Chapter 12

Network Programming

What is in This Chapter ?

This chapter explains how to connect your JAVA application to a network. You will learn how to read files from over the internet as well as have two or more programs communicate with one another over a network connection (wired or wireless). You will learn about **Uniform Resource Locators** as well as **Client/Server** communications using **TCP** and **Datagram Sockets**.

12.1 Networking Basics

Network Programming *involves writing programs that communicate with other programs across a computer network.*

There are many issues that arise when doing network programming which do not appear when doing single program applications. However, JAVA makes networking applications simple due to the easy-to-use libraries. In general, applications that have components running on different machines are known as ***distributed*** applications ... and usually they consist of client/server relationships.

A ***server*** *is an application that provides a "service" to various* ***clients*** *who request the service.*

There are many client/server scenarios in real life:

- Bank tellers (server) provide a service for the account owners (client)
- Waitresses (server) provide a service for customers (client)
- Travel agents (server) provide a service for people wishing to go on vacation (client)

In some cases, servers themselves may become clients at various times.

- E.g., travel agents will become clients when they phone the airline to make a reservation or contact a hotel to book a room.

In the general networking scenario, everybody can either be a client or a server at any time. This is known as ***peer-to-peer*** computing. In terms of writing java applications it is similar to having many applications communicating among one another.

- E.g., the original **Napster** worked this way. Thousands of people all acted as clients (trying to download songs from another person) as well as servers (in that they allowed others to download their songs).

There are many different strategies for allowing communication between applications. JAVA technology allows:

- internet clients to connect to servlets or back-end business systems (or databases).
- applications to connect to one another using sockets.
- applications to connect to one another using RMI (remote method invocation).
- some others

We will look at the simplest strategy of connecting applications using sockets.

A ***Protocol*** *is a standard pattern of exchanging information.*

It is like a set of rules/steps for communication. The simplest example of a protocol is a phone conversation:

1. **JIM** dials a phone number
2. **MARY** says "Hello..."
3. **JIM** says "Hello..."
4. ... the conversation goes on for a while ...
5. **JIM** says "Goodbye"
6. **MARY** says "Goodbye"

Perhaps another person gets involved:

1. **JIM** dials a phone number
2. **MARY** says "Hello..."
3. **JIM** says "Hello" and perhaps asks to speak to **FRED**
4. **MARY** says "Just a minute"
5. **FRED** says "Hello..."
6. **JIM** says "Hello..."
7. ... the conversation goes on for a while ...
8. **JIM** says "Goodbye"
9. **FRED** says "Goodbye"

Either way, there is an "expected" set of steps or responses involved during the initiation and conclusion of the conversation. If these steps are not followed, confusion occurs (like when you phone someone and they pick up the phone but do not say anything).

Computer protocols are similar in that a certain amount of "*handshaking*" goes on to establish a valid connection between two machines. Just as we know that there are different ways to shake hands, there are also different protocols. There are actually layered levels of protocols in that some low level layers deal with how to transfer the data bits, others deal with more higher-level issues such as "where to send the data to".

Computers running on the internet typically use one of the following high-level **Application Layer** protocols to allow applications to communicate:

- **Hyper Text Transfer Protocol** (HTTP)
- **File Transfer Protocol** (FTP)
- **Telnet**

This is analogous to having multiple strategies for communicating with someone (in person, by phone, through electronic means, by post office mail etc...).

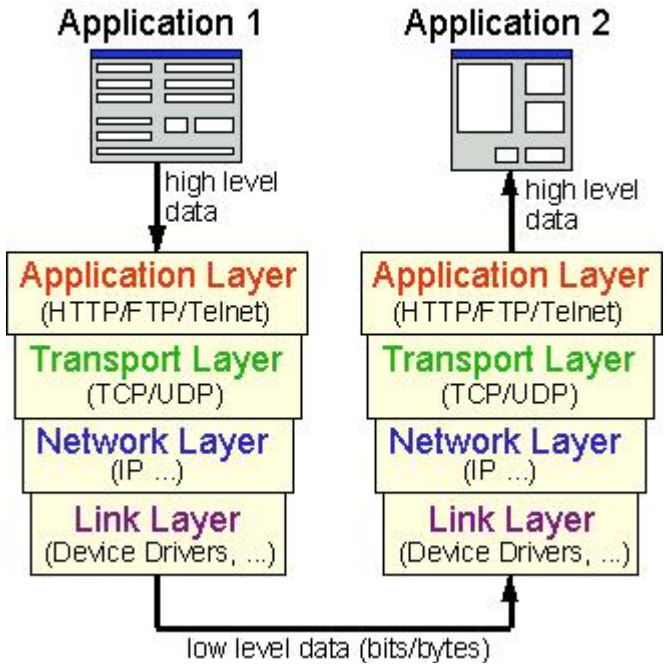
In a lower **Transport Layer** of communication, there is a separate protocol which is used to determine how the data is to be transported from one machine to another:

- **Transport Control Protocol** (TCP)
- **User Datagram Protocol** (UDP)

This is analogous to having multiple ways of actually delivering a package to someone (Email, Fax, UPS, Fed-Ex etc...)

