Java Network Programming

http://usmans.info

Things we will learn:

- Internet Addresses
- TCP/IP
- Sockets
- Server Sockets
- UDP

Assumptions

- You already know about:
- Understanding basic Java syntax and I/O
- User's view of the internet
- No prior network programming experience

Some Background

- Hosts
- Internet Addresses
- Ports
- Protocols

Hosts

- Devices connected to the Internet are called hosts
- Most hosts are computers, but hosts also include routers, printers, fax machines, soda machines, bat houses, etc.

Internet Addresses

- Every host on the Internet is identified by a unique, four-byte Internet Protocol (IP) address.
- This is written in dotted quad format like 199.1.32.90 where each byte is an unsigned integer between 0 and 255.
- There are about four billion unique IP addresses, but they aren't very efficiently allocated
- IPv6 expands the address space to 2^{128}

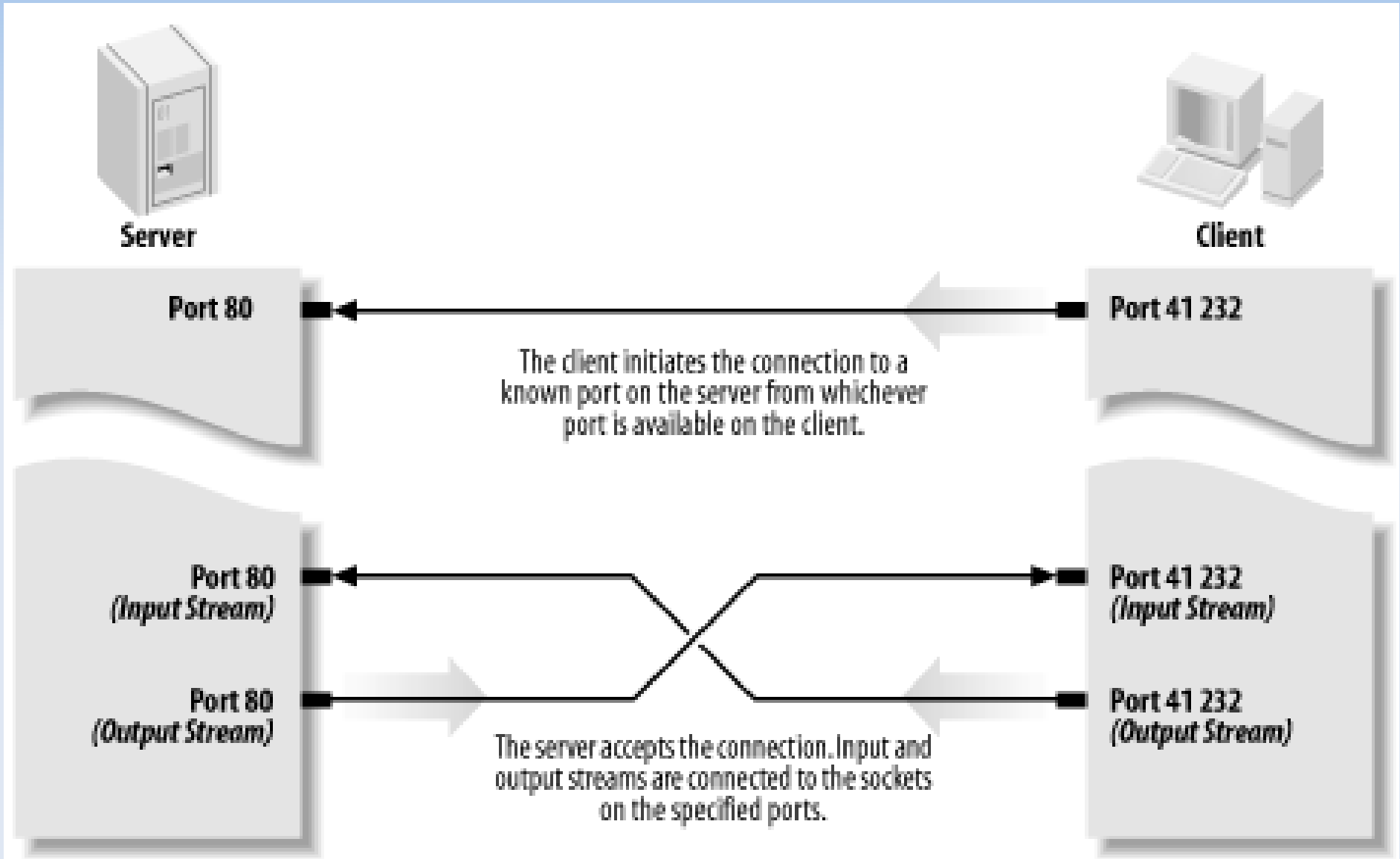
Cont.

