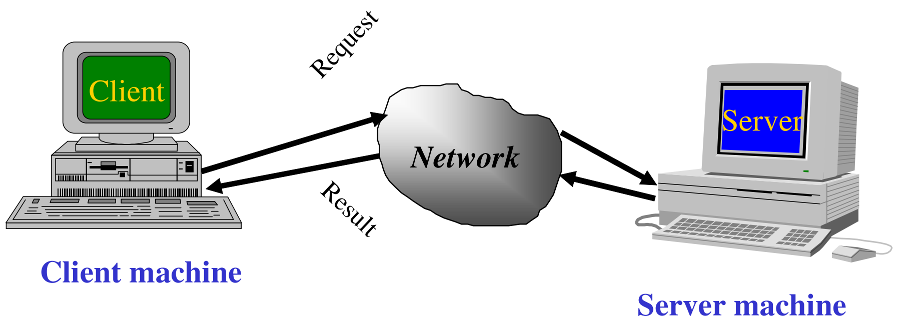

Figure 13.1: Client – Server communication

Generally, programs running on client machines make requests to a program (often called as server program) running on a server machine. They involve networking services provided by the transport layer, which is part of the Internet software stack, often called as *TCP/IP (Transport Control Protocol/Internet Protocol) stack*, shown in Figure 13.2. The transport layer comprises two types of protocols, *TCP (Transport Control Protocol)* and *UDP (User Datagram Protocol)*. The most widely used programming interfaces for these protocols are sockets.

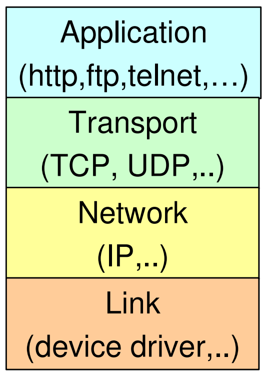

Figure 13.2: TCP/IP software stack

TCP is a connection-oriented protocol that provides a reliable flow of data between two computers. Example applications that use such services are HTTP, FTP, and Telnet.

UDP is a protocol that sends independent packets of data, called *datagrams*, from one computer to another with no guarantees about arrival and sequencing. Example applications that use such services include Clock server and Ping.

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer. Port is represented by a positive (16-bit) integer value. Some ports have been reserved to support common/well known services:

• ftp	21/tcp
• telnet	23/tcp
• smtp	25/tcp
• login	513/tcp
• http	80/tcp,udp

- https 443/tcp,udp

User level process/services generally use port number value >= 1024.

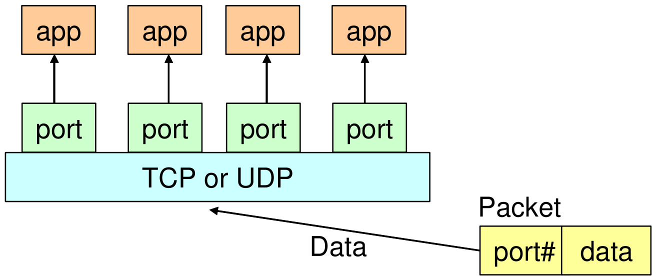

Figure 13.3: TCP/UDP mapping of incoming packets to appropriate port/process

Object-oriented Java technologies – Sockets, threads, RMI, clustering, Web services – have emerged as leading solutions for creating portable, efficient, and maintainable large and complex Internet applications.

13.1.2 Hosts Identification and Service Ports

Every computer on the Internet is identified by a unique, four-byte *IP address*. This is typically written in dotted quad format like 128.250.25.158 where each byte is an unsigned value between 0 and 255. This representation is clearly not user friendly because does not tell us anything about the content and then it is difficult to remember. Hence, IP addresses are mapped to names like *www.buyya.com* or *www.google.com* which are easier to remember. Internet supports name servers that translate these names to IP addresses.

