Assignment 1: Implementing Chat Application

Deadline for Part 1: Thursday, 13 February 2020 at 11pm

Deadline for Part 2: Wednesday, 26 February 2020 at 11pm

The goal of this assignment is to (i) introduce you to socket programming and (ii) to implement a reliable transport protocol. In this assignment, you will implement a chat application (like messenger) which will allow users to transfer messages and files. The assignment must be done individually and you should use Python. The application has the following two parts:

Part 1: A simple chat application using UDP, which is a transport protocol that does not ensure reliable communication.

Part 2: Extending the chat application in Part 1 to ensure reliable communication of messages and files, by implementing a reliable protocol on top of UDP.

Note: You should post any assignment related query **ONLY** on **campuswire**. Do not directly email the course staff.

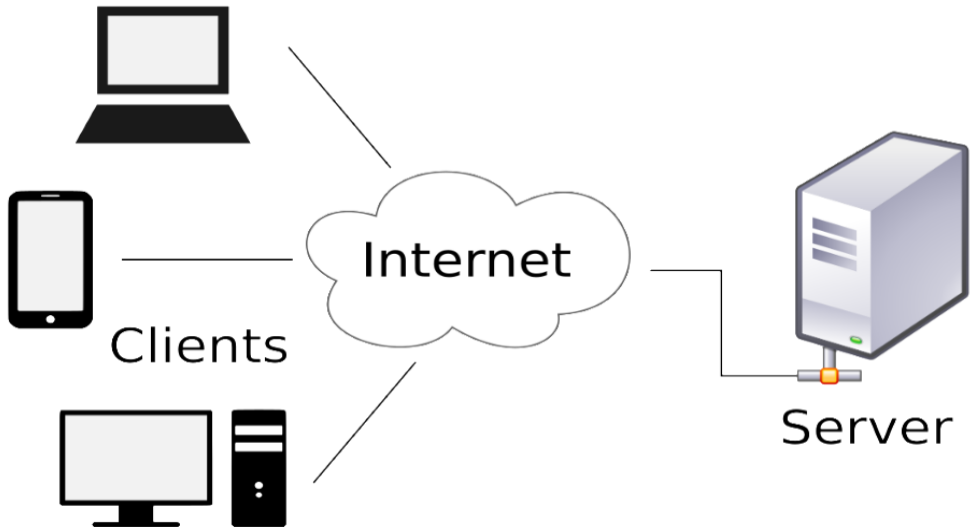
Note: Course policy about **plagiarism** is as follows:

- This assignment should be done individually.
- Students must not share actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection. We will run MOSS on your assignment and compare with both online solutions as well as solutions from previous years.
- Students cannot copy the code from the Internet. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Late day policy: You have a total pool of 5 days for late submissions across all the assignments, without deductions. Once you have used up all the 5 days, no late submissions for any assignments will be accepted.

Overview

In this assignment, you will implement a chat application (like Messenger). The application architecture is as follows:



There is one server and multiple clients. The clients are the users of the chat application. All clients must implement the functionality needed to reliably exchange messages and files. In this architecture, we will use a central server to keep track of which users are active and how to reach individual users. The server knows all the clients that are currently active (i.e., can receive and send messages) and how to reach them (e.g., current address). All message and file exchanges happen through the server. A client (e.g., Client_1) which wants to send a message to another client (e.g., Client_2), first sends the message to the server, which then sends it to the destined client.

Your job is to write Server and Client code. Implementation and testing should be done on the linux machines. Test cases will be available only for programs written on these machines.

1) Part-1: Simple Chat Application

The main goal of this part of the assignment is to introduce you to socket programming in the context of client-server application architectures. We begin by describing the role of the application server (1.1), application client (1.2), then provider a short introduction to socket programming (1.3), and finally describe in detail the protocol (1.4) to be used for exchanging messages between the client and server.

