Part 1

Design

The **pythonClient.py** file contains a single function called echo_client which takes as input an integer representing a port and a string representing a server hostname. The function then begins by creating a socket object and connecting it using the server_address variable; a tuple containing the hostname and port. Once connected, the client will request the user to enter some text. It then sends the message to the server and waits for a response. Upon receiving a response, the client prints the message and then closes the socket. I have also included an argument parsing section outside of the function in order to handle the inputs from BASH. The parser is included in all 4 python files and introduces a parser object which takes two arguments: a port and a hostname. It then will call the echo_client function on the given port and host.

The **pythonServer.py** file contains a single function called echo_server which takes as input an integer representing a port which the server will operate on. The server then creates a socket object and binds it to the server address using itself as the hostname and the given port as the port. The socket is then set up to listen for incoming messages. The program then runs a while loop which waits for data. Upon receiving data, the server prints the message and sends it back to the client. It finishes by closing the socket. This program again includes a parser section. See the section on pythonClient.py for more information on this section.

Experimentation

To test part 1 of the assignment, I typed various arbitrary text messages and ensured that they were all successfully printed by the server and then sent back to the client. As long as the server prints the correct message and the client prints the correct message, I determined that the program would successfully echo just about any message.

Problems and Expansion

The largest oversight in this program is its simplicity. It is designed to send data which has a maximum length of 2048 bytes. Should a message be sent that is too long or is truncated by TCP, it would fail to properly echo the message. The expansion necessary for this to work would be the inclusion of a while loop which checks for delimiter characters at the end of the message. The loop would constantly receive data and append it to the end of an accumulator variable until it reaches a delimiter, at which point the loop would break and the message would have been completely received.

Part 2

Design

echoClient.py - echo_client

The echoClient.py program contains 2 functions. Echo_client takes as input a port and a hostname. It runs a while loop which connects to the socket of the server and requests input in the proper format of a CSP or MP message. It then sends that to the server and waits for a response. If the message was a properly formatted connection setup message, the function splits the message at the spaces and stores each part of the message in descriptive variables. It then runs a loop which calls the second function, rtt(), as many times as the number of probes it needs to send. In the loop, it ensures that the server didn't receive an incorrectly formatted message and then takes the time returned by the rtt function and adds it to an accumulator. Once the loop is finished, it calculates the RTT and checks to see which measurement type was inputted. If RTT was inputted, it displays the RTT. If tput was inputted, it calculates the throughput and prints that. If the user inputs a properly formatted termination message, the program closes the socket and exits gracefully.

echoClient.py - rtt

The rtt function takes as input an int representing the number of probes to send, a payload size int, the id of the current probe as sent by echo_client, a socket, and a server address. It constructs a measurement message by sending payloadSize characters in a message with the protocol phase and the probe ID. It takes a timestamp and sends the message. The function then enters a loop which ensures that any truncated messages are fully received before taking another timestamp. The difference in the times is the rtt for a single probe which is then sent back to the echo_client function. If a probe was sent out of order or if too many probes were sent, the function returns -500 which will result in an error message being printed. This file also contains the code for a parser. Details are included above under **pythonClient.py**.

echoServer.py - echo_server

The echoServer.py file contains two functions. Echo_server takes as input a port number and then creates a socket to listen for incoming data on that port. The program enters a loop which waits for data. Upon receiving a message, it determines whether or not the message is a properly formatted CSP or TP message. It does so by splitting the message at the spaces and determining whether each section of the message is valid as well as a bunch of try-catch statements which look out for indexing errors as a result of bad user input. If the server received a properly formatted termination message, it returns a 200: OK message to the client and closes the socket and exits gracefully. If it receives a properly formatted connection setup message, it stores each value of the message and calls the rtt() function. If it receives a message that looks like a termination message but is incorrectly formatted, it returns a 404: ERROR and closes the connection. It functions similarly when receiving improper connection setup messages.