In general, each computer only has one Internet address. However, computers often need to communicate and provide more than one type of service or to talk to multiple hosts/computers at a time. For example, there may be multiple ftp sessions, web connections, and chat programs all running at the same time. To distinguish these services, a concept of *ports*, a logical access point, represented by a 16-bit integer number is used. That means, each service offered by a computer is uniquely identified by a port number. Each Internet packet contains both the destination host address and the port number on that host to which the message/request has to be delivered. The host computer dispatches the packets it receives to programs by looking at the port numbers specified within the packets. That is, IP address can be thought of as a house address when a letter is sent via post/snail mail and port number as the name of a specific individual to whom the letter has to be delivered.

13.1.3 Sockets and Socket-based Communication

Sockets provide an interface for programming networks at the transport layer. Network communication using Sockets is very much similar to performing file I/O. In fact, socket handle is treated like file handle.

The streams used in file I/O operation are also applicable to socket-based I/O. Socket-based communication is independent of a programming language used for implementing it. That means, a socket program written in Java language can communicate to a program written in non-Java (say C or C++) socket program.

A server (program) runs on a specific computer and has a socket that is bound to a specific port. The server listens to the socket for a client to make a connection request (see Figure 13.4a). If everything goes well, the server accepts the connection (see Figure 13.4b). Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (consequently a different port number) so that it can continue to listen to the original socket for connection requests while serving the connected client.

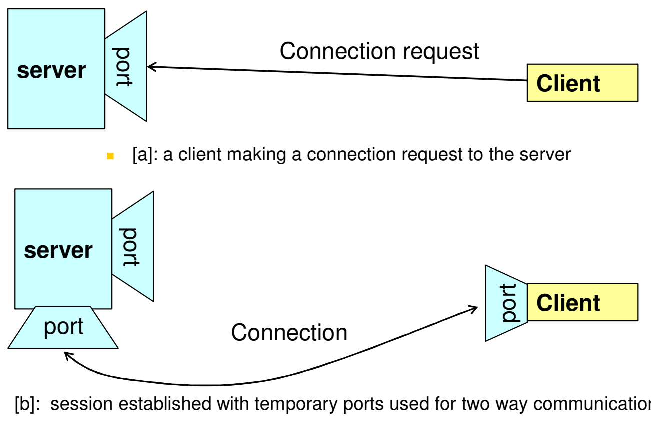
server
port
Connection request
Client
[a]: a client making a connection request to the server
server
port
port
Connection
port
Client
[b]: session established with temporary ports used for two way communication.

Figure 13.4: Establishment of path for two-way communication between a client and server

13.2 Socket Programming and java.net Class

A socket is an endpoint of a two-way communication link between two programs running on the network. Socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent. Java provides a set of classes, defined in a package called `java.net`, to enable the rapid development of network applications. Key classes, interfaces, and exceptions in `java.net` package simplifying the complexity involved in creating client and server programs are:

The Classes

- `ContentHandler`
- `DatagramPacket`
- `DatagramSocket`
- `DatagramSocketImpl`
- `HttpURLConnection`
- `InetAddress`
- `MulticastSocket`
- `ServerSocket`
- `Socket`
- `SocketImpl`
- `URL`
- `URLConnection`
- `URLEncoder`
- `URLStreamHandler`

The Interfaces

- `ContentHandlerFactory`
- `FileNameMap`
- `SocketImplFactory`
- `URLStreamHandlerFactory`

Exceptions

- `BindException`
- `ConnectException`
- `MalformedURLException`
- `NoRouteToHostException`
- `ProtocolException`
- `SocketException`
- `UnknownHostException`
- `UnknownServiceException`

13.3 TCP/IP Socket Programming

The two key classes from the `java.net` package used in creation of server and client programs are:

- `ServerSocket`
- `Socket`

A server program creates a specific type of socket that is used to listen for client requests (server socket), In the case of a connection request, the program creates a new socket through which it will exchange data with the client using input and output streams. The socket abstraction is very similar to the file concept: developers have to open a socket, perform I/O, and close it. Figure 13.5 illustrates key steps involved in creating socket-based server and client programs.