Beneath that layer is a **Network Layer** for determining how to locate destinations for the data (i.e., address). And at the lowest level (for computers) there is a **Link Layer** which actually handles the transferring of bits/bytes.

So, internet communication is built of several layers:

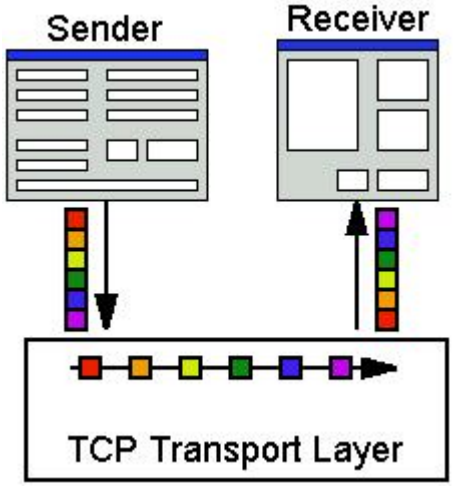


When you write JAVA applications that communicate over a network, you are programming in the **Application Layer**.

JAVA allows two types of communication via two main types of **Transport Layer** protocols:

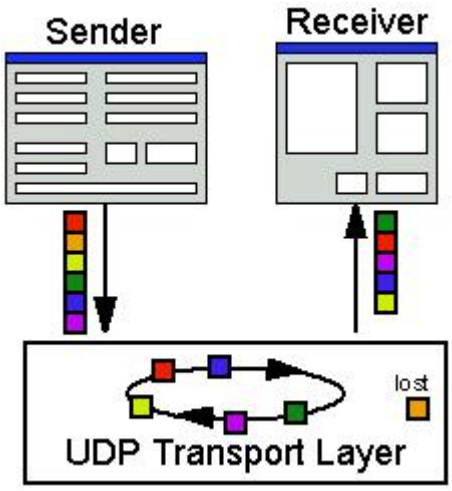
TCP

- a ***connection-based*** protocol that provides a reliable flow of data between two computers.
- guarantees that data sent from one end of the connection actually gets to the other end and in the same order
  - similar to a phone call. Your words come out in the order that you say them.
- provides a point-to-point channel for applications that require **reliable communications**.
- **slow overhead time** of setting up an end-to-end connection.

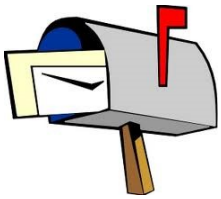


UDP

- a protocol that sends independent packets of data, called ***datagrams***, from one computer to another.
- no guarantees about arrival. UDP is not connection-based like TCP.
- provides **communication that is not guaranteed** between the two ends
  - sending packets is like sending a letter through the postal service
  - the order of delivery is not important and not guaranteed
  - each message is independent of any other
- **faster** since no overhead of setting up end-to-end connection
- many firewalls and routers have been configured NOT TO allow UDP packets.



Why would anyone want to use UDP protocol if information may get lost ? Well, why do we use email or the post office ? We are never guaranteed that our mail will make it to the person that we send it to, yet we still rely on those delivery services. It may still be quicker than trying to contact a person via phone to convey the data (i.e., like a TCP protocol).



One more important definition we need to understand is that of a *port*:

> *A **port** is used as a gateway or "entry point" into an application.*

Although a computer usually has a single physical connection to the network, data sent by different applications or delivered to them do so through the use of ports configured on the same physical network connection. When data is to be transmitted over the internet to an application, it requires that we specify the address of the destination computer as well as the application's port number. A computer's address is a 32-bit IP address. The port number is a 16-bit number ranging from 0 to 65,535, with ports 0-1023 restricted by well-known applications like HTTP and FTP.

12.2 Reading Files From the Internet (URLs)

*A **Uniform Resource Locator** (i.e., **URL**) is a reference (or address) to a resource over a network (e.g., on the Internet).*

So, a URL can be used to represent the "location" of a webpage or web-based application. A URL is really just a String that represents the name of a resource ... which can be files, databases, applications, etc.. A resource name consists of a host machine name, filename, port number, and other information. It may also specify a ***protocol identifier*** (e.g., http, ftp) Here are some examples of URLs:

```
http://www.cnn.com/
http://www.apple.com/ipad/index.html
http://en.wikipedia.org/wiki/Computer_science
```

Here, **http://** is the protocol identifier which indicates the protocol that will be used to obtain the resource. The remaining part is the ***resource name***, and its format depends on the protocol used to access it.

