uOttawa

School of Information Technology and Engineering

CEG 4188: Higher Layer Network Protocols

Assignment 1

Part I

Question 1:

Consider two hosts, **A** and **B**, connected by a single link of rate **R** bps. Suppose that the two hosts are separated by **m** meters, and suppose the propagation speed along the link is **s** meters/sec. Host **A** is to send a packet of size **L** bits to
Host **B**.

a) Ignoring processing and queuing delays, obtain an expression for the end-to-end delay.

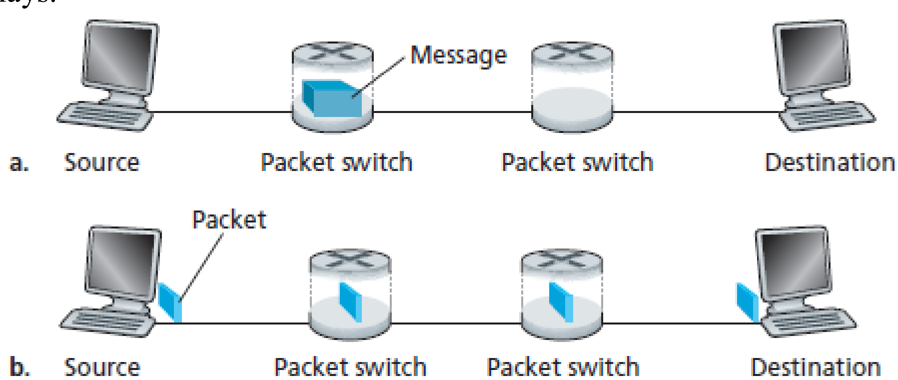
b) Suppose $\mathbf{d_{prop}}$ is greater than $\mathbf{d_{trans}}$. At time $\mathbf{t} = \mathbf{d_{trans}}$, where is the first bit of the packet?

c) Suppose $\mathbf{s} = 2.5 * 10^8$, $\mathbf{L} = 120$ bits, and R = 56 kbps. Find the distance m so that $\mathbf{d_{prop}}$ equals $\mathbf{d_{trans}}$.

Question2:

In modern packet-switched networks, including the Internet, the source host segments long, application-layer messages (for example, an image or a music file) into smaller packets and sends the packets into the network. The receiver then reassembles the packets back into the original message. We refer to this process as message segmentation. The following figure illustrates the end-to-end transport of a message without (a.) and with (b.) message segmentation.

Consider a message that **is $8 * 10^6$ bits** long that is to be sent from source to destination in the figure above. Suppose each link in the figure is 2 Mbps. Ignore propagation, queuing, and processing delays.



a) Consider sending the message from source to destination without message segmentation. How long does it take to move the message from the source host to the first packet switch? Keeping in mind that each switch uses store-and-forward packet switching, what is the total time to move the message from source host to destination host?

b) Now suppose that the message is segmented into 800 packets, with each packet being 10,000 bits long. How long does it take to move the first packet from source host to the first switch? When the first packet is being sent from the first switch to the second switch, the second packet is being sent from the source host to the first switch. At what time will the second packet be fully received at the first switch?

c) How long does it take to move the file from source host to destination host when message segmentation is used? Compare this result with your answer in part (a) and comment.

d) In addition to reducing delay, what are reasons to use message segmentation?

e) Discuss the drawbacks of message segmentation.

Question3 (HTTP):

Consider Figures 2.12 and 2.13, for which there is an institutional network connected to the Internet. However, assume the bandwidth on the access link is 100Mbps and the bandwidth within the institutional LAN is 1000Mbps. Also, suppose that the average object size is 850,000 bits and that the average request rate from the institution's browsers to the origin servers is 16 requests per second. Suppose that the amount of time it takes from when the router on the Internet side of the access link forwards an HTTP request until it receives the response is 3 seconds. Model the total average response time as the sum of the average access delay and the average Internet delay. For the average access delay, use the following formula:

$$\frac{\Delta}{(1-\Delta\times\beta)}$$

where Δ is the average time required to send an object over the access link, and β is the arrival rate of objects to the access link. Answer the following two questions:

a) Find the total average response time.

b) Now suppose a cache is installed in the institutional LAN. Suppose the miss rate is 0.4. Find the total response time.

Question 4 (DNS):

Assume you want to retrieve a web page from www.eecs.uottawa.ca. As your PC or the local DNS server do not have the IP address, then the root, TLD, and authoritative servers should all be visited to resolve the IP address. Now suppose that the round-trip time between your PC and the local DNS server is 4 msec. Further, assume that the subsequent delay between the local DNS server and each of the Root, TLD, and authoritative servers is 11 msec and that each department has its own authoritative name server for its labs.

a) How long would it take for your PC to obtain the IP address of the web server?

b) Now assume that we want to obtain another web page from www.med.uottawa.ca immediately after retrieving a web page from www.eecs.uottawa.ca. How long would the DNS resolution take?

