The University of Calgary

Department of Electrical & Computer Engineering

ENSF 462 Networked Systems

(Fall 2024)

Due, Oct 16, 2024

Lab 2 Web Server and UDP Pinger[1]

Lab Section	Section Date	Location
B01	October 9th, 2024	ICT 319 (8:00am-9:50am)
B02	October 9th, 2024	ENA 305 (3:00pm-4:50pm)

Part 1 – Web Server

Simple Web Server

In this section of the lab, you will continue practicing socket programming for TCP connections in Python and will learn how to receive an 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.

Acknowledgement:

Part of the content in Lab 2 manual is modified based on the material provided on the authors' website for the textbook, *Computer Networks, A Top-down Approach, 8h ed., J.F. Kurose and K.W. Ross, Addison-Wesley/Pearson, 2020.*

Code

Along with this manual you will find the skeleton code for the web server in a file named WebServer.py (posted on D2L). 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 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 on the server. You should get a "404 Not Found" message.

Two sample HTML files have been provided (posted on D2L) to test your code with.

Assumptions

- Only well formatted GET requests are received by your server.
- All requests are for a single html object on the Internet (i.e., you can IGNORE the case of base html and embedded objects in it).

Multi-threaded Webserver

Currently, the web server handles only one HTTP request at a time. In this section, we want to implement a server that is capable of serving multiple requests simultaneously. For this, we are going to use multi-threading. A thread is a light-weight process that does not need much memory overhead. Multithreading refers to executing multiple threads simultaneously in a single process.

Using multi-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.

A supplementary reading will be provided and posted on D2L to show an example of multi-threading for socket programming in Python.

Part 2 – UDP Pinger

Introduction

In this lab, you will first study a simple Internet ping server written in Python, and implement a corresponding client. These programs provide functionality 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 program 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). The ping protocol allows hosts to determine round-trip times to other machines.

Server Code and Packet Loss

The Ping server code is provided in "`UDPPingerServer.py`". You need to run the server code before running your client program. *You do not need to modify the server code.*

The server sits in an infinite loop listening for incoming UDP packets. As 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/home 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. 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. In other words, 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.

Client Code Requirements

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 one ping message per second using UDP. The ping message is a one-line message 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.
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"
5. report the minimum, maximum, and average RTTs at the end of all pings
6. calculate the packet loss rate (in percentage).

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.

Submit a lab report that includes the following:

- **Your name and UCID #**
- **For Web Server: The complete server code and the screen shots of your client browser, verifying that you actually receive the contents of the HTML file from the server.**
- **For UDP Pinger: Python file for your client and the screenshots of the output of the client**