A URL resource name may generally contain:

- **a Host Name -** The name of the machine on which the resource lives.
  http://**www.apple.com**:80/ipad/index.html
- **a Port # (optional) -** The port number to which to connect.
  http://www.apple.com**:80**/ipad/index.html
- **a Filename -** The pathname to the file on the machine.
  http://www.apple.com:80**/ipad/index.html**

In JAVA, there is a **URL** class defined in the **java.net** package. We can create our own URL objects as follows:

```
URL webPage = new URL("http://www.apple.com/ipad/index.html");
```

JAVA will "dissect" the given String in order to obtain information about protocol, hostName, file etc....

Due to this, JAVA may throw a **MalformedURLException** ... so we will need to do this:

```
try {
    URL webPage = new URL("http://www.apple.com/ipad/index.html");
} catch(MalformedURLException e) {
    ...
}
```

Another way to create a URL is to break it into its various components:

```
try {
    URL webPage = new URL("http","www.apple.com",80,"/ipad/index.html");
} catch(MalformedURLException e) {
    ...
}
```

If you take a look at the JAVA API, you will notice some other constructors as well.

The URL class also supplies methods for extracting the parts (protocol, host, file, port and reference) of a URL object. Here is an example that demonstrates what can be accessed. Note that this example only manipulates a URL object, it does not go off to grab any web pages:

```
import java.net.*;

public class URLTestProgram {
    public static void main(String[] args) {
        URL webpage = null;
        try {
            webpage = new URL("http", "www.apple.com", 80, "/ipad/index.html");
        } catch(MalformedURLException e) {
            e.printStackTrace();
        }
        System.out.println(webpage);
        System.out.println("protocol = " + webpage.getProtocol());
        System.out.println("host = " + webpage.getHost());
        System.out.println("filename = " + webpage.getFile());
        System.out.println("port = " + webpage.getPort());
        System.out.println("ref = " + webpage.getRef());
    }
}
```

Here is the output:

```
http://www.apple.com:80/ipad/index.html
protocol = http
host = www.apple.com
filename = /ipad/index.html
port = 80
ref = null
```

After creating a URL object, you can actually connect to that webpage and read the contents of the URL by using its **openStream()** method which returns an **InputStream**. You actually read from the webpage as if it were a simple text file. If an attempt is made to read from a URL that does not exist, JAVA will throw an **UnknownHostException**

Example:

Here is an example that reads a URL directly. It actually reads the file on wikipedia and displays it line by line to the console. Notice that it reads the file as a text file, so we simply get the HTML code. Also, you must be connected to the internet to run this code:

```
import java.net.*;
import java.io.*;

public class URLReaderProgram {
    public static void main(String[] args) {
        URL wiki = null;
        try {
            wiki = new URL("https://en.wikipedia.org/wiki/Computer_science");
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(wiki.openStream()));
            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);
            in.close(); // Close the connection to the net
        } catch(MalformedURLException e) {
            System.out.println("Cannot find webpage " + wiki);
        } catch(IOException e) {
            System.out.println("Cannot read from webpage " + wiki);
        }
    }
}
```

The output should look something like this, assuming you could connect to the webpage:

```
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Computer science - Wikipedia</title>
<script>document.documentElement.className =
document.documentElement.className.replace( /(^|\s)client-nojs(\s|$)/, "$1client-js$2"
);</script>
<script>(window.RLQ=window.RLQ||[]).push(function(){mw.config.set({"wgCanonicalNamespa
ce":"","wgCanonicalSpecialPageName":false,"wgNamespaceNumber":0,"wgPageName":"Computer
_science","wgTitle":"Computer
science","wgCurRevisionId":771021423,"wgRevisionId":771021423,"wgArticleId":5323,"wgIs
Article":true,"wgIsRedirect":false,"wgAction":"view","wgUserName":null,"wgUserGroups":
["*"],"wgCategories":["Pages using web citations with no URL","Pages using citations
with accessdate and no URL","Webarchive template wayback links","Pages using ISBN
magic links","Wikipedia pages semi-protected against vandalism","Wikipedia
...
```

Example:

Here is a modification to the above example that reads the URL by making a **URLConnection** first. Since the tasks of opening a connection to a webpage and reading the contents may both generate an **IOException**, we cannot distinguish the kind of error that occurred. By trying to establish the connection first, if any **IOExceptions** occur, we know they are due to a connection problem. Once the connection has been established, then any further **IOException** errors would be due to the reading of the webpage data.

```
import java.net.*;
import java.io.*;

public class URLConnectionReaderExample {
    public static void main(String[] args) {
        URL wiki = null;
        BufferedReader in = null;
        try {
            wiki = new URL("https://en.wikipedia.org/wiki/Computer_science");
        } catch(MalformedURLException e) {
            System.out.println("Cannot find webpage " + wiki);
            System.exit(-1);
        }
        try {
            URLConnection  aConnection = wiki.openConnection();
            in = new BufferedReader(
                    new InputStreamReader(aConnection.getInputStream()));
        }
        catch (IOException e) {
            System.out.println("Cannot connect to webpage " + wiki);
            System.exit(-1);
        }
        try {
            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);
            in.close(); // Close the connection to the net
        } catch(IOException e) {
            System.out.println("Cannot read from webpage " + wiki);
        }
    }
}
```

12.3 Client/Server Communications

Many companies today sell services or products. In addition, there are a large number of companies turning towards E-business solutions and various kinds of web-server/database technologies that allow them to conduct business over the internet as well as over other networks. Such applications usually represent a client/server scenario in which one or more servers serve multiple clients.

*A **server** is any application that provides a service and allows clients to communicate with it.*

Such services may provide:

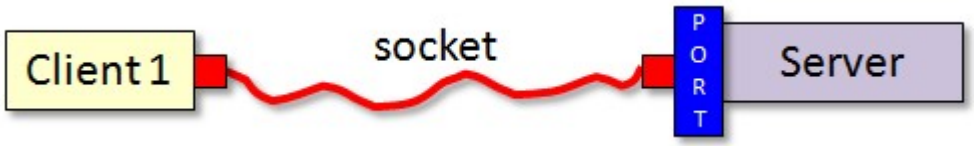
- a recent stock quote
- transactions for bank accounts
- an ability to order products
- an ability to make reservations
- a way to allow multiple clients to interact (Auction)

*A **client** is any application that requests a service from a server.*

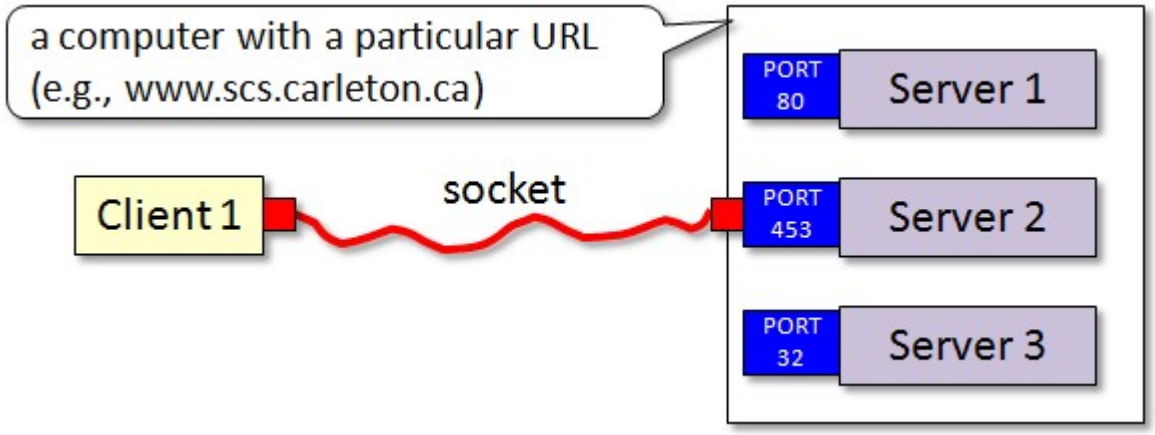
The client typically "uses" the service and then displays results to the user. Normally, communication between the client and server must be reliable (no data can be dropped or missing):

- stock quotes must be accurate and timely
- banking transactions must be accurate and stable
- reservations/orders must be acknowledged

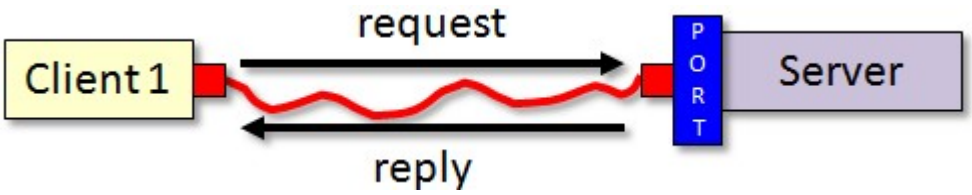
The TCP protocol, mentioned earlier, provides reliable point-to-point communication. Using TCP the client and server must establish a connection in order to communicate. To do this, each program binds a ***socket*** to its end of the connection. A ***socket*** is one endpoint of a two-way communication link between 2 programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application to which the data is to be sent. It is similar to the idea of plugging the two together with a cable.



