

# To java.net and Beyond Teaching Networking Concepts Using the Java Networking API

Greg Gagne  
Westminster College  
Salt Lake City, UT  
ggagne@westminstercollege.edu

## Abstract

This paper covers the use of Java and its API for developing networking programs in an undergraduate computer networks class. Through the use of TCP and UDP sockets provided in the `java.net` package, students are able to write several client-server applications such as web and mail servers and a chat room. Additionally, the `java.rmi` API provides a library for invoking methods on remote objects, similar to remote procedure calls. Remote method invocation (RMI) is used to write a distributed version of the Producer-Consumer problem.

## 1 Background

The computer network class at Westminster College is a required course that computer science students typically take their junior or senior years. Students have had significant Java programming experience as our CS1, CS2 and CS7 classes use Java. There are several prerequisites for the class, however there is no expectation of previous network programming experience.

The course is focused around TCP/IP and how the Internet works. Rather than covering the OSI reference model, we instead use the hybrid reference model outlined in [3]. This model is shown below:

Level	Layer
5	Application
4	Transport
3	Network
2	Data Link
1	Physical

Various protocols such as HTTP, FTP, SMTP, etc. are

used as examples of the Application layer. TCP/IP is used to demonstrate the Transport and Network layers. Ethernet is used primarily as a reference for the Data Link and Physical layers, although coverage of other Medium Access Protocols (MAC) such as Token Ring is also presented. Various accompanying issues such as security, caching, wireless networking, wide-area networking, etc. is also covered where appropriate.

Coverage of the application layer focuses on the client-server paradigm. There has been significant interest recently in the peer-to-peer model, especially with the growth of services such as Napster. However, despite the popularity of peer-to-peer, most services are likely to remain client-server [1]. Furthermore, the role of the client-server model in relationship to Curricula 2001 has also been explored [2].

For several years I taught computer networks by programming sockets using C on a variety of UNIX platforms, including Linux and Solaris. The difficulty in this approach primarily lies with the barrier of C programming using the BSD socket API. Even programming a simple echo server is a non-trivial task. In 1996 I began using Java and its API for distributed programming in our computer networks class. Using Java has raised the bar where students are now writing more complex and more interesting programs than when C was being used.

## 2 Course Goals

The programming goals of the computer networking course are for students to understand: (1) the client-server model, (2) the role of protocols including distributed object protocols, and (3) performance issues. After brief coverage of connection-oriented socket and I/O programming in Java, this paper will illustrate several different client-server assignments used to reinforce these goals. Coverage of unreliable, connection-less sockets and Java's remote method invocation (RMI) and associated assignments follows. Lastly, this paper concludes by evaluating the effectiveness of using Java to teach network programming according to the goals

outlined in this section.

### 3 Socket Programming in Java

This section describes the the construction of reliable, connection-oriented sockets (TCP) using the `java.net` API. This section intends to highlight the simplicity of socket creation and management in `java.net` and does not provide detailed coverage. A “server socket” refers to the socket on the server end, a “socket” refers to the client-side socket. For this example, the server socket resides on host `cs.wcslc.edu` and is listening on port 6000.

#### 3.1 Server Sockets

A server socket is created with the statement

```
ServerSocket sock = new ServerSocket(6000);
```

Once created, the `accept()` method is used to listen for incoming connections to the server socket:

```
Socket client = sock.accept();
```

`accept()` returns the socket connecting from the client side. The server can communicate with the client by opening an input or output stream for the client socket.

After accepting a connection, a single-threaded server services the client. Alternatively, a multi-threaded server hands the connection off to a separate thread for service and the server resumes listening for more connection requests.

#### 3.2 Client Sockets

A client connects to a server socket by creating a `Socket` specifying the IP address and port of the server. The following statement connects to port 6000 on host `cs.wcslc.edu`:

```
Socket server = Socket("cs.wcslc.edu",6000);
```

Once the connection is made, the client may open an input or output stream to communicate with the server socket.

#### 3.3 Streams

Streams are part of the `java.io` API and are the mechanism by which input and output is done in Java. Streams may be either character or byte oriented. An *input stream* is used to read from a data source, data is written to an *output stream*. There is an associated input and output stream with each TCP socket.

