Assignment 1:

Due Date: Friday October 6th, 11:59 p.m.

Objectives:

- Practicing socket programming in python using stream sockets (TCP) and datagram sockets (UDP)
- Implementing a simple protocol similar to those you will encounter in this course
- knowing many terminologies used in computer networks

General Instruction

- You may work in groups of at most two students or individually.
- **Start early.** This assignment involves a substantial amount of work.
- Begin with the provided `UDPServer.py` and `UDPClient.py` files.
- You will be implementing a protocol consisting of four phases: the first two using UDP for communication, and the latter two using TCP.
- Ensure you use python version 3.6 or higher for this assignment.

Description

In this assignment, you will develop a client and server application that communicate using a specific protocol called **HaveFunCoding (HFC)**. The protocol consists of four phases. The initial two phases use UDP for communication, while the last two phases use TCP. Communication occurs through the exchange of packets, each comprising a header and data. The general format of a packet is shown in the following figure:

As you can see, a packet consists of a **header** and a **data** part. The header is located in the initial part of the packet and is 8 bytes. Within the header, the first four bytes represent **data_length**, indicating the length of the data part of the packet (Note: this length excludes the header length, it only specifies the data part's length).

The **pcode** is the code generated and sent by the server in the previous phase of the protocol. In the initial phase, pcode is set to zero. The client must extract the code sent by the server in each phase and use it in the subsequent phase. The server verifies that the client adheres to the protocol and will terminate the connection in case of any deviation from the protocol.

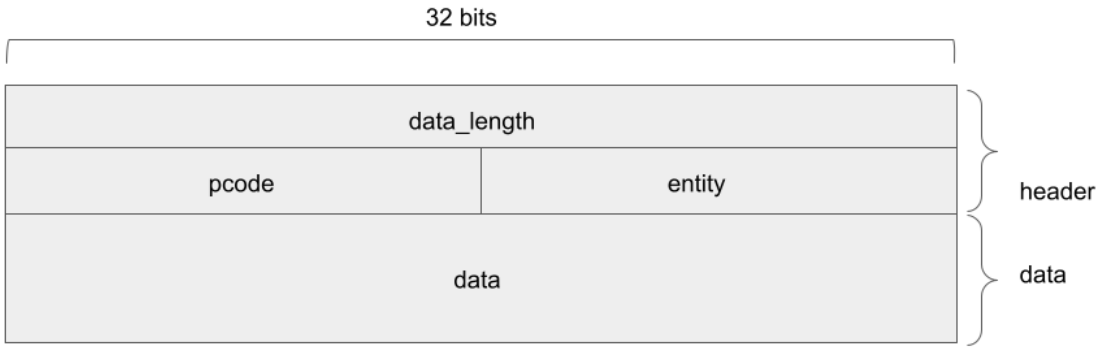
The following two bytes indicate **entity**. This field specifies whether the sender is the client (always represented as 1) or the server (always represented as 2).

The **data** part can vary in size; what's important is that the packet's overall size must be divisible by 4. In other words, when constructing a packet, the data part must be padded to ensure the packet length is evenly divisible by 4.

Server

The server initiates by running on UDP port 12000 and listens for incoming packets. These packets are expected to conform to the specified format outlined above. The specific value for each field will be detailed for each phase of the protocol.

The server validates the incoming packets to ensure they contain a header and follows the specified format. Additionally, it verifies that the packet's size is divisible by four. All numeric values within the packet should be either unsigned shorts or unsigned integers (as specified). The server will expect the characters to be in network order (big-endian). The server will close the connection with the client under the following circumstances:



- If the server does not receive any packet from the client for 3 seconds.
- If a packet arrives with a size that is not divisible by 4.
- If any of the packet's length, data length, data, pcode or entity values do not match the expectation for the specific phase of the protocol.
- If unsigned short or unsigned integer numbers are not used as required.
- If the characters are not properly aligned in network order (big-endian)

