THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 4436A Networking: Principles, Protocols and Architectures

Assignment # 1: Socket Programming

Python 3.x must be used

Release Date: September 28, 2018 - Due Date: October 15, 2018 at 5:00 pm

You are required to complete the assignment individually. Required files must be submitted on OWL. Do not show your code to any other class members code. Do not put your code in any public domain. TAs will be available at 3C+4435 during lab hours every week and next week to answer any question that you may have. *Please note that you can attend only the Lab section that you are registered in*. You may ask questions related to the Lab only during Lab hours and office hours.

Due to the large enrollment in this course we will not be able to answer questions by email.

This assignment has 2 parts.

Report:

- Cover page including official name and Student ID
- File name: "LastNameMember1_LastNameMember2_Lab1.pdf"
- Add screenshots to each question

When you complete your assignment, please submit the **source code & other required files using OWL**. Although this is a programming assignment, you must comment the code so that it can be read and understood by another programmer that is not necessarily experienced with Python.

Two files need to be submitted:

1. PDF file with the report and results shown through screenshots.
2. Source code compressed into **.zip** format. Please **do not use** other compression formats.

The assignment consists of 2 parts:

Part 1: Develop a Pinging program (Socket Programming Assignment UDP) - pages 2-4
Part 2: Develop an HTTP Web Server (Server Socket Programming Assignment) - pages 5-8

Materials:

See Sections 2.7 of the textbook for an introduction to socket programming in Python. Furthermore, you can download the source code for the ping server from OWL to test your program. Additional examples (including most of this assignment) can be found at the textbook's website http://www.awl.com/kurose-ross.

Part A: UDP Pinger [50 Marks]

In this lab, you will learn the basics of socket programming for UDP in Python. You will learn how to send and receive datagram packets using UDP sockets and also, how to set a proper socket timeout. Throughout the lab, you will gain familiarity with a Ping application and its usefulness in computing statistics such as packet loss rate.

You will first study a simple Internet ping server written in the Python, and implement a corresponding client. The functionality provided by these programs is similar to the functionality provided by standard ping programs available in modern operating systems. However, these programs use a simpler protocol, UDP, rather than the standard Internet Control Message Protocol (ICMP) to communicate with each other. The ping protocol allows a client machine to send a packet of data to a remote machine, and have the remote machine return the data back to the client unchanged (an action referred to as echoing). Among other uses, the ping protocol allows hosts to determine round-trip times to other machines.

You are given the complete code for the Ping server below. Your task is to write the Ping client.

Server Code

The following code fully implements a ping server. You need to compile and run this code before running your client program. *You do not need to modify this code.*

In this server code, 20% of the client's packets are simulated to be lost. You should study this code carefully, as it will help you write your ping client.

```
# UDPPingerServer.py
# We will need the following module to generate randomized lost packets
import random
from socket import *

# Create a UDP socket
# Notice the use of SOCK_DGRAM for UDP packets
serverSocket = socket(AF_INET, SOCK_DGRAM)
# Assign IP address and port number to socket
serverSocket.bind(('', 12000))

while True:
    # Generate random number in the range of 0 to 10
    rand = random.randint(0, 10)
    # Receive the client packet along with the address it is coming from
    message, address = serverSocket.recvfrom(1024)
    # Capitalize the message from the client
    message = message.upper()
    # If rand is less is than 3, we consider the packet lost and do not respond
    if rand < 3:
        continue
    # Otherwise, the server responds
    serverSocket.sendto(message, address)
```

The server sits in an infinite loop listening for incoming UDP packets. When a packet comes in and if a randomized integer is greater than or equal to 3, the server simply capitalizes the encapsulated data and sends it back to the client.

Packet Loss

UDP provides applications with an unreliable transport service. Messages may get lost in the network due to router queue overflows, faulty hardware or some other reasons. Because packet loss is rare or even non-existent in typical campus networks, the server in this lab injects artificial loss to simulate the effects of network packet loss. The server creates a variable randomized integer which determines whether a particular incoming packet is lost or not.

