CS 176A: Homework #3

Note: Parts 1 and 2 of the assignment are to be completed independently.
For the programming assignment (part 3), you are allowed to work with one partner.

Your solutions/results/write-ups from parts 1 and 2 should be tarred or gzip'd into one file and turned in through the Canvas homework link. Your code from part 3 should be tarred or gzip'd into a separate file and uploaded to the Canvas link.

Part 1: Answer the questions on the following page. (25 points total)

Part 2: Wireshark Lab (25 points total)

Complete the Wireshark Lab on TCP.

Part 3: Programming Assignment (50 points total)

Complete the programming assignment to build a UDP Ping Agent.

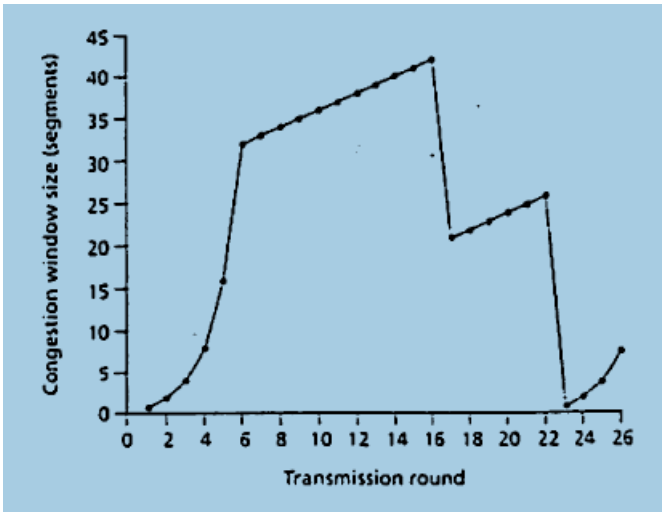
1. For each of the following fields in the TCP header, describe the function/purpose of that field:
   a) Sequence number
   b) Acknowledgement number
   c) ACK bit
   d) Receiver advertised window
   e) Source port number

2. For each of the following TCP connection management stages, name the message type that accomplishes that function (i.e. SYN, FIN, etc.)
   a) A message indicating that the sending side is terminating the connection
   b) A message from server to client ACKing receipt of a SYN message and indicating the willingness of the server to establish a TCP connection with the client
   c) A message from client to server initiating a connection request
   d) A message sent in response to a request to terminate a connection
   e) A general purpose error message during connection set up or tear down, indicating the referenced connection should be shut down.

3. Describe what is meant by transport-layer multiplexing and demultiplexing.

4. Why is the UDP header length field needed?

5. Consider the following plot of TCP window size as a function of time. Assuming TCP Reno is the protocol experiencing the behavior shown in the figure, answer the following questions. In all cases, you should provide a short discussion justifying your answer.
a) Identify the intervals of time when TCP slow start is operating.
b) Identify the intervals of time when TCP congestion avoidance is operating.
c) After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?
d) After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?
e) What is the initial value of `ssthresh` at the first transmission round?
f) What is the value of `ssthresh` at the 18th transmission round?
g) What is the value of `ssthresh` at the 24th transmission round?
h) Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the congestion window size and of `ssthresh`?



6. In the following question, you will compare the performance of GBN, SR and TCP Reno (no delayed ACK). Assume that the timeout values for all three protocols are sufficiently long such that 5 consecutive data segments and their corresponding ACKs can be received (if not lost in the channel) by the receiving host (Host B) and the sending host (Host A) respectively. Suppose Host A sends 5 data segments to Host B, and the 2nd segment (sent from A) is lost. In the end, all 5 data segments are correctly received by Host B.

a) For each of the three protocols, how many segments has Host A sent in total and how many ACKs has Host B sent in total? What are their sequence numbers?
b) If the timeout values for all three protocols are much longer than 5 RTT, then which protocol will successfully deliver all 5 data segments in the shortest time interval?

