A Simple Client-Server Application

The Java language has several classes that are used for network programming. They are in the java.net package and are adaptations of the corresponding structures in C. The C structures were introduced in the early 1980s by researchers in Berkeley while working with the UNIX operating system. The **Socket** class is used for connecting client computers to a network, and the **ServerSocket** class is used by servers to wait for incoming requests from clients.

The **SocketImpl** class is an abstract class that is the super class of all classes that implement sockets. SocketImpl has four fields, the address of the *host* computer, the *file descriptor*, the *localport* on the client computer, and the *port* on the host computer to which the client is connected. The host address may be something like www.cnn.com, the Internet address of the Cable News Network. The port (an integer) could be 80, the standard port for accessing web pages on web servers.

The Socket class is a subclass of SocketImpl and is used to create a client connection. The name comes from a wall socket that is used to connect an electrical device, such as a lamp, to a source of electrical power. The connection can be over a local area network (LAN), the Internet, or even using the *local loop* within the computer itself. The local loop has the network address **127.0.0.1**. It is often given the name, *localhost*.[1]

When a socket is created and opened by a Java program, it uses the Transmission Control Protocol (TCP) /Internet Protocol (IP) or the User Datagram Protocol (UDP). TCP/IP is the principal network protocol architecture used on the Internet. UDP is simpler and used when network reliability is not a problem. TCP is a *stream* oriented protocol. That means that applications see input and output as streams of data rather than discrete packets or frames. Therefore programmers can treat network input and output in the same way as they do keyboard, screen and file I/O.

The World Wide Web primarily uses the Hypertext Transfer Protocol (HTTP). HTTP sits on top of TCP/IP and adds functionality needed for web actions such as sending requests, receiving responses and following hyperlinks from one web address to another. It is designed for rapid hops across the Internet and so keeps a connection open for just one transaction.

HTTP is said to be *stateless*. That means that a web server has no memory of its clients. Internet companies manage this either by depositing a *cookie* on the client's computer or by issuing a session identification number included in the *URL string*. For example, the following URL string was generated by the barnesandnoble.com server:

    http://www.barnesandnoble.com/bookstore.asp?userid=0FJHK58GK6

The userid for this specific session is 0FJHK58GK6. It follows the user as he or she moves around the web site. However, it is dropped when the user clicks on the Back button. Users that use the Back button and do not accept cookies may lose the contents of their shopping carts.

Web browsers such as Internet Explorer and Netscape are configured for HTTP. When you use one of these browsers, it will open a client socket and send a request to the URL (Uniform Resource Locator) address given.

---

[1] In Windows 2000 or XP, you can set localhost as an Environment Variable. Go into **Settings/Control Panel/System/Advanced/System Variables.** Choose **New** and then enter localhost as the Variable name and 127.0.0.1 as the Variable value. In Windows 98, use Windows Explorer to find Autoexec.bat. It is in the C:\System folder. Edit it and add the line SET localhost=127.0.0.1. When you next boot up your computer, this file will be executed and will set the environment variable.

When the server sends back a web page, the browser formats it for display on your computer. The formatting instructions are written in HyperText Markup Language (HTML). The World Wide Web Consortium (W3C) publishes recommendations for browser and web page designers to follow. W3C has issued a number of updates and is now working on Extensible HyperText Markup Language (XHTML). XHTML "is a family of current and future document types and modules that reproduce, subset, and extend HTML, reformulated in XML."[2] (XML stands for Extensible Markup Language.)

We will use a server socket in a simple application that functions as a server on a single machine using the local loop. It can receive requests from a web page sent by a browser, process the request, and then send a response back to the browser. However, before looking at this program, we will see how to use a client's socket to obtain information from the Internet.

Using a Java program to access the date and time.

A simple Java program can be created that will connect to one of the servers maintained by the National Institute of Standards and Technology (NIST). NIST has several atomic clocks that are the most accurate ones in the US. (The world clock is in Paris, France.) These clocks are kept synchronized and can be accessed by anyone using the Internet. NIST has several sites in this country. The one used by the program below is in Gaithersburg, Maryland. Its URL is time.nist.gov. NIST keeps this site open all the time; the port that services date and time requests is 13.

