CS360 Lecture 14
Networking with Sockets

Thursday, March 18, 2004

Reading

Networking: Chapter 18

Back to Basics

Question: What is a *network*?

Nodes are any machines that are connected to a network. What are examples of nodes?

Fully functional computers on a network are also called *hosts*.

Every node on a network has an *address*. Nodes also usually have names to aid us human folks because remembering network addresses is hard for us.

All modern computer networks are *packet-switched* networks. All data being sent across the network is broken into *packets* and each packet contains the information about who sent it and where it is going.

A *protocol* is a precise set of rules defining how computers communicate. Examples of protocols include http that defines how web browsers and servers communicate.

Layers of a Network

There are different layers of communication on a network.

The lowest level is the physical layer. This includes the actual wires, fibre optic cables or whatever is used to connect the different nodes on the network. Java never sees the physical layer.

The next layer up is the data link layer. This converts the analog data that is being sent on the physical later to the digital data needed by the computer. Error correction is also built into this layer since analog data contains noise. The most common data link layer is Ethernet. Again Java is not concerned with the data link layer.

The Internet or network layer is the first one that we need to concern ourselves with as Java programmers. In this layer a protocol defines how bits and bytes are organised into packets. The Internet Protocol (IP) is the only Internet protocol that Java understands. It's also the most popular protocol in the world. Examples of other protocols include AppleTalk and IPX. Data is sent across the Internet layer in packets called datagrams.

In the Internet layer there is no guarantee that datagrams will be delivered. They also do not arrive at their destination in the order they were delivered. The transport layer is responsible for ensuring

that packets are received in the order they were sent and making sure that no data was lost. There are two protocols at this level; TCP and UDP. TCP is the reliable protocol that allows for retransmission of lost data and delivery of bytes in the order they were sent. UDP is the unreliable protocol that is much faster but does not guarantee the delivery of the packets.

Finally, the application layer delivers the data to the user. It decides what to do with the data. The application layer is where most of the network parts of Java programs spend their time. Application layer protocols include HTTP, FTP, SMTP, POP and many more.

In summary, the application produces some data, adds a header to it and tells the receiving application how to handle it, and hands it to the transport layer. The transport layer adds another header and hands the result to the Internet layer.

Application Layer	Adds application header
Transport Layer	Adds TCP header
Internet/Network Layer	Adds IP header
Data Link Layer	Adds Ethernet header
Physical Layer	

IP

IP was developed with military sponsorship during the Cold War. It is robust and platform independent. If one computer in the network goes down the whole network does not go down. Therefore, IP was designed to allow multiple routes between any two points.

TCP

There are multiple routes between two points and the shortest path between two points changes over time due to network traffic and other factors. This means that packets that make up a particular data stream may not arrive in the order sent or arrive at all. TCP was added to give each connection the ability to acknowledge the receipt of IP packets and request retransmission of lost packets. TCP also puts the packets back together. This means that TCP carries some overhead.

UDP

If the order of the data isn't particularly important and if the loss of individual packets won't completely corrupt the data stream, packets are sent without the guarantee that TCP provides. This is accomplished through UDP. Examples of such data are video and audio signals.

IP Addresses

Java programmers do not need to be concerned with IP, but should know about addressing. Every computer on an IP network has a unique four-byte number. An example is 144.1.32.90

Ports

If computers did only one thing at a time then IP addresses will be all we need. However, computers do more than one thing at once. For example email needs to be separated from web and ftp requests. This is achieved through ports.

Purpose	Port Number
HTTP	80
SMTP	25
Telnet	23
POP3	110
IRC	194
Kazaa	1214 but can be reconfigured

Sockets

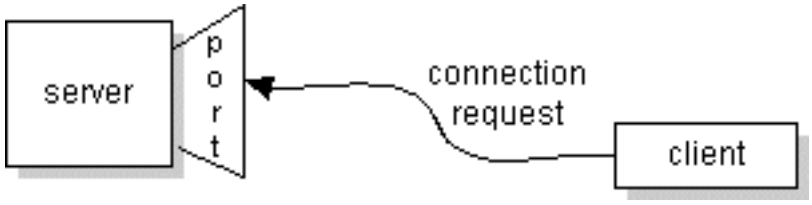
In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, 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.

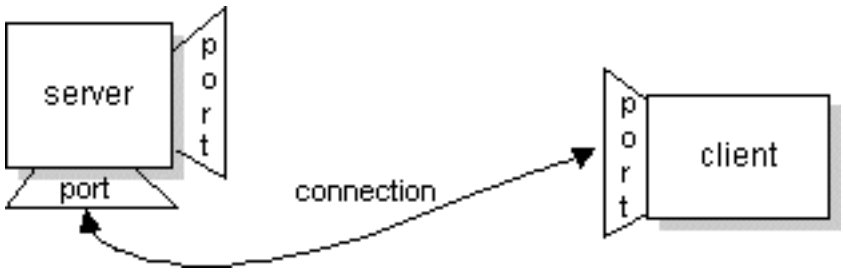
A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively.