Wireshark Lab: TCP

In this lab, we'll investigate the behavior of TCP in detail. We'll do so by analyzing a trace of the TCP segments sent and received in transferring a 150KB file (containing the text of Lewis Carrol's *Alice's Adventures in Wonderland*) from a computer to a remote server. We'll study TCP's use of sequence and acknowledgement numbers for providing reliable data transfer; we'll see TCP's congestion control algorithm – slow start and congestion avoidance – in action; and we'll look at TCP's receiver-advertised flow control mechanism. We'll also briefly consider TCP connection setup and we'll investigate the performance (throughput and round-trip time) of the TCP connection between the client computer and the server.

Before beginning this lab, you'll probably want to review sections 3.5 and 3.7 in the text.

1. Capturing a bulk TCP transfer from the client computer to a remote server

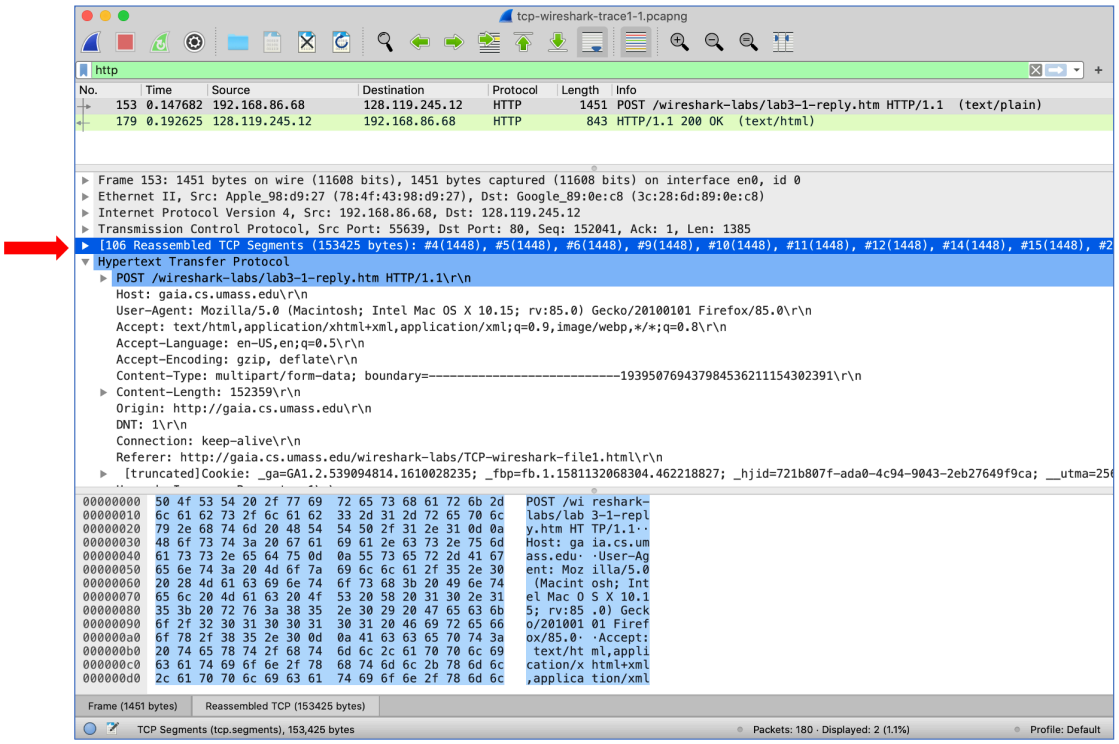
To allow you to investigate TCP's behavior, Wireshark was used to obtain a packet trace of the TCP transfer of a file from a client computer to a remote server. The trace was created by accessing a Web page that will allow the user to enter the name of a file stored on the client computer (which contains the ASCII text of *Alice in Wonderland*), and then transfer the file to a Web server using the HTTP POST method (see section 2.2.3 in the text). The POST method was used rather than the GET method because we'd like to transfer a large amount of data *from* the client computer to another computer. During the transfer, Wireshark was running to obtain the trace of the TCP segments sent and received from the client computer.

