# 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...).

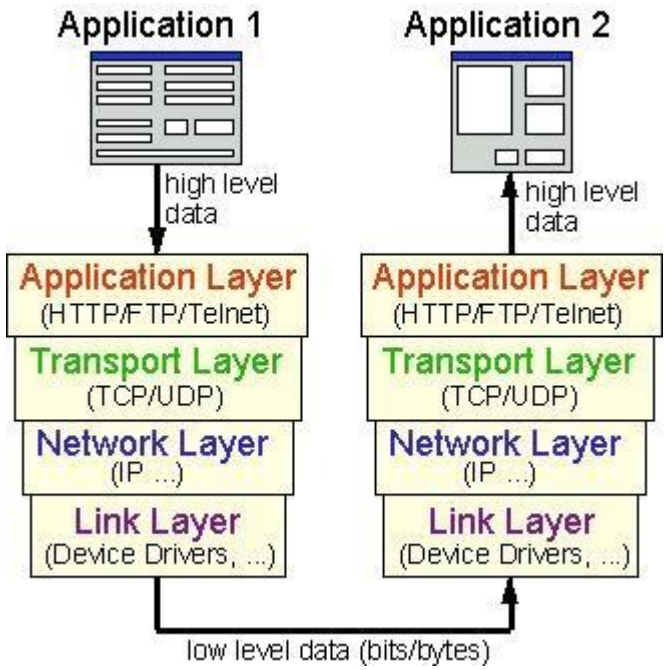
In a lower **Transport Layer** of communication, there is a separate protocol which is used to determine how the data is to be transported from one machine to another:

- **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:

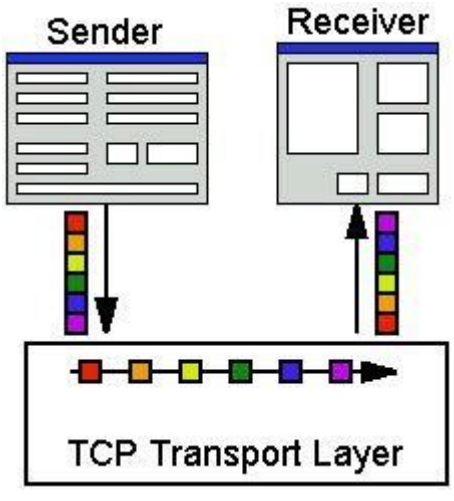



When you write JAVA applications that communicate over a network, you are programming in the **Application Layer**.

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

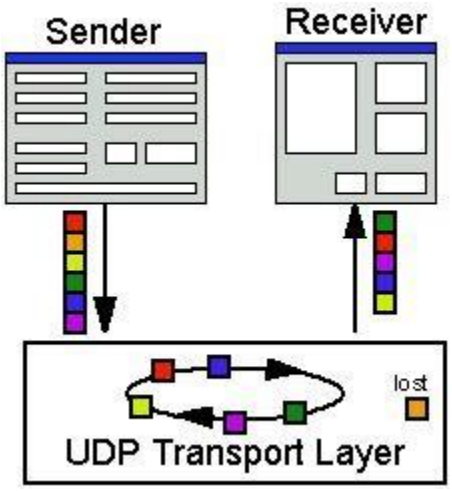
## TCP

- 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.

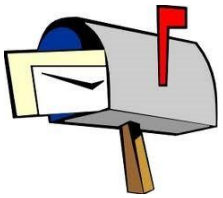


## UDP

- 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.

## 12.3 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)

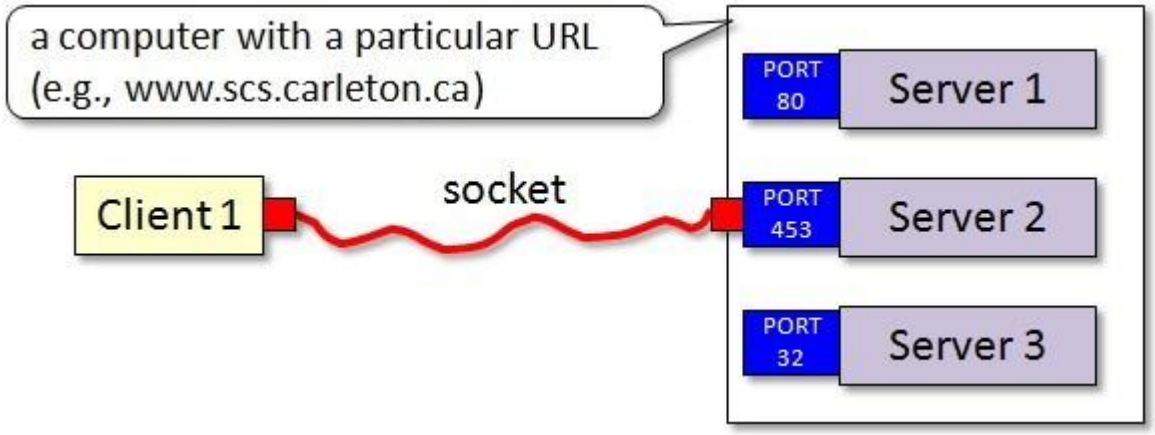
*A **client** is any application that requests a service from a server*.