As mentioned above, the Socket class is in java.net, which must be imported into the program. Also just about anything that you do with networks can throw an exception, so one will have to be caught or re-thrown. The creation of an instance of a socket throws an IOException and an UnknownHostException. The latter is a subclass of the former, therefore it is only necessary to catch the first.

The first thing that NIST sends is a blank line. The second thing is the date and time using Greenwich Mean Time (GMT). The following is a sample of the output from the program.

```
    52780 03-05-21 15:51:53 50 0 0 502.7 UTC<NIST> *
```

When a new instance of a socket is created, it is associated with an I/O stream. We can use both getInputStream () and getOutputStream () in order to use this stream. As usual, we need a BufferedReader and a PrintWriter to use them efficiently.

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

public class NIST
{   public static void main (String [] args)
    {   try
        {
            // Create an instance of a stream socket connected to NIST on port 13.
            Socket socket = new Socket ("time.nist.gov", 13);

            // Get a BufferedReader to read data from the socket's InputStream.
            BufferedReader reader = new BufferedReader (new InputStreamReader
                socket.getInputStream ()));
```

[2] http://www.w3.org/MarkUp/

```
            // Read two lines from the BufferedReader and display them in the console window.
            for (int count = 0; count < 2; count++)
            {
                String time = reader.readLine ();
                System.out.println (time);
            }
        } catch (IOException e) {System.out.println ("Network error." + e);}
    }
} // NIST
```

Servers

A *server* is a computer process that handles requests from *client* computers. It requires special software to do so. There are a number of commercial servers available from companies such as Microsoft, IBM and SUN. There is also an open-source server created by the Apache open software project. This is an international consortium that continually modifies and improves software. The software is then freely shared. However commercial users need to get a license. The Apache server is called *Tomcat* and is available from http://jakarta.apache.org/tomcat/tomcat-4.0-doc/.

Before we use a full strength server, however, we will look at a very simple version that can be used on any computer with a Java interpreter. It was developed by Cathy Zura and extended by myself. It does only a fraction of the work that Tomcat does, but it demonstrates some of the things that a server must do.

This server first gets a port from the user. This can be a default port, 8080 in this example, or it can be some other number. In any case, the port chosen must be the same one that will be used by the client's web page. Next the server gets an instance of the ServerSocket class. This class is used for sockets on a server to wait for a request from a client.

```
    ServerSocket serverSocket = new ServerSocket (port);
```

The server program now is ready to receive a request and act upon it. When it receives a request, it will *accept* it with the following code:

```
    Socket clientSocket = serverSocket.accept ();
```

This completes the connection between the server and this particular client.

The next thing it does is to create a new instance of the Server class. This class is a thread and can be created and started with one command.

```
    new Server (clientSocket).start ();
```

All this is done within an infinite loop (while (true)). This way, the server's socket will stay open for as long as it is needed to receive web page requests. To close the server program, you have to click on the X in the upper right hand corner of the console window. The full code for the WebServer class follows:

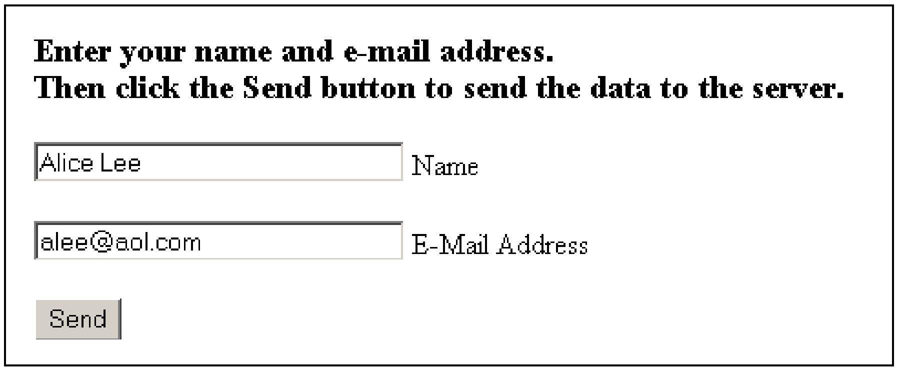
```
/**
The Web Server opens a port and gets a new ServerSocket. When a web page client opens a socket on the same port, it accepts the connection and creates a thread to handle it. It also keeps a count of the number of threads created.
**/
import java.io.*;
import java.net.*;                  // The Socket classes are in the java.net package.
public class WebServer
{
	public static void main (String [] args)
	{
		final int DefaultPort = 8080;
		try
		{
			// Set the port that the server will listen on.
			BufferedReader stdin = new BufferedReader (new InputStreamReader (System.in));
			System.out.print ("Port: ");
			String portStr = stdin.readLine ();
			int port;
			if (portStr.equals ("")) port = DefaultPort;	// Use the default port.
			else port = Integer.parseInt (portStr);		// Use a different port.

