Project 1 Report

Mehroz Akhtar - 922734865
Abdul Nawab - 917412765

Part 1 - iPerf

Files

Initial Implementation Files (With quite a few bugs / needed fixes)

- *Located in initial_impl folder*

udp_server_first[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py
udp_client_first[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py

ChatGPT / AI Assisted Implementation:

Note: *Lost track of links to various LLM Chats, used a bunch of tabs / different LLM models, however, we note below what we took away and learned from the LLM.*
This is our final implementation.

udp_server[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py
udp_client[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py

So, how to run the code?

In order to use our implementation. The first thing that you need to do is **instantiate** the server. Before running **any** commands, you need to ensure that you are in the proper directory for where the files are located, and be sure that you're running the *final (AI Assisted)* implementation files. Whereas the initial implementation files are buggy, and not as smooth as the AI Assisted implementation.

Open **two** terminals [ideally horizontally split to make your life easier], and do these steps in order:

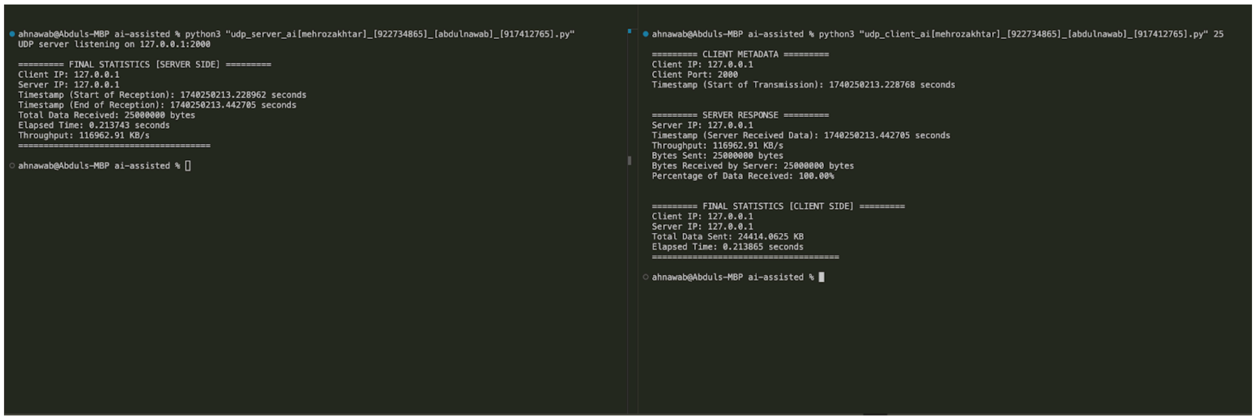
1. Make sure each terminal's directory is in **Part1**
- **cd Part1**

2. Run the server in a terminal:
- **python "**udp_server[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py"

3. Run the client in the 2nd terminal that you have open:
- **python "**udp_client[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py" <*Value between 25 and 200*>

Example Input for Client Side File:
python udp_client_ai[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py **25**