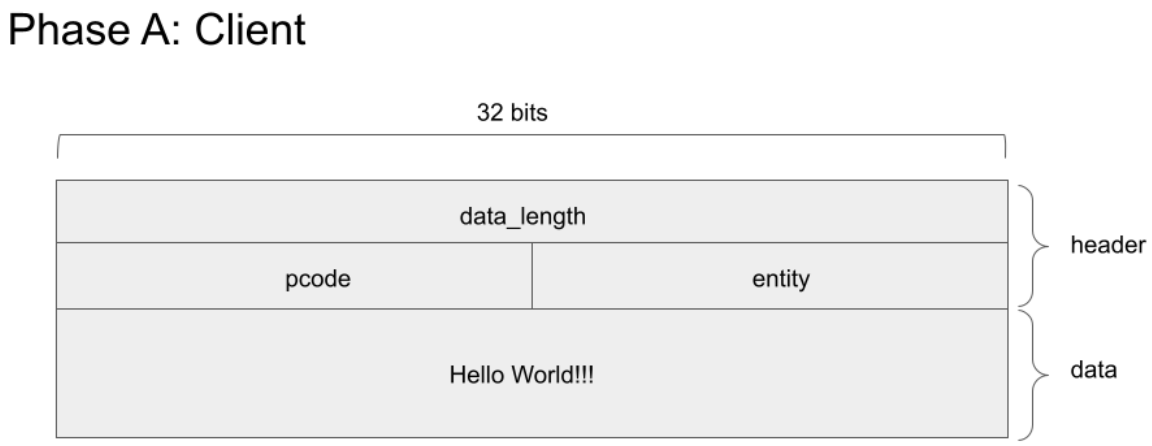
Below, you will find the description of the packet details for each phase of the protocol.

Phase A

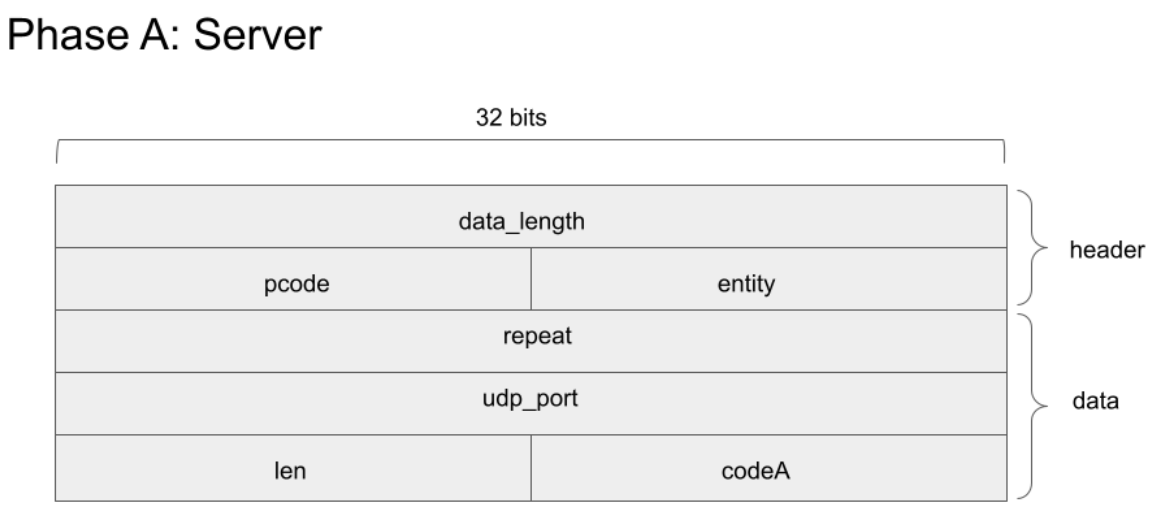
Client: The client sends a single UDP packet to the server with the format shown in the following figure.

- **data**: Hello World!!!. (without quotation marks and with exclamation marks)
- **data_length**: specifies the size of the data in bytes.
- **pcode:** initially set to zero.
- **entity**: Client.

Ensure the packet length is divisible by 4..



Server: Upon receiving the client's packet, the server verifies it and responds with a packet containing:

