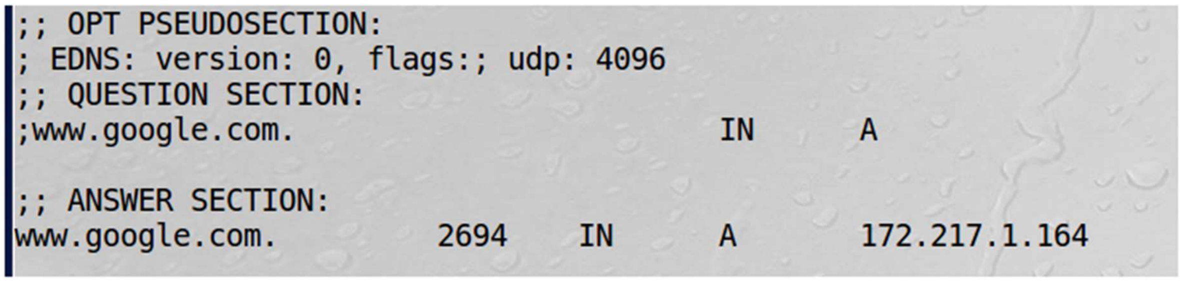
Lab 2

[Due your class day of May 26/31]

Note: in all the lab questions, please include the screen shots as evidence of your solutions.

1. Command **dig** is used to find the ip address of a hostname such as www.yahoo.com. To do this, you can simply run **dig www.google.com**. You might see the following result.



```
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.google.com.                        IN      A

;; ANSWER SECTION:
www.google.com.         2694    IN      A       172.217.1.164
```

This is the result returned by your local DNS server when you run **dig** command to request it to find out the ip address of www.google.com. Question section is to repeat your question and **A** stands for the record of IP address. We can see that www.google.com has an ip address 172.217.1.164.

a. Try **dig www.example.net** to find out its ip address.

b. run Wireshark on your VM, then **dig www.example.net** and stop wireshark. Look at the DNS request packet, confirm that the transport layer protocol is UDP. What are the values of this UDP header (you need to first check the header fields learned in class)?

c. In the DNS request packet in step **b**, the destination IP is your local DNS server's IP. What is this value? As said, DNS is serviced by UDP and has **no** connection setup before sending DNS request. You can confirm this by checking that there is no any packet in Wireshark exchanged between your VM and local DNS server, prior to the DNS request packet.

2. Run Wireshark and then access **www.example.net** using Firefox. Then stop the Wireshark.

Check your list of packets in Wireshark window with filtered by the ip address of **www.example.net**. You can see that before the HTTP request to **www.example.net**, there is a connection stage with three packets: SYN packet, SYN-ACK packet and ACK packet. This is to provide the connection setup between your VM and **www.example.net**. Confirm this. Also, confirm that the transport layer protocol in these packets is TCP. When the message exchange starts, you can see ACK packet. This is to confirm the receipt of a packet. Find out such a packet. This is to find a packet with flags bit A=1. This provides an evidence that TCP is a reliable protocol. This is different from the UDP protocol.

3. Applications communicate using sockets. Practice UDP sockets to send/receive messages.

Run the following UDP client on one VM (change ip in your setting):

```
#!/usr/bin/python3
from socket import *
serverName = "10.0.2.5"
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input("input your message:")
clientSocket.sendto(message.encode(), (serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode())
clientSocket.close()
```

Rune the UDP server at 10.0.2.5 (change ip in your setting)

```
#!/usr/bin/python3
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((’’, serverPort))
print(”The server is ready to receive”)
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

Describe what the programs are doing. You need to specify the meaning of the socket functions in the programs. You can see that the client send functions always uses the server address. This is because the client socket does not maintain the server state. Whenever it needs to send a message to the server, you need to tell the socket the destination address again. This similar holds for the server socket.

4. Run wireshark and access **www.example.net** and then close your webpage and stop your wireshark. Answer the following questions.

a. Find out the first packet from your VM to **www.example.net**. This should be the SYN-packet (i.e., the first packet of the 3-way handshake protocol). What is source port # and destination port #? Confirm that they are in the TCP header in the Wireshark packet window. What is source IP and destination IP? Confirm that they are in the ip header in the Wireshark packet window.

b. Look at SYN-packet. What is the sequence #? It is a random number. Confirm this.

c. Find out in the TCP header the flag bits U|A|P|R|S|F in the SYN packet and in the reply packet (called SYN-ACK packet).

d. The receive window field is to tell its partner the current **receive-buffer** size it has. Find out the window size of SYN-ACK packet and that of http response packet. Are they equal?

e. Find out the sequence # of http request packet and its payload size (the **segment len** is the payload size). The next sequence # is the sum of these two numbers. Verify that this is indeed the sequence # of the next packet sent by your VM.

f. Find out the acknowledgement # in http response packet. Is this the same as the next sequence # you calculated above for the request packet?

g. What is the flags bits U|A|P|R|S|F in the http response packet?

h. Find out the packet your VM requests to terminate the TCP connection. This packet will be sent when you close the webpage. What is the flags bit U|A|P|R|S|F in this packet?

5. In this experiment, you run a TCP client on one VM and a TCP server on another VM. Confirm that the TCP server sockets (welcome socket and new socket) always uses the same port #. To do this, you need to run the Wireshark to find the packet of server sending the capitalizedSentence and verify its source port number is 12000. Outline how server is working.

Server:

```
#! /usr/bin/python3
from socket import *
from _thread import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("10.0.2.5", serverPort))
serverSocket.listen(1)
print("The server is ready to receive")

def multi_threaded_client(connectionSocket):
    sentence = connectionSocket.recv(1024)
    while sentence:
        capitalizedSentence = sentence.decode().upper()
        connectionSocket.send(capitalizedSentence.encode())
        print(sentence.decode())
        sentence = connectionSocket.recv(1024)
    connectionSocket.close()

while True:
   connectionSocket, addr = serverSocket.accept()
   start_new_thread(multi_threaded_client, (connectionSocket, ))
```

Client:

```
from socket import *

serverName = "10.0.2.5"
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

sentence = input("Input lowercase sentence:")
while sentence:
    clientSocket.send(sentence.encode())
    modifiedSentence = clientSocket.recv(1024)
    print("From Server: ", modifiedSentence.decode())
    sentence=input("Input your lowercase sentence:")
clientSocket.close()
```

6 In the following code, client sends three messages through three send functions. But it is not necessary that there are three outgoing **TCP packets** (for sending these messages). Use Wireshark to confirm this. This is because the sending buffer has no boundary for different messages. It creates the TCP segment by taking the message from the buffer with any length it prefers.

Client:
```
from socket import *

serverName = "10.0.2.5"
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

sentence = input("Input lowercase sentence:")
while sentence:
    clientSocket.send(sentence.encode())
    S1="fff1"
    clientSocket.send(S1.encode())
    S2="fff2"
    clientSocket.send(S2.encode())
    S3="fff3"
    clientSocket.send(S3.encode())
    modifiedSentence = clientSocket.recv(1024)
    print("From Server: ", modifiedSentence.decode())
    sentence=input("Input your lowercase sentence:")
clientSocket.close()
```