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):

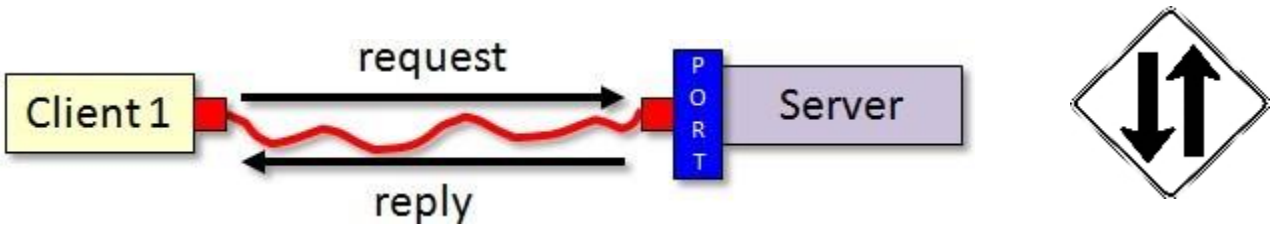
- stock quotes must be accurate and timely
- banking transactions must be accurate and stable
- reservations/orders must be acknowledged

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:

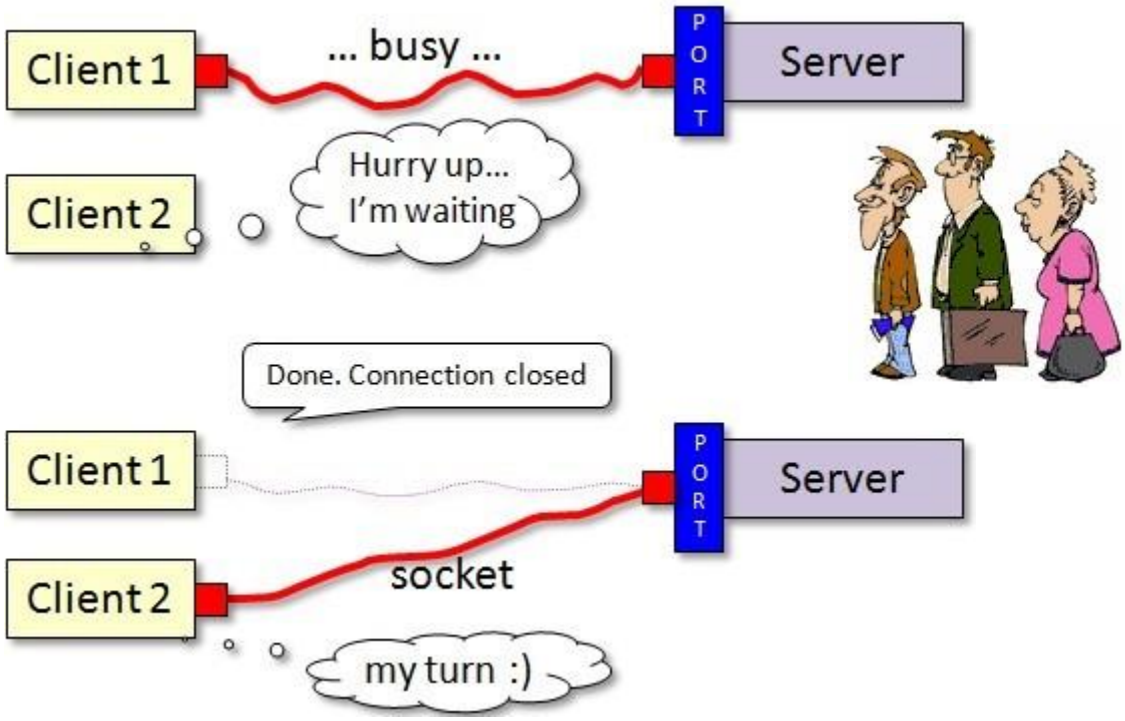
```
Socket aClientSocket;

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.