Using the example shown above, once the client connects to the server socket, the server may write the message “Hello There” to the socket with the following statements:

```
OutputStream raw = client.getOutputStream();  
Writer pout = new PrintWriter(raw);  
pout.write("Hello There");
```

The client would read this message from the server socket with the statements:

```
InputStream raw = server.getInputStream();  
Reader pin = new InputStreamReader(raw);  
BufferedReader bin =  
    new BufferedReader(bin);  
String msg = bin.readLine();
```

Streams provide an abstract mechanism for performing input and output, regardless whether the I/O is being done on the local filesystem or a socket. To enforce this, an early assignment is to have students write two versions of a program that reads files - the first program reads from the local filesystem, the second reads a file from a URL (using the `java.net.URL` class) on a remote host. This simple assignment emphasizes the polymorphism in the `java.io` package as both programs essentially use the same methodology for performing the input.

#### 3.4 Threads

All programming assignments are client-server. As servers are often multithreaded, this requires coverage of thread creation in Java. Although some students have had significant exposure to threads and thread synchronization in our operating systems class, I have discovered that students without prior multithreading experience quickly grasp the concept. However, to ease their understanding, I often provide the necessary code details to both create a thread per message (i.e. create a new thread when a web server receives a request for a resource) and using a thread pool.

## 4 Client Server Assignments Using TCP

This section illustrates four different client-server assignments using reliable, connection-oriented TCP sockets in Java. Students worked on each project in pairs. Prior to these assignments, I provide students with several example programs such as an echo server and a time-of-day server. I also have the students learn some protocols in advance by acting as a client and requesting services “by hand”. For example, students telnet to a web server and manually request a web page. Students also manually deliver an email message using a telnet connection. First acting as a manual client is a useful step for students to better understand the role of protocols.

#### 4.1 Web Server

The first detailed assignment is for students to write a web server implementing a subset of the HTTP 1.0 protocol. Specifically, students must support the HEAD and GET methods. The web server is multithreaded using a thread pool. When the server receives a request from a client, it passes the request off to an available thread in the pool. After handing this request off, the server resumes listening for more requests. If there are no available threads in the pool, the client must wait until one becomes available.

The server returns the following to the client: (1) HTTP status code, (2) the date, (3) server name, (4) content length, (5) content type, and lastly (6) the document. For example, the following represents the headers returned by the server “Super-Caffeinated WWW Server” after it has received the GET request for the file `index.html`:

```
HTTP/1.0 200 OK
Date: Tue Jul 17 15:54:42 MDT 2001
Super-Caffeinated WWW Server 1.1
Content-length: 145
Content-type: text/html
```

After sending these headers, the contents of the document `index.html` is delivered to the client.

In addition to delivering resources to clients, the web server must also implement logging and provide a document root.

Students must also implement a caching mechanism on the server using a policy of their choice (i.e. LRU). Despite the simplicity of cache control in this assignment, merely incorporating caching allows students to better understand the role of web and proxy caches, and the different levels of cooperative caches for the WWW.

Although not required, requiring support for the POST and PUT methods or providing support for persistent connections are possibilities for enhancing this assignment.

#### 4.2 Mail Server

Following the web server, students implement a variation of the SMTP protocol - VSMTP (for Very Simple Mail Transport Protocol). The VSMTP is defined as

Command	Meaning
<b>HELO</b>	Connect to the mail server
<b>MAIL FROM</b>	The sender
<b>RCPT TO</b>	The receiver
<b>DATA</b>	The beginning of data

Students first write the mail server that receives client email messages according to VSMTP. Upon receipt of

an email message, it is stored in the mailbox for the intended recipient. Prior to writing the client application, many students first debug their server by establishing a telnet session to the server and delivering a mail message manually.

Students implement the client application using a graphical interface. The interface must allow the client to compose an email message and transmit it to the mail server according to VSMTP. The client must also connect to a recipient’s mailbox and retrieve messages.

#### 4.3 Chatroom

Students also write a client-server application for a chatroom. Specifically, the server acts as the chatroom where clients may connect and communicate with others in the same chatroom. When a chatroom member sends a message, it is delivered to all clients in the same chatroom. However, for added functionality, members can also *whisper* to another specific member in the group whereby the message is only delivered to the intended member. For performance reasons, the server uses a separate thread to communicate with each client thread although variations could incorporate peer-to-peer designs.