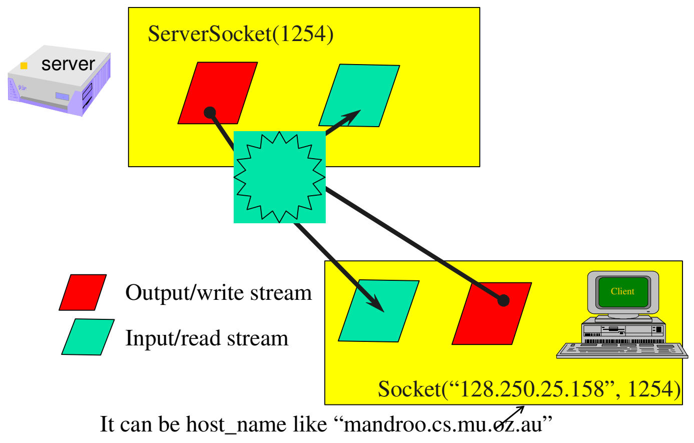
server
ServerSocket(1254)
Output/write stream
Input/read stream
Client
Socket("128.250.25.158", 1254)
It can be host_name like "mandroo.cs.mu.oz.au"

Figure 13.5: Socket-based client and server programming

A simple Server Program in Java

The steps for creating a simple server program are:

1. Open the Server Socket:

```
ServerSocket  server = new ServerSocket( PORT );
```

2. Wait for the Client Request:

```
Socket client = server.accept();
```

3. Create I/O streams for communicating to the client

```
DataInputStream is = new DataInputStream(client.getInputStream());
DataOutputStream os = new DataOutputStream(client.getOutputStream());
```

4. Perform communication with client

```
Receive from client: String line = is.readLine();
Send to client: os.writeBytes("Hello\n");
```

5. Close socket:

```
client.close();
```

An example program illustrating creation of a server socket, waiting for client request, and then responding to a client which requested for connection by greeting it as given below:

Program 13.1

```
// SimpleServer.java: A simple server program.
import java.net.*;
import java.io.*;
public class SimpleServer {
   public static void main(String args[]) throws IOException {
      // Register service on port 1254
      ServerSocket s = new ServerSocket(1254);
      Socket s1=s.accept(); // Wait and accept a connection
      // Get a communication stream associated with the socket
      OutputStream s1out = s1.getOutputStream();
      DataOutputStream dos = new DataOutputStream (s1out);
      // Send a string!
      dos.writeUTF("Hi there");
      // Close the connection, but not the server socket
      dos.close();
      s1out.close();
      s1.close();
   }
}
```

A simple Client Program in Java

The steps for creating a simple client program are:

1. Create a Socket Object:

```
Socket client = new Socket(server, port_id);
```

2. Create I/O streams for communicating with the server.

```
   is = new DataInputStream(client.getInputStream());
   os = new DataOutputStream(client.getOutputStream());
```

3. Perform I/O or communication with the server:

```
Receive data from the server:  String line = is.readLine();
Send data to the server:  os.writeBytes("Hello\n");
```

4. Close the socket when done:

```
client.close();
```

An example program illustrating establishment of connection to a server and then reading a message sent by then server and displaying it on the console is given below:

Program 13.2

```
// SimpleClient.java: A simple client program.
import java.net.*;
import java.io.*;
public class SimpleClient {
   public static void main(String args[]) throws IOException {
     // Open your connection to a server, at port 1254
     Socket s1 = new Socket("localhost",1254);
     // Get an input file handle from the socket and read the input
     InputStream s1In = s1.getInputStream();
     DataInputStream dis = new DataInputStream(s1In);
     String st = new String (dis.readUTF());
     System.out.println(st);
     // When done, just close the connection and exit
     dis.close();
     s1In.close();
     s1.close();
   }
}
```