			int count = 1;  // Track the number of clients.
			ServerSocket serverSocket = new ServerSocket (port);
			while (true)
			{
				Socket clientSocket = serverSocket.accept ();	// Respond to the client.
				System.out.println ("Client " + count + " starting:");
				new Server (clientSocket).start ();
				count ++;
			}
		} catch (IOException e) {System.out.println ("IO Exception");}
		catch (NumberFormatException e) {System.out.println ("Number error");}
	} //main
} // WebServer
```

Clients

The server is designed to connect with a client through a web page. The client downloads the web page from the server and then fills out a *form* on the page. This might be an order form for buying a product or a registration form that will sign a client up for some service. The following is a sample form that only requests the client's name and e-mail address.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
	<head><title>E-Mail Form</title></head>
	<body>
		<h3>Enter your name and e-mail address.
		<br />Then click the Send button to send the data to the server.</h3>
		<form method = "get" action="http://localhost:8080/EmailProcessor">
			<p><input type = "text" name = "name" value = "" size = 30 /> Name </p>
			<p><input type = "text" name = "email" value = "" size = 30 /> E-Mail Address </p>
			<p><input type="submit" value="Send" /></p>
		</form>
	</body>
</html>
```

Displayed by a browser, the form looks as follows:


Enter your name and e-mail address.
Then click the Send button to send the data to the server.

Alice Lee Name

alee@aol.com E-Mail Address

The form uses the default port number, 8080. If the server is assigned a different port, the web page will not be processed. Since the web page is normally downloaded from the server before it is filled out, the programmer knows the port that will be used. (The standard port for web pages on the Internet is 80.)

Server Class

The web page form is used to send a request to the server. When that request is received, the WebServer class creates a thread to process the request. Here this is done by a class simply called Server. This class performs several tasks.

The Server class first uses the clientSocket sent to it by the WebServer class to get a BufferedReader and then reads the first line. This line is the URL string created by the browser. The one generated from the form above is

```
GET /EmailProcessor?name=Alice+Lee&email=alee%40aol.com HTTP/1.1.
```

The browser creates part of the string from the method and action line in the form and the rest from the input data. It uses the action statement of the form,

```
action="http://localhost:8080/EmailProcessor"
```

to find the address of the server, here the local host with port number 8080. It also uses the action statement to find the name of the program on the server that is to process the request.

The URL string, then, starts with the method, here GET, followed by a space and a '/'. The processor name is next. It is separated from the rest of the data by a question mark, '?'. After all the data from the form, the browser adds a space and the version of HTTP used by the browser, here HTTP/1.1. The request data is taken from the input boxes of the form. It can also come from other form objects such as list boxes or radio buttons.

The first box contributes 'name=Alice+Lee' to the URL string, and 'email=alee%40aol.com' comes from the second box. In general, the URL string is coded with all spaces replaced by the '+' sign, and data items separated by ampersands (&). Letters and digits are not changed, but a number of other characters are replaced by the percent sign (%) followed by the ascii code for the character. For example, the 'at' sign (@) is replaced by %40 (in Netscape, but not Internet Explorer).

A StringTokenizer is used to separate the string into its parts. It is instantiated by

```
StringTokenizer tokenizer = new StringTokenizer (urlString, "/?&= ");
```

where urlString contains the data above. The delimiters for the tokenizer are '/', '?', '&', '=', and space. They are all to be discarded. The '+' sign is retained in order to determine the location of spaces in the data. After the method and processor name are saved, the tokenizer is sent to a class called Request that uses it to retrieve and store the remainder of the data. This class will be discussed later. The server also gets an instance of the Response class. It will be used to get a PrintWriter for sending responses back to the client.