- Network Address Translation (NAT) allows hosts to share same addresses by using proxy servers.

Domain Name System (DNS)

- Numeric addresses are mapped to names like usmans.info or www.usmans.info by DNS.
- Each site runs domain name server software that translates names to IP addresses and vice versa
- DNS is a distributed system

The Client/Server Model



Cont...

- A client/server application typically stores large quantity of data on (expensive) server machines and softwares, while logic and user interface is handled on (cheap) client machines and softwares. For example Database Server, Web Server etc.
- Some server process and analyze data before sending results back to client. They are often refer as ”Application Server”.
- Peer-To-Peer networking is different than client/server

The InetAddress class

- The java.net.InetAddress class represents an IP address.
- It converts numeric addresses to host names and host names to numeric addresses.
- It is used by other network classes like Socket and ServerSocket to identify hosts

Creating InetAddresses

- There are no public InetAddress() constructors. Arbitrary addresses may not be created.
- All addresses that are created must be checked with DNS

The getByName factory method

- `public static InetAddress getByName(String host) throws UnknownHostException`
- For example,

```
try {
  InetAddress usmanbyhost =
InetAddress.getByName("usmans.info");
  InetAddress usmanbyip =
InetAddress.getByName("64.191.15.246");
  // work with address....
}
catch (UnknownHostException e) {
  System.err.println(ex);
}
```

Other ways to create InetAddress

- `public static InetAddress[] getAllByName(String host) throws UnknownHostException`
- `public static InetAddress getLocalHost() throws UnknownHostException`

Getter Methods

- `public boolean isMulticastAddress()`
- `public String getHostName()`
- `public byte[] getAddress()`
- `public String getHostAddress()`

Utility Methods

- `public int hashCode()`
- `public boolean equals(Object o)`
- `public String toString()`

Ports

- In general a host has only one Internet address
- This address is subdivided into 65,536 ports
- Ports are logical abstractions that allow one host to communicate simultaneously with many other hosts
- Many services run on well-known ports. For example, http tends to run on port 80

Protocols

- A protocol defines how two hosts talk to each other.
- The daytime protocol, RFC 867, specifies an ASCII representation for the time that's legible to humans.
- The time protocol, RFC 868, specifies a binary representation for the time that's legible to computers.
- There are thousands of protocols, standard and non-standard

Datagrams

- Before data is sent across the Internet from one host to another using TCP/IP, it is split into packets of varying but finite size called datagrams.
- Datagrams range in size from a few dozen bytes to about 60,000 bytes.
- Packets larger than this, and often smaller than this, must be split into smaller pieces before they can be transmitted.

Packets Allow Error Correction

- If one packet is lost, it can be retransmitted without requiring redelivery of all other packets.
- If packets arrive out of order, they can be reordered at the receiving end of the connection.

Abstraction

- Datagrams are mostly hidden from the Java programmer.
- The host's native networking software transparently splits data into packets on the sending end of a connection, and then reassembles packets on the receiving end.
- Instead, the Java programmer is presented with a higher level abstraction called a socket.