1.1) Application-Server

The application-server is a single-threaded server, listening to new connections from clients at a given **host** and **port**, accepting them and handling the messages sent from each client. The server can concurrently have up to **MAX_NUM_CLIENTS** clients use its service (i.e., while MAX_NUM_CLIENTS are connected to the server, another client can only join if someone disconnects). When a client requests to join a server which already has

MAX_NUM_CLIENTS, its request will be rejected. Furthermore, the server does not store information of disconnected clients. The server must handle all possible errors and remain alive until explicitly terminated.

1.2) Application-Client

The application-client represents the interface for your chat application, that connects to the application-server with a unique username (not already taken by any clients connected to the server). It reads user input from standard input and sends messages to the server, accordingly. It also receives and handles messages from the server. If the client gets any possible error from the server, it will end its connection with the server and shut down. It should show the user an informative message about why it's shutting down.

For the Application-Client and Application-Server to communicate with each other over the Internet, we will use socket programming. Below we describe it.

1.3) Socket Programming

Sockets allows applications to send and receive messages across a network. Sockets are an Operating System mechanism that connects processes to the networking stack. A **port number** identifies a socket. If an application has to send a message, it writes to a specific socket. If an application is expecting to receive messages, it listens on a socket for incoming messages. To create a socket, you need to (i) specify the transport protocol you will use, the (ii) the address of the machine (referred to as IP address) and (iii) port number you will use to identify the socket.

In this assignment, your client should connect with the server using the transport protocol, **UDP.** UDP is a basic transport protocol that just sends a message from the sender to the receiver with no guarantees about its delivery. For Part 2 of the assignment you will improve UDP to offer reliability guarantees about message delivery.

Furthermore, you will use the local host as your IP address as you will be running the server and client on the same machine. The port number must be different for each entity as one port can only be used by one process. The server should listen on a fixed port number which all the clients need to know beforehand. The clients should pick a random open port number when creating their sockets. Once you have established a socket, you can send and receive messages.

1.4) Application API and Protocols

Before describing the Application API and protocols in detail, we begin by listing down the sequence of events you should follow for establishing communication between clients.

- First establish a connection with the server by sending a "**`join`**" message from the client to an already running server.
- The server should add the new client to its list of clients and store its address so that it can send messages back to the client.
- Any client can then send messages to any other client(s) connected to the server, using the protocol described later.

For example, the user *client_spiderman* may type:
`msg 2 client_superman client_batman Hey there folks, Marvel is better than DC!`

- The client application should interpret this as a message being sent to '2' other clients, namely *client_superman* and *client_batman*. And the message text is *'Hey there folks, Marvel is better than DC!'*
- The client should then compile a message to send to the server. The server must receive the message, understand that it is incoming from *client_spiderman* and the 2 intended recipients are *client_superman* and *client_batman*.
- The recipients, *client_superman* and *client_batman,* should then receive a message from the server and display on the screen to their respective users:

`msg: client_spiderman: Hey there folks, Marvel is better than DC!`

- This describes the basic crux of the chat application; you must build up from here to include all the required functionality and exceptional handling, using the protocols described below.

Below we describe in detail the Application API and the protocols you will need to implement.

1.4a) Application API

Each Application-Client should be able to perform the tasks given below. For each function, the client must get the user-input from standard input (stdin) and process the input according to the below mentioned formats. If the input does not match with any format, the client should print on stdout: incorrect userinput format

Your chat application should support the following API:

1) **Message:**
   Func: Sends a message from this client to other clients connected to the server using the names specified in the user-input. The application-server must ensure that the client (whom the message is sent to) will only receive the message once even if his/her username appears more than once in the user-input.
   Input: **msg <number_of_users> <username1> <username2> … <message>**
2) **Available Users:**
   Func: Lists all the usernames of clients connected to the application-server (including itself) one name per line in ascendingly sorted order.
   Input: **list**
3) **File Sharing:**
   Func: Sends a file to other clients connected to the server. The other clients should save the file with the same filename (as specified by the sender) prefixed with their username (e.g.ammar_solution.py).
   Format: **file <number_of_users> <username1> <username2> … <file_name>**
4) **Help:**
   Func: Prints all the possible user-inputs and their format
   Input: **help**
5) **Quit:**