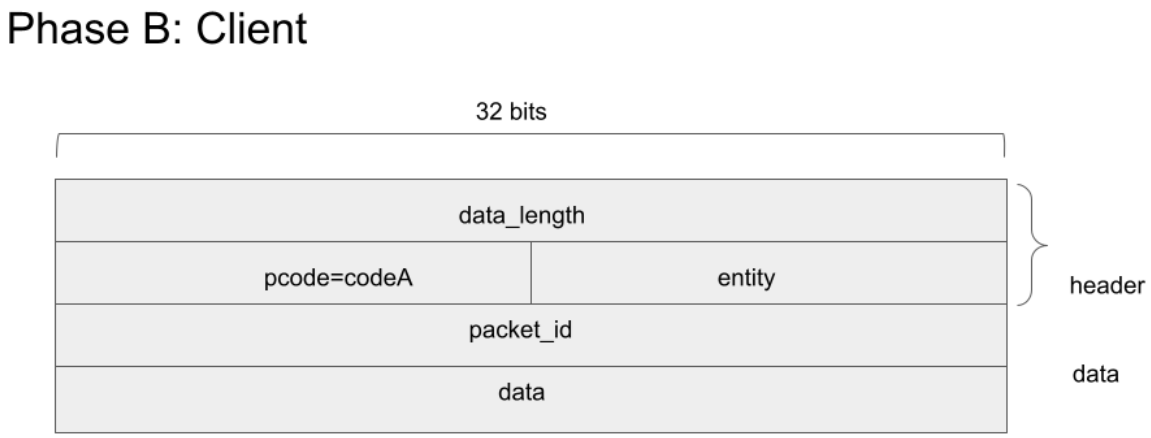


- **repeat**: a random integer (4 bytes) between 5 and 20.
- **udp_port**: a random integer (4 bytes) between 20000 and 30000.
- **len:** a short random number between 50 and 100.
- **codeA**: a short random number between 100 and 400

The server subsequently starts listening on the **udp_port** just sent to the client.

Phase B:

Client: The client **reliably** sends **repeat** UDP packets to the server on **udp_port**, each having the following format:



- **pcode:** is set to **codeA,** which was received in the previous phase.
- **entity**: Client.
- **packet_id:** an integer that uniquely identifies the packet, ranging from zero to “repeat-1”.
- **data**: This field comprises all 0’s, with its length adjusted to ensure the packet’s length is divisible by 4.The length of the data is determined by the “len” received from the server in the previous phase. To clarify, all measurements mentioned are in bytes. For instance, if “len” received in phase A is 70, two additional bytes are added to **data to** ensure divisibility of packet’s length by four. Note: In python you can use `bytearray(2)` to add two bytes containing zero.
- **data_length:** this field contains the length of the data (len + if any padding needed) and the length of the “packet_id”, which is 4 bytes.

Server:

Step B-1

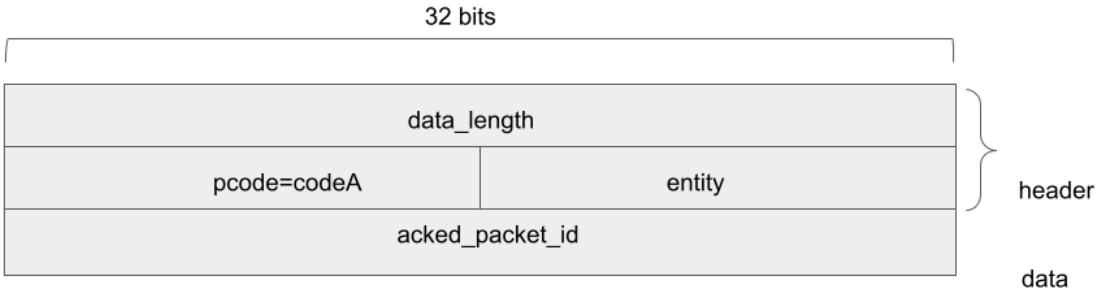
The server expects to receive **repeat** packets from the client. The **repeat** value was sent to the client in the previous phase. Upon receiving a packet, the server performs the following verification actions:

- The server checks that the received packet has the correct length. Remember, the **len** value sent in the previous phase may not necessarily match the length of the data part of the received packet because the data part may have been padded to ensure that the

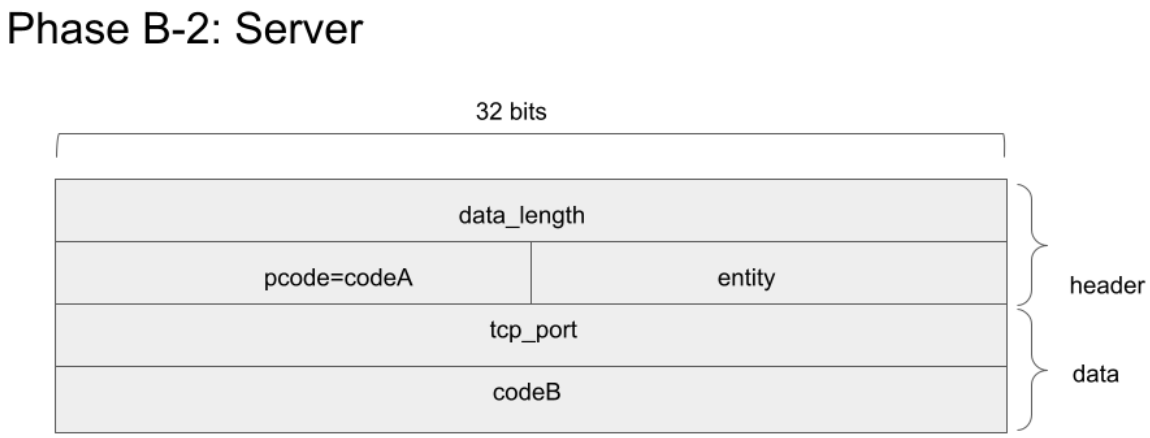
packet's length is divisible by 4. Additionally, the 4 bytes corresponding to "packet_id" should be taken into consideration.

