HOMEWORK 1

UNTU GEORGE-ADRIAN, MISS1

**Explanations regarding implementation and tests:**

1. **TCP**

The TCP approach used in the provided code demonstrates a basic TCP server-client communication model in Python using the socket library. TCP (Transmission Control Protocol) is a connection-oriented protocol, ensuring reliable, ordered delivery of a stream of bytes between applications running on hosts communicating via an IP network. Here’s how the TCP approach is implemented and works within the code:

**TCP Server**

Socket Creation: The server starts by creating a TCP socket with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM). AF\_INET specifies the IPv4 address family, and SOCK\_STREAM indicates that this is a TCP socket.

Binding and Listening: The server socket binds to an empty string for the host (''), meaning it accepts connections on all available IPv4 interfaces, and to a specified port (port). After binding, it listens for incoming connections with s.listen(). This prepares the server to accept connections.

Accepting Connections: With conn, addr = s.accept(), the server accepts an incoming connection. The accept method blocks and waits for an incoming connection. When a client connects, it returns a new socket object representing the connection (conn) and a tuple holding the address of the client (addr).

Receiving Data: Once a connection is established, the server enters a loop where it receives data from the client using conn.recv(buffer\_size). The buffer\_size parameter determines the maximum amount of data to be received at once. The server accumulates the total data received in total\_data\_received.

Session End and Reporting: When the client closes the connection, recv returns an empty bytes object (b''), signaling the end of data transmission. The server then calculates the session duration, prints the protocol used (TCP), the estimated number of messages read (assuming each "message" is of size buffer\_size), and the total bytes received.

**TCP Client**

Socket Creation: Similar to the server, the client creates a TCP socket.

Connecting to the Server: The client uses s.connect((server\_ip, port)) to initiate a connection to the server specified by server\_ip and port.

Sending Data: The client breaks down the data into chunks of buffer\_size and sends each chunk in a loop using s.send(). The total bytes sent are tracked.

Transmission Time and Reporting: The client measures the time it takes to send all the chunks and, after the transmission is complete, prints the total transmission time, the number of messages sent (chunks), and the total bytes sent.

1. **UDP**

The UDP (User Datagram Protocol) approach in your code illustrates how to set up a basic UDP server-client communication using Python's socket library. Unlike TCP, UDP is a connectionless protocol, which means it does not establish a dedicated connection before sending data. This results in a simpler, faster communication process, though it lacks the built-in mechanisms for reliability, order, and data integrity that TCP provides. Here's how the UDP approach is implemented in the code:

**UDP Server**

Socket Creation: The server initiates a UDP socket with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM). AF\_INET specifies the IPv4 address family, and SOCK\_DGRAM indicates a UDP socket.

Binding: The server binds to an empty string for the host ('') and a specified port (port), indicating it's ready to receive data sent to this port from any interface on the machine.

Receiving Data: The server enters a loop, continuously calling s.recvfrom(buffer\_size), which waits for data from clients. recvfrom returns the received data and the address of the sender. The server processes each received datagram independently since UDP is connectionless.

Acknowledgment (ACK): In the stop-and-wait implementation, for each packet received, the server sends back an "ACK" message to the sender's address. This requires the client to wait for this acknowledgment before sending the next piece of data, introducing reliability into the otherwise unreliable UDP communication.

Handling Session End: The server does not inherently know when a client is done sending data, as UDP lacks connection termination signals. Implementing a mechanism for ending the server's listening loop (like checking for a specific "shutdown" message or setting a timeout) is left to the developer.

**UDP Client**

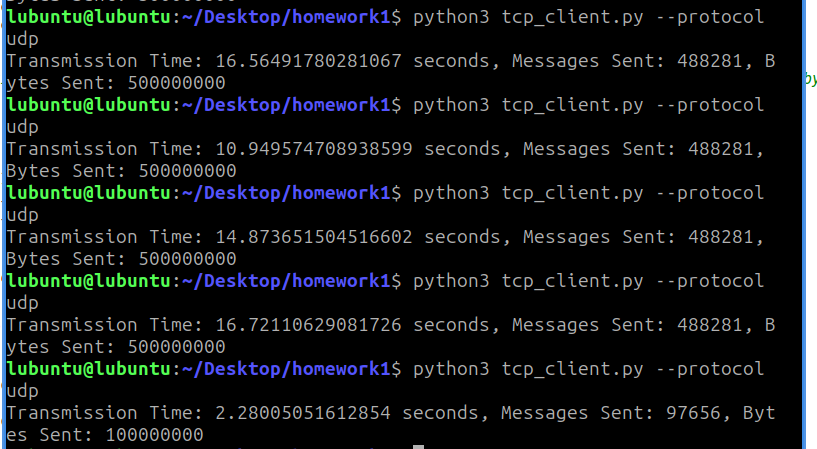
Socket Creation: Similar to the server, the client creates a UDP socket.