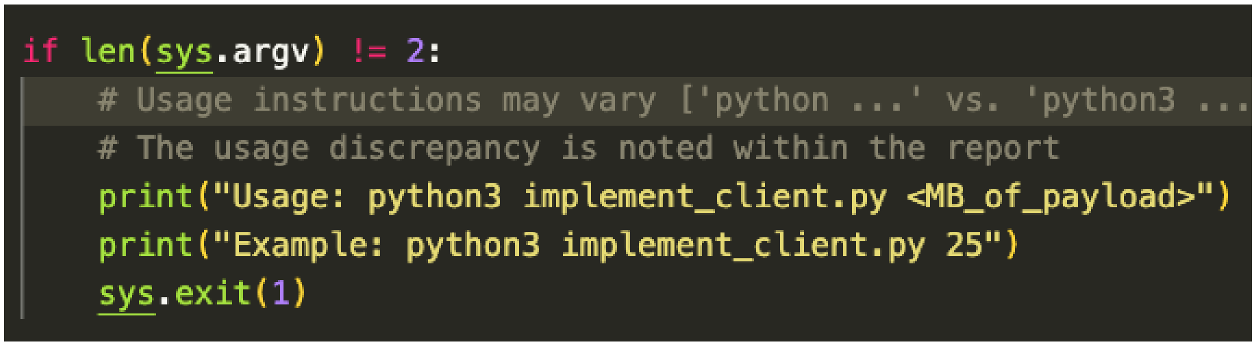
<u>Visual of split terminal view w/ input & server running:</u>



```
ahnawab@Abduls-MBP ai-assisted % python3 "udp_server_ai[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py"
UDP server listening on 127.0.0.1:2000

========= FINAL STATISTICS [SERVER SIDE] =========
Client IP: 127.0.0.1
Server IP: 127.0.0.1
Timestamp (Start of Reception): 1740250213.228962 seconds
Timestamp (End of Reception): 1740250213.442705 seconds
Total Data Received: 25000000 bytes
Elapsed Time: 0.213743 seconds
Throughput: 116962.91 KB/s
======================================

ahnawab@Abduls-MBP ai-assisted %
```

```
ahnawab@Abduls-MBP ai-assisted % python3 "udp_client_ai[mehrozakhtar]_[922734865]_[abdulnawab]_[917412765].py" 25

========= CLIENT METADATA =========
Client IP: 127.0.0.1
Client Port: 2000
Timestamp (Start of Transmission): 1740250213.228768 seconds


========= SERVER RESPONSE =========
Server IP: 127.0.0.1
Timestamp (Server Received Data): 1740250213.442705 seconds
Throughput: 116962.91 KB/s
Bytes Sent: 25000000 bytes
Bytes Received by Server: 25000000 bytes
Percentage of Data Received: 100.00%


========= FINAL STATISTICS [CLIENT SIDE] =========
Client IP: 127.0.0.1
Server IP: 127.0.0.1
Total Data Sent: 24414.0625 KB
Elapsed Time: 0.213865 seconds
=====================================

ahnawab@Abduls-MBP ai-assisted %
```

Left-hand side: server
Right-hand side: client



```python
if len(sys.argv) != 2:
    # Usage instructions may vary ['python ...' vs. 'python3 ...
    # The usage discrepancy is noted within the report
    print("Usage: python3 implement_client.py <MB_of_payload>")
    print("Example: python3 implement_client.py 25")
    sys.exit(1)
```

Screenshot of a comment within the client-side file that notes the usage discrepancy. In-depth explanation is written below.

<u>Note:</u>
Running the script may vary depending on the manner in which you have Python installed. For Abdul's machine, he needed to run both files prefixed with 'python3' instead of the conventional 'python' noted above. This difference may affect how the grader / Professor would have to run the program. Also, be sure to encapsulate the filename with " ", I was running into this problem when trying to run the filename as one would usually. I think the issue arises from the fact that there are square brackets within the file name.

<u>Sufficiently detailed explanation of the implementation at a high-level:</u>
For Part 1 of Project 1, we were asked to build a simple UDP Client / Server application that could send a certain amount of data within the range of 25 to 200 MB from a client to a server. The easiest way to think of this is that we are sending information from one computer, to **another** computer. The amount of data that we want to send is also referred to as the 'payload'

