CSUS, College of Engineering and Computer Science
Department of Computer Science
CSC/CPE 138 – Computer Networks and Internets

Socket Programming
PLab 1a – UDP/TCP Example, PLab 1b – Web Server, PLab 1c – UDP Pinger

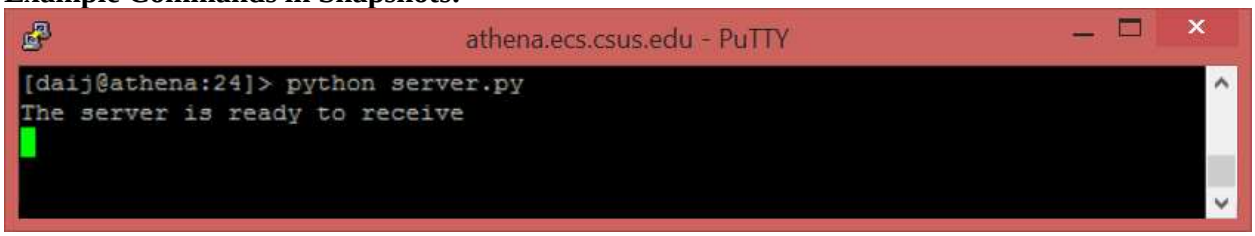
Goal: Practice makes perfect! Socket programming assignments are to help you review and apply your conceptual knowledge from this class.

Attention: Code plagiarism is absolutely **NOT** allowed! Please prepare for a **demonstration** of running your program in front of the instructor/grader and answer their questions.

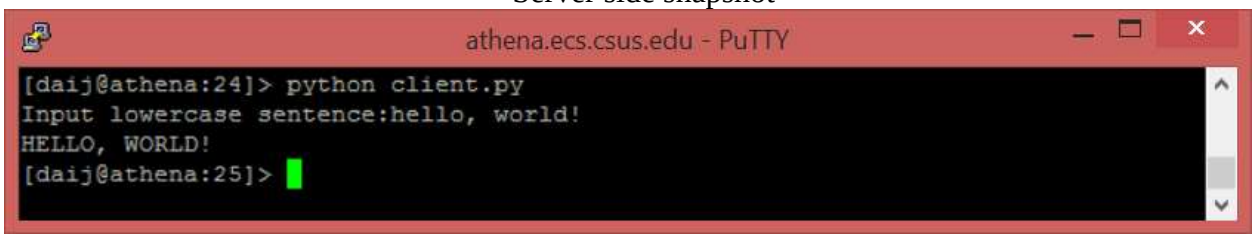
Instructions: Please repeat what's done in the course slides about implementing the **UDP** and **TCP** client/server interactions with Python. If you prefer C or Java implementation, that's OK. If you choose to do so, the caveat is that there is more help if you do it in Python.

Create a separate submission for each of the other two labs: Web Server and UDP Pinger

Example Commands in Snapshots:



Server side snapshot



Client side snapshot

Deliverable: A project report, an **electronic submission** to Canvas, is expected to include both your **source code** and some **screenshots** that can help you demonstrate your work (**commands**, **operations**, **results** and **analysis**). Code plagiarism is absolutely **NOT** allowed! Please also prepare for a **demonstration** of running your program in front of the instructor/grader and answer their **questions** (which are about your code). You grade will be based on both the report and your performance during demonstration.

Requirement: The report will all be evaluated based on the following grading criteria.

Report Correctness, Completeness, Clarity	20%+15%+15%
Demonstration Correctness, Completeness, Question	20%+15%+15%

Programming Lab 1b: 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.

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.

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.

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
```

Programming Lab 1c: UDP Pinger Lab

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, 30% 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 4, we consider the packet lost and do not
respond
      if rand < 4:
            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 4, 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 one second for a reply; if no reply is received within one second, 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 seconds, 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.