The Wireshark packet trace was captured using the following steps:

- The user visited, retrieved and stored the ASCII file at URL http://gaia.cs.umass.edu/wireshark-labs/alice.txt
- Then the user went to http://gaia.cs.umass.edu/wireshark-labs/TCP-wireshark-file1.html
- Before pressing the Upload button, the packet capture with Wireshark was started.
- Then the user pressed the "*Upload alice.txt file*" button to upload the file to the gaia.cs.umass.edu server. Once the file had been uploaded, a short congratulations message was be displayed in the browser window.
- Wireshark packet capture was then stopped.

Load the tcp_wireshark_trace_1 file into Wireshark. Before analyzing the behavior of the TCP connection in detail, let's take a high-level view of the trace. Start by looking at

the HTTP POST message that uploaded the alice.txt file to gaia.cs.umass.edu. Find that file in the Wireshark trace, and expand the HTTP message so we can take a look at the HTTP POST message more carefully. Your Wireshark screen should look something like the following:



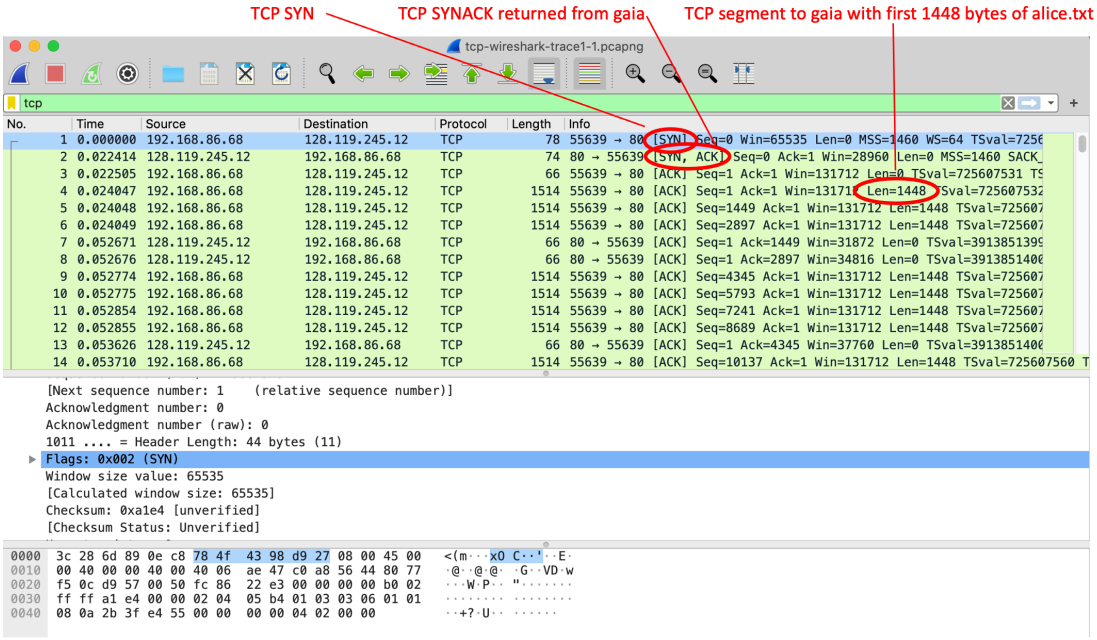
There are a few things to note here:

- The body of the application-layer HTTP POST message contains the contents of the file alice.txt, which is a large file of more than 152K bytes. OK – it's not *that* large, but it's going to be too large for this one HTTP POST message to be contained in just one TCP segment!
- In fact, as shown in the Wireshark window in the figure above we see that the HTTP POST message was spread across 106 TCP segments. This is shown where the red arrow is placed in the figure [Wireshark doesn't have a red arrow like that; we added it to the figure to be helpful ☺]. If you look even more carefully there, you can see that Wireshark is being really helpful to you as well, telling you that the first TCP segment containing the beginning of the POST message is packet #4 in this particular trace 2. The second TCP segment containing the POST message in packet #5 in the trace, and so on.

