**CS 6043/5143: Computer Networking**

**FALL 2019**

**PROJECT 1**

**Given: Sept. 19, 2019**

**Due: Oct. 7 (Monday), 2019 (NO LATER THAN 11:59PM)**

**Submission Instructions:**

1. Submit only on-line files on Blackboard before midnight. No hard copy will be accepted.

2. Wireshark files for this project can be found in the zip file “Project\_1\_Wireshark\_Traces\_Fall\_2019.zip”.

**Total possible points: 10**

**Part I: HTTP**

1. (0.5 pts) Load the file ‘http-trace-1.pcapng’ in Wireshark and answer the following questions. The traces were collected when a simple and very short HTML file was downloaded by the client.
2. What languages (if any) does the client browser indicate that it can accept to the server? What versions of HTTP are being run by the client and the server?
3. What is the status code returned from the server to the client browser for the second GET request? What does the code mean?
4. (0.5 pts) Load the file ‘http-trace-2.pcapng’ in Wireshark and answer the following questions. The traces were collected when a particular web page is accessed twice from the browser within a short interval.
5. Inspect the contents of the first server response. Did the server explicitly return the contents of the file? If so, what was returned? (provide a screenshot)
6. What is the HTTP status code and phrase returned from the server in response to this second HTTP GET? Did the server explicitly return the contents of the file? Explain why it did or did not.
7. (1 pts) Load the file ‘http-trace-3.pcapng’ in Wireshark and answer the following question.

The traces were collected when HTTP GET requests were sent for four image files (one PNG, three JPEGs). Can you tell whether the client browser downloaded the first two images (one JPEG and one PNG) serially or in parallel? Explain how you came to your conclusion.

1. (1 pts) Enter the following URL into your browser and type the requested user name and password into the pop up box.

<http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html>

The username is “wireshark-students”, and the password is “network” (without the quotes).

1. What is the server’s response (status code and phrase) in response to the initial HTTP GET message from your browser?
2. When your browser sends the HTTP GET message for the second time, what new field is included in the HTTP GET message? Are the login credentials encrypted or sent as plain text?

**Part II: DNS**

1. (0.5 pts) Run *nslookup* to determine the authoritative DNS server and its IP for www.uc.edu. Include screenshot in your answer.
2. (0.5 pts) Run *nslookup* so that the authoritative DNS server obtained in Question 1 is queried for mail.uc.edu. Mention the IP address(es) for mail.uc.edu. Include screenshot.
3. (0.5 pts) Load the file ‘dns-trace-1.pcapng’ in Wireshark and answer the following questions.
4. Locate the DNS query and response messages. Are they sent over UDP or TCP? What is the destination port for the DNS query message and the source port of DNS response message?
5. Examine the DNS query message. What “Type” of DNS query is it?
6. Examine the DNS response message. How many “answers” are provided? What does each of these answers contain?
7. (0.5 pts) Load the file ‘dns-trace-2.pcapng’ in Wireshark and answer the following questions.
8. Examine the fourth DNS response message (serial no. 19). What are the type, class, and address received for google.com?
9. Examine the tenth DNS response message (serial no. 39). What are the refresh interval and minimum TTL for UC mail server?

**Part III: Socket Programming**

1. (2.5 pts) Web Server

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.

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.

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.

*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 \*

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()

2. (2.5 pts) Simple Mail Client

You will develop a simple mail client that sends email to any recipient. Your client will need to connect to a mail server, dialogue with the mail server using the SMTP protocol, and send an email message to the mail server. Python provides a module, called smtplib, which has built in methods to send mail using SMTP protocol. However, we will not be using this module in this lab, because it hide the details of SMTP and socket programming.

In order to limit spam, some mail servers do not accept TCP connection from arbitrary sources. For the experiment described below, you may want to try connecting both to your university mail server and to a popular Webmail server, such as a AOL mail server. You may also try making your connection both from your home and from your university campus.

Below you will find the skeleton code for the client. 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.

In some cases, the receiving mail server might classify your e-mail as junk. Make sure you check the junk/spam folder when you look for the e-mail sent from your client.

*In your submission, you are to provide the complete code for your SMTP mail client as well as a screenshot showing that you indeed receive the e-mail message.*

--------------

In gmail

Disable 2 step verification

Enable less secure app access

----

Use smtp Port 465

Bare 64 encoding for pwds

-----

Skeleton Python Code for the Mail Client:

from socket import \*

msg = "\r\n I love computer networks!"

endmsg = "\r\n.\r\n"

# Choose a mail server (e.g. Google mail server) and call it mailserver

mailserver = #Fill in start #Fill in end

# Create socket called clientSocket and establish a TCP connection with mailserver

#Fill in start

#Fill in end

recv = clientSocket.recv(1024).decode()

print(recv)

if recv[:3] != '220':

print('220 reply not received from server.')

# Send HELO command and print server response.

heloCommand = 'HELO Alice\r\n'

clientSocket.send(heloCommand.encode())

recv1 = clientSocket.recv(1024).decode()

print(recv1)

if recv1[:3] != '250':

print('250 reply not received from server.')

# Send MAIL FROM command and print server response.

# Fill in start

# Fill in end

# Send RCPT TO command and print server response.

# Fill in start

# Fill in end

# Send DATA command and print server response.

# Fill in start

# Fill in end

# Send message data.

# Fill in start

# Fill in end

# Message ends with a single period.

# Fill in start

# Fill in end

# Send QUIT command and get server response.

# Fill in start

# Fill in end