Client Code

You need to implement the following client program.
The client should send 10 pings to the server. Because UDP is an unreliable protocol, a packet sent from the client to the server may be lost in the network, or vice versa. For this reason, the client cannot wait indefinitely for a reply to a ping message. You should get the client wait up to two seconds for a reply; if no reply is received within two seconds, your client program should assume that the packet was lost during transmission across the network. You will need to look up the Python documentation to find out how to set the timeout value on a datagram socket.

Specifically, your client program should
(1) send the ping message using UDP (Note: Unlike TCP, you do not need to establish a connection first, since UDP is a connectionless protocol.)
(2) print the response message from server, if any
(3) calculate and print the round trip time (RTT), in milliseconds, of each packet, if server responses
(4) otherwise, print "Request timed out"

During development, you should run the `UDPPingerServer.py` on your machine, and test your client by sending packets to *localhost* (or, 127.0.0.1). After you have fully debugged your code, you should see how your application communicates across the network with the ping server and ping client running on different machines.

Message Format

The ping messages in this lab are formatted in a simple way. The client message is one line, consisting of ASCII characters in the following format:

Ping *sequence_number time*

where *sequence_number* starts at 1 and progresses to 10 for each successive ping message sent by the client, and *time* is the time when the client sends the message.

What to Hand in

You will hand in the complete **client code** and **screenshots** at the client verifying that your ping program works as required. *[20 Marks]*

Along with the above specifications, your program should do also the following:

1. Currently, the program calculates the round-trip time for each packet and prints it out individually. Modify this to correspond to the way the standard ping program works. In this part, the client should send 200 pings to the server.
    - You will need to report the minimum, maximum, average, and standard deviation RTTs at the end of all pings from the client. *[5 Marks]*
    - In addition, you need to calculate and show the packet loss rate (in percentage). *[5 Marks]*
    - You need to visulize the histogram of the RTTS for the different packets. Create a histogram, which bins the values into ranges and count how many RTTs fall into each range. You may use different Python libraries for this part (e.g., matplotlib, Seaborn, numpy, …). *[8 Marks]*
2. Another similar application to the UDP Ping would be the UDP Heartbeat. The Heartbeat can be used to check if an application is up and running and to report one-way packet loss. The client sends a sequence number and current timestamp in the UDP packet to the server, which is listening for the Heartbeat (i.e., the UDP packets) of the client. Upon receiving the packets, the server calculates the time difference and reports any lost packets. If the Heartbeat packets are missing for some specified period of time, we can assume that the client application has stopped. Implement the UDP Heartbeat (both client and server). You will need to modify the given `UDPPingerServer.py`, and your UDP ping client. *[12 Marks]*

Part B: Web Server *[50 Marks]*

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.

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.

Running the Server

Put an HTML file (e.g., HelloWorld.html) in the same directory that the server is in. 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.

Skeleton Python Code for the Web Server

```
#import socket module

from socket import *

import sys # In order to terminate the program


serverSocket = socket(AF_INET, SOCK_STREAM)

#Prepare a sever socket

#Fill in start

#Fill in end

while True:

    #Establish the connection

    print('Ready to serve...')

    connectionSocket, addr =   #Fill in start              #Fill in end

    try:

        message =   #Fill in start          #Fill in end

        filename = message.split()[1]

        f = open(filename[1:])

        outputdata = #Fill in start       #Fill in end

        #Send one HTTP header line into socket

        #Fill in start

        #Fill in end

        #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 start
        #Fill in end

        #Close client socket

        #Fill in start

        #Fill in end

serverSocket.close()

sys.exit()#Terminate the program after sending the corresponding data
```

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. *[20 Marks]*

Along with the above specifications, your program should do also the following:

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. *[15 Marks]*
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. *[15 Marks]*

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