Sockets

- A socket is a reliable connection for the transmission of data between two hosts.
- Sockets isolate programmers from the details of packet encodings, lost and retransmitted packets, and packets that arrive out of order.
- There are limits. Sockets are more likely to throw IOExceptions than files, for example.

Socket Operations

- There are four fundamental operations a socket performs. These are:
  - Connect to a remote machine
  - Send data
  - Receive data
  - Close the connection
- A socket may not be connected to more than one host at a time.
- A socket may not reconnect after it's closed.

The java.net.Socket class

- The `java.net.Socket` class allows you to create socket objects that perform all four fundamental socket operations.
- You can connect to remote machines; you can send data; you can receive data; you can close the connection.
- Each Socket object is associated with exactly one remote host. To connect to a different host, you must create a new Socket object.

Constructing a Socket

- Connection is established through the constructors

```
•public Socket(String host, int port) throws
UnknownHostException, IOException
•public Socket(InetAddress address, int port)
throws IOException
•public Socket(String host, int port, InetAddress
localAddr, int localPort) throws IOException
•public Socket(InetAddress address, int port,
InetAddress localAddr, int localPort) throws
IOException
```

Opening Sockets

- The Socket() constructors do not just create a Socket object. They also attempt to connect the underlying socket to the remote server.
- All the constructors throw an IOException if the connection can't be made for any reason.

Choosing the Host and Port

- You must at least specify the remote host and port to connect to.
- The host may be specified as either a string like "usmans.info" or as an InetAddress object.
- The port should be an int between 1 and 65535.
- Socket webUsmanInfo = new Socket("usmans.info", 80);
- You cannot just connect to any port on any host. The remote host must actually be listening for connections on that port.

Cont...

- You can use the constructors to determine which ports on a host are listening for connections.

Port Scanner

```
public static void scan(InetAddress remote) {

  String hostname = remote.getHostName();
  for (int port = 1; port < 65536; port++) {
    try {
      Socket s = new Socket(remote, port);
      System.out.println("A server is listening on port "
       + port + " of " + hostname);
      s.close();
    }
    catch (IOException ex) {
      // The remote host is not listening on this port
    }
  }

}
```

Picking an IP Address

- The last two constructors also specify the host and port you're connecting from.
- On a system with multiple IP addresses, like many web servers, this allows you to pick your network interface and IP address.

Choosing a local port

- You can also specify a local port number,
- Setting the port to 0 tells the system to randomly choose an available port.
- If you need to know the port you're connecting from, you can always get it with getLocalPort().
- ```
  Socket usmanInfo = new
  Socket("usmans.info", 80, "localhost",
  0);
  ```
  ```

Sending and Receiving Data

- Data is sent and received with output and input streams.
- There are methods to get an input stream for a socket and an output stream for the socket.

```
public InputStream  getInputStream()
throws IOException
public OutputStream getOutputStream()
throws IOException
```

- There's also a method to close a socket:

```
public synchronized void close() throws
IOException
```

Reading Input from a Socket

- The `getInputStream()` method returns an `InputStream` which reads data from the socket.
- You can use all the normal methods of the `InputStream` class to read this data.
- Most of the time you'll chain the input stream to some other input stream or reader object to more easily handle the data.

Time Client

- The following code fragment connects to the daytime server on port 13 of tock.usno.navy.mil, and displays the data it sends.