Running Socket Programs

Compile both server and client programs and then deploy server program code on a machine which is going to act as a server and client program which is going to act as a client. If required, both client and server programs can run on the same machine. To illustrate execution of server and client programs, let us assume that a machine called `mundroo.csse.unimelb.edu.au` on which we want to run a server program as indicated below:

```
[raj@mundroo] java SimpleServer
```

The client program can run on any computer in the network (LAN, WAN, or Internet) as long as there is no firewall between them that blocks communication. Let us say we want to run our client program on a machine called `gridbus.csse.unimelb.edu.au` as follows:

```
[raj@gridbus] java SimpleClient
```

The client program is just establishing a connection with the server and then waits for a message. On receiving a response message, it prints the same to the console. The output in this case is:

```
Hi there
```

which is sent by the server program in response to a client connection request.

It should be noted that once the server program execution is started, it is not possible for any other server program to run on the same port until the first program which is successful using it is terminated. Port numbers are a mutually exclusive resource, they cannot be shared among different processes at the same time.

13.4 UDP Socket Programming

The previous two example programs used the TCP sockets. As already said, TCP guarantees the delivery of packets and preserves their order on destination. Sometimes these features are not required and it since they do not come without performance costs, it would be use a lighter transport protocol. This kind of service is accomplished by the UDP protocol which conveys *datagram packets*.

Datagram packets are used to implement a connectionless packet delivery service supported by the UDP protocol. Each message is transferred from source machine to destination based on information contained within that packet. That means, each packet needs to have destination address and each packet might be routed differently, and might arrive in any order. Packet delivery is not guaranteed.

The format of datagram packet is:

▪ | Msg | length | Host | serverPort |

Java supports datagram communication through the following classes:

- `DatagramPacket`
- `DatagramSocket`

The class `DatagramPacket` contains several constructors that can be used for creating packet object. One of them is:

```
DatagramPacket(byte[] buf,     int length,     InetAddress address,
int port);
```

This constructor is used for creating a datagram packet for sending packets of length `length` to the specified port number on the specified host. The message to be transmitted is indicated in the first argument. The key methods of `DatagramPacket` class are:

`byte[] getData()`
    Returns the data buffer.

`int getLength()`
  Returns the length of the data to be sent or the length of the data received.

`void setData(byte[] buf)`
  Sets the data buffer for this packet.

`void setLength(int length)`
  Sets the length for this packet.

The class `DatagramSocket` supports various methods that can be used for transmitting or receiving data a datagram over the network. The two key methods are:

`void send(DatagramPacket p)`
  Sends a datagram packet from this socket.

`void receive(DatagramPacket p)`
  Receives a datagram packet from this socket.

A simple UDP server program that waits for client's requests and then accepts the message (datagram) and sends back the same message is given below. Of course, extended server program can manipulate client's messages/request and send a new message as a response.

Program 13.3

```
// UDPServer.java: A simple UDP server program.
import java.net.*;
import java.io.*;
public class UDPServer {
   public static void main(String args[]){
      DatagramSocket aSocket = null;
      if (args.length < 1) {
           System.out.println("Usage: java UDPServer <Port Number>");
           System.exit(1);
      }
      try {
          int socket_no = Integer.valueOf(args[0]).intValue();
        aSocket = new DatagramSocket(socket_no);
        byte[] buffer = new byte[1000];
        while(true) {
          DatagramPacket request = new DatagramPacket(buffer,
                                                       buffer.length);
          aSocket.receive(request);
          DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(),request.getAddress(),
                    request.getPort());
          aSocket.send(reply);
        }
      }
      catch (SocketException e) {
          System.out.println("Socket: " + e.getMessage());
      }
      catch (IOException e) {
          System.out.println("IO: " + e.getMessage());
      }
      finally {
          if (aSocket != null)
            aSocket.close();
      }
   }
}
```

A corresponding client program for creating a datagram and then sending to the above server and then accepting a response is listed below.

