Networking Programming

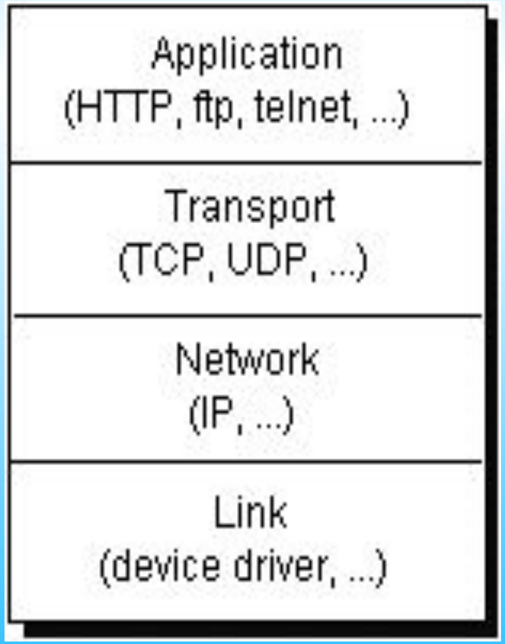
Session

Of

Advance Java

Networking Basics

- Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP), as this diagram illustrates:
- Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the java.net package. These classes provide system-independent network communication
- To decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.



TCP

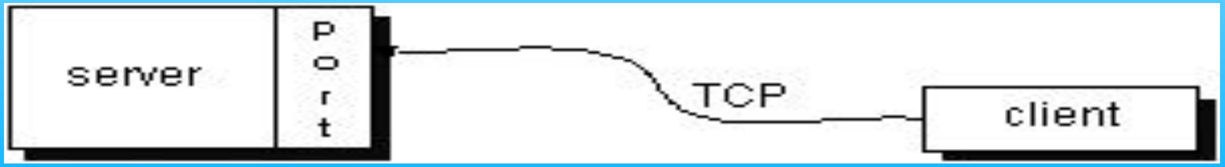
- TCP (Transmission Control Protocol) is a connection-based protocol that provides a reliable flow of data between two computers
- When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection
- TCP provides a point-to-point channel for applications that require reliable communications.
  - ✔ The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel.
- The order in which the data is sent and received over the network is critical to the success of these applications.

UDP

- UDP (User Datagram Protocol) is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival. UDP is not connection-based like TCP
  - ✔ Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other
- Another example of a service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network.

Ports

- Generally, A computer has a single physical connection to the network.
  - ✔ All data destined for a particular computer arrives through that connection.However, the data may be intended for different applications running on the computer. So the computer use **ports** to forward the data to corespond application
- The computer is identified by its 32-bit IP address, whereas ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.
  - ✔ The port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called well-known ports.



*

Java Network Classes

- Through the classes in java.net, Java programs can use TCP or UDP to communicate over the Internet
- Socket : TCP endpoint (a “telephone”)
- ServerSocket : TCP endpoint (“telephone company”)
- DatagramSocket: UDP endpoint (a “mailbox”)
- DatagramPacket: UDP endpoint (a “letter”)
- URL :Uniform Resource Locator (an “address”)
- URLConnection : Connection to web object