echoServer.py - rtt

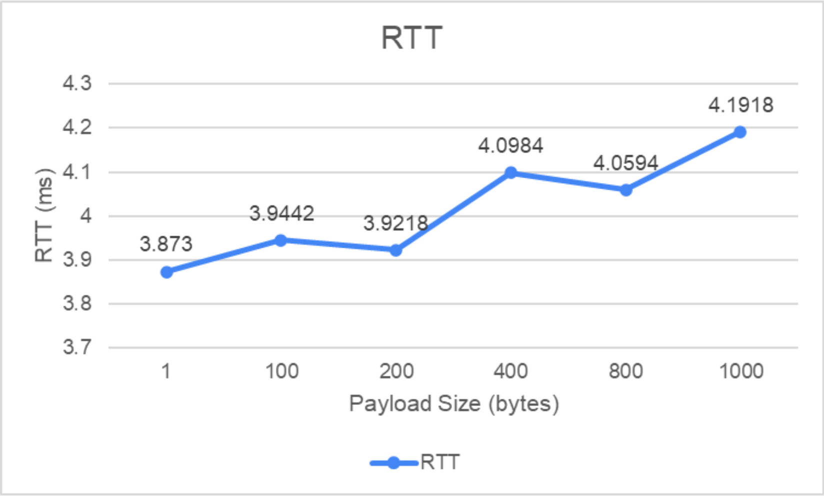
The RTT function takes as inputs the number of probes to be sent, a server delay value, a socket, and a server address. It begins by initializing a counter variable for the number of probes and converts the server delay value to milliseconds. It responds to the client with 200: OK Ready and then enters a loop. This loop begins by initializing an accumulator variable for the message. It enters another loop and begins receiving data until it reaches a newline character, at which point the message is complete. It then splits the message to confirm that the probe number is 1 more than the last probe received and that the number of probes hasn't exceeded the number to be sent. If these conditions hold, the server will sleep for the duration of the delay and then echo the message back to the client. If the conditions do not hold, it throws an error. This file again contains a parser. Details can be found in the **pythonClient.py** section.

Experimentation

In general, when testing this program I attempted to leave out parts of the connection setup message as well as messing up the format of the termination message in ways that would seem common. Additionally, I passed in invalid arguments for different parts of these messages and ensured that I received the proper error messages. I then created real, properly formatted messages and ensured that data of just about any size would be sent across.

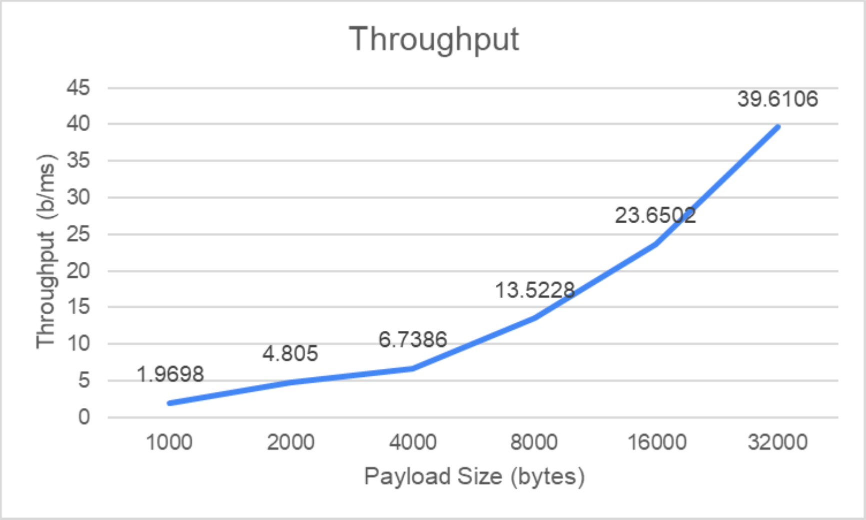
Testing the RTT

For testing the RTT, I did 5 trials averaged together for each point, each with 50 probes of sizes 1, 100, 200, 400, 800, and 1000 bytes with no server delay. These trials yielded an average of how RTT would be affected by larger message size. In general, larger message sizes yielded higher RTT values, but sometimes there were outliers. The below graph shows RTT in milliseconds.