Each chatroom client connects to the server and identifies the client according to their *handle* - their name in the chatroom. When a client joins the chatroom, their handle is broadcast to all members in the chatroom. Likewise, when the client exits, the departure is also broadcast. The client application is written as a graphical interface.

#### 4.4 File Transfer

Students must write a client-server program that acts similarly to the well-known `ftp` program for transferring files. However, unlike with the web server and mail server assignments, students must devise a protocol on their own. The following contains a few commands from the protocol designed by one pair of students:

Command	Meaning
<b>PUTFL &lt;file&gt;</b>	Put file
<b>GETFL &lt;file&gt;</b>	Get file
<b>CHDIR &lt;directory&gt;</b>	Change directory
<b>ASCII</b>	ASCII mode
<b>BNARY</b>	Binary mode
<b>DISCN</b>	Disconnect
<b>LOGNO</b>	Invalid login
<b>BADCM</b>	Unknown command

The server listens to a port awaiting client connections to place or retrieve files. The required version of this program specifies only a command-line syntax for the

client, similar to the UNIX `ftp` utility. However, although not required, many students implemented the client with a graphical interface, similar to the file transfer tools `wsftp` and `fetch`.

## 5 UDP Sockets

The TCP sockets described in Section 3 are connection-oriented and reliable. In many networking applications, the reliability provided by TCP is unnecessary. Streaming audio and video come to mind. The `java.net` API also provides connectionless, unreliable UDP sockets. In Java, the fundamental differences between TCP and UDP sockets are: (1) UDP does not distinguish between a client and server socket, all UDP sockets use the class `DatagramSocket`. (2) UDP sockets are connectionless - the client and server cannot obtain an input or output stream at the other end of the connection. Rather than using streams, UDP data must be inserted into - and extracted from - a datagram.

The UDP socket for the server is created with the statement

```
DatagramSocket socket =  
    new DatagramSocket(port);
```

The `port` parameter identifies the port the server is listening to. The server then constructs a `DatagramPacket` that will be used to encapsulate the datagram received from the client. Arriving data will be stored in a byte array `buffer`:

```
DatagramPacket data =  
    new DatagramPacket(buffer,buffer.length);
```

The server then waits for datagram packets using the blocking `receive()` method of the `DatagramSocket` class:

```
socket.receive(data);
```

Once the server receives a datagram, it can retrieve its contents from the `buffer` byte array.

The client communicates with the server by first constructing an empty `DatagramSocket`:

```
DatagramSocket server =  
    new DatagramSocket();
```

The client then packages the data to send into a byte array which we will name `info`. Using this byte array, it then constructs a `DatagramPacket` containing (1) the `info` array and its length, (2) the IP address of the server (IP), and (3) the port the server is listening on (`port`). The following statement accomplishes this

```
DatagramPacket thePacket = new  
    DatagramPacket(info,info.length,IP,port);
```

This `DatagramPacket` is now delivered to the socket

using the `send` method of the `DatagramSocket` class:

```
server.send(thePacket);
```

Delivery of a `datagramPacket` best describes the differences between TCP and UDP. With connection-oriented TCP sockets, all data can be handled using the input or output stream associated with the connection. Because UDP is connectionless, the data and the IP address and port of the server must all be part of the datagram.

### 5.1 UDP Assignment

To reflect the usefulness of an unreliable protocol such as UDP, students write a simple client-server program where the server transfers a specific file to the client. For example, a server may deliver a certain MP3 file. Depending upon how the datagrams are routed and the traffic load on the medium, the datagrams often arrive out of order or may not arrive at all. However, for some content - such as certain media files - students discover the file remains relatively usable despite the unreliability.

Although not reflected in a programming assignment, the use of an unreliable protocol at the transport layer does introduce a discussion on how reliability *could* be implemented at the application layer. For example, Sun's Network File Service (NFS) and the Domain Name System (DNS) can be implemented using UDP at the transport layer. Any desired reliability must be provided by these applications.

