

**HTTP Protocol and Writing Java Client to Interact with Web Server****Objectives:**

1. Explore the HTTP Protocol with `telnet`.
2. Write Java client to interact with web server.
3. Explore sending and receiving bytes in Java.

**Preparation:** Read and study 160-164 in our Distributed Systems text on HTTP response-reply protocol.

Bring the Java and Distributed Systems texts to lab.

**Introduction:**

The HyperText Transfer Protocol (HTTP) is the main protocol of the World Wide Web.

“The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods (commands). A feature of HTTP is the typing of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects common usage of the protocol referred to as 'HTTP/1.0'.”

The above quote is from the Internet Engineering Task Force's (IETF) *Request For Comments* (RFC) 1945 document on HTTP/1.0 written by Tim Berners-Lee and others in May 1996. The RFC is very readable and worth reading. To acquire a feel for what is in a RFC, we recommend you read the first ten pages or so (out of 59). Available at URL:

<http://www.w3.org/Protocols/rfc1945/rfc1945>

The main World Wide Web Consortium (W3C) web page on HTTP is at

<http://www.w3.org/Protocols/>

**Laboratory Assignment:****1. HTTP Protocol with `telnet`**

Make sure you have read pages 160-164 in the Distributed Systems Text on HTTP response-reply protocol before starting this part of lab.

Most of us have used `telnet` to logon to a remote host. However, `telnet` is a richly endowed tool and can do much more. The brave should read the man page on `telnet`. In this exercise we will start `telnet` by connecting to a server in the following form:

```
telnet <server> <port>
```

This will allow us to issue commands to the server.

The server can be any server including a web server such as `www.eg.bucknell.edu`. The port number for a web server is 80. Try the following:

```
telnet www.eg.bucknell.edu 80
```

which connects you to the Engineering web server.

Try the following HTTP command after `telnet` connects: (in RFC called “Simple-Request”)

```
GET http://www.eg.bucknell.edu/~cs355/simple.html
```

`telnet` should display the html code for that web page. Notice that the protocol is case sensitive. The command must be “GET” not “get”. Also, you **must** type carefully! `telnet` is unforgiving.

Now try the following HTTP command: (in RFC an example of “Full-Request”)

```
GET http://www.eg.bucknell.edu/~cs355/simple.html HTTP/1.0
```

You must press **Return** key twice.

Now, `telnet` should display several lines before the html code for the web page. Make sure you read the first few lines that are returned. What is the **status code** returned? \_\_\_\_

See page 163 in Distributed Systems text if you don’t know about the **status code**.

The **GET** command requires a legal URL or a legal URL and a Mime type field, e.g., **text/plain**, **text/http/1.0**, **image/gif** or **image/jpeg**, of what is requested. See page 164 for discussion on **Mime type** field.

Try the following variations on the above **GET** command and record the returned **status code**.

- |   |  |
|---|--|
| 1. Remove: <b>simple.html</b>                           | What is <b>status code</b> ? ____ What is printed? _____ |
| 2. Remove: <b>/simple.html</b>                          | What is <b>status code</b> ? ____                        |
| 3. Remove: <b>http://</b>                               | What is <b>status code</b> ? ____                        |
| 4. Replace: <b>simple.html</b> with <b>simple2.html</b> | What is <b>status code</b> ? ____                        |

Try some other commands and alternatives.

`telnet` is a useful tool for determining the correct commands to place in your Java program when you want your Java program to communicate with a server.

## 2. Writing Java Client to Interact with Web Server

In this exercise you are to write a Java application which will have similar behavior as in the previous exercise, i.e., connect to a stream socket (TCP-based) on a port of a server and issue commands on the output stream and receive the replies on the input stream.

Your Java application (client) should ask for the host name, the port number and open a socket to that port on that host. You should be able to issue commands from the keyboard to the server and print the response to the screen.

## Some Details

1. If you issue an improper command to a web server, it will respond with a text response and close the connection. By default, the server will also close the connection on proper commands unless you request a http 1.1 session. You don't need to open an http 1.1 session in this lab.
2. To communicate with a web server, you can *not* use the **ObjectInputStream** and **ObjectOutputStream** streams of earlier labs. Your Java client is no longer communicating with a Java server that understands Java serialized objects i.e., ones that are sophisticatedly packed including object name, version number, data fields, etc.  
Since a web server needs to communicate with a client written in any programming language, e.g., C, C++, Visual Basic or Perl, the web server's approach must be plain vanilla. Therefore, a web server expects a stream of bytes (in the jargon called *octets* - any 8-bit sequence of data) and responses with bytes embedded with end of lines.
3. To send and receive bytes, you should use the streams **DataOutputStream** and **DataInputStream**.
4. A good way to send the Java String **s1** as a bunch of bytes is **out.writeBytes( s1 )**. You will need to append a "**\n**" to the end of the string before sending.
5. A *line* of the response from the server can be handled with **s2 = in.readLine()**; where **s2** is a Java String. You will need to loop and continue to read lines and print lines to the screen until **s2 == null**. When I used **readLine()**, I received a "deprecated API" message. Ignore this message for this lab.

**Test 1:** To test your client, do the following HTTP command after connecting to host `www.eg.bucknell.edu` on port 80:

```
GET http://www.eg.bucknell.edu/~cs355/simple.html
```

You should receive the same output as you did in exercise 1.

**Test 2:** Alter your program to append " HTTP/1.0\n\n" to the end of the String instead of a "\n". And redo Test 1.

You should receive the same output as you did in that part of exercise 1.

## Hand In:

For exercise 1, hand in the five status codes.

For exercise 2, hand in the Java code of your client and the results of running the two tests.