NTNU
Norwegian University of
Science and Technology
Department of Telematics

TTM4100

Communication – Services and Networks

Programming lab 1: “Web Server”

There are three programming labs in this course, and you have to do at least two of them. The exercises are designed to focus on learning and understanding – not debugging code. The labs are good exercise for the upcoming project, as they cover some of the same tasks. All lab-answers should be delivered on itslearning for review. Any questions about the exercises can be posted on the forum.

Deadline of submission: 07.02.2016

Lab 1: Web Server Lab

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, as well as send and receive a HTTP packet. You will also learn some basics of 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 in the server, the server should send an HTTP "404 Not Found" message back to the client.

N.B. It is strongly recommended to read **Sect. 2.7.2** of the textbook before (and while) doing this programming lab.

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 START` and `# FILL IN END`. Each place may require one or more lines of code. The same code can also be found in the file "Skeleton WebServer.py".

Running the Server

Put an HTML file (e.g., HelloWorld.html) in the same directory that the server is in. If you do not know how to create an HTML file, there are many examples in the web; a possible one is: http://www.w3schools.com/html/html_editors.asp.

Run the server program. Determine the IP address of the host 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

'HelloWorld.html' is the name of the file you placed in the server directory. Note also the use of the port number after the colon. You need to replace this port number with whatever port you have used in the server code. In the above example, we have used the port number 6789. The browser should then display the contents of HelloWorld.html. If you omit ":6789", the browser will assume port 80 and you will get the web page from the server only if your server is listening at port 80.

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 screenshots of your client browser, verifying that you actually receive the contents of the HTML file from the server.

Skeleton Python Code for the Web Server

The same code can also be found in the file “Skeleton WebServer.py”.

```
# Import socket module
from socket import *

# Create a TCP server socket
#(AF_INET is used for IPv4 protocols)
#(SOCK_STREAM is used for TCP)
serverSocket = socket(AF_INET, SOCK_STREAM)

# Prepare a server socket
# FILL IN START

# Assign a port number
serverPort =

# Bind the socket to server address and server port


# Listen to at most 1 connection at a time


# FILL IN END

# Server should be up and running and listening to the incoming connections
while True:
	print 'Ready to serve...'

	# Set up a new connection from the client
	connectionSocket, addr = # FILL IN START		# FILL IN END

	# If an exception occurs during the execution of try clause
	# the rest of the clause is skipped
	# If the exception type matches the word after except
	# the except clause is executed
	try:
		# Receives the request message from the client
		message =  # FILL IN START	# FILL IN END

		# Extract the path of the requested object from the message
		# The path is the second part of HTTP header, identified by [1]
		filepath = message.split()[1]

		# Because the extracted path of the HTTP request includes
		# a character '\', we read the path from the second character
		f = open(filepath[1:])

		# Read the file "f" and store the entire content of
		# the requested file in a temporary buffer.
		# Check the python documentation for how to read from a file.
		outputdata = # FILL IN START		# FILL IN END
```

```
            # Send the HTTP response header line to the connection socket
            # Format: "HTTP/1.1 *code-for-successful-request*\r\n\r\n"
            # FILL IN START

            # FILL IN END

            # Send the content of the requested file to the connection socket
            for i in range(0, len(outputdata)):
                  connectionSocket.send(outputdata[i])
            connectionSocket.send("\r\n")

            # Close the client connection socket
            connectionSocket.close()

      except IOError:
            # Send HTTP response message for file not found
            # Same format as above, but with code for "Not Found"
            # FILL IN START

            # FILL IN END
            connectionSocket.send("<html><head></head><body><h1>404 Not
            Found</h1></body></html>\r\n")

            # Close the client connection socket
            # FILL IN START

            # FILL IN END

serverSocket.close()
```

Optional Exercises

1. 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 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 services the client request in a separate thread. There will be a separate TCP connection in a separate thread for each request/response pair.

2. 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 an input command format to run the client.
   ```
   client.py server_host server_port filename
   ```