Sending Data with Stop-and-Wait: The client breaks the data into chunks and sends each chunk using s.sendto(), specifying the server's address. After sending each chunk, it waits for an acknowledgment from the server before proceeding. This is achieved through a loop that resends the current chunk if no acknowledgment is received within a specified timeout, introducing reliability into the transmission.

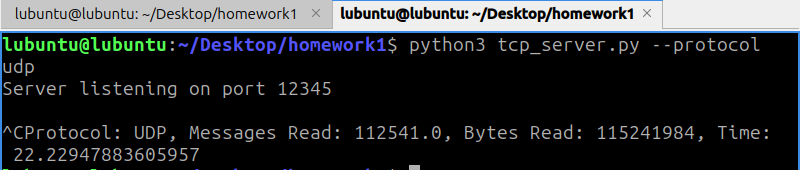
Timeouts: The client uses s.settimeout() to specify how long it should wait for an acknowledgment before considering the attempt failed and resending the data. This is crucial for handling packet loss or delays in the network.

Transmission Report: The client measures and reports the transmission time, the number of chunks (messages) sent, and the total bytes sent.

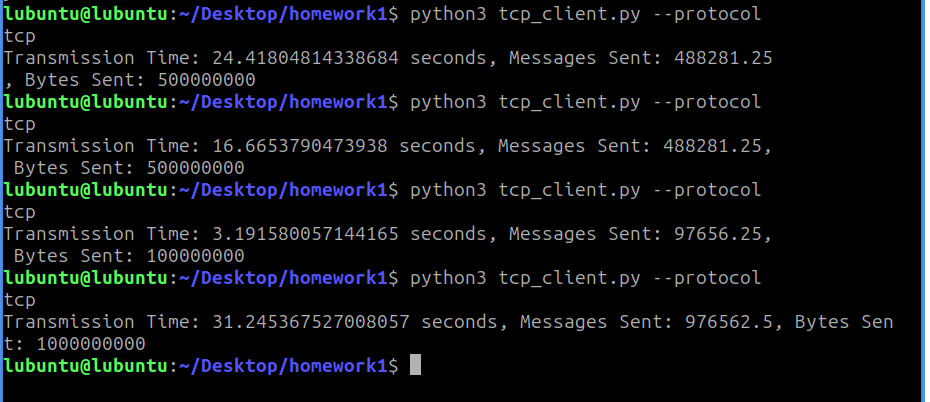
1. **Test results:**
2. For UDP:

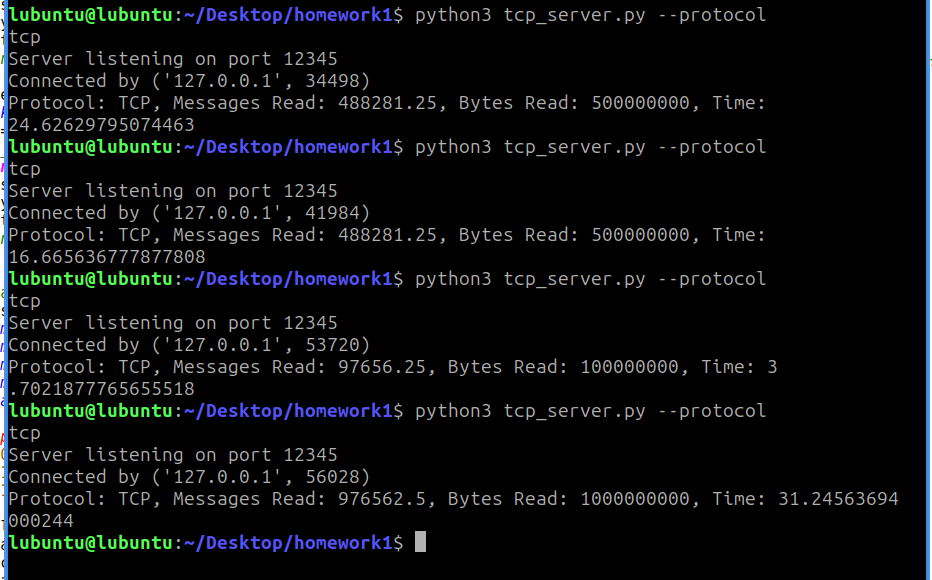
Client: 

Server:



1. For TCP

Client: 

Server: 

By the test results, the TCP connection is generally slower from my tests, which included sending 100MB/500MB/1GB of bytes messages from a client to a server and to transfer the data. Messages are generally 1MB in size, so the comparison is fair between the two methods. I implemented the tests in Python and they were done in Lubuntu from a virtual machine configured using VirtualBox.