Let's now "get our hands dirty" by looking at some TCP segments.

- First, filter the packets displayed in the Wireshark window by entering "tcp" (lowercase, no quotes, and don't forget to press return after entering!) into the display filter specification window towards the top of the Wireshark window.

Your Wireshark display should look something like the figure below. In this figure, we've noted the TCP segment that has its SYN bit set – this is the first TCP message in the three-way handshake that sets up the TCP connection to gaia.cs.umass.edu that will eventually carry the HTTP POST message and the alice.txt file. We've also noted the SYNACK segment (the second step in TCP three-way handshake), as well as the TCP segment (packet #4, as discussed above) that carries the POST message and the beginning of the alice.txt file.



3. TCP Basics

Answer the following questions for the TCP segments:

1. What is the IP address and TCP port number used by the client computer (source) that is transferring the alice.txt file to gaia.cs.umass.edu? To answer this question, it's probably easiest to select an HTTP message and explore the details of the TCP packet used to carry this HTTP message, using the "details of the selected packet header window" (refer to Figure 2 in the "Getting Started with Wireshark" Lab if you're uncertain about the Wireshark windows).
2. What is the IP address of gaia.cs.umass.edu? On what port number is it sending and receiving TCP segments for this connection?

Since this lab is about TCP rather than HTTP, now change Wireshark's "listing of captured packets" window so that it shows information about the TCP segments containing the HTTP messages, rather than about the HTTP messages, as in the figure

above. This is what we're looking for–a series of TCP segments sent between the client and gaia.cs.umass.edu.