Program 13.4

```
// UDPClient.java: A simple UDP client program.
import java.net.*;
import java.io.*;
public class UDPClient {

   public static void main(String args[]){
      // args give message contents and server hostname
      DatagramSocket aSocket = null;
      if (args.length < 3) {
        System.out.println(
          "Usage: java UDPClient <message> <Host name> <Port number>");
        System.exit(1);
      }
      try {
        aSocket = new DatagramSocket();
        byte [] m = args[0].getBytes();
        InetAddress aHost = InetAddress.getByName(args[1]);
        int serverPort = Integer.valueOf(args[2]).intValue();
        DatagramPacket request =
               new DatagramPacket(m, args[0].length(), aHost, serverPort);
        aSocket.send(request);
        byte[] buffer = new byte[1000];
        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
        aSocket.receive(reply);
        System.out.println("Reply: " + new String(reply.getData()));
      }
      catch (SocketException e) {
        System.out.println("Socket: " + e.getMessage());
      }
      catch (IOException e) {
        System.out.println("IO: " + e.getMessage());
      }
      finally {
        if (aSocket != null)
                   aSocket.close();
      }
   }
}
```

13.5 Math Server

It is time to implement a more comprehensive network application by using the socket programming APIs you have learned so far. A sample math client-server interaction demonstrating online math server that can perform basic math operations is shown in Figure 13.6.

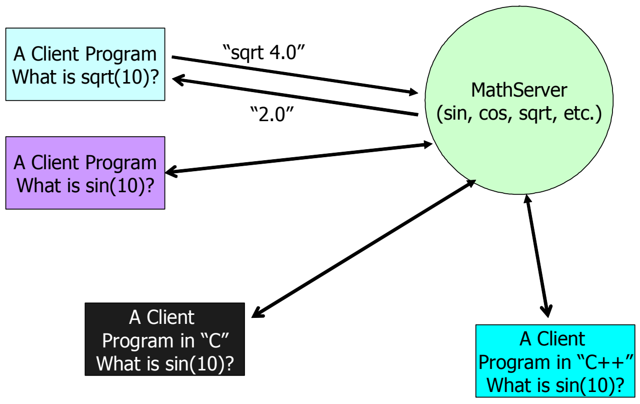

Figure 13.6: A socket-based math server and clients.

The basic math interface is shown as follows:

Program 13.5

```
// MathService.java: A basic math interface.
public interface MathService {
   public double add(double firstValue, double secondValue);
   public double sub(double firstValue, double secondValue);
   public double div(double firstValue, double secondValue);
   public double mul(double firstValue, double secondValue);
}
```

The implementation of this interface is not related to any network operation. The following code shows a very simple implementation of this interface:

Program 13.6

```
// PlainMathService.java: An implementation of the MathService interface.
public class PlainMathService implements MathService {

   public double add(double firstValue, double secondValue) {
      return firstValue+secondValue;
   }
   public double sub(double firstValue, double secondValue) {
      return firstValue-secondValue;
   }
   public double mul(double firstValue, double secondValue) {
          return firstValue * secondValue;
   }
   public double div(double firstValue, double secondValue) {
      if (secondValue != 0)
       return firstValue / secondValue;
      return Double.MAX_VALUE;
   }
}
```

The implementation of the `MathServer` is quite straightforward which looks pretty similar to the echo server mentioned previously. The difference is that the `MathServer` have to consider the specific protocol defined by the math server and client communication. The program uses a very simple protocol *operator:first_value:second_value*, it is the math server's responsibility to understand this protocol and delegate to the proper methods such as *add*, *sub*, *mul* or *div*.