Func: Close the connection to application-server, print the following message to stdout and shutdown gracefully.

quitting

Input: **quit**

1.4b) Protocols and Message Formats

A message is a basic unit of data transfer between client and server. The different types of messages that can be exchanged between a client and a server are as follows (the packet formats mentioned are given below):

1) **join**

   Packet Format: Type 1

   Sender Action: This message serves as a request to join the chat. Whenever a new client comes, it will send this message to the server.

   Receiver Action: When a server receives this message, 3 things can happen at the server:

   - The server has already MAX_NUM_CLIENTS, so it will reply with ERR_SERVER_FULL message and will print

     disconnected: server full
   - The username is already taken by another client. In this case, the server will reply with ERR_USERNAME_UNAVAILABLE message and will print:

     disconnected: username not available
   - The server allows the user to join the chat. In this case, it will not reply but will print:

     join: <username>

2) **request_users_list**

   Packet Format: Type 2

   Sender Action: A client sends this message to the server when it reads a message **list** user-input.

   Receiver Action: The server will reply with RESPONSE_USERS_LIST message and will print:

   request_users_list: <username>

3) **response_users_list**

   Packet Format: Type 3

   Sender Action: The server will send the list of all usernames (including the one that has requested this list) to the client.

   Receiver Action: Upon receiving this message, the client will print:

   list: <username-1> <username-2> <username-3> … <username-k>

4) **send_message**

   Packet Format: Type 4

   Sender Action: A client sends this message to the server.

   Receiver Action: The server forwards this message to each user whose name is specified in the request. It will also print:

   msg: <sender username>

   For each username that does not correspond to any client, the server will print:

   msg: <sender username> to non-existent user <recv. username>

5) **forward_message**

   Packet Format: Type 4

Sender Action: The server will forward the messages it receives from clients. It will specify the username of sender in the message.

Receiver Action: The client, upon receiving this message, will print:

msg: <sender username>: <message>

6) **send_file**

Packet Format: Type 4

Sender Action: A client sends this message to the server. In the place of message, it will place the file (the filename will be placed before actual file content, separated by space).

Receiver Action: The server forwards this file to each user whose name is specified in the request. It will also print:

file: <sender username>

For each username that does not correspond to any client, the server will print:

file: <sender username> to non-existent user <recv. username>

7) **forward_file**

Packet Format: Type 4

Sender Action: The server will forward the files it receives from clients. It will specify the username of sender in the message.

Receiver Action: The client, upon receiving this message, will save the file and print:

file: <sender username>: <filename>

8) **disconnect**

Packet Format: Type 1

Sender Action: The client will send this to the server to let it know it's disconnecting

Receiver Action: The server upon receiving this, shuts down the connection and removes this user from its list of online users. The server will also print:

disconnected: <username>

9) **err_unknown_message**

Packet Format: Type 2

Sender Action: The server will send this message to a client if it receives a message, it does not recognize, from that client. The server will also print:

disconnected: <username> sent unknown command

Receiver Action: The client, upon receiving this message, closes the connection to server and shuts down with the following message on screen:

disconnected: server received an unknown command

10) **err_server_full**

Packet Format: Type 2

Sender Action: The server will send this message.

Receiver Action: The client, upon receiving this message, closes the connection to server and shuts down with the following message on screen:

disconnected: server full

11) **err_username_unavailable**

Packet Format: Type 2

Sender Action: The server will send this message.

Receiver Action: The client, upon receiving this message, closes the connection to server and shuts down with the following message on screen:

disconnected: username not available

Furthermore, there are 4 different message formats for your chat application as given below. Since the messages are sent/received as strings, the different fields in a message will be separated by a single space character (“ “).

