Programming Assignment 3

CSE 310 Spring 2024

Due date: May 10 2024; 11:59 PM

Submission via Brightspace

In this assignment, you will implement a chat application (like messenger) that will allow users to transfer messages. You will also implement a custom reliable transport protocol on top of UDP. The assignment 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 2.1: Extending the chat application in Part 1 to implement sequences and acknowledgements just like TCP.

Part 2.2: Extending the chat application in Part 2.1 to ensure reliable communication of messages in case of packet loss.

Overview

In your chat application, there will be 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. 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 (i.e. current address). All message exchanges happen through the server. A client (e.g., Client1) that wants to send a message to another client (e.g., Client2), first sends the message to the server, which then sends it to the destined client.

Your job is to write Server and Client code. You can add your own additional utility functions in the util.py file (if you want).

Part-1: Simple Chat Application

We begin by describing the role of the application server (1.1), application client (1.2), then provide 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 **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 that 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. 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.

1.3) Socket Programming

For the application-client and application-server to communicate with each other over the Internet, we will use socket programming. You have already been introduced to socket programming in your programming assignment-1. 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 localhost 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 it on the screen to their respective users:

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

- This describes the crux of the chat application; you must build up from here to include all the required functionality and exception handling, using the protocols described 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 user input format**

Your chat application should support the following API:

1) Message:

Function: 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:

Function: Lists all the usernames of clients connected to the application-server (including itself). All names must be in one line in ascending order.

Input: **list**

3) Help:

Function: Prints all the possible user-inputs and their format Input:

Input: **help**

4) Quit:

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

Input: **quit**

1.4b) Protocols and Message Format

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 message formats mentioned are given below):

1) **join**

Message Format: Type 1

Sender Action: This message serves as a request to join the chat. Whenever a new client connects, 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**

Message Format: Type 2

Sender Action: A client sends this message to the server when it receives a message **list** via standard input.

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

request_users_list: <username>

3) **response_users_list**

Message 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>

where the usernames are sorted A-Z.

4) **send_message**

Message 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

Message Format: Type 4

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

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

msg: <sender username>: <message>

6) disconnect

Message 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>

7) err_unknown_message

Message 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 the server and shuts down with the following message on screen:

disconnected: server received an unknown command

8) err_server_full

Message Format: Type 2

Sender Action: The server will send this message.

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

disconnected: server full

9) err_username_unavailable

Message Format: Type 2

Sender Action: The server will send this message.

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

disconnected: username not available

Each message type can be grouped according to the following 4 different message formats. 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 the above messages is the length of the message (excluding Type and Length fields). You have been provided with a function **make_message()** in the util.py file to format the messages for you.

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, set to data)
- 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.

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. You will be building up on your part1 implementation. Your goal is to provide reliable delivery of UDP datagrams in the presence of packet loss event

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 (size defined in util.py) and append a checksum to each packet. seq_num should be incremented by one for each additional packet in a connection.

To start a connection, you will send a START packet. After transferring the entire message, you should send an END packet to mark the end of the connection (the server will keep the client in its online users list until the client sends a **disconnect** message). To handle cases where a packet is lost and you don't receive an ACK for it, you should implement a 500-millisecond retransmission timer to automatically retransmit that packet that was never acknowledged and then move forward to send the next packet.

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 (i.e. current seq_num+1)

2.3) Packet Formation

In this part, your packet formation will stay the same. 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, ack, data
- 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** packet along with a random seq_num value and waits for an ACK for this START packet. After sending the START packet, additional packets in the same connection are sent using the **DATA** packet type, adjusting seq_num appropriately. After everything has been transferred, the connection should be terminated with the sender sending an **END** packet and waiting for the corresponding ACK for this packet.
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 function in your util.py file.

