Lab 1: Web Server

In this lab, you will learn the basics of socket programming for TCP connections in Python: how to create a socket, bind it to a specific address and port, and send and receive an HTTP packet. You will also learn some basics of the HTTP header format.

You will develop a web server that handles one HTTP request at a time. Your web server should accept and parse the HTTP request, get the requested file from the server's file system, create an HTTP response message consisting of the requested file (preceded by header lines), and then send the response directly to the client. If the requested file is not present on the server, the server should send an HTTP 404 "Not Found" message back to the client.

Security

Note that this Web server will allow its clients to access ANY file that they can name. To prevent strangers from snooping on your files, you should include some basic protections:

1. If the requested filename begins with one or more "/" characters, those slashes should be removed.
2. If the requested filename contains the sequence "/../ anywhere in the string, your server should return an HTTP 404 "Not Found" response.

Code

Below you will find the skeleton code for the Web server. You are to complete the skeleton code. The places where you need to fill in code are marked with `#Fill in`. Each place may require one or more lines of code.

We recommend that you write your server to use a port number equal to your numeric user ID (type "id" on Knuth or Wilkes). That will ensure that you're using a unique port that won't conflict with anybody else in the course.

Running the Server

Put an HTML file (e.g., HelloWorld.html) in the same directory that the server is in. On Wilkes, change into that directory and run the server program. Determine the IP address of the Wilkes that is running the server (e.g., 128.238.251.26). From another host, open a browser and provide the corresponding URL. For example:

http://128.238.251.26:6789/HelloWorld.html

where 6789 is the port number you hardwired into your server code and 'HelloWorld.html' is the name of the file you placed in the server directory (case matters!). The browser should then display the contents of HelloWorld.html. Don't forget the port number; if you leave it out the browser will use the wrong port and you won't get a result.

Then try to get a file that is not present at the server. You should get a 404 "Not Found" message.

What to Hand in

You will hand in the complete server code along with the screen shots of your client browser, verifying that you actually receive the contents of the HTML file from the server. Use cs125submit on Knuth or Wilkes to hand the code in.

Skeleton Python Code for the Web Server

```
from socket import *       # Get socket functions

serverSocket = socket(AF_INET, SOCK_STREAM)
#  Prepare a server socket on a particular port
#Fill in code to set up the port
while True:
    # Establish the connection
    print('Ready to serve...')
    connectionSocket, addr =   #Fill in code to get a connection
    try:
        message =    # Fill in code to read GET request
        filename = message.split()[1]
        # Fill in security code
        f = open(filename)
        outputdata = # Fill in code to read data from the file
        # Send HTTP header line(s) into socket
        # Fill in code to send header(s)
        # Send the content of the requested file to the client
        for i in range(0, len(outputdata)):
            connectionSocket.send(outputdata[i].encode())
        connectionSocket.send("\r\n".encode())
        connectionSocket.close()
    except IOError:
        #Send response message for file not found
        # Fill in
        # Close client socket
        # Fill in
```

Optional Exercises

1. Currently, the Web server reads the entire file into memory before sending it. That's an inefficient approach if the file is large, and if the file is *really* large the server will break. Fix the server to read and send in small chunks (4K-8K is a good size).

2. Currently, the Web server handles only one HTTP request at a time. Implement a multithreaded server that is capable of serving multiple requests simultaneously. Using Python threading, first create a main thread in which your modified server listens for clients at a fixed port. When it receives a TCP connection request from a client, it will set up the TCP connection through another port and service the client request in a separate thread. There will be a separate TCP connection in a separate thread for each request/response pair.

3. Extend the server to support HTTP/1.1's ability to fetch multiple files over one connection. Note that you can do so only if the original GET specifies HTTP/1.1. You will need to generate a Content-Length header, loop to service multiple requests, and respond appropriately when the client closes the connection.
4. Instead of using a browser, write your own HTTP client to test your server. Your client will connect to the server using a TCP connection, send an HTTP request to the server, and display the server response as an output. You can assume that the HTTP request sent is a GET method.
   The client should take command-line arguments specifying the server IP address or host name, the port at which the server is listening, and the path at which the requested object is stored at the server. The following is a sample input command format for running the client.

```
./client.py server_host server_port filename
```