Type: 1

Type	Length	UserName

Type: 2

Type	Length

Type: 3

Type	Length	List of UsersNames

Type: 4

Type	Length	List of UsersNames	Message

List of Usernames (in Type 3 and 4) will be formatted as:

Num of Users	User1	User2	...	UserN

The **Length** field in above messages is the length of message (excluding Type and Length fields).

1.4c) Packet Formation

In this chat application, your messages will be structured as follows: first header, followed by a chunk of your chat application data. A Packet will consist of the following pieces of information.

Packet:

- packet_type # start, end, ack, data (ignore for this part)
- seq_num # the packet number (ignore for this part, set to 0)
- data # the contents to be sent to the receiver, e.g., “Join message” above
- checksum # 32-bit CRC (ignore for this part)

In the first part of the assignment, you do not have to worry much about this. You will only be sending messages with packet_type equal to data.

2) Part-2: Reliable Communication

In this part, you will build a simple reliable transport protocol (like TCP) on top of **UDP** for your Chat Application. Your goal is to provide in--order, reliable delivery of UDP datagrams in the presence of events like packet loss, delay, corruption, duplication, and re-ordering.

2.1) Message Sender

The message sender will get an input message and transmit it to a specified receiver using UDP sockets following the new reliable transfer protocol. It should split the input message into appropriately sized chunks of data, and append a checksum to each packet. seq_num should be incremented by one for each additional packet in a connection.
You will implement reliable transport using a *sliding window mechanism* and will accept *cumulative ACKs* from the receiver. After transferring the entire message, you should send an END message to mark the end of connection. You must ensure reliable data transfer under the following network conditions:

1. Loss of arbitrary messages
2. Re-ordering of ACK messages
3. Duplication for any packet
4. Delay in the arrivals of ACKs.

To handle cases where ACK packets are lost, you should implement a 500 milliseconds retransmission timer to automatically retransmit packets that were never acknowledged. Whenever the window moves forward (i.e., some ACK(s) are received and some new packets are sent out), you reset the timer. If after 500ms the window still has not advanced, you retransmit all packets in the window because they are all never acknowledged.

2.2) Message Receiver

The message receiver must receive and store the message sent by the sender completely and correctly. It should calculate the checksum value for the data in each packet it receives. If the calculated checksum value does not match the checksum provided in the header, it should drop the packet (i.e. not send an ACK back to the sender).
For each packet received, it sends a cumulative ACK with the seq_num it expects to receive next. If it expects a packet of sequence number N, the following two scenarios may occur:

1. If it receives a packet with seq_num not equal to N, it will send back an ACK with seq_num=N.
2. If it receives a packet with seq_num=N, it will check for the highest sequence number (say M) of the in-order packets it has already received and send ACK with seq_num=M+1.

If the next expected seq_num is N, receiver will drop all packets with seq_num greater than or equal to N + window_size to maintain a window_size window.

Packet Formation

In this chat application, you will be sending messages in the format of a header, followed by a chunk of data.

The messages have four header types: start, end, data and ack, all following the same format:

```
Packet:
        packet_type  # start; end; data; ack
        seq_num    # the number of the packet as described below
        data   # the contents to be sent to the receiver
        checksum  # 32-bit CRC
```

In this part of the assignment, to initiate a connection, the sender starts with a START message along with a random seq_num value, and wait for an ACK for this START message. After sending the START message, additional packets in the same connection are sent using the DATA message type, adjusting seq_num appropriately. After everything has been transferred, the connection should be terminated with the sender sending an END message, and waiting for the corresponding ACK for this message.

The checksum is used to validate whether the contents of the packet have not been corrupted or changed from what the sender sent. The sender after making the packet, calculates its checksum and then adds that to the end of the packet.
The receiver upon receiving the packet, calculates the checksum of the packet themselves and compares it with the given checksum to see if they are the same. If they are not, then the receiver discards the packet. The checksum can be calculated using the given function in your skeleton code.