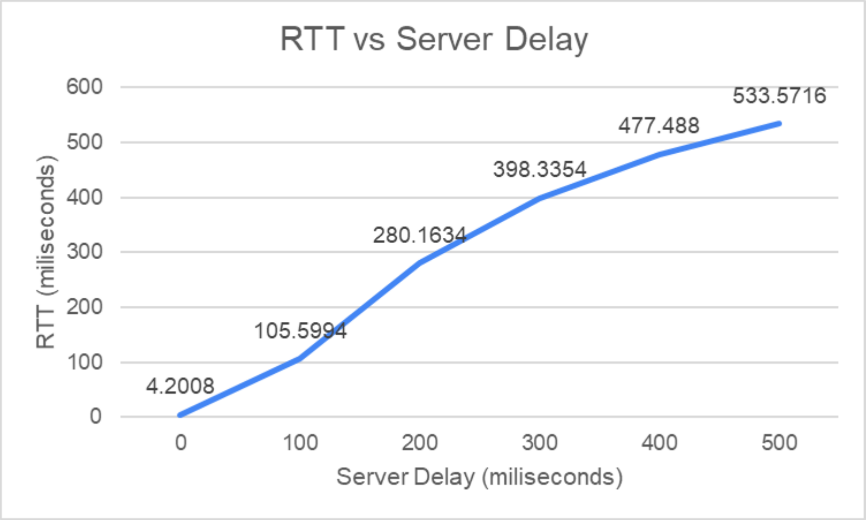
Testing the Throughput

To test the throughput, I did 5 trials averaged together for each point. Each trial sent 50 probes of sizes 1000, 2000, 4000, 8000, 16000, and 32000 bytes with no server delay. These trials show how throughput is impacted by the size of the messages being sent across. In general, larger message sizes with no delay had higher throughput considering more data was being sent across. The below graph shows throughput at various data sizes in bytes per millisecond.



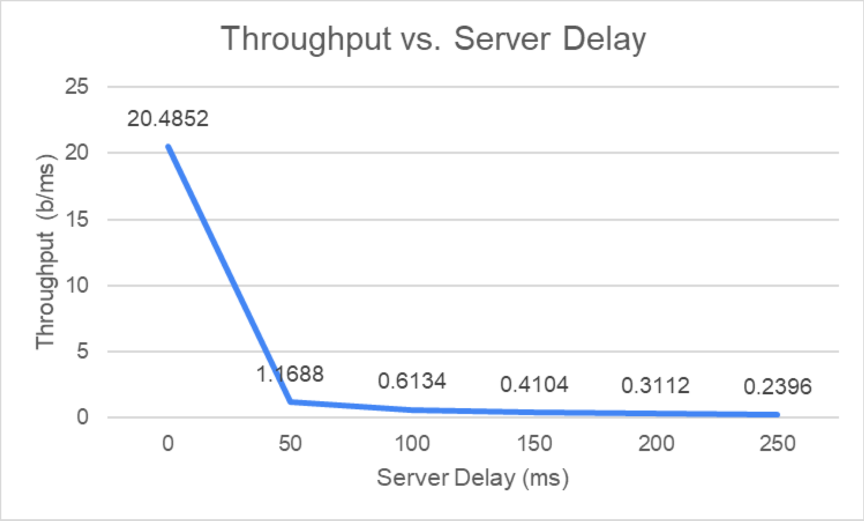
Testing the RTT vs Server Delay

In order to see the effects of server delay on RTT, I did 5 trials averaged together for each delay value. Each trial contained 10 probes and every trial used probes of size 800 bytes. I did 5 trials for each of the 5 server delay values - 0ms, 100ms, 200ms, 300ms, 400ms, 500ms - and the general trend is that RTT increases linearly with the server delay value as seen in the below graph.



Testing the Throughput vs Server Delay

In order to see the effects of server delay on the throughput, I again did 5 trials averaged together at 5 different server delay values. For these trials I used 10 probes each with an 8000 byte payload with varied server delay. For the 5 delay values, I did 0ms, 50ms, 100ms, 150ms, 200ms, and 250ms. The reason for less delay in this case was due to the large payload of the probes. The graph shows an exponential decrease in throughput as the server delay is increased linearly.



Problems and Expansion

The main issue with this program is interoperability. Although it functions perfectly fine with the server file I wrote, the client program will not function properly when being tested on another server file. Because debugging this issue was difficult without being able to edit the server code, I can only assume that there is either an issue with how my client is receiving server messages or there is an issue with the client formatting and sending the messages.

A way this code could be improved is in the try-catch series. Rather than hard coding the different checks, I could've implemented a for loop which checked each index of the split message individually rather than a hard coded set of try-catch messages.