Networking Introduction

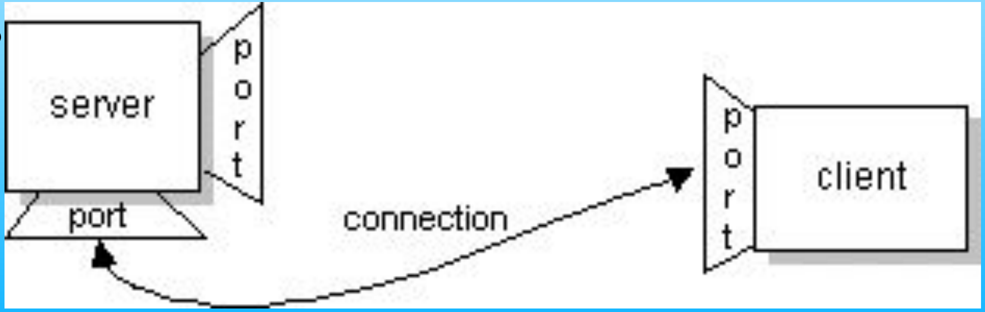
- There are four major technologies for building client/server applications in Java.
  - ✔ Java Sockets, the backbone of all networked communication
  - ✔ Java Remote Method Invocation (RMI), Java-to-Java client/server systems
  - ✔ JDBC, Java Database Connectivity to relational databases (already examined)
  - ✔ Java Interface Definition Language (IDL), the Java binding to the CORBA standard.
- In this session we look at Java Sockets

Sockets Introduction

- To communicate over network, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.
- A socket is one endpoint of a two-way communication link between two programs running on the network.
- A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.
- In Java, each program reads from and writes to a socket in much the same way that you open, read, write to and close a file.

Sockets Introduction

- Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request
- On the client-side: The client knows the hostname of the machine on which the server is running and the port number to which the server is connected.
- If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket so that it can continue to listen to the original socket for connection requests
- The client and server can now communicate by writing to or reading from their sockets



Sockets Introduction

- There are two types of sockets in Java
  - ✔ One is analogous to the telephone connecting to a service. In Java, this is a connection oriented service using the Transmission Control Protocol (TCP).
    - When using TCP connection sockets, the TCP protocol makes sure that everything you send gets to the other end - there is a direct connection between one end and the other.
  - ✔ The other is analogous to a mailbox, a connectionless “datagram” service called the Unreliable Datagram Protocol (UDP).

Socket program

- The basic steps for socket programming are much the same as following:
  1. Open a socket.
  2. Open an input stream and output stream that bind to the socket.
  3. Read from and write to the stream according to the server's protocol.
  4. Close the streams.
  5. Close the socket.

ServerSocket

- The *ServerSocket* class is used to associate a *Socket* with a port.
  ServerSocket s=new ServerSocket(8189);
- The statement
  Socket incoming = s.accept();
  waits indefinitely until a client connects to the chosen port, then creates a socket for the Client to connect to.
- The accept() method blocks so that any thread including this statement will also be blocked.

Simple EchoServer

```
public class EchoServer {
 public static void main(String[] args ){
 try {
  ServerSocket s = new ServerSocket(8189);
  Socket incoming = s.accept( );
  BufferedReader in = new BufferedReader
          (new InputStreamReader(incoming.getInputStream()));
  PrintWriter out = new PrintWriter
      (incoming.getOutputStream(), true);
  out.println( "Hello! Enter BYE to exit." );
  boolean done = false;
  while (!done){
  String str = in.readLine();
  if (str == null) done = true;
  else{ out.println ("Echo: " + str);
  if (str.trim().equals("BYE")) done = true; }
   }  incoming.close();
} catch (Exception e) ….
```

Simple EchoServer

- The Socket object that represents the connection to the client can be used to get an input reader and an output writer
- BufferedReader in = new BufferedReader (new InputStreamReader(incoming.getInputStream()));
PrintWriter out = new PrintWriter(incoming.getOutputStream(), true);
- This means that everything the server sends to the out object, becomes the input to the connected client, and all output from the client ends up in the server's input stream object in.

Simple EchoServer

- We will concentrate on transmitting text through sockets, so we turn the streams into readers and writers. Then we can use the readLine() method and the print() method.
- In order to transmit binary data we would need to turn the streams into DataInputStreams and DataOutputStreams
- Having initialized the connection and set up the input and output streams, the program sends a greeting to the client.