3) Instruction for Setup and Starter Code

The assignment requires you to use a Linux distribution and python3. Please install a VM (e.g. using VirtualBox) if you do not have this set up.

The starter code consists of 2 folders, each containing some python files:

1. server.py and client.py are the only files you should modify.
2. util.py contains some constants and utility functions (like calculating or validating checksum) that you can use.
3. TestChatApp.py is the test file that you can run to check your implementation of client and server code.

4) Testing

To run the server of chat application, execute following command:

```
$ python3 server.py -p <port_num>
```

Similarly, execute following command to run a client (with same port_num that you have provided to server.py):

```
$ python3 client.py -p <server_port_num> -u <username>
```

To test your submission, we will execute the following command from each folder (part1 and part2):

```
$ python3 TestChatApp.py
```

5) Grading

You can earn upto 30 points from this assignment.

Part-1

1. **SingleClientTest - 3 points**

A single client can exchange messages with the server.

2. **MultipleClientsTest - 3 points**

Multiple clients can exchange messages with each other.

3. **FileSharingTest - 3 points**

Multiple clients can exchange files with each other

4. **ErrorHandlingTest - 3 points**

Your code can handle both server and client errors.

Part-2

1. **BasicFunctionalityTest - 5 points**

Connections are correctly setup and data messages can be exchanged between clients.

2. **OutOfOrderPacketsTest - 2 points**

Packets are received in-order by clients even when the packets are reordered by the network.

3. **PacketLossTest - 2 points**

Packets are reliably received, even when there are losses inside the network.

4. **WindowSizeTest - 3 points**

We will check whether you set and adapt the sending window correctly.

5. **DuplicatePacketsTest - 2 points**

You code can handle the event of a receiver receiving packets from the network.

6. **Code Quality - 4 points**

- Your submission should have good coding style that adheres to the *formatting* and *name conventions* of Python.
- Your code should be modular, e.g., you may lose points if your code consists of a single large function that encapsulates all the functionality.
- Your code should be well commented.

Notes and Additional Tips

1. First, start off early. The assignment is not very difficult but there is a lot of new information that might overwhelm you, so it's important that you give it time. Read this handout carefully. Also, debugging will take a lot of time so make sure you avoid deadline day panic.
2. Unless otherwise stated, each line should end with a newline. The delimiter between each word of your input and output should be a single space.
3. Google is your friend. Google can give you answers to many questions faster than any TA can. Use these resources effectively. Questions ranging from string

manipulation to list-trimming or parsing through user input and dictionaries; these can be answered in less than a minute with a few simple keywords being searched for.

4. All the output specified in this document is done to stdout. Since we will be using stdout for testing, you must ensure that only the specified output goes there.
5. You are provided with helper functions in ***util.py***, that you are recommended to use. Before sending any messages over sockets, the messages **MUST** be in the format as output by the function ***util.make_packet()***.
6. ***util.make_message()***, although not mandatory, can be used to create messages for the protocol described in part 1. ***util.make_packet()*** is to be used to package the message in a *packet* with a pseudo header and checksum. This will be important in the second part of the assignment. For Part 1, the *msg_type* argument in ***util.make_message()*** will always be set to "*data*", and the *seqno* (sequence number) can be set to any value since it is not used in part 1.
7. `TestChatApp.py` will itself invoke instances of ***client.py*** and ***server.py*** so you do not need to have them running when testing. However, we recommend that you start working on your code by testing your ***server.py*** and ***client.py*** file manually, since that will be a better way to track your progress in the earlier stages.
8. Your server code **must not crash** at any point unless explicitly being told to do so. Make sure you have done exceptional handling to preclude failures.
9. Your code MUST follow pep8 standard and it must not get any warnings when running *pylint3 <filename>*.
10. Read point 1 again.