When the Request and Response classes have been created, the server is ready to create an instance of the class that is to process the data. In this example, it is called EmailProcessor. It has saved the name previously, so all it has to do is instantiate it. This is done using the method, newInstance (), which is in Class, a subclass of Object. First it is necessary to initialize the class, and this is done with Class.forName (processName). forName is a static method that returns the Class object associated with the class or interface with the given string name, here processName.

The server then has to start the processor. For this, it must know the name of the method in the processing class that does the work. For Java servlets, there are two such methods, doGet and doPost. This example uses a single method called process. It has two parameters, the Request and Response classes. Every program that is instantiated by the server has to have this method, so it is included in an abstract class called WebRequestProcessor. All processor classes must extend this class.

```
package client_server;

// An abstract class that defines a set of processing classes.
public abstract class WebRequestProcessor
{
    // An abstract method that processes a request.
    public abstract void process (Request request, Response response);
} // WebRequestProcessor
```

Note that instead of a class, the above could just as easily be an interface. It would work the same way.

The lines of code in the server now are

```
    WebRequestProcessor processor =
        (WebRequestProcessor) Class.forName (processName).newInstance ();
    processor.process (request, response);
```

As described above, processor is a new instance of a WebRequestProcessor class with the name, processName, obtained from the URLString. The method that does the work is called process, and it has instances of the Request and Response classes as parameters.

```
/**
The Server class is a thread. It reads the URL string from the client's socket. It then gets a
StringTokenizer for the string and uses the tokenizer to parse it. The first two tokens in the string are the
method (get or post) and the name of the class that is to process the request. The remainder of the tokens
is sent to the Request class for further processing. The process method in the processor class is then
started.
**/

class Server extends Thread
{
    WebRequestProcessor processor;
    Socket clientSocket;

    public Server (Socket clientSocket) {this.clientSocket = clientSocket;}
```

```
	public void run ()
	{
		String urlString, method, processName;
		try
		{
			// Get an input stream for the client’s socket.
			InputStream inStream = clientSocket.getInputStream ();
			BufferedReader in = new BufferedReader (new InputStreamReader (inStream));

			// Read the URL string and tokenize it.
			urlString = in.readLine();
			System.out.println (urlString);
			StringTokenizer tokenizer = new StringTokenizer(urlString, "/?&= ");

			// Get the first two tokens and send the rest of the tokens to the Request class.
			method = tokenizer.nextToken();
			System.out.println (method);
			processName = tokenizer.nextToken();
			System.out.println (processName);

			// Set up the Request and Response clases.
			Request request = new Request (tokenizer, method);
			OutputStream outStream = clientSocket.getOutputStream ();
			Response response = new Response (outStream);

			// Get a new instance of the processor class and start processing.
			System.out.println ("Processing: " + processName);
			processor = (WebRequestProcessor) Class.forName (processName).newInstance ();
			processor.process (request, response);

			clientSocket.close (); // Close the client's socket.
		} catch (IOException e) {System.out.println ("Processing Exception");}
		catch (ClassNotFoundException ce) {System.out.println("Class not found");}
		catch (InstantiationException ie) {System.out.println ("Instantiation exception");}
		catch (IllegalAccessException ie) {System.out.println ("Illegal Access Exception");}
		System.out.println("Client at "+ clientSocket.getInetAddress() + " disconnected");
	} // run
} // class Server
```

EMailProcessor

The program that processes the web request must extend the WebRequestProcessor class and contain a method called, process. It can have other methods and classes like any Java program. It uses methods in the Request and Response classes to get the data and return a web page to the client.

The Request class has a method getParameter (parameterName) that returns the contents of the box in the form with the given name. To get the values of the parameters, the program uses

```
	String name = request.getParameter ("name");
	String email = request.getParameter ("email");
```

In the example, name will have the value "Alice Lee" and email the value "alee@aol.com". (Among other things, the Request class changes all '+' signs to spaces.)

The Response class has a method called getWriter () that returns a PrintWriter object. This is used to send a web page back to the client. The web page will be displayed by a browser, so the output of the program consists of strings of html statements. Web pages generally begin with a heading containing the title and end with the lines "</body>" and "</html>". Since this is common to most web pages, it can be included in a separate class, here called Page.

```
// Class with static methods that add standard lines to the html output page.
class Page
{
	public static void createHeader (PrintWriter out, String title)
	{
		out.println ("<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0 Transitional//EN'>");
		out.println ("<html>");
		out.println ("<head>");
		out.println ("<title>" + title + "</title>");
		out.println ("</head>");
		out.println ("<body>");
	} // createHeader

	public static void createFooter (PrintWriter out){out.println ("</body></html>");}
} // class Page
```

The rest of the html is written by the process method in the main class.