```
try {
  Socket s = new Socket("tock.usno.navy.mil", 13);
  InputStream in = s.getInputStream();
  InputStreamReader isr = new InputStreamReader(in);
  BufferedReader br = new BufferedReader(isr, "ASCII");
  int c;
  while ((c = br.read()) != -1) {
    System.out.print((char) c);
  }
  System.out.println();
}
catch (IOException ex) {
  return (new Date()).toString();
}
```

Writing Output to Socket

- The `getOutputStream()` method returns an output stream which writes data to the socket.
- Most of the time you'll chain the raw output stream to some other output stream or writer class to more easily handle the data.

Reading and Writing to a Socket

- It's unusual to only read from a socket. It's even more unusual to only write to a socket.
- Most protocols require the client to both read and write.

Synchronous

- Some protocols require the reads and the writes to be interlaced. That is:

```
write
read
write
read
write
read
```

Request-Response

- Other protocols, such as HTTP 1.0, have multiple writes, followed by multiple reads, like this:

```
write
write
write
read
read
read
read
```

Asynchronous

- Other protocols don't care and allow client requests and server responses to be freely intermixed.
- Java places no restrictions on reading and writing to sockets.
- One thread can read from a socket while another thread writes to the socket at the same time.

Very basic HTTP client

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

public class SocketCat {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      int port = 80;
      String file = "/";
      try {
        URL u = new URL(args[i]);
        if (u.getPort() != -1) port = u.getPort();
        if (!(u.getProtocol().equalsIgnoreCase("http"))) {
          System.err.println("I only understand http.");
          continue;
        }
```

Cont...

```
    if (!(u.getFile().equals(""))) file = u.getFile();

    Socket s = new Socket(u.getHost(), port);

    OutputStream theOutput = s.getOutputStream();
    PrintWriter pw = new PrintWriter(theOutput, false);
    pw.println("GET " + file + " HTTP/1.0");
    pw.println("Accept: text/plain, text/html, text/*");
    pw.println();
    pw.flush();

    InputStream in = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(in);
    BufferedReader br = new BufferedReader(isr, "ASCII");

    String theLine;
    while ((theLine = br.readLine()) != null) {
      System.out.println(theLine);
    }
}
```

Cont...

```
      catch (MalformedURLException e) {
        System.err.println(args[i] + " is not a valid URL");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

Socket Options

- Several methods set various socket options. Most of the time the defaults are fine.

```
public void setTcpNoDelay(boolean on) throws
SocketException
public boolean getTcpNoDelay() throws
SocketException
public void setSoLinger(boolean on, int val)
throws SocketException
public int getSoLinger() throws
SocketException
public void setSoTimeout(int timeout) throws
SocketException
public int getSoTimeout() throws
SocketException
```

Getter Methods

- Methods to return information about the socket:

```
public InetAddress getInetAddress()
public InetAddress getLocalAddress()
public int getPort()
public int getLocalPort()
```

- Finally there's the usual toString() method:

```
public String toString()
```

Servers

- There are two ends to each connection: the client, that is the host that initiates the connection, and the server, that is the host that responds to the connection.
- Clients and servers are connected by sockets.
- A server, rather than connecting to a remote host, a program waits for other hosts to connect to it.

Server Sockets

- A server socket binds to a particular port on the local machine.
- Once it has successfully bound to a port, it listens for incoming connection attempts.
- When a server detects a connection attempt, it accepts the connection. This creates a socket between the client and the server over which the client and the server communicate.

Multiple Clients

- Multiple clients can connect to the same port on the server at the same time.
- Incoming data is distinguished by the port to which it is addressed and the client host and port from which it came.
- The server can tell for which service (like http or ftp) the data is intended by inspecting the port.
- It can tell which open socket on that service the data is intended for by looking at the client address and port stored with the data.

Queuing

- Incoming connections are stored in a queue until the server can accept them.
- On most systems the default queue length is between 5 and 50.
- Once the queue fills up further incoming connections are refused until space in the queue opens up.

The java.net.ServerSocket class

- The `java.net.ServerSocket` class represents a server socket.
- A `ServerSocket` object is constructed on a particular local port. Then it calls `accept()` to listen for incoming connections.
- `accept()` blocks until a connection is detected. Then `accept()` returns a `java.net.Socket` object that performs the actual communication with the client.

Constructors

- There are three constructors that let you specify the port to bind to, the queue length for incoming connections, and the IP address to bind to:

```
public ServerSocket(int port) throws
IOException
public ServerSocket(int port, int backlog)
throws IOException
public ServerSocket(int port, int backlog,
InetAddress networkInterface) throws
IOException
```

Constructing Server Sockets

- Normally you only specify the port you want to listen on, like this:

```
try {
    ServerSocket ss = new ServerSocket(80);
}
catch (IOException ex) {
    System.err.println(ex);
}
```

Binding to Ports

- When a ServerSocket object is created, it attempts to bind to the port on the local host given by the port argument.
- If another server socket is already listening to the port, then a java.net.BindException, a subclass of IOException, is thrown.
- No more than one process or thread can listen to a particular port at a time. This includes non-Java processes or threads.
- For example, if there's already an HTTP server running on port 80, you won't be able to bind to port 80.

Port numbers

- On Unix systems (but not Windows or the Mac) your program must be running as root to bind to a port between 1 and 1023.
- 0 is a special port number. It tells Java to pick an available port.
- The getLocalPort() method tells you what port the server socket is listening on. This is useful if the client and the server have already established a separate channel of communication over which the chosen port number can be communicated.

Expanding the queue

- if you think you aren't going to be processing connections very quickly you may wish to expand the queue when you construct the server socket. For example,

```
try {
  ServerSocket httpd = new ServerSocket(80, 50);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Choosing an IP Address

- Many hosts have more than one IP address.
- By default, a server socket binds to all available IP addresses on a given port.
- You can modify that behavior with this constructor:

```
public ServerSocket(int port, int backlog,
  InetAddress bindAddr) throws IOException
```

Example

```
try {
 InetAddress ia =
  InetAddress.getByName("199.1.32.90");
 ServerSocket ss = new ServerSocket(80, 50,
  ia);
}
catch (IOException ex) {
 System.err.println(ex);
}
```

Checking the Port number

- On a server with multiple IP addresses, the `getInetAddress()` method tells you which one this server socket is listening to.

```
public InetAddress getInetAddress()
```

- The `getLocalPort()` method tells you which port you're listening to.

```
public int getLocalPort()
```

Accepting Connections

- The accept() and close() methods provide the basic functionality of a server socket.

```
public Socket accept() throws IOException

public void close() throws IOException
```

- A server socket can't be reopened after it's closed

Reading Data with Server Socket

- ServerSocket objects use their accept() method to connect to a client.

```
public Socket accept() throws IOException
```

- There are no getInputStream() or getOutputStream() methods for ServerSocket.
- accept() returns a Socket object, and its getInputStream() and getOutputStream() methods provide streams.

A Simple Server

```
try {
   ServerSocket ss = new ServerSocket(2345);
   Socket s = ss.accept();
   PrintWriter pw = new
   PrintWriter(s.getOutputStream());
   pw.println("Hello There!");
   pw.println("Goodbye now.");
   s.close();
}
catch (IOException ex) {
   System.err.println(ex);
}
```

A Better Server

```
try {
 ServerSocket ss = new ServerSocket(2345);
 Socket s = ss.accept();
 BufferedWriter out = new BufferedWriter(
                      new OutputStreamWriter(
                        s.getOutputStream(), "UTF-8"
                      ));
 out.write("Hello There!");
 out.newLine();
 out.write("Goodbye now.");
 out.newLine();
 out.flush();
 s.close();
}
catch (IOException ex) {
 System.err.println(ex);
}
```

Writing data to client one by one

```
try {
  ServerSocket ss = new ServerSocket(port);
  while (true) {
    try {
      Socket s = ss.accept();
      BufferedWriter out = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
      out.write("Hello " + s.getInetAddress() + " on port " +
s.getPort()); out.writeLine();
      out.write("This is " + s.getLocalAddress() + " on port "
+ s.getLocalPort()); out.writeLine();
      out.flush();
      s.close();
    }
    catch (IOException ex) { // skip this connection; continue
with the next }
   }
 } catch (IOException ex) { System.err.println(ex); }
```