The ***port number*** is used as the server's location on the machine that the server application is running. So if a computer is running many different server applications on the same physical machine, the port number uniquely identifies the particular server that the client wishes to communicate with:



The client and server may then each read and write to the socket bound to its end of the connection.



In JAVA, the server application uses a **ServerSocket** object to wait for client connection requests. When you create a **ServerSocket**, you must specify a port number (an **int**). It is possible that the server cannot set up a socket and so we have to expect a possible **IOException**. Here is an example:

```
public static int SERVER_PORT = 5000;

ServerSocket serverSocket;
try {
     serverSocket  = new ServerSocket(SERVER_PORT);
}
catch(IOException e) {
     System.out.println("Cannot open server connection");
}
```

The server can communicate with only one client at a time.

The server waits for an incoming client request through the use of the **accept()** message:

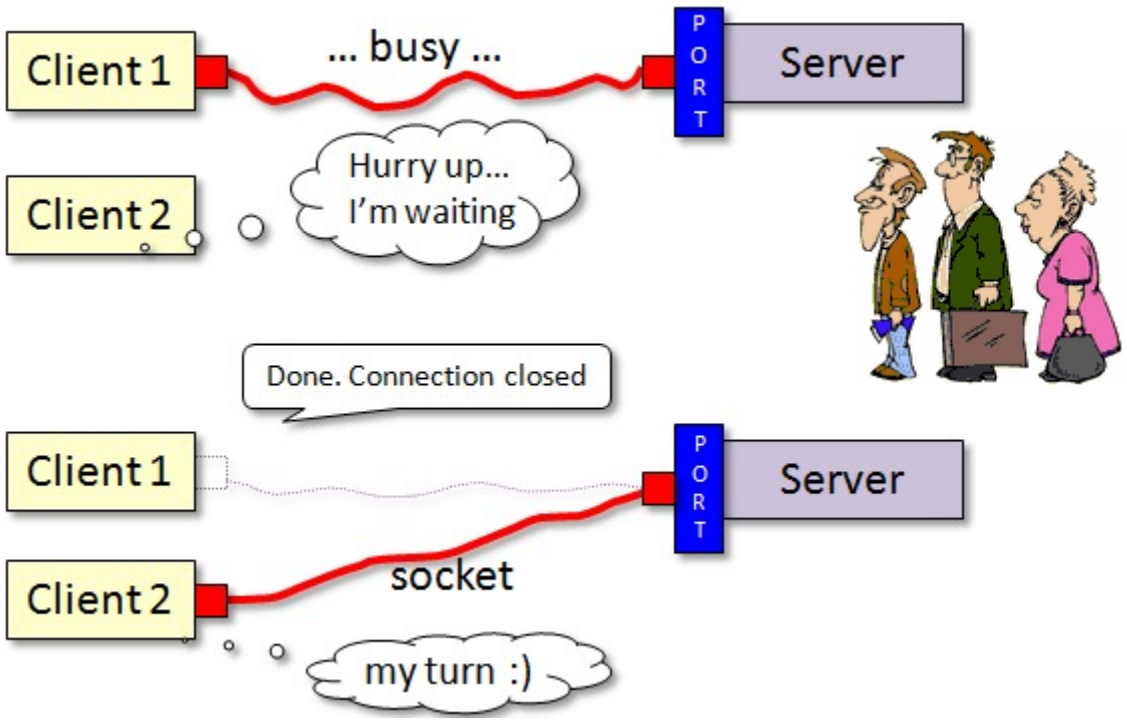
```
Socket aClientSocket;

try {
    aClientSocket = serverSocket.accept();
}
catch(IOException e) {
    System.out.println("Cannot connect to client");
}
```

When the **accept()** method is called, the server program actually waits (i.e., ***blocks***) until a client becomes available (i.e., an incoming client request arrives). Then it creates and returns a **Socket** object through which communication takes place.

Once the client and server have completed their interaction, the socket is then closed:

```
aClientSocket.close();
```

Only then may the next client open a socket connection to the server. So, remember ... if one client has a connection, everybody else has to wait until they are done:



So how does the client connect to the server ? Well, the client must know the address of the server as well as the port number. The server's address is stored as an **InetAddress** object which represents any IP address (i.e., an internet address, an ftp site, local machine etc,...).

