Part 1:

- <u>How my code works:</u>
- I approached part 1 of this project by splitting up the total amount of data the client should send into many smaller packets
    - I chose each payload segment to be 1000 bytes as this ended up being small enough for the .sendto() function to properly send packets
    - I included meta data of the time that the client sends the data (4 bytes), and whether the client plans to send more packets or not (1 byte = 0 for more packets, = 1 if the current packet is the last packet)
- I made the server socket accept up to 1050 bytes to make sure that the entire packet is able to be received, since each packet from the client should be 5 bytes of metadata + 1000 bytes for the payload
- The server records the time that the packet is sent, decodes the metadata from the received packet, and calculates throughput by finding the difference in the time from the metadata and the server-recorded time, then adding the difference to a total sum to get the time for the entire message to be sent from the client
    - The server knows the total time for the entire message has been calculated when the incoming packet has the “last byte” set to 1
    - The overall throughput is calculated when the last packet has been received
- The server sends a response of more metadata back to the client so that the client can print the necessary information
    - The response metadata includes throughput, timestamp of the time that the last packet was received, the client’s IP, and the time since epoch that the last packet was received
        - I included 1 byte to represent the lengths for each of the metadata categories that precedes the given category so that the client can properly decode the received metadata
- The meta data could have been reported for each packet by having an array on the client side and recording the server responses by putting them into arrays

- <u>How to run my code:</u>
- To run my code, I opened 2 separate terminal pages and ran the following commands in order:
    - Terminal 1: $python udp_server_IsabelleLai_919259175.py
    - Terminal 2: $python udp_client_IsabelleLai_919259175.py [user-defined size of payload]
        - Replace user-defined size of payload with a number to represent the size of the payload in MB

- <u>Thoughts:</u>
- My code worked with my initial attempt/logic behind the process
- I struggled getting used to the socket package, but I used the documentation for the package provided on the internet to help understand what was going on

    - The key functions I learned were sendto() and recvfrom() which allows me to use udp without establishing a connection between the client and the server
- The ChatGPT part of this question made me realize that my response message from the server could have been simplified by formatting the response on the server side, converting the entire message to a string, or a dictionary with JSON format, and sending to the client; the client would just decode and print the response

ChatGPT:

I wasn't sure how to share a link of the conversation, so I recorded my responses and the final output that ChatGPT gave me.

My code already functions properly, so I chose not to implement any of the ideas from the ChatGPT code.

ChatGPT was able to send timestamp information and calculate throughput in a way that appears more concise in code, but logically differs from my understanding of how to calculate throughput. My code attempts to start recording time right before each packet is sent, ends recording time right after each packet is sent, then adds up each packet's sending time to get the total time for the entire message to be sent. The ChatGPT code records the time the first packet is sent to the time the last packet is received. I probably could have been more specific in the chat to get a better response that matches my logic more.

I also think that ChatGPT did a better job of sending the response back to the client. I did not think to encode all of the data in the response message as a single string before sending it back to the client.

My input:

- Code a python client and server using udp sockets. The client takes in a number from the user that represents the number of megabytes of data in a message that the client should send to the server. Make sure packets do not get dropped by sending information in smaller packets. Make the client print a timestamp of when the first packet was sent. The server should calculate throughput in kilobytes per second for the entire message to get received. Server should also make a timestamp of when the last data packet was received. The server should send the throughput, the ending timestamp, and the IP address of the client back to the client. The client the beginning timestamp, and the server response. The server should have no print statements"
- Fix it so that the time for the throughput is the time when the client sends the message to the time the server receives the message
- The client should send its time stamp to the server so throughput can be calculated correctly.

ChatGPT produced code:

Updated UDP Client:

python

Copy

```python
import socket
import time
import sys
import struct

def udp_client():
    # User input for message size in megabytes
    message_size_mb = int(input("Enter the number of megabytes of data to send: "))
    message_size_bytes = message_size_mb * 1024 * 1024  # Convert MB to bytes

    # Split the message into smaller packets (each packet will be 1024 bytes in this
    packet_size = 1024
    num_packets = message_size_bytes // packet_size
    if message_size_bytes % packet_size != 0:
        num_packets += 1

    # Create UDP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_address = ('localhost', 12345)  # Server address and port

    # Prepare a dummy message
    message = b'A' * packet_size  # A simple repeating pattern to send in packets

    # Record client start time and include it in the first packet
    start_time = time.time()  # Record the start time when first packet is sent
    start_time_bytes = struct.pack('d', start_time)  # Pack the time as a double (8 b

    # Send packets with timestamp in the first packet
    for i in range(num_packets):
        if i == 0:
            # Attach the start timestamp to the first packet
            client_socket.sendto(start_time_bytes + message, server_address)
        else:
            client_socket.sendto(message, server_address)

    # Send final empty packet to signal the end
    client_socket.sendto(b'', server_address)

    # Receive server response (which includes throughput and server timestamp)
    data, server = client_socket.recvfrom(4096)
    end_time = time.time()

    # Output the response from the server
    print(f"Sent {message_size_mb}MB of data.")
    print(f"Start time (Client): {start_time}")
    print(f"Server response: {data.decode()}")
    print(f"End time (Client): {end_time}")

    # Close the socket
    client_socket.close()

if __name__ == '__main__':
    udp_client()
```

Updated UDP Server:

python

Copy

```python
import socket
import time
import struct

def udp_server():
    # Set up server socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind(('localhost', 12345))  # Bind to the same port as client

    # Variables for throughput calculation
    total_received_data = 0
    start_time = None
    end_time = None
    client_address = None
    client_start_time = None  # To store the timestamp received from the client

    while True:
        # Receive data from client
        data, client_address = server_socket.recvfrom(1024)

        if data:
            # Extract the client timestamp from the first packet
            if start_time is None:
                client_start_time = struct.unpack('d', data[:8])[0]  # Extract client
                start_time = time.time()  # Record the server's start time when first

            # Count the number of bytes received
            total_received_data += len(data)

        if len(data) == 0:
            # End of message (empty packet received)
            end_time = time.time()  # Record end time when last packet is received
            break

    # Calculate throughput (Time from client sending to server receiving)
    total_seconds = end_time - client_start_time  # Time based on client timestamp
    throughput_kbps = (total_received_data / 1024) / total_seconds  # Kilobytes per s

    # Prepare the server's response to the client
    response = f"Throughput: {throughput_kbps:.2f} KB/s, End Time: {end_time}, Client

    # Send response to client
    server_socket.sendto(response.encode(), client_address)

    # Close the socket
    server_socket.close()

if __name__ == '__main__':
    udp_server()
```

Part 2:

- <u>How my code works:</u>
- I made the client have the addresses of the proxy and the server
    - The proxy and the server and each bound to their addresses, while the client remains unbound
- The proxy server has a blocklist of addresses that it will not communicate with if the server has an address from the list
- The client connects to the proxy server, takes the user's message and converts it to JSON format, and then bytes that can be sent over the connection, and sends the message to the proxy server
    - The client waits for a response from the proxy server
- The proxy server waits for a client to connect to it ("listens"), accepts the connection when it comes in, which produces a separate socket where information between the two end hosts can be transmitted between each other, and receives a message
    - The proxy pulls out the JSON formatted data from the message to get the IP address of the server the client wants to send the message to, and the message to send to the server
    - The proxy attempts to connect to the server address, and checks if the server address is in the blocklist; if it is, an error message is sent back to the client to say that the message did not go through
    - If the server is not on the block list, the proxy server sends the message and waits for a server response
- The server waits for incoming connections (from the proxy server), and accepts them when they come
    - The message it receives from the proxy server is decoded and printed, and the string "RESPONSE || " is appended to the start of the message and sent back to the proxy server as the server response
    - The connecting socket to the proxy server is then closed, and the server continues waiting for incoming connections
- When the proxy server receives a response from the server, it prints the response that the server sent it, and forwards the response to the client that sent the initial message
    - The server connection and the client connection are then both closed, and the proxy server waits for more incoming requests for connections
- When the client receives a response from the proxy server, it prints the response and closes its connection with the proxy server

- <u>How to run my code:</u>
- To run my code I ran the following commands in order:
    - Terminal 1: $python proxy_server_IsabelleLai_919259175.py
    - Terminal 2: $python server_IsabelleLai_919259175.py
    - Terminal 3: $python client_IsabelleLai_919259175.py [4 letter message to send]
        - Replace 4 letter message to send with a message

- <u>Thoughts:</u>

- I struggled understanding how some of the functions worked with the TCP connection
    - I had to search up how the listen() and accept() functions worked, and I learned that the accept() function works by creating a new socket between the client and server that allows them to communicate, while the listening socket remains bounded to the server and always listens for more connections to be made
- The JSON format went smoothly and made sending data easier