- The server verifies that the **packet_id** field is correct. The first four bytes of the data part of the packet represent the integer number identifying the packet.
- The server ensures that packets arrive in the correct order.
- After completing the last three checks for each received packet, the server sends an acknowledgement packet, with the format shown in the following figure to the client. However, randomly, it may decide not to send an acknowledgement packet. To establish reliable communication, the client must receive the acknowledgement packets from the server for all **repeat** packets sent to the server. If the server does not send an acknowledgement for a packet, the client initiates resending of those packers and the server should anticipate receiving the same packet (with the same **packet_id**) once again. This process continues until the server acknowledges the packet. The client should use a retransmission interval of at least 5 seconds. You can use **settimeout** method from the socket library in python to do so.

Phase B-1: Server



Step B-2. Once the server receives all the **repeat** packets, it sends a packet to the client having the following format:



The **tcp_port** and **codeB** are random numbers generated by the server. Then the server starts listening on this **tcp_port** waiting for connection from the client.

- **tcp_port** is a random integer (4 bytes) between 20000 and 30000
- **codeB** is a random integer number between 100 and 400

Phase C:

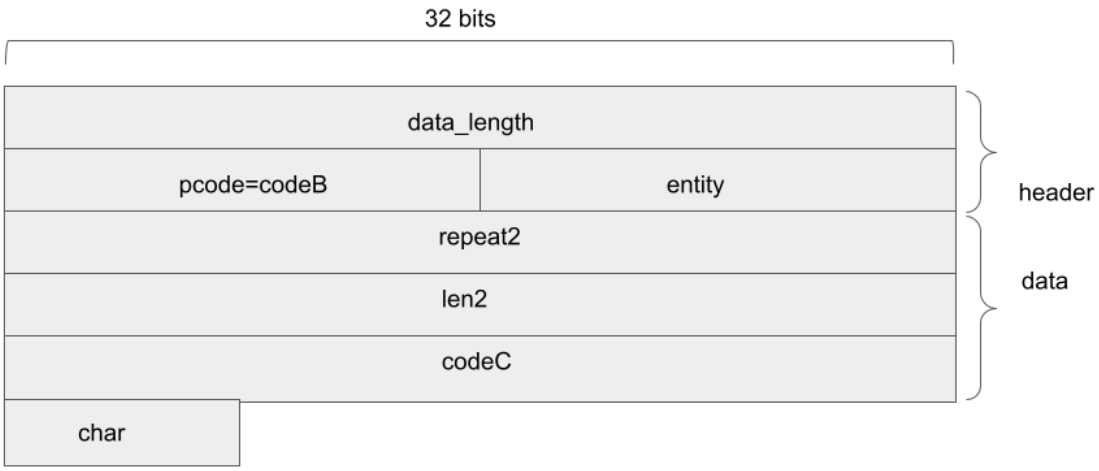
client:

The client connects to the server on the TCP port specified by **tcp_port** and initiates a TCP connection. The **tcp_port** value is received from the server in the previous phase. Before initiating the connection, it's important to ensure a brief wait (achieved by calling the "time.sleep" method in python) because the server may not have had sufficient time to start listening. Alternatively, make sure the server starts listening on **tcp_port** before sending the packet to the client in the phase B-2.