If the server and client are on the same machine, the static method **getLocatHost()** in the **InetAddress** class may be used to get an address representing the local machine as follows:

```
public static int SERVER_PORT = 5000;
try {
     InetAddress address = InetAddress.getLocalHost();
     Socket socket = new Socket(address, SERVER_PORT);
}
catch(UnknownHostException e) {
     System.out.println("Host Unknown");
}
catch(IOException e) {
     System.out.println("Cannot connect to server");
}
```

Once again, a socket object is returned which can then be used for communication.
Here is an example of what a local host may look like:

```
cr850205-a/169.254.180.32
```

The **getLocalHost()** method may, however, generate an **UnknownHostException**.
You can also make an **InetAddress** object by specifying the network IP address directly or the machine name directly as follows:

```
InetAddress.getByName("169.254.1.61");
InetAddress.getByName("www.scs.carleton.ca");
```

So how do we actually do communication between the client and the server ? Well, each socket has an **inputStream** and an **outputStream**. So, once we have the sockets, we simply ask for these streams … and then reading and writing may occur.

```
try {
    InputStream   in = socket.getInputStream();
    OutputStream  out = socket.getOutputStream();
}
catch(IOException e) {
    System.out.println("Cannot open I/O Streams");
}
```

Normally, however, we actually wrap these input/output streams with text-based, datatype-based or object-based wrappers:

```
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
PrintWriter out = new PrintWriter(socket.getOutputStream());

DataInputStream in = new DataInputStream(socket.getInputStream());
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
```

You may look back at the notes on file I/O to see how to write to the streams. However, one more point ... when data is sent through the output stream, the **flush()** method should be sent to the output stream so that the data is not buffered, but actually sent right away.

Also, you must be careful when using **ObjectInputStreams** and **ObjectOutputStreams**. When you create an **ObjectInputStream**, it blocks while it tries to read a header from the underlying **SocketInputStream**. When you create the corresponding **ObjectOutputStream** at the far end, it writes the header that the **ObjectInputStream** is waiting for, and both are able to continue. If you try to create both **ObjectInputStreams** first, each end of the connection is waiting for the other to complete before proceeding which results in a deadlock situation (i.e., the programs seems to hang/halt). This behavior is described in the API documentation for the **ObjectInputStream** and **ObjectOutputStream** constructors.

Example:

Let us now take a look at a real example. In this example, a client will attempt to:

1. connect to a server
2. ask the server for the current time
3. ask the server for the number of requests that the server has handled so far
4. ask the server for an invalid request (i.e., for a pizza)

Here is the server application. It runs forever, continually waiting for incoming client requests:

```
import java.net.*;   // all socket stuff is in here
import java.io.*;

public class Server {
    public static int SERVER_PORT = 5000;  // arbitrary, but above 1023
    private int counter = 0;

    // Helper method to get the ServerSocket started
    private ServerSocket goOnline() {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(SERVER_PORT);
            System.out.println("SERVER online");
        } catch (IOException e) {
            System.out.println("SERVER: Error creating network connection");
        }
        return serverSocket;
    }

    // Handle all requests
    private void handleRequests(ServerSocket serverSocket) {
        while(true) {
            Socket socket = null;
            BufferedReader in = null;
            PrintWriter out = null;

            try {
                // Wait for an incoming client request
                socket = serverSocket.accept();
                // At this point, a client connection has been made
                in = new BufferedReader(new InputStreamReader(
                                                    socket.getInputStream()));
                out = new PrintWriter(socket.getOutputStream());
```

```
            } catch(IOException e) {
                System.out.println("SERVER: Error connecting to client");
                System.exit(-1);
            }
            // Read in the client's request
            try {
                String request = in.readLine();
                System.out.println("SERVER: Client Message Received: " + request);
                if (request.equals("What Time is It ?")) {
                    out.println(new java.util.Date());
                    counter++;
                }
                else if (request.equals("How many requests have you handled ?"))
                    out.println(counter++);
                else
                    System.out.println("SERVER: Unknown request: " + request);

                out.flush();      // Now make sure that the response is sent
                socket.close();    // We are done with the client's request

            } catch(IOException e) {
                System.out.println("SERVER: Error communicating with client");
            }
        }
    }

    public static void main (String[] args) {
        Server s = new Server();
        ServerSocket ss = s.goOnline();
        if (ss != null)
            s.handleRequests(ss);
    }
}
```