Simple EchoServer

- The server then receives text from the client and displays it.
- To test this program
  - ✔ start the server
  - ✔ start telnet
  - ✔ connect to 127.0.0.1, port 8189
  - ✔ (127.0.0.1 is a special address, called the local loopback address that denotes the local machine. Convenient for setting up test systems)
- You should see the prompt
  - ✔ Hello! Enter BYE to exit
  - ✔ Enter anything, and it should be echoed on your screen.

A Simple Threaded Server

- One problem with the previous example is that only one client can connect to the server at a time.
- We overcome this by using threads.
  - ✔ Whenever a new client attempts to connect, we spawn a new thread to handle the communications with that client. Each client has its own connection thread.
- With new server, start telnet 2 or 3 times and connect as before. We can have multiple clients running connecting to the server.

A Simple Threaded Server

```
public class ThreadedEchoServer {
    public static void main(String[] args ){
       int i = 1;
       try{
         ServerSocket s = new ServerSocket(8189);
     while(true){
          Socket incoming = s.accept( );
          System.out.println("Spawning " + i);
          new ThreadedEchoHandler(incoming, i).start();
         i++;
            }
        }catch (Exception e) {
     System.out.println(e);
        }
    }
```

A Simple Threaded Server

```
class ThreadedEchoHandler extends Thread {
   public ThreadedEchoHandler(Socket i, int c) {
   incoming = i; counter = c;
   }
   public void run() { ….
    // code as before to handle the receipt and
   transmission of text.
   }
}
```

A Client/Server System

- We can combine the ideas for the client and server to build a Client/Server System.
- Simplify with both the server and the client running on one machine
- The client can talk to the server over the network. The server's response is similar. The system could easily be extended to make it much more sophisticated, however, the network code is very similar to what we have seen.
- Look at ChatServer.java and ChatClient.java

Datagrams

- The UDP protocol provides a mode of network communication whereby applications send packets of data, called datagrams, to one another.
- A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.
- The DatagramPacket and DatagramSocket classes in the java.net package implement system-independent datagram communication using UDP
- An application can send and receive DatagramPackets through a DatagramSocket.

DatagramPacket

- DatagramPacket are used to implement a connectionless packet delivery service.
  - ✔ message is routed from one machine to another based on information contained within that packet.
  - ✔ packets sent from one machine to another might be routed differently, and might arrive in any order
- Constructor
  DatagramPacket(byte[] buf, int length,InetAddress address, int port)
  - ✔ buf - the packet data.
  - ✔ length - the packet length.
  - ✔ address - the destination address.
  - ✔ port - the destination port number

InetAddress class

- InetAddress class represents an Internet Protocol (IP) address
- Applications should use the methods getLocalHost, getByName, or getAllByName to create a new InetAddress instance
  - ✔ internetAddress = InetAddress.getByName("localhost");
  - ✔ internetAddress = InetAddress.getLocalHost()
  - ✔ InetAddress[] addr= InetAddress.getAllByName(“trainsoft.com”)
    - Determines all the IP addresses of a host,
    - The host name can either be a machine name, such as "java.sun.com", or a string representing its IP address, such as "206.26.48.100“
  - ✔ getHostAddress()
    - Returns the IP address string "%d.%d.%d.%d".

DatagramSocket

- The **DatagramSocket** class represents a socket for sending and receiving datagram packets
- A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed.
- Constructor
  - ✔ **DatagramSocket(int port)**: Constructs a datagram socket and binds it to the specified port on the local host machine
  - ✔ **DatagramSocket(int port, InetAddress localAddr)** Creates a datagram socket, bound to the specified local address
- The receive() method waits forever until a packet is received. If no packet is received, the server makes no further progress and just waits.