Server:

The server sends a packet to the client with the following format:

Phase C: Server



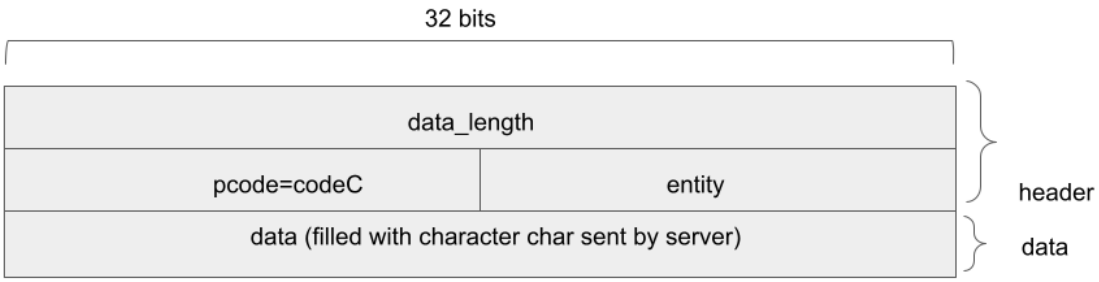
- **repeat2**: a random integer (4 bytes) between 5 and 20
- **len2**: a random Integer number between 50 and 100
- **codeC**: a random Integer number between 100 and 400
- **Char**: any of the 26 characters in English alphabet, from A to Z

Phase D:

Client:

The client sends **repeat2** packets to the server each having the following format:

Phase D: Client

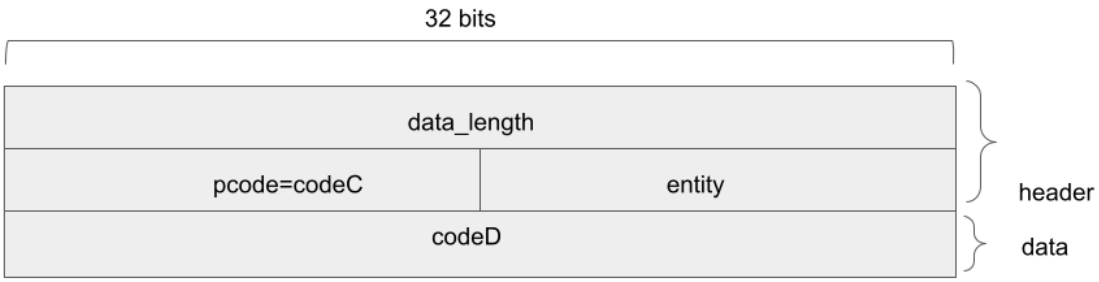


The length of data in each packet sent by the client to the server is **len2** (the packet may be padded to ensure that the packet length is divisible by 4), and each byte is set to the character **char** sent by the server in the previous phase.

Server:

Once the server receives all the packets, it responds by sending a packet to the client with the following format. The **codeD** is a random integer between 100 and 400.

Phase D: Server



Submission Instructions

- Create one file named **`server.py`**
- Create one file name **`client.py`**
- Put the above two files in a folder named **a1_x_y** where x is your username and y is your assignment partner username (e.g., if I were to develop this assignment with my partner having username barn407, then I would name my project as **a1_mrudafshani_barn407**)
- Zip the above folder (e.g., a1_mrudafshani_barn407.zip) and submit the zip file to the appropriate dropbox folder.
- **Do NOT submit an eclipse project!**
- **NOTE: Make sure to use python version 3.6 or higher.**
- **DO NOT USE python 2.7.**
- Make sure to put your names (as well as your partner's name if any) at the top of both the server.py and client.py.
- **Make sure to have one submission per group**
- **Assign yourself to a group if you want to work in groups**

Tips

- When using the TCP socket and testing your application, you may run the code several times and encounter the following error:

```
OSError: [Errno 98] Address already in use
```

To prevent this error, you can set a socket flag named **SO_REUSEADDR**. For more information, please refer to this resource.

- The `struct.pack()` and `struct.unpack()` functions are needed to create packets in the specified format. You need to use **I** for formatting unsigned integers and **H** for formatting unsigned short, and **!** to make sure the characters are in big-endian order. For more information, refer to this resource.
- You can utilize the **settimeout** function from the socket library to define a timeout for a socket
- To simplify development, use hardcoded values in your code where it instructs you to use a randomly generated number.

Grading

To test your code, I will run your client against my server and your server against my client. Here is the marking scheme:

[5 points] Following Submission Instruction

Server:
[10 points] Close connection when the client sends faulty packets or times out.
[10 points] Phase A
[10 points] Phase B
[10 points] Phase C
[15 points] Phase D

Client:
[10 points] Phase A
[10 points] Phase B
[10 points] Phase C
[10 points] Phase D

A significant portion of your mark will be deducted if your code does not run.
Try to develop and run your code step by step. Follow the hints and tips provided