Here is the client application:

```
import java.net.*;
import java.io.*;

public class ClientProgram {
    private Socket          socket;
    private BufferedReader  in;
    private PrintWriter     out;

    // Make a connection to the server
    private void connectToServer() {
        try {
            socket = new Socket(InetAddress.getLocalHost(), Server.SERVER_PORT);

            in = new BufferedReader(new InputStreamReader(
                                                    socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream());
        } catch(IOException e) {
            System.out.println("CLIENT: Cannot connect to server");
            System.exit(-1);
        }
    }
```

```
    // Disconnect from the server
    private void disconnectFromServer() {
        try {
            socket.close();
        } catch(IOException e) {
            System.out.println("CLIENT: Cannot disconnect from server");
        }
    }

    // Ask the server for the current time
    private void askForTime() {
        connectToServer();
        out.println("What Time is It ?");
        out.flush();
        try {
            String time = in.readLine();
            System.out.println("CLIENT: The time is " + time);
        } catch(IOException e) {
            System.out.println("CLIENT: Cannot receive time from server");
        }
        disconnectFromServer();
    }

    // Ask the server for the number of requests obtained
    private void askForNumberOfRequests() {
        connectToServer();
        out.println("How many requests have you handled ?");
        out.flush();
        int count = 0;
        try {
            count = Integer.parseInt(in.readLine());
        } catch(IOException e) {
            System.out.println("CLIENT: Cannot receive num requests from server");
        }
        System.out.println("CLIENT: The number of requests are " + count);
        disconnectFromServer();
    }

    // Ask the server to order a pizza
    private void askForAPizza() {
        connectToServer();
        out.println("Give me a pizza");
        out.flush();
        disconnectFromServer();
    }

    private static void Delay() {
        try{Thread.sleep(3000);}catch(InterruptedException e){}
    }

    public static void main (String[] args) {
        ClientProgram c = new ClientProgram();
        Delay(); c.askForTime();
        Delay(); c.askForNumberOfRequests();
        Delay(); c.askForAPizza();
        Delay(); c.askForTime();
        Delay(); c.askForNumberOfRequests();
    }
}
```

12.4 Datagram Sockets

Recall that with the datagram protocol (i.e., UDP) there is no direct socket connection between the client and the server. That is, packets are received "in seemingly random order" from different clients. It is similar to the way email works. If the client requests or server responses are too big, they are broken up into multiple packets and sent one packet at a time. The server is not guaranteed to receive the packets all at once, nor in the same order, nor is it guaranteed to receive all the packets !!

Let us look at the same client-server application, but by now using **DatagramSockets** and **DatagramPackets**. Once again, the server will be in a infinite loop accepting messages, although there will be no direct socket connection to the client. We will be setting up a ***buffer*** (i.e., an array of bytes) which will be used to receive incoming requests.

Each message is sent as a ***packet***. Each packet contains:

- the ***data*** of the message (i.e., the message itself)
- the ***length*** of the message (i.e., the number of bytes)
- the ***address*** of the sender (as an InetAddress)
- the ***port*** of the sender

The code for packaging and sending an outgoing packet involves creating a **DatagramSocket** and then constructing a **DatagramPacket**. The packet requires an array of bytes, as well as the address and port in which to send to. A byte array can be obtained from most objects by sending a **getBytes()** message to the object. Finally, a **send()** message is used to send the packet:

```
byte[]          sendBuffer;
DatagramSocket  socket;
DatagramPacket  packetToSend ;

socket = new DatagramSocket();
sendBuffer = "This is the data ... need not be a String".getBytes();
packetToSend = new DatagramPacket(sendBuffer, sendBuffer.length,
                                  anInetAddress, aPort);
socket.send(packetToSend);
```

The server code for receiving an incoming packet involves allocating space (i.e., a byte array) for the **DatagramPacket** and then receiving it. The code looks as follows:

```
byte[]          recieveBuffer;
DatagramPacket  receivePacket;

recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
receivePacket = new DatagramPacket(recieveBuffer,
                                   recieveBuffer.length);
socket.receive(receivePacket);
```

We then need to extract the data from the packet. We can get the address and port of the

sender as well as the data itself from the packet as follows:

```
InetAddress  sendersAddress = receivePacket.getAddress();
int          sendersPort = receivePacket.getPort();
String       sendersData = new String(receivePacket.getData(), 0,
                                      receivePacket.getLength());
```

In this case the data sent was a **String**, although it may in general be any object. By using the sender's address and port, whoever receives the packet can send back a reply.

Example:

Here is a modified version of our client/server code ... now using the **DatagramPackets**:

```
import java.net.*;
import java.io.*;

public class PacketServer {
    public  static int SERVER_PORT = 5000;
    private static int INPUT_BUFFER_LIMIT = 500;
    private int counter = 0;

    // Helper method to get the DatagramSocket started
    private DatagramSocket goOnline() {
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket(SERVER_PORT);
            System.out.println("SERVER online");
        } catch (SocketException e) {
            System.out.println("SERVER: no network connection");
            System.exit(-1);
        }
        return socket;
    }

    // Handle all requests
    private void handleRequests(DatagramSocket socket) {
        while(true) {
            try {
                // Wait for an incoming client request
                byte[] recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
                DatagramPacket receivePacket;
                receivePacket = new DatagramPacket(recieveBuffer,
                                                   recieveBuffer.length);
                socket.receive(receivePacket);

                // Extract the packet data that contains the request
                InetAddress address = receivePacket.getAddress();
                int clientPort = receivePacket.getPort();
                String request = new String(receivePacket.getData(), 0,
                                            receivePacket.getLength());
                System.out.println("SERVER: Packet received: \"" + request +
                                   "\" from " + address + ":" + clientPort);
```

```
                // Decide what should be sent back to the client
                byte[] sendBuffer;
                if (request.equals("What Time is It ?")) {
                    System.out.println("SERVER: sending packet with time info");
                    sendResponse(socket, address, clientPort,
                                 new java.util.Date().toString().getBytes());
                    counter++;
                }
                else if (request.equals("How many requests have you handled ?")) {
                    System.out.println("SERVER: sending packet with num requests");
                    sendResponse(socket, address, clientPort,
                                 ("" + ++counter).getBytes());
                }
                else
                    System.out.println("SERVER: Unknown request: " + request);
            } catch(IOException e) {
                System.out.println("SERVER: Error receiving client requests");
            }
        }
    }

    // This helper method sends a given response back to the client
    private void sendResponse(DatagramSocket socket, InetAddress address,
                              int clientPort, byte[] response) {
        try {
            // Now create a packet to contain the response and send it
            DatagramPacket sendPacket = new DatagramPacket(response,
                                            response.length, address, clientPort);
            socket.send(sendPacket);
        } catch (IOException e) {
            System.out.println("SERVER: Error sending response to client");
        }
    }

    public static void main (String args[]) {
        PacketServer s = new PacketServer();
        DatagramSocket ds = s.goOnline();
        if (ds != null)
            s.handleRequests(ds);
    }
}
```

Notice that only one **DatagramSocket** is used, but that a new **DatagramPacket** object is created for each incoming message. Now let us look at the client:

```
import java.net.*;
import java.io.*;

public class PacketClientProgram {
    private static int  INPUT_BUFFER_LIMIT = 500;
    private InetAddress localHost;

    public PacketClientProgram() {
        try {
            localHost = InetAddress.getLocalHost();
        } catch(UnknownHostException e) {
            System.out.println("CLIENT: Error connecting to network");
            System.exit(-1);
        }
    }

    // Ask the server for the current time
    private void askForTime() {
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();
            byte[] sendBuffer = "What Time is It ?".getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                                    sendBuffer.length, localHost,
                                                    PacketServer.SERVER_PORT);
            System.out.println("CLIENT: Sending time request to server");
            socket.send(sendPacket);
        } catch(IOException e) {
            System.out.println("CLIENT: Error sending time request to server");
        }
        try {
            byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
            DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
                    receiveBuffer.length);
            socket.receive(receivePacket);
            System.out.println("CLIENT: The time is " + new String(
                              receivePacket.getData(), 0,
                              receivePacket.getLength()));
        } catch(IOException e) {
            System.out.println("CLIENT: Cannot receive time from server");
        }
        socket.close();
    }

    // Ask the server for the number of requests obtained
    private void askForNumberOfRequests() {
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();
            byte[] sendBuffer = "How many requests have you handled ?".getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                                    sendBuffer.length, localHost,
                                                    PacketServer.SERVER_PORT);
            System.out.println("CLIENT: Sending request count request to server");
            socket.send(sendPacket);
        } catch(IOException e) {
            System.out.println("CLIENT: Error sending request to server");
        }
```

```
        try {
            byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
            DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
                                                              receiveBuffer.length);
            socket.receive(receivePacket);

            System.out.println("CLIENT: The number of requests are " +
                               new String(receivePacket.getData(), 0,
                                          receivePacket.getLength()));
        } catch(IOException e) {
            System.out.println("CLIENT: Cannot receive num requests from server");
        }
        socket.close();
    }

    // Ask the server to order a pizza
    private void askForAPizza() {
        try {
            byte[] sendBuffer = "Give me a pizza".getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                                           sendBuffer.length, localHost,
                                                           PacketServer.SERVER_PORT);
            DatagramSocket socket = new DatagramSocket();
            System.out.println("CLIENT: Sending pizza request to server");
            socket.send(sendPacket);
            socket.close();
        } catch(IOException e) {
            System.out.println("CLIENT: Error sending request to server");
        }
    }

    private static void Delay() {
        try{Thread.sleep(3000);}catch(InterruptedException e){}
    }

    public static void main (String[] args) {
        PacketClientProgram c = new PacketClientProgram();
        Delay(); c.askForTime();
        Delay(); c.askForNumberOfRequests();
        Delay(); c.askForAPizza();
        Delay(); c.askForTime();
        Delay(); c.askForNumberOfRequests();
    }
}
```