Question 5 (C-s vs p2p):

Consider distributing a file of F = 150 Gbits to 1,000,000 peers. The server has an upload rate of u_s = 100 Gbps, and each peer has a download rate of d_i = 3 Mbps and an upload rate of u_i = 600 Kbps. Calculate the minimum distribution time for two cases:

a) If client-server file distribution is used.

b) If a P2P file distribution is used.

Question 6 (persistent vs- none-persistent HTTP):
Suppose you use the HTTP protocol to download a Web page that resides on some Web server. The cases of persistent and non-persistent connections will be considered. Define the

response time as the time from when you request the URL of the Web page to the time when the page and its embedded images are displayed.

The assumptions are:

- The base html file has a size of B_f bytes.
- The Web page contains m embedded images $I_1, I_2, \ldots, I_m$, each image has a size of B_i bytes.
- The m images are stored on the same Web server.
- The propagation delay between your host and the Web server is T_p seconds.
- The network path between your host and the Web server has a link capacity of R bytes/second.
- The time it takes to transmit a GET message into the link is zero (i.e., the transmission delay is zero for the GET messages).
- Other than propagation and transmission delays, ignore any other types of delays (such as the DNS-related delay).

In terms of m, T_p, B_f , B_i, R give answers to the following questions:

a) What is the two-way propagation delay between the client and server (denoted T_{tp})?

b) What is the transmission delay (denoted T_{tf}) of the base html file?

c) What is the transmission delay (denoted T_{ti}) of each of the embedded images?

d) What is the required time to conduct the two-step handshaking?

e) Assuming a non-persistent HTTP connection, how long is the response time? Describe and detail the various components that contribute to this delay.

f) Assuming a persistent HTTP connection, how long is the response time? Describe and detail the various components that contribute to this delay.

<u>Deliverables</u>
1. A document file with your Answers.

Part II: Programming

For this question you will need to build a simple application that connects users over a network: a chat server. Your chat server will allow users to converse in different channels. Users can create and join channels; once a user is in a particular channel, all messages that he/she sends will be relayed to all other users in that channel.

This assignment should be implemented in Python 2. This assignment will introduce you to the socket programming API.

Download and use the following help files from BBL:

- client_split_messages.py is intended to help you ensure your server is correctly buffering messages, and is described in more detail below.
- simple_test.py tests a basic scenario where two clients communicate in a simple channel.
- utils.py file that has error messages that you should use.

Important note:

Unfortunately, stdin does not count as a socket in Windows due to how file descriptors work differently. Since the tests will be run on a Linux machine, this will cause issues if you write the code to use the Windows version of file descriptors. Your options are to download a Linux virtual machine, or use the instructional servers through SSH, or use the lab machines to test your code. Side note: If you are using Windows 10, you can use the Linux subsystem for Windows and run your code from there and it may work.

Deliverables

1. `client.py`
2. `server.py`.

Requirements:

1. Server Functionality

The server should accept a single command line argument that's the port that the server should run on.

Unlike your server in part 0, your server in this part of the assignment must allow many clients to be connected and sending messages concurrently. Each client should have an associated name (so that other connected clients know who each message is from) and channel that they're currently subscribed to. When a client first connects, it won't have an associated name and channel. The first message that the server receives from the client should be used as the client's name.

Future messages from the client to the server can take one of two forms. The first type of message is a control message; control messages always begin with "/". There are three different control messages your server should handle from clients:

/join <channel> should add the client to the given channel. Clients can only be in one channel at a time, so if the client is already in a channel, this command should remove the client from that channel. When a client is added to a channel, a message should be broadcasted to all existing members of the channel stating that the client has joined. Similarly, if the client left a channel, a message should be broadcasted stating that the client left.
/create <channel> should create a new channel with the given name, and add the client to the given channel. As with the /join call, the client should be removed from any existing channels.
/list should send a message back to the client with the names of all current channels, separated by newlines.
The second type of message is normal messages to the client's current channel. All messages that are not preceded by a / are considered normal messages. These messages should be broadcasted to all other clients in the channel, preceded by the client's name in brackets (see the example below). Messages should not be sent to client in different channels. If the client is not currently in a channel, the server should send an error message back to the client.

When a client disconnects, a message should be broadcasted to all members of the client's channel saying that the client disconnected.

Sockets provide a datastream functionality, but they don't delineate different messages. When a given recv call returns some data, the socket won't tell you whether the data returned is a single message, or multiple messages, or part of one message. As a result, you'll need a way to determine when a message ends. For this assignment, use fixed length messages that have 200 characters for all messages (including messages from the server to the client). If a message is shorter than 200 characters, you should pad the message with spaces (and the receiver should strip any spaces off of the end of the message). You can assume that no messages are longer than 200 characters.