Instruction for Setup and Starter Code

The assignment test cases require you to use a Linux/MacOS and python3. Windows users can either use a linux VM or WSL.
The starter code consists of:

1. **server_1.py** and **client_1.py** are the files that you have to implement for Part 1.
2. **util.py** contains some constants and utility functions (like calculating or validating checksum) that you can use. Don't change the utility functions, but you can play around with different constant values. Please make sure you test your code with the default constant values. You can also add your own utility functions to it (if you want).
3. **TestPart1.py** is the test file that you can run to check your implementation of client and server code for Part 1. Once your implementation passes all the Part 1 tests, make new files **client_2.py** and **server_2.py** with the same code as **server_1.py** and **client_1.py**. Now you should use these files to implement logic for Part 2.
4. **TestPart2.1.py** is the test file that you can run to check your implementation of client and server code for Part 2.1.
5. **TestPart2.2.py** is the test file that you can run to check your implementation of client and server code for Part 2.2.

Manual 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 commands:

```
$ python3 TestPart1.py
```

```
$ python3 TestPart2.1.py
```

```
$ python3 TestPart2.2.py
```

Grading

Total Points: 100

Part-1 Test Cases

1. **ListUsersTest - 5 points**
   A client can see the list of online clients including itself.

2. **MessageTest1 - 5 points**
   A client can send a message to itself.

3. **MessageTest2 - 5 points**
   A client can send a message to another client.

4. **SingleClientTest - 10 points**
   A single client can exchange messages with the server.

5. **MultipleClientsTest - 20 points**
   A client can send messages to multiple clients simultaneously

6. **ErrorHandlingTest - 10 points**
   Your code can handle both server and client errors.

Part-2.1

1. **BasicFunctionalityTest - 20 points**

Connections are correctly setup and data messages can be exchanged between clients. (This test doesn't check your protocol's reliability. It just makes sure that the APIs provided in Part 1 are working correctly when you include seq/acks logic in them)

Part-2.2

1. **PacketLossTest - 15 points**

Packets are reliably received, even when there are losses inside the network. It is best to test your code against packet loss using TestPart2.2.py rather than testing it manually.

Code Quality - 10 points

- Your submission should have a 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.

Note: It is quite possible that sometimes a particular test can fail, while using it repeatedly and it's okay. First you should do manual testing and then move towards usage of test cases. Moreover, all the testcases make use of **quit** to disconnect the client, so make sure your is working correctly before using any testcase.

Notes and Additional Tips

1. First, start 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. 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. For debugging, Python's Logger library will be your best friend. Write all the debugging related output to a file using this library.
4. 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()***.
5. ***util.make_message()*** 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_packet()*** 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.
6. TestPart1.py, TestPart2.1.py, and TestPart2.2.py will invoke instances of ***client_1.py*** and ***server_1.py*** (for Part 1 tests) ***client_2.py*** and ***server_2.py*** (For Part 2 tests) by themselves 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*** and ***client*** files manually since that will be a better way to track your progress in the earlier stages.
7. The test cases are very robust and rigorously checks your **standard output**. Make sure that you are printing your output exactly as specified in 1.4b and are not printing extra spaces in between or extra newline characters at the end. These are very easy to overlook and frustrating to debug, so be extra careful. 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.
8. Your code must not get any warnings when running because it can lead to test failures even with correct logic.
9. To run just one test case of Part1, you can simply comment out the tests in **tests_to_run()** function in TestPart1.py
10. For Part-1, we have already implemented threading for you in the client file and you won't be needed to implement newer threads even for part-2. However, you will have to use Threading at the server side for Part-2 which you will implement yourself. (Part-1 server code doesn't require Threading). Additionally, you can use **Python Queues** to communicate among threads because of their blocking nature.

Submission Instructions:

Submit a single zip file that contains:

- testpart1 and testpart2 folder along with the given files inside them
- TestPart1.py, TestPart2.1.py and TestPart2.2.py
- Your client_1.py, server_1.py for part1 and client_2.py, server_2.py for part2 along with util.py

The zip file should be named using your last name, first name, and the assignment number, all separated by a dash ('-') e.g. lastname-firstname-assignment3.zip. Not following the submission instructions can result in partial deductions.