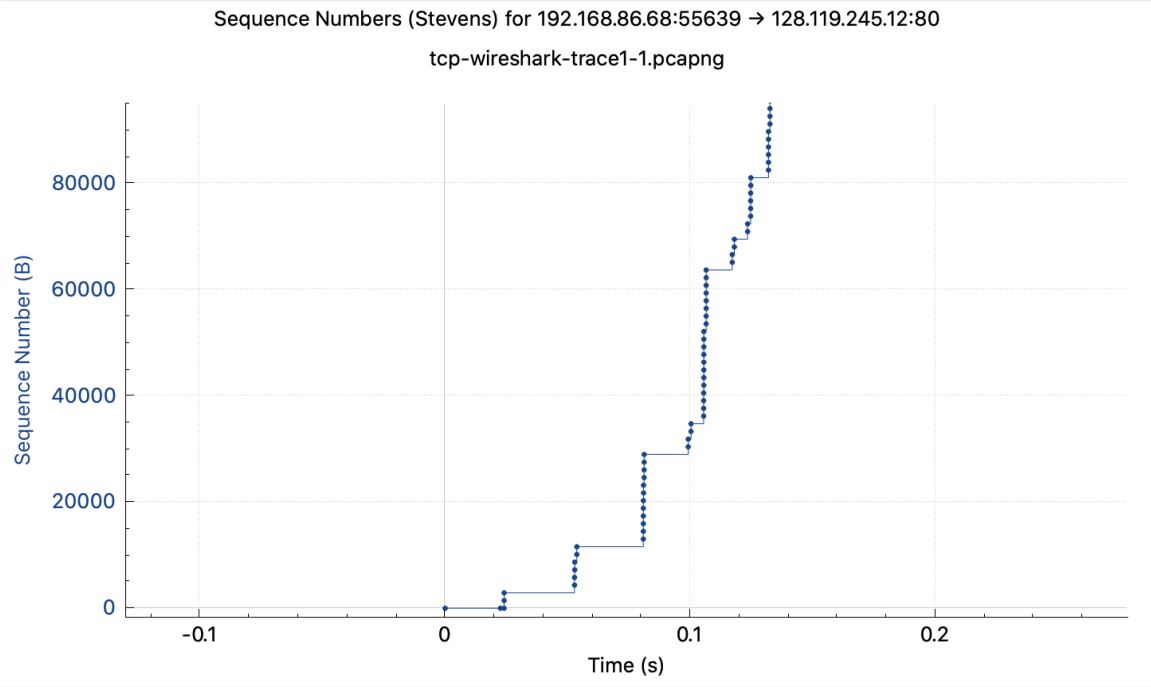
3. What is the sequence number of the TCP SYN segment that is used to initiate the TCP connection between the client computer and gaia.cs.umass.edu? (Note: this is the "raw" sequence number carried in the TCP segment itself; it is *NOT* the packet # in the "No." column in the Wireshark window. Remember there is no such thing as a "packet number" in TCP or UDP; as you know, there *are* sequence numbers in TCP and that's what we're after here. Also note that this is not the relative sequence number with respect to the starting sequence number of this TCP session.). What is it in the segment that identifies the segment as a SYN segment?
4. What is the sequence number of the SYNACK segment sent by gaia.cs.umass.edu to the client computer in reply to the SYN? What is the value of the ACKnowledgement field in the SYNACK segment? How did gaia.cs.umass.edu determine that value?
5. What is the **raw (not relative!)** sequence number of the TCP segment containing the HTTP POST command? Note that in order to find the POST command, you'll need to dig into the packet content field at the bottom of the Wireshark window, looking for a segment with the ASCII text "POST" within its DATA field.
6. Consider the TCP segment containing the HTTP POST as the first segment in the TCP connection.
   - At what time was the first segment (the one containing the HTTP POST) in the data-transfer part of the TCP connection sent?
   - At what time was the ACK for this first data-containing segment received?
   - What is the RTT for this first data-containing segment?
   - What is the RTT value the second data-carrying TCP segment and its ACK?
   - What is the `EstimatedRTT` value (see Section 3.5.3, in the text) after the ACK for the second data-carrying segment is received? Assume that in making this calculation after receiving the ACK for the second segment, that the initial value of `EstimatedRTT` is equal to the measured RTT for the first segment, and then is computed using the `EstimatedRTT` equation on page 242, and a value of $\alpha = 0.125$.

     Note: Wireshark has a nice feature that allows you to plot the RTT for each of the TCP segments sent. Select a TCP segment in the "listing of captured packets" window that is being sent from the client to the gaia.cs.umass.edu server. Then select: *Statistics->TCP Stream Graph->Round Trip Time Graph* if you would like to have a look at how the RTT fluctuates. Click 'Switch Direction' to view the RTTs computed from traffic in the other direction.

4. TCP congestion control in action

Let's now examine the amount of data sent per unit time from the client to the server. Rather than (tediously!) calculating this from the raw data in the Wireshark window, we'll use one of Wireshark's TCP graphing utilities - *Time-Sequence-Graph(Stevens)* - to plot out data.

- Select a client-sent TCP segment in the Wireshark's "listing of captured-packets" window corresponding to the transfer of alice.txt from the client to gaia.cs.umass.edu. Then select the menu: *Statistics->TCP Stream Graph-> Time-Sequence-Graph (Stevens*[1]). You should see a plot that looks similar to the plot in the figure below. You may have to expand, shrink, and fiddle around with the intervals shown in the axes in order to get your graph to look exactly like this figure.

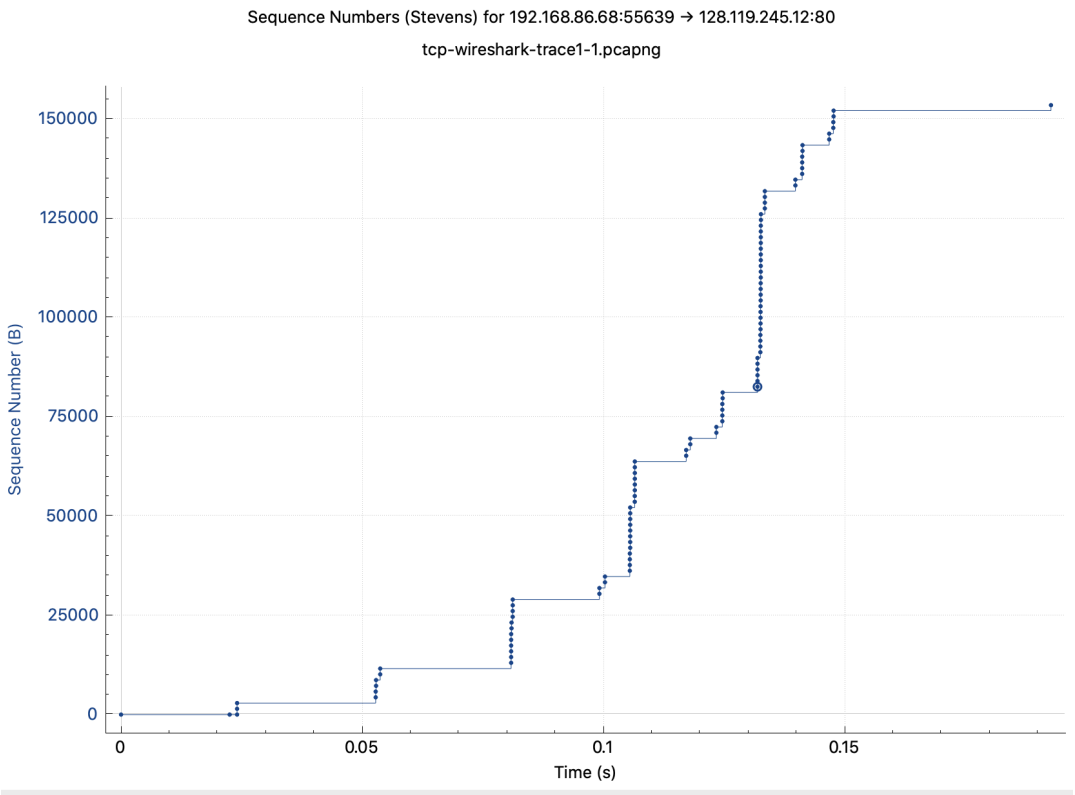


Here, each dot represents a TCP segment sent, plotting the sequence number of the segment versus the time at which it was sent. Note that a set of dots stacked above each other represents a series of packets (sometimes called a "fleet" of packets) that were sent back-to-back by the sender.

Answer the following question:

7. Use the *Time-Sequence-Graph(Stevens)* plotting tool to view the sequence number versus time plot of segments being sent from the client to the gaia.cs.umass.edu server. Consider the "fleets" of packets sent around $t = 0.025$, $t = 0.053$, $t = 0.082$ and $t = 0.1$. Comment on whether this looks as if TCP is in its slow start phase, congestion avoidance phase or some other phase. The figure below shows a slightly different view of this data.

[1] William Stevens wrote the "bible" book on TCP, known as *TCP Illustrated*.


Sequence Numbers (Stevens) for 192.168.86.68:55639 → 128.119.245.12:80
tcp-wireshark-trace1-1.pcapng
Sequence Number (B)
150000
125000
100000
75000
50000
25000
0
0
0.05
0.1
0.15
Time (s)

Programming Assignment: UDP Ping Lab

In this lab, you will learn the basics of socket programming for UDP in C. 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 Python, and implement a corresponding client in C. 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 in C.** You must test your code in CSIL. That is where we will grade your program.

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)
      # 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 returns the encapsulated data 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 should write the client so that it sends 10 ping requests to the server, separated by approximately one second. Each message contains a payload of data that includes the keyword PING, a sequence number, and a timestamp. After sending each packet, the client waits up to one second to receive a reply. If one seconds goes by without a reply from the server, then the client assumes that its packet or the server's reply packet has been lost in the network.
You should write the client so that it starts with the following command:

```
PingClient host port
```

where host is the name of the computer the server is running on and port is the port number it is listening to. Note that you can run the client and server either on different machines or on the same machine.

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 C documentation for DatagramSocket to find out how to set the timeout value on a datagram socket.

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 in CSIL.

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.

Client Output

Your client should produce output modeled after the actual ping command (as shown below). To see the ping command output, type `ping <hostname>`. For example, `ping www.google.com`, or `ping csil-01`.

```
PING 128.111.52.176 (128.111.52.176): 56 data bytes
64 bytes from 128.111.52.176: icmp_seq=0 ttl=64 time=90.608 ms
64 bytes from 128.111.52.176: icmp_seq=1 ttl=64 time=2.097 ms
64 bytes from 128.111.52.176: icmp_seq=2 ttl=64 time=1.570 ms
Request timeout for icmp_seq 3
Request timeout for icmp_seq 4
Request timeout for icmp_seq 5
Request timeout for icmp_seq 6
Request timeout for icmp_seq 7
Request timeout for icmp_seq 8
Request timeout for icmp_seq 9
64 bytes from 128.111.52.176: icmp_seq=10 ttl=64 time=2.459 ms
64 bytes from 128.111.52.176: icmp_seq=11 ttl=64 time=1.438 ms
64 bytes from 128.111.52.176: icmp_seq=12 ttl=64 time=1.671 ms
64 bytes from 128.111.52.176: icmp_seq=13 ttl=64 time=1.439 ms
64 bytes from 128.111.52.176: icmp_seq=14 ttl=64 time=1.429 ms
^C
--- 128.111.52.176 ping statistics ---
15 packets transmitted, 8 packets received, 46.7% packet loss
round-trip min/avg/max/stddev = 1.429/12.839/90.608/29.396 ms
```

Similar to the output you see from this command, your output should be in the following format:

PING received from machine_name: seq#=X time=Y ms

where X is the sequence number of the received packet and Y is the RTT in milliseconds. Note that if there is no response to the ping and the request times out, then you still must print a "Request timeout" message to the screen as indicated and as formatted in the figure above. As your client terminates after it receives the tenth ping response (or timeout), it should print the following:

--- ping statistics ---
X packets transmitted, Y received, Z% packet loss rtt min/avg/max = MIN AVG MAX ms

where X is the number of packets transmitted, Y is the number of packets received, and Z is the percent that were lost. To produce the last line of output you will need to calculate the minimum, average, and maximum RTTs and fill in the appropriate values.

An example output of what the autograder expects is listed below:

```
Request timeout for seq#=1
PING received from 127.0.0.1: seq#=2 time=0.156 ms
PING received from 127.0.0.1: seq#=3 time=0.262 ms
Request timeout for seq#=4
PING received from 127.0.0.1: seq#=5 time=0.271 ms
PING received from 127.0.0.1: seq#=6 time=0.225 ms
Request timeout for seq#=7
Request timeout for seq#=8
PING received from 127.0.0.1: seq#=9 time=0.244 ms
Request timeout for seq#=10
--- 127.0.0.1 ping statistics ---
10 packets transmitted, 5 received, 50% packet loss rtt min/avg/max = 0.156 0.232 0.271 ms
```

What to Hand in

You will hand in the complete client code (named PingClient.c) and a Makefile that compiles your program. If you had a partner for the assignment, you MUST include a README file that lists your and your partners name. Only one of you needs to turn in the program. Without the README, we will not know to give credit to the partner! Combine the two (or three, if you have a README) files into a single tar file and use the Canvas/Gradescope weblink to turn in your file.

Finally, **be sure to test your code in CSIL!!!**