Program 13.7

```
// MathServer.java : An implementation of the MathServer.
import java.io.*;
import java.net.*;

public class MathServer{
   protected MathService mathService;
   protected Socket socket;

   public void setMathService(MathService mathService) {
      this.mathService = mathService;
   }
   public void setSocket(Socket socket) {
      this.socket = socket;
   }
   public void execute() {
      try {
```

```
      BufferedReader reader = new BufferedReader(
           new InputStreamReader(socket.getInputStream()));
      // read the message from client and parse the execution
      String line = reader.readLine();
      double result = parseExecution(line);
      // write the result back to the client
      BufferedWriter writer = new BufferedWriter(
           new OutputStreamWriter(socket.getOutputStream()));
      writer.write(""+result);
      writer.newLine();
      writer.flush();
      // close the stream
      reader.close();
      writer.close();
    }
        catch (Exception e) {
      e.printStackTrace();
    }
}
// the predefined protocol for the math operation is
// operator:first value:second value
protected double parseExecution(String line)
                 throws IllegalArgumentException {
   double result = Double.MAX_VALUE;
   String [] elements = line.split(":");
   if (elements.length != 3)
     throw new IllegalArgumentException("parsing error!");
   double firstValue = 0;
   double secondValue = 0;
   try {
     firstValue = Double.parseDouble(elements[1]);
     secondValue = Double.parseDouble(elements[2]);
   }
       catch(Exception e) {
     throw new IllegalArgumentException("Invalid arguments!");
   }
   switch (elements[0].charAt(0)) {
     case '+':
       result = mathService.add(firstValue, secondValue);
     break;
     case '-':
       result = mathService.sub(firstValue, secondValue);
```

```
        break;
        case '*':
          result = mathService.mul(firstValue, secondValue);
        break;
        case '/':
          result = mathService.div(firstValue, secondValue);
        break;
        default:
          throw new IllegalArgumentException("Invalid math  operation!");
      }
      return result;
   }

   public static void main(String [] args)throws Exception{
      int port = 10000;
      if (args.length == 1) {
        try {
          port = Integer.parseInt(args[0]);
        }
             catch(Exception e){
             }
      }
      System.out.println("Math Server is running...");
      // create a server socket and wait for client's connection
      ServerSocket serverSocket = new ServerSocket(port);
      Socket socket = serverSocket.accept();
      // run a math server that talks to the client
      MathServer mathServer = new MathServer();
      mathServer.setMathService(new PlainMathService());
      mathServer.setSocket(socket);
      mathServer.execute();
      System.out.println("Math Server is closed...");
  }
}
```

A test client program that can access the math server is shown below:

Program 13.8

```
// MathClient.java: A test client program to access MathServer.
import java.io.*;
import java.net.Socket;
public class MathClient {
   public static void main(String [] args){
      String hostname = "localhost";
      int port = 10000;
      if (args.length != 2) {
        System.out.println("Use the default setting...");
      }
          else {
        hostname = args[0];
        port = Integer.parseInt(args[1]);
      }
      try {
        // create a socket
        Socket socket = new Socket(hostname, port);
        // perform a simple math operation "12+21"
        BufferedWriter writer = new BufferedWriter(
                new OutputStreamWriter(socket.getOutputStream()));
        writer.write("+:12:21");
        writer.newLine();
        writer.flush();
        // get the result from the server
        BufferedReader reader = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
        System.out.println(reader.readLine());
        reader.close();
        writer.close();
      }
          catch (Exception e) {
        e.printStackTrace();
      }
   }
}
```

13.6 URL Encoding

It is very important that the web can be accessed via heterogeneous platforms such as Windows, Linux or Mac. The characters in the URL must come from a fixed subset of ASCII in order to maintain the interoperability between various platforms. Specifically, the capital letters A-Z, the lowercase letters a-z, the digits 0-9 and the punctuation characters. Encoding is very simple, any characters that are not ASCII numerals, letters, or the punctuation marks allowed are converted into bytes and each byte is written as a percentage sign followed by two hexadecimal digits. For example, the following program helps encode a query string if non ASCII characters are present.