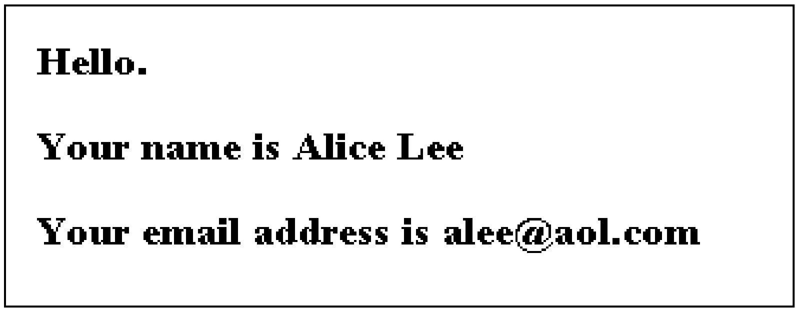
```
/* EmailProcessor processes a request from a client's web page. It responds to the request by sending
back a second web page echoing the name and email address. */

import java.io.*;

// The Request, Response and WebRequestProcessor classes are in the client_server package.
import client_server.Request;
import client_server.Response;
import client_server.WebRequestProcessor;

public class EmailProcessor extends WebRequestProcessor
{
	public void process (Request request, Response response)
	{
		// Get the request parameters with types name and email.
		String name = request.getParameter ("name");
		String email = request.getParameter ("email");

		// Get a PrintWriter object and respond to the request.
		PrintWriter out = response.getWriter ();
		Page.createHeader (out, "Test Data");
		out.println ("<h3>Hello.</h3>");
		out.println ("<h3>Your name is " + name + "</h3>");
		out.println ("<h3>Your email address is " + email + "</h3>");
```

```
            Page.createFooter (out);
        }
} // EmailProcessor
```

As you can see, this class only echoes the data on the form. It could also save the data to a file or store it in a database. If the data contained prices and quantities it could calculate and return the bill for an order. Also note that the output consists only of strings. All the html code must be contained in quotation marks, including tags such as <html> and <h3>. The resulting web page sent back to the client is

```
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0 Transitional//EN'>
<html>
<head>
<title>Test Data</title>
</head>
<body>
<h3>Hello.</h3>
<h3>Your name is Alice Lee</h3>
<h3>Your email address is alee@aol.com</h3>
</body></html>
```

The page when displayed looks like the following:



Hello.

Your name is Alice Lee

Your email address is alee@aol.com

Request Class

The Request class is used to store the data entered into the form by the client. It has methods that return either a parameter value, given its name, or the entire list of values. Among other things, it changes all plus signs to spaces and hex values to the character given by that value. It also discards the HTTP/1.1 at the end of the string.

The client data consists of a set of name-value pairs. These are stored in a small wrapper class called Param. Each pair holds one pair of the parameters.

```
class Param // A class that stores a name-value pair of parameters.
{
    private String name, value;

    Param (String n, String v){name = n; value = v;} //constructor
    protected String getName(){return name;}
    protected String getValue(){return value;}
}//Param
```

