Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-eight)

8.7: Socket Programming in Java

In this section, I assume you know the rudiments of network programming in C on UNIX or Win32. The books by Stevens and Harold (see **Recommended Reading** in the syllabus) contain some pointers to additional reading if you need to come up to speed on this topic. The Stevens book is a definitive but accessible introduction to everything you'd ever want to know about Internet programs and protocols at the application programmer's level. It does not specifically mention Java, but the material is useful to Java programmers. The Harold book contains many Java-specific network-programming techniques, and includes an exhaustive guide through the java.net package.

Java's java.awt.Socket class is very easy to use. You can open a TCP client connection simply by creating a Socket object. You can then obtain an InputStream and an OutputStream for two-way communication:

```
// host and port
Socket s = new Socket("hostname.com", 6543);
PrintWriter pw =
  new PrintWriter(new
                  OutputStreamWriter
                  (s.getOutputStream());
BufferedReader br =
  new BufferedReader(new
                     InputStreamReader
                     (s.getInputStream());
```

You can use these streams just like any other local streams. Here we are preparing to exchange textual data, but of course you can use the bare byte streams to exchange raw data. The DataInputStream and DataOutputStream classes are very useful for network programming, as they have methods such as writeInt and readFloat that send and receive data in network (Big-Endian) byte order, regardless of the hardware the Java Virtual Machine is running on. This makes them handy when you're writing a Java client to an existing network service. Note that there is also a DatagramSocket class, which lets you write UDP clients.

It is possible to extend the class Socket and alter its functionality. This provides some interesting opportunities. For example, you could write a CryptoSocket class that encrypted all data that passed through it. As long as both ends of the connection used your CryptoSocket class, they would remain completely unaware of the encryption, and in fact you could change the algorithm at will without affecting any client code. The most flexible way to do this would be to put the encryption code into two custom stream classesone InputStream and one OutputStream and have your CryptoSocket class simply return these classes from getOutputStream( ) and getinputStream( ), wrapped around the streams returned from super.get(X)Stream( ). Here you would be deliberately applying the Factory design pattern we discussed above. Similarly, you could produce a CompressedSocket, or a FilterSocket that otherwise altered some of the data that passed through.

Network server programming in Java is equally simple. You create an instance of java.net.ServerSocket, then call accept( ), which blocks until a connection request is received. accept( ) returns a Socket, as above, and you can then make a two-way connection the the client:

```
ServerSocket ss = new ServerSocket(6543);
Socket s = ss.accept();
DataOutputStream dos =
   new DataOutputStream(s.getOutputStream());
DataInputStream dis =
   new DataInputStream(s.getInputStream());
```

Many Internet servers are written as *concurrent* servers, meaning that they can handle multiple connections simultaneously. This is usually done on the UNIX platform by using fork( ) to create a new child process to handle each incoming request. In Java, the idiom is to use threads:

```
public class RequestHandler extends Thread
{
    private Socket m_s;
    public RequestHandler(Socket s) { m_s = s;}
    public void run() { /* handle the request on s */ }
}

...

ServerSocket ss = new ServerSocket(6543);

while (true)   // accept connections forever
{
   // wait for connection
   Socket s = ss.accept();
   // start a Thread to handle it
   new RequestHandler(s).start();
}
```

Note that you can also extend ServerSocket to create a ServerSocket that returns CryptoSockets from accept( ):

```
public class CryptoServerSocket
   extends ServerSocket
{
    // constructor omitted, but you'd have to
    //include them!
    public Socket accept()
    {
       return new CryptoSocket(super.accept());
    }
}
```

Again, the encryption is completely transparent to the server code.