in computer networking terms. The actual data in our implementation doesn't really matter, it is the **size** of it that matters in how it is being transferred over. The actual data is just a bunch of A's that are concatenated (connected) together and transformed into the proper data form that is viable to be sent over the network. Within this project, we use a popular Python library called socket that helps us achieve building both Part 1 / 2 of this project. The socket library has extensive documentation on how to use it, and was heavily referenced during the development process of this project. Within the client side, we declare global variables such as the IP address, PORT number, as well as the buffer size. We then incorporate some basic error handling within our program. After running through the basic error handling, we then have a try catch block with an exception statement that will print out the following error associated with it. After completing all of the error handling, I then wrote some basic command line parsing. We grab the value that the user inputted. And then we convert this value from a string type to an integer type since that is how we will be able to work with converting it into the proper data form. We then convert the MB that the user inputted into bytes with the following conversion. We multiply the MB inputted by 1000000. This allows us to work with python's socket API. What we then do is generate the data that we will be sending to the server. We only create specific buffer sizes of this data. We then calculate how many chunks we will need to send, which will be found by taking the bytes, and dividing that value by the BUFFER_SIZE variable. What we do then is start a timer before we start sending data. We print some of the client metadata, such as the Client IP, Port, as well as the timestamp to indicate the start of the transmission of the data. Then, we iterate through all of the 'chunks' and send our data of the designated buffer size each time! Within the for loop of sending all of the chunks, I also added in a **timer**, which was an AI suggestion to solve some issues of packet dropping. The timer is designated to just be .00001 seconds. After iterating through all of the chunks, we also send the remaining data that could exist. This is also sent to the server. After sending all of our payload. We then send an end marker denoted by the string "STOP". This will essentially be read by the server and server-side logic will be implemented. After sending the "STOP" marker, we then end our timer. After sending everything from the client, we progress to receiving data back from the server. I implemented a try-catch block to receive the throughput calculation, server_address, as well as the bytes received from the server end. The client also receives the timestamp data, as well as the percentage of data that was successfully transmitted. The following lines print the entirety of the server response with all designated metrics and metadata. We then clean up our socket. Which cleans up all of the resources associated with it. And there is some additional printing of metadata as designated in the rubric. Now, I will explain the server-side implementation. For the server side, we declare the same variables such as the UDP_IP, PORT, and the same buffer size. All of the global variables match the client side global variables. We then create, and bind the UDP socket just like we did on the client side. After creating our socket. We instantiate a couple of variables: **total_bytes**, **start_time**, and **client_addr**. We then have a while loop which will start listening for data from the client side. We utilize the 'recvfrom()' method to listen for the data and address associated with the client-side. We then will store the client address on the client-side. We also start the timer on the first arrival of a packet. We also do a check if the data we are receiving is the end marker, in the case of receiving an end marker. We will break out of our while loop. At the end portion of our while-loop, we also increment the total bytes of data that we have received and continuously increment the value of that as we keep receiving data.

After receiving everything from the client side, we then will end our timer. After ending our timer, we will calculate the amount of time that it took for our data to be received from the client. This is done by taking the end time and subtracting from it the start time. This is only done if the start time is greater than ‘0’. If the start time is negative [which is an edge case] or simply is ‘0’, we will return ‘0’ to the client. This simple line takes care of potential edge cases and is specific enough for the user to know if there is an issue. We then calculate the throughput, which will take the total bytes, divided by the value of 1000, and divided by duration only if the duration of time is **greater** than ‘0’. Then, we print the final stats from the server-side. This includes the IP of the client and server, the timestamp [start of the reception], as well as the total data received, the elapsed time, and the throughput. After printing some stats on the server-side, we still need to send the **throughput**, the **bytes received**, and the **timestamp** to the **client.** We encode these values, and then call socket’s sendto() method, and send them to the client [Which is addressed as the client_addr]. After sending some final stats, we finally will clean up our socket on the server side.

<u>**LLM Section:**</u>

The usage of LLM was used to fine-tune our initial implementation as well as give us some tips on what we could make better. In our initial implementation, we kept running into issues with packet loss. In prompting the LLM, we asked what some good options for mitigating the issue were. As we all know, UDP isn’t notoriously known for being a secure transport protocol. It is just going to send data without any type of secure connection established. The failure of delivering some packets was an expected issue. Here are some of the solutions that various LLM have given to us in order to mitigate these issues. The first suggestion from the LLM was to add in a fairly small sleeping time between sending packets on the client side of our application. This allowed for there to be some time between sending packets which only ensured that our socket was not being overloaded. This was just one of the fixes that the LLM suggested. Another suggestion was to add in a more secure buffer size after instantiating the socket object. The following line that the LLM suggested was to adjust options that were able to be adjusted at the socket level. We set the buffer size to be 40 MB, this essentially specifies how much buffer space that the OS can use when receiving data. The LLM noted that this line of code assists us in having a higher level throughput value and from observation, it did make our program better / more efficient. Other miscellaneous tasks that the LLM assisted us with was code organization, better commenting, and just overall readability of our code. They were also extremely helpful in debugging syntax issues with our code and understanding sockets at a deeper level since they were not really discussed in class as in depth as other concepts. The LLM was also incorrect in explaining some aspects of how sockets worked, which required us to heavily refer to other resources such as general Googling, and go straight to the documentation of the Socket API. Overall, we still had to be wary with the LLM as opposed to fully trusting it.