Program 13.9

```
// QueryStringFormatter.java: encodes a string with non-ASCII characters.
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;

public class QueryStringFormatter {
   private String queryEngine;
   private StringBuilder query = new StringBuilder();

   public QueryStringFormatter(String queryEngine) {
      this.queryEngine = queryEngine;
   }
   public String getEngine() {
      return this.queryEngine;
   }
   public void addQuery(String queryString, String queryValue)
            throws Exception {
      query.append(queryString+"="+
                  URLEncoder.encode(queryValue,"UTF-8")+"&");
   }
   public String getQueryString(){
      return "?"+query.toString();
   }
   public static void main(String[] args) throws Exception {
      QueryStringFormatter formatter =
            new
QueryStringFormatter("http://www.google.com.au/search");
      formatter.addQuery("newwindow","1");
      formatter.addQuery("q","Xingchen Chu & Rajkumar Buyya");
      System.out.println(formatter.getEngine()+
                        formatter.getQueryString());
   }
}
```

The output of this program is shown as follows:

```
http://www.google.com.au/search?newwindow=1&q=Xingchen+Chu+%26+Rajkumar+Buyya&
```

It can be seen that the whitespace has been encoded as '+' and the '&' has been encoded as '%26' the percentage sign following by its byte value. Other characters remain the same. These conversions have been performed by the `URLEncoder` class,which is part of the Java base class library and provides facilities for encoding strings (urls) in different formats. There is also an `URLDecoder` class that performs the reverse process. These two classes are useful not only for URLs but for managing HTML form data.

13.6.1 Writing and Reading Data via URLConnection

Besides the socket and datagram introduced in the previous section, `java.net` package provides another useful class that can be used to write and read data between the server and the client:

`URLConnetion` It is not possible to instantiate the `URLConnection` class directly; instead you have to create the `URLConnection` via the `URL` object.

```
   URLConnection connection = new URL("www.yahoo.com").openConnection();
```

Then the `URLConnection` class provides the `getInputStream` and `getOutputStream` methods that are very similar to the `getInputStream` and `getOutputStream` provided by the `Socket` class. The following example shows how to use the `URLConnection` and the `URLEncoder` class to send queries to the Yahoo search engine.

Program 13.10

```
// TextBasedSearchEngine.java:
import java.io.*;
import java.net.*;

public class TextBasedSearchEngine {
   private String searchEngine;

   public TextBasedSearchEngine(String searchEngine) {
      this.searchEngine = searchEngine;
   }
   public void doSearch(String queryString) {
     try {
       // open a url connection
       URL url = new URL(searchEngine);
       URLConnection connection = url.openConnection();
       connection.setDoOutput(true);
       // write the query string to the search engine
       PrintStream ps = new PrintStream(connection.getOutputStream());
       ps.println(queryString);
       ps.close();
       // read the result
       DataInputStream input =
               new DataInputStream(connection.getInputStream());
       String inputLine = null;
       while((inputLine = input.readLine())!= null) {
         System.out.println(inputLine);
       }
     }
     catch(Exception e) {
       e.printStackTrace();
     }
   }
   public static void main(String[] args) throws Exception{
      QueryStringFormatter formatter =
                new QueryStringFormatter("http://search.yahoo.com/search");
      formatter.addQuery("newwindow","1");
      formatter.addQuery("q","Xingchen Chu & Rajkumar Buyya");
      // search it via yahoo
      TextBasedSearchEngine search =
          new TextBasedSearchEngine(formatter.getEngine());
      search.doSearch(formatter.getQueryString());
   }
}
```

Output:
The program first creates an encoded query string that can be used by the web application, then it utilises the `URLConnection` class to send/receive data from the Yahoo search engine. The output is the entire html page will be something like below:

```
<!doctype      html      public      "-//W3C/DTD      HTML      4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
…
…
</html>
```

13.7 Summary

Developing network applications is made possible in Java by using sockets, threads, RMI, clustering and Web services. These technologies allow for the creation of portable, efficient, and maintainable large and complex Internet applications. The `java.net` package provides a powerful and flexible set of classes for implementing network applications.