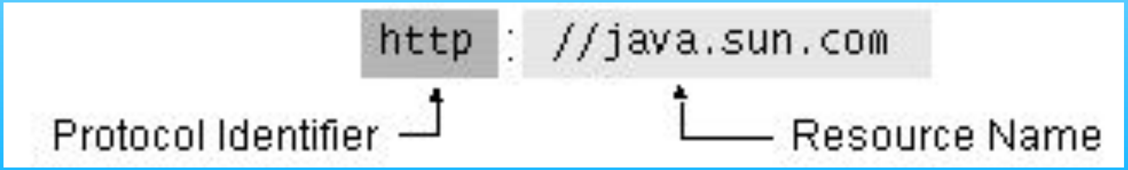
DatagramSocket

- The send(DatagramPacket) method sends a datagram packet from this socket.
  - ✔ The DatagramPacket includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.
- With a received packet, the DatagramSocket (server - receiver) can then send a response message to the sender base on address embbed inside the received packet
  - ✔

```
socket.receive(packet);
InetAddress address = packet.getAddress();
int port = packet.getPort();
byte[] buf="OK".getBytes();
DatagramPacket OKpack =
        new DatagramPacket(buf, buf.length, address, port);
socket.send(OKpack);
```

URL

- URL stand for Uniform Resource Locator. It is a reference (an address) to a resource on the Internet.
- Java programs can use a class called URL in the java.net package to represent a URL address
- a URL has two main components:
  - ✔ Protocol identifier
  - ✔ Resource name



*

URL

- The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the components listed in the following table
  - ✔ Host Name: The name of the machine on which the resource lives.
  - ✔ Filename: The pathname to the file on the machine.
  - ✔ Port Number: The port number to which to connect (typically optional).
  - ✔ Reference: A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).
  - ✔ **http://www.gamelan.com:8080/pages/Gamelan.html#note1**
- **For many protocols, the host name and the filename are required, while the port number and reference are optional.**

URL class

- The URL class provides various constructors for creating a URL object
  - ✔ new URL("http", "www.gamelan.com", "/pages/Gamelan.net.html");
  - ✔ new URL("http://www.gamelan.com/pages/Gamelan.net.html");
  - ✔ new URL("http", "www.gamelan.com", 80, "pages/Gamelan.html");
  - ✔ Each of the URL constructors throws a MalformedURLException if the arguments to the constructor refer to a null or unknown protocol.
- URLs are "write-once" objects. Once you've created a URL object, you cannot change any of its attributes (protocol, host name, filename, or port number)
- The URL class provides several methods that let you query URL objects.
- the URL's openStream() method return a stream from which the contents of the URL can be read
- the URL's openConnection() method open a connection to it

URLConnection

- The abstract class URLConnection is the superclass of all classes that represent a communications link between the application and a URL.
  - ✔ Instances of this class can be used both to read from and to write to the resource referenced by the URL.
- In general, creating a connection to a URL is a multistep process:
  1. The connection object is created by invoking the openConnection method on a URL.
  2. The setup parameters and general request properties are manipulated.
  3. The actual connection to the remote object is made, using the connect method.
  4. The remote object becomes available. The header fields and the contents of the remote object can be accessed

URLConnection

- The following methods are used to access the header fields and the contents after the connection is made to the remote object:
  - ✔ getContent
  - ✔ getHeaderField
  - ✔ getInputStream
  - ✔ getOutputStream
- Certain header fields are accessed frequently by:
  - ✔ getContentEncoding
  - ✔ getContentLength
  - ✔ getContentType
  - ✔ getDate
  - ✔ getExpiration
  - ✔ getLastModifed

URLEncoder

- The URLEncoder class contains a utility method for converting a String into a MIME format called "x-www-form-urlencoded" format.
- To convert a String, each character is examined in turn:
  - ✔ The ASCII characters 'a' through 'z', 'A' through 'Z', '0' through '9', and ".", "-", "*", "_" remain the same.
  - ✔ The space character ' ' is converted into a plus sign '+'.
  - ✔ All other characters are converted into the 3-character string "%xy", where xy is the two-digit hexadecimal representation of the lower 8-bits of the character.
- This class is basically used for converting text strings to a suitable form useable as part of an URL