## 6 Remote Method Invocation

Remote method invocation (RMI) is Java's version of a remote procedure call (RPC). RMI is a typical client-server architecture that allows clients to invoke methods on remote objects. In RMI, "remote" refers to a separate Java virtual machine (JVM). Hence, the client and server reside in separate JVMs. Although fairly easy to implement, code highlights and the steps involved in the development of even a simple RMI application are well beyond the page limitations of this paper. Therefore, this section will only provide an overview of an RMI assignment using the well-known Producer-Consumer problem.

The buffer which stores items that are produced and consumed is placed on the server. This server contains a remote object - `buffer` which accepts remote invocations of the `enter()` method from producers and the `remove()` method from consumers. A call to `enter(item)` inserts `item` into the buffer, invoking `remove()` removes an item from the buffer. This application is constructed using a separate thread for each producer and consumer, therefore it is important to use a thread-safe data structure such as `java.util.Vector`

for implementing the buffer. The specification for the assignment indicates that `enter()` and `remove()` are to be implemented as non-blocking and the Java exception mechanism is used to indicate if the buffer is either full or empty. Implementing blocking methods would most likely require coverage of thread synchronization in Java.

There are several other options for an RMI assignment. One obvious choice is to rewrite the chatroom application described in Section 4.3 using RMI.

## 7 Evaluation

Java has been an effective tool to teach network programming. There is a rich API for the construction of connection-oriented and connection-less sockets that allows students to easily write client-server applications. Furthermore, RMI allows the architecture of distributed applications allowing a thread in one Java virtual machine to invoke methods on remote objects residing in other virtual machines. The outcomes of using the Java networking API to teach network programming concepts will be evaluated according to the goals outlined in Section 2.

### 7.1 Client-Server Model

All assignments are client-server thus providing students a solid understanding of this paradigm. This is helped by the `java.net` API distinguishing client and server sockets. RMI is also based on the client-server model where the server provides the remote object whose methods are invoked by client applications. Furthermore, by providing several different unrelated assignments – as opposed to a single incremental project – students are exposed to many client-server applications that are all fundamentally different from one another.

### 7.2 Protocols

Students understand the role of protocols early on in this curriculum. Despite only implementing a small subset of the HTTP protocol and the simplicity of the VSMTP protocol, many students developed protocols for the chatroom even without it being a formal part of the assignment. Students cited the use of a protocol made for cleaner designs and provided a *state* which made debugging the client or server easier. Many of the protocols designed for the file transfer assignment were quite robust, handling different failure situations and providing a wide range of services.

RMI provides the opportunity to study a distributed object protocol, a contrast to the socket-based protocols of the earlier assignments.

### 7.3 Performance

Performance issues are addressed throughout these assignments. Because of the multithreaded nature of client-server programming, students understand how different threading models can lead to different performance results. For example, a server that creates a separate thread to service each incoming connection may quickly become overloaded if there is a sudden increase in connections. Thread pools can limit the number of concurrent connections on a server, providing a mechanism for avoiding server overload. The role of the cache in the web server is also evident as a performance benefit. Also, the mechanism by which files are read from the local file system has an impact on performance. Students saw significant benefits by adopting a buffered input stream over the non-buffered method. Performance is also highlighted when contrasting the TCP and UDP protocols.

There are some performance issues that are not addressed in this curriculum, for example the size of the congestion window used by TCP. However, the API does provide access to this and other performance-related parameters.

### 7.4 Security

Although not explicitly stated as a goal, security issues are covered throughout the computer networks class. However, its coverage has not been as effective as desired in the programming curriculum presented in this paper. For example, students *know* that their chatroom application isn't secure, however the proposed outline does not provide sufficient coverage of security mechanisms such as secure sockets (SSL). Significant coverage of security issues such as public and private key encryption is presented in lectures, however no practical programming exercises are provided. Java does provide SSL and it is to be incorporated into the chatroom assignment when the network course is offered in the Spring of 2002.

## References

- [1] Parameswaran, M., Susarla, A., and Whinston, A. B. P2p networking: An information-sharing alternative. *IEEE Computer* 34, 7 (June 2001), 31–37.
- [2] Stiller, E., and LeBlanc, C. Teaching client server computing in the context of curricula 2001. *The Journal of Computing in Small Colleges* 16, 4 (May 2001), 122–133.
- [3] Tanenbaum, A. *Computer Networks, Third Edition*. Prentice Hall, 1996.