Be sure that your code correctly handles the case where less than one message is available in the socket's buffer (so a recv call returns fewer than 200 characters of data) and the case where more than one message is available in the socket's buffer. You should handle partial messages by buffering: if a recv call returns only part of a message, your code should hold on to the part of the message until the remainder of the message is received, and then handle the complete message. For example, if a client receives 150 characters from the server, it should hold those 150 characters until 50 more characters are received.

The client should only write the message to stdout once all 200 characters have been received. To help you check for your server's handling of these cases, we've provided a special client (client_split_messages.py) that splits messages into many smaller messages before sending them to the server. This client only tests some of the scenarios your server should handle! You'll likely want to modify this client to test for additional cases.

2. Error Handling

Your server should handle cases where the client sends an invalid message by returning an appropriate error message to the client. For example, if a client uses the /join command but doesn't provide the name of a channel to join, the server should send back an error message. The provided utils.py includes format strings for all of the errors you should handle. You can use these format strings using Python's string formatting operations. For example, utils.py defines the following error message:

CLIENT_SERVER_DISCONNECTED = "Server at {0}:{1} has disconnected"
You can use the .format function to replace the brackets with strings as follows:

error_message = CLIENT_SERVER_DISCONNECTED.format("localhost", 12345)
You are required to use the error messages defined in utils.py. If you do not use these error messages (with appropriate formatting), you will not get credit for error handling.

When commands lead to an error, the command should not cause any changes at the server. For example, if a client is currently in the cs168_tas channel and then tries to join a channel that doesn't exist, the client should not be removed from the cs168_tas channel.

We will only test for errors that have associated error messages in utils.py in our testing. You're welcome to check for additional errors if you'd like, but you will not be graded on them.

3. Client Functionality

Each client connects to a particular chat server, and is associated with a name. Your client should be started as follows:

$ python client.py Scott 127.0.0.1 55555
This command should connect to the server at the given address and port number, and then send a message with the name Scott.

After being started, the client should listen for messages from the server and from stdin. Messages from the server should be printed to the command line (after being stripped of any spaces at the end) and messages from stdin should be sent to the server (after being padded with spaces, as needed). The client should print [Me] to each new line, to make it clear which messages in the history were typed by the client. When the client gets a message from the server, it should write over the [Me] with the message from the server. For an example of this, take a look at the demo video linked above.

Here's an example of a client's interaction with a server that was started locally on port 55555:

python chatv3_client.py Panda localhost 55555
[Me] Hello world!
Not currently in any channel. Must join a channel before sending messages.
[Me] /list
[Me] /create 168_tas
[Me] /list
168_tas
[Me] Hello world!
Alice has joined
[Alice] Hi everyone! Does anyone know what we're doing on the first day of lecture?

After seeing Alice's message, Panda stopped his client. After Panda created the 168_tas channel, a second client started:

```
python chatv3_client.py Alice 127.0.0.1 55555
[Me] /list
168_tas
[Me] /join 168_tas
[Me] Hi everyone! Does anyone know what we're doing on the first day of lecture?
Panda has left
```

4. Some helpful hints:

In Python, you can create a socket and connect to a remote endpoint by using the socket library as follows:

```
import socket
# The socket constructor accepts a few arguments; the defaults are fine for this class.
client_socket = socket.socket()
client_socket.connect(("1.2.3.4", 5678))
client_socket.sendall("Hello World")
```

The example above created a socket and connected it to port 5678 at IP address 1.2.3.4. Then, it sent a "Hello World" message to the server at 1.2.3.4:5678.

The example above created a client socket that was connected to exactly one remote endpoint. When you create a server, you'll typically want to allow multiple remote clients to connect, and you don't usually know the address of those clients when the socket is created. As a result, server sockets work differently:

```
server_socket = socket.socket()
server_socket.bind(("1.2.3.4", 5678))
server_socket.listen(5)
```

After creating the socket, rather than connecting to a particular remote destination, the code above bound the socket to a particular IP address and port, which essentially tells the operating system to associate the given IP address and port with the socket. Finally, the listen call listens for connections made to the socket. When a new client connects to the socket, the socket library will create a new socket to use to communicate with that client, so that the server socket can continue to be used to wait for inbound connections from other clients:

```
(new_socket, address) = server_socket.accept()
```

This call blocks until a client connects (using a connect() call, as in the example above), and then returns a newly created socket, new_socket, that can be used to send and receive data to and from the client. For example, the call

```
message = new_socket.recv(1024)
```

will block until there is data to receive from the client, and will return up to 1024 bytes of data.

Use The Python Socket Programming HOWTO https://docs.python.org/2/howto/sockets.html for additional help.

Note: All submitted files must have your name and your student number at the beginning of the file.