The constructor for the Request class has the StringTokenizer as a parameter and uses it to retrieve the data from the URL string. All the pairs are added to a Vector, except for the HTTP/1.1 at the end.

```
/* The Request class uses the StringTokenizer created in the WebServer class to create a Vector with the
parameters encoded in the URL string.  */
public class Request
{
    private Vector parameters;
    private String method;

    /* Use the tokenizer to get the name and value parameters and store them in the Properties table.  */
    public Request (StringTokenizer tokenizer, String method)
    {
        String name, value;
        this.method = method;
        int size = 0;
        parameters = new Vector ();
        while (tokenizer.hasMoreTokens())
        {
            name = tokenizer.nextToken ();
            value = tokenizer.nextToken ();
            System.out.println(name + " " + value);
            if (!name.equals ("HTTP"))
            {
                value = replaceHexCode (value);
                Param param = new Param (name, value);
                parameters.addElement (param);
                size++;
            }
        }
    } // constructor

    /* Some characters are sent in hex by the URL string.  They are preceded by a '%' sign.  The
   following method replaces them with the corresponding characters.  It also replaces the '+' sign in the
   string with a space.  */
    private String replaceHexCode (String value)
    {
        value = value.replace ('+', ' ');
        int index = value.indexOf ('%');
        while (index >= 0)  // If no such index occurs in the string, the method returns -1.
        {   try
            {   // Get the hex code and covert it to decimal.
                String hex = value.substring (index+1, index+3);
                int decimal = Integer.parseInt (hex, 16);

                /* Get the character with the decimal code, change the characters '<' to &lt,
                and '>' to &gt.  Include the resulting string in the value parameter. */
                char ch = (char) decimal;
                String code;
                if (ch == '<') code = "&lt";
```

```
                    else if (ch == '>') code = "&gt";
                    else code = String.valueOf (ch);
                    value = value.substring (0, index) + code + value.substring (index+3);
               } catch (NumberFormatException e) {}
               index = value.indexOf ('%', index+1);
          }
          return value;
     } // replaceHexCode
```

There are several other methods in the Request class. The first is used to return the value of a property given its name.

```
     // Given a name, return the corresponding value.
     public String getParameter (String name)
     {
          int index = 0;
          boolean found = false;
          Param param = null;
          while (index < parameters.size () && !found)
          {
               param = (Param) parameters.elementAt (index);
               if (param.getName ().equals (name)) found = true;
               else index ++;
          }
          if (found) return param.getValue ();
          else return null;
     } // getParameter
```

The second method returns a list of all the values in the Vector that have the same name. This list is an array of Strings. The method, getParameterValues (), is useful for retrieving data from check boxes, list boxes and radio buttons.

```
     // Return an array containing just the parameter values, not the names.
     public String [] getParameterValues (String name)
     {
          String [] values = new String [10];
          int index = 0;
          Param param;
          for (int count = 0; count < parameters.size (); count ++)
          {
               param = (Param) parameters.elementAt (count);
               if (param.getName ().equals (name))
                    values [index] = param.getValue ();
               index++;
          }
          return values;
     } // getParameterValues
```

Response Class

The Response class is very short. Its main function is to return a PrintWriter for the OutputStream.

```
package client_server;
import java.io.*;

// The Response class uses the OutputStream sent by the Server to get a PrintWriter object.
public class Response
{
    private PrintWriter writer;

    public Response (OutputStream output)
    {
        /* The 'true' in the PrintWriter constructor indicates whether or not to use autoflush.  If it is
        omitted, the buffer is not automatically emptied at the end of each line. */
        writer = new PrintWriter (new OutputStreamWriter (output), true);
    } // constructor

    public PrintWriter getWriter () {return writer;}
} // Response
```

Using the WebServer

In order to use the WebServer described above, you should follow the directions outlined below.

1. Create a folder to hold all your files.
2. Create an html page and save it in the folder. You can use Notepad or an IDE like JCreator.
3. Download the WebServer files from http://csis.pace.edu/~wolf/Documents/ and copy them to the folder.
4. Load the files into your IDE (JCreator) and compile them. Compile the Request and Response classes first, then the WebRequestProcessor. After that you can compile the Server and the WebServer classes.
5. Create a program to process the data such as the EmailProcessor program listed above. Store it in the same folder and compile it.
6. Run the WebServer class. If you are using port 8080, just hit the return key. Otherwise enter the number of the port you are using.
7. Open InternetExplorer or Netscape, browse to your folder, and load the html page with the form.
8. Fill in the form and send it to the server. Observe the printout on the console (black) screen.
9. View the html page returned to the browser.
10. Use the Back button to try different data.

References

1. Marty Hall & Larry Brown, *Core Servlets and Java Server Pages*, First Edition , Sun Microsystems Press/Prentice-Hall PTR Book, 2003.
2. Elliotte Rusty Harold, *Java Network Programming*, O'Reilly & Associates, Inc., 2000.
3. Karl Moss, *Java Servlets Developer's Guide*, McGraw-Hill/Osborne, 2002.
4. William Stallings, *Data & Computer Communications*, 6th Edition, Prentice-Hall, Inc., 2000.
5. Cathy Zura, Class Notes for CS 396N, http://matrix.csis.pace.edu/~czura/cs396n/, 2003.