Ben Pogacar
CS 455
10/13/2023

Part 1 Documentation

class echoClient:

Private Attributes:

- clientSocket
  - A Socket variable that is used to create the socket with a given ip,port in order to connect to the server
- out
  - A PrintWriter variable that is used to write data to the server that the client has connected to.
- in
  - A BufferedReader that is used to read data in from the server that the client has connected to.

Public Methods:

- main(ip, port)
  - When running the file it accepts two inputs, an ip address and a port number, the main method is used to ensure this, as well as the validity of those inputs, then it calls the startConnection method to start the client. Does not return anything.
- startConnection(ip, port)
  - Takes in the ip and port parameters and initializes the clientSocket, out, and in attributes. This will fail if there is no server open at the corresponding port. Then it accepts an input from the reader, sends it to the server, waits for the response, prints the response, and then closes. Does not return anything.
- stopConnection()
  - This is called to close the connection, it closes all 3 attributes, in, out, and clientSocket, which will close both the server and the client. Does not return anything.

class echoServer:

Private Attributes:

- serverSocket
  - A ServerSocket variable that is used to create the socket with a given port number in order to connect to the client
- clientSocket
  - A Socket variable that is used to store the connected client socket
- out
  - A PrintWriter variable that is used to write data to the client that the server has connected to.
- in
  - A BufferedReader that is used to read data in from the client that the server has connected to.

Public Methods:

- main(port)
  - When running the file it accepts one input, a port number, the main method is used to ensure this, as well as the validity of that input, then it calls the start method using that port to start the server. Does not return anything.
- start(port)
  - Initializes the 3 attributes, in, out, and serverSocket, and then uses serverSocket's .accept() method to listen for and connect to a clientSocket. Once connection is established it will listen for a message from the client using the in attribute, and then echo that message back with the out attribute. Upon completion of those steps it will call stop() to close the server (this is redundant in my implementation of the client, since it already closes both, but is to ensure it will work with other clients that do not do that). Does not return anything.
- stop()
  - Closes the serverSocket in order to shut down the server. Does not return anything.

Part 2 Documentation

class echoClient:

Private Attributes:

- clientSocket
  - A Socket variable that is used to create the socket with a given ip,port in order to connect to the server
- out
  - A PrintWriter variable that is used to write data to the server that the client has connected to.
- in
  - A BufferedReader that is used to read data in from the server that the client has connected to.
- numProbes
  - An int that is used to store the number of probes to be sent to the server during the measurement phase.
- payloadSize
  - An int that is used to store the payload size of each probe sent during the measurement phase.
- measurementType
  - A string used to identify which type of measurement the user wants to perform during the measurement phase.

Public Methods:

- main(ip, port)
  - When running the file it accepts two inputs, an ip address and a port number, the main method is used to ensure this, as well as the validity of those inputs, then it calls the startConnection method to start the client. Does not return anything.
- startConnection(ip, port)
  - Takes in the ip and port parameters and initializes the clientSocket, out, and in attributes. This will fail if there is no server open at the corresponding port. Then it goes through the 3 phases of operation in the correct order; connection, measurement, then termination. Does not return anything.
- stopConnection()
  - This is called to close the connection, it closes all 3 attributes, in, out, and clientSocket, which will close both the server and the client. Does not return anything.

Private Methods

- connectionPhase()
    - This method gathers input from the user to determine and build the corresponding connection setup message to send to the server. It ensures that each input is valid for whatever type it is supposed to be. It then sends the message to the server. It does not return anything.
- measurementPhase()
    - This method