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. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.



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 (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. Note that the socket on the client side is not bound to the port number used to rendezvous with the server. Rather, the client is assigned a port number local to the machine on which the client is running.

The client and server can now communicate by writing to or reading from their sockets.

Socket Client/Server Example

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

public class KnockKnockServer
{
  public static void main( String[] args ) throws IOException
  {
    ServerSocket serverSocket = null;
    try
    {
      serverSocket = new ServerSocket( 7777 );
    }
    catch ( IOException e )
    {
      System.err.println("Could not listen on port: 4444.");
      System.exit(1);
```

```
    }

    Socket clientSocket = null;
    try
    {
      clientSocket = serverSocket.accept();
    }
    catch ( IOException e )
    {
      System.err.println( "Accept failed." );
      System.exit(1);
    }

    ObjectOutputStream output =
         new ObjectOutputStream( clientSocket.getOutputStream() );
    ObjectInputStream input =
         new ObjectInputStream( clientSocket.getInputStream() );

    String inputLine, outputLine;
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput( null );
    output.writeObject( outputLine );
    output.flush();

    try
    {
      while((inputLine = (String) input.readObject()) != null)
      {
        outputLine = kkp.processInput( inputLine );
        output.writeObject( outputLine );
        output.flush();
        if ( outputLine.equals( "Bye." ) )
          break;
      }
    }
    catch ( ClassNotFoundException classNotFoundException )
    {
      System.err.println( "\nUnknown object type received" );
        System.exit(1);
    }
    output.close();
    input.close();
    clientSocket.close();
    serverSocket.close();
  }
}
```

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

public class KnockKnockProtocol
{
  private static final int WAITING = 0;
  private static final int SENTKNOCKKNOCK = 1;
  private static final int SENTCLUE = 2;
  private static final int ANOTHER = 3;

  private static final int NUMJOKES = 5;

  private int state = WAITING;
  private int currentJoke = 0;

  private String[] clues = { "Turnip", "Little Old Lady",
                             "Atch", "Who", "Who" };
  private String[] answers = { "Turnip the heat, it's cold!",
                               "I didn't know you could yodel!",
                               "Bless you!",
                               "Is there an owl in here?",
                               "Is there an echo in here?" };

public String processInput( String theInput )
{
  String theOutput = null;

  if( state == WAITING )
  {
    theOutput = "Knock! Knock!";
    state = SENTKNOCKKNOCK;
  }
  else if( state == SENTKNOCKKNOCK )
  {
    if( theInput.equalsIgnoreCase( "Who's there?" ) )
    {
      theOutput = clues[currentJoke];
      state = SENTCLUE;
    }
    else
    {
      theOutput = "You're supposed to say \"Who's there?\"! " +
                          "Try again. Knock! Knock!";
    }
  }
```

```
  else if ( state == SENTCLUE )
  {
    if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?"))
    {
      theOutput = answers[currentJoke] + " Want another? (y/n)";
      state = ANOTHER;
    }
    else
    {
      theOutput = "You're supposed to say \"" +
                  clues[currentJoke]
                  + " who?\"! Try again. Knock! Knock!";
      state = SENTKNOCKKNOCK;
    }
  }
  else if ( state == ANOTHER )
  {
    if( theInput.equalsIgnoreCase( "y" ) )
    {
      theOutput = "Knock! Knock!";
      if( currentJoke == ( NUMJOKES - 1 ) )
        currentJoke = 0;
      else
        currentJoke++;
        state = SENTKNOCKKNOCK;
    }
    else
    {
      theOutput = "Bye.";
      state = WAITING;
    }
  }

  return theOutput;
  }
}

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

public class KnockKnockClient
{
  public static void main( String[] args ) throws IOException
  {

    final String HOST = "";
```

```
Socket kkSocket = null;
try
{
  kkSocket = new Socket( HOST, 7777 );
}
catch ( IOException e )
{
  System.err.println( "Couldn't get I/O for connection to:"
                      + HOST );
  System.exit(1);
}

ObjectOutputStream output =
     new ObjectOutputStream( kkSocket.getOutputStream() );
ObjectInputStream input =
     new ObjectInputStream( kkSocket.getInputStream() );

BufferedReader stdIn =
    new BufferedReader( new InputStreamReader( System.in ) );
String fromServer, fromUser;

try
{
  while ((fromServer = (String) input.readObject()) != null)
  {
    System.out.println( "Server: " + fromServer );
    if ( fromServer.equals( "Bye." ) )
      break;

    fromUser = stdIn.readLine();

    if ( fromUser != null )
    {
      System.out.println( "Client: " + fromUser );
      output.writeObject( fromUser );
    }
  }
}
catch ( IOException e )
{
  System.err.println( "Couldn't get I/O for connection to: "
                      + HOST );
  System.exit(1);
}
catch ( ClassNotFoundException classNotFoundException )
```

```
    {
      System.err.println( "\nUnknown object type received" );
    }
    output.close();
    input.close();
    stdIn.close();
    kkSocket.close();
  }
}
```