Typically, programs running on client machines make requests to programs on a server machine. These involve networking services provided by the transport layer. The most widely used transport protocols on the Internet are *TCP (Transmission control Protocol)* and *UDP (User Datagram Protocol)*. TCP is a connection-oriented protocol providing a reliable flow of data between two computers. It is used by applications such as the World Wide Web, e-mail, ftp, and secure shell. On the other hand, UDP is a simpler message-based connectionless protocol which sends packets of data known as *datagrams* from one computer to another with no guarantees of arrival. Network applications using UDP include *Domain Name Systems (DNS)*, streaming media applications as IPTV, VoIP, and online games. Both protocols use ports for application-to-application communication. A port is a logical access point represented by a positive 16-bit integer value. Each service offered by a computer is uniquely identified by a port number. Each Internet packet contains both destination IP address and the port to which the request is to be delivered. Some ports are reserved for common services such as *ftp*, *telnet*, *smtp*, *http*, *https*, and *login*. Port numbers >=1024 are generally used for user level services.

Sockets provide an interface for programming networks at the transport layer. Using sockets, network communication is very much similar to performing file I/O. A socket is an endpoint of a two-way communication link between two programs running on the network. The source and destination IP address, and the port numbers constitute a network socket. Two key classes from `java.net` package used in creation of server and client programs are `ServerSocket`, which represents a server socket, and `Socket`, an instantiation of which performs the actual communication with the client. Datagram communication is also supported through `DatagramPacket` and `DatagramSocket` classes. Writing and reading data between server and the client is also supported thru the `URLconnection` class.

13.8 Exercises

Objective Questions

13.1 ______ is a connection-oriented and reliable protocol, ______ is a less reliable protocol.

13.2 The TCP and UDP protocols use ______ to map incoming data to a particular process running on a computer.

13.3 *Datagrams* are normally sent by ______ protocol.

13.4 Java uses ______ class representing a server and ______ class representing the client that uses TCP protocol.

13.5 ______ is used to wait for accepting client connection requests.

13.6 Class ______ is used to create a packet of data for UDP protocol.

13.7 If something goes wrong related to the network, ______ will be thrown when dealing with TCP/UDP programming in Java.

13.8 The main purpose of URL encoding is to maintain the ______ between various platforms.

13.9 Based on the URL encoding mechanism, "www.test.com/test me&test you" becomes ______.

13.10 ______ method is used to instantiate a URLConnection instance.

13.11 UDP is more reliable than TCP protocol: True or False

13.12 The same port number can be reused for many times when binding with sockets simultaneously: True or False

13.13 In order to create a client socket connecting to a particular server, the IP address must be given to the client socket, otherwise, it cannot connect to the server: True or False.

13.14 Sockets provide an interface for programming networks at the transport layer: True or False

13.15 Call Socket.close() method will close the TCP server that socket connects to: True or False

13.16 The socket instance does not need to be explicitly closed to release all the resources it occupies, as the Java Garbage Collection mechanism will do it automatically: True or False

13.17 The following line of code

```
Socket socket = new Socket("localhost", 1254);
```

will create a TCP server at localhost port 1254: True or False

13.18 Java TCP Socket uses the InputStream/OutputStream to read/write data to the network channel: True or False

13.19 Java UDP sends/receives data as raw Java primitive types: True or False

13.20 URL Encoding is useful for translating Chinese to English that presents in a particular URL: True or False

Review Questions

13.21 Explain the client-server architecture via a simple example.

13.22 Explain the TCP/IP stacks. Briefly explain the TCP and UDP protocols and the difference between the two protocols.

13.23 What is port? List some well-known ports and explain the applications associated with them.