Please type your homework. We will discuss each problem in class but you must do your own work for submission. The date on which each problem is discussed is indicated at the beginning of the problem.

1) (9/30, 5 points) Multiplexing and DeMultiplexing
Suppose Client A initiates a Telnet session with Server S. At about the same time, Client B also initiates a Telnet session with Server S. Provide possible source and destination port numbers for
a. The segments sent from A to S.
b. The segments sent from B to S.
c. The segments sent from S to A.
d. The segments sent from S to B.
e. If A and B are different hosts, is it possible that the source port number in the segments from A to S is the same as that from B to S?
f. How about if they are the same host?

2) (9/30, 5 points) UDP and TCP use 1’s complement for their checksums. Suppose you have the following three 8-bit bytes: 01010011, 01010100, 01110100. What is the 1’s complement of the sum of these 8-bit bytes? (Note that although UDP and TCP use 16-bit words in computing the checksum, for this problem you are being asked to consider 8-bit sums.) Show all work. Why is it that UDP takes the 1’s complement of the sum; that is, why not just use the sum? With the 1’s complement scheme, how does the receiver detect errors? Is it possible that a 1-bit error will go undetected? How about a 2-bit error?

3) (10/2, 5 points) Consider our motivation for correcting protocol rdt2.1. Show that the receiver, shown in Figure 3.57, when operating with the sender shown in Figure 3.11, can lead the sender and receiver to enter into a deadlock state, where each is waiting for an event that will never occur.

4) (10/4, 5 points) Give a trace of the operation of protocol rdt3.0 when data packets and acknowledgment packets are garbled.
5) (10/4, 10 points) Consider two network entities, A and B, which are connected by a perfect bi-directional channel (i.e., any message sent will be received correctly; the channel will not corrupt, lose, or re-order packets). A and B are to deliver data messages to each other in an alternating manner: First, A must deliver a message to B, then B must deliver a message to A, then A must deliver a message to B and so on. If an entity is in a state where it should not attempt to deliver a message to the other side, and there is an event like rdt_send(data) call from above that attempts to pass data down for transmission to the other side, this call from above can simply be ignored with a call to rdt_unable_to_send(data), which informs the higher layer that it is currently not able to send data. [Note: This simplifying assumption is made so you don’t have to worry about buffering data.]

Draw a FSM specification for this protocol (one FSM for A, and one FSM for B!). Note that you do not have to worry about a reliability mechanism here; the main point of this question is to create a FSM specification that reflects the synchronized behavior of the two entities. You should use the following events and actions that have the same meaning as protocol rdt1.0 in Figure 3.9: rdt_send(data), packet = make_pkt(data), udt_send(packet), rdt_rcv(packet), extract(packet, data), deliver_data(data).

Make sure your protocol reflects the strict alternation of sending between A and B. Also, make sure to indicate the initial states for A and B in your FSM descriptions.

6) (10/7, 10 points) In the generic SR protocol that we studied in Section 3.4.4, the sender transmits a message as soon as it is available (if it is in the window) without waiting for an acknowledgment. Suppose now that we want an SR protocol that sends messages two at a time. That is, the sender will send a pair of messages and will send the next pair of messages only when it knows that both messages in the first pair have been received correctly. Suppose that the channel may lose messages but will not corrupt or reorder messages. Design an error-control protocol for the unidirectional reliable transfer of messages. Give an FSM description of the sender and receiver. Describe the format of the packets sent between sender and receiver, and vice versa. If you use any procedure calls other than those in Section 3.4 (for example, udt_send(), start_timer(), rdt_rcv(), and so on), clearly state their actions. Give an example (a timeline trace of sender and receiver) showing how your protocol recovers from a lost packet.

7) (10/7, 10 points) Consider the GBN protocol with a sender window size of 4 and a sequence number range of 1,024. Suppose that at time t, the next in-order packet that the receiver is expecting has a sequence number of k. Assume that the medium does not reorder messages. Answer the following questions:
   a) What are the possible sets of sequence numbers inside the sender’s window at time t? Justify your answer.
   b) What are all possible values of the ACK field in all possible messages currently propagating back to the sender at time t? Justify your answer.
8)  (9/30, 20 points) Wireshark

In this lab, we’ll take a quick look at the UDP transport protocol. As we saw in Chapter 3 of the text\(^1\), UDP is a streamlined, no-frills protocol. You may want to re-read section 3.3 in the text before doing this lab. Because UDP is simple and sweet, we’ll be able to cover it pretty quickly in this lab. So if you’ve another appointment to run off to in 30 minutes, no need to worry, as you should be able to finish this lab with ample time to spare.

At this stage, you should be a Wireshark expert. Thus, we are not going to spell out the steps as explicitly as in earlier labs. In particular, we are not going to provide example screenshots for all the steps.

The Assignment

Start capturing packets in Wireshark and then do something that will cause your host to send and receive several UDP packets. It’s also likely that just by doing nothing (except capturing packets via Wireshark) that some UDP packets sent by others will appear in your trace. In particular, the Simple Network Management Protocol (SNMP - chapter 9 in the text) sends SNMP messages inside of UDP, so it’s likely that you’ll find some SNMP messages (and therefore UDP packets) in your trace.

After stopping packet capture, set your packet filter so that Wireshark only displays the UDP packets sent and received at your host. Pick one of these UDP packets and expand the UDP fields in the details window. If you are unable to find UDP packets or are unable to run Wireshark on a live network connection, you can download a packet trace containing some UDP packets.\(^2\)

Whenever possible, when answering a question below, you should hand in a printout of the packet(s) within the trace that you used to answer the question asked. Annotate the printout\(^1\) to explain your answer. To print a packet, use File->Print, choose Selected packet only, choose Packet summary line, and select the minimum amount of packet detail that you need to answer the question.

---

\(^1\) References to figures and sections are for the 6\(^{th}\) edition of our text, *Computer Networks, A Top-down Approach, 6\(^{th}\) ed.*, J.F. Kurose and K.W. Ross, Addison-Wesley/Pearson, 2012.

\(^2\) Download the zip file [http://gaia.cs.umass.edu/wireshark-labs/wireshark-traces.zip](http://gaia.cs.umass.edu/wireshark-labs/wireshark-traces.zip) and extract the file http-etheral-trace-5, which contains some UDP packets carrying SNMP messages. The traces in this zip file were collected by Wireshark running on one of the author’s computers. Once you have downloaded the trace, you can load it into Wireshark and view the trace using the File pull down menu, choosing Open, and then selecting the http-etheral-trace-5 trace file.
1. Select one UDP packet from your trace. From this packet, determine how many fields there are in the UDP header. (You shouldn’t look in the textbook! Answer these questions directly from what you observe in the packet trace.) Name these fields.

2. By consulting the displayed information in Wireshark’s packet content field for this packet, determine the length (in bytes) of each of the UDP header fields.

3. The value in the Length field is the length of what? (You can consult the text for this answer). Verify your claim with your captured UDP packet.

4. What is the maximum number of bytes that can be included in a UDP payload? (Hint: the answer to this question can be determined by your answer to 2. above)

5. What is the largest possible source port number? (Hint: see the hint in 4.)

6. What is the protocol number for UDP? Give your answer in both hexadecimal and decimal notation. To answer this question, you’ll need to look into the Protocol field of the IP datagram containing this UDP segment (see Figure 4.13 in the text, and the discussion of IP header fields).

7. Examine a pair of UDP packets in which your host sends the first UDP packet and the second UDP packet is a reply to this first UDP packet. (Hint: for a second packet to be sent in response to a first packet, the sender of the first packet should be the destination of the second packet). Describe the relationship between the port numbers in the two packets.

9) (10/2, 30 points) Programming

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.

```python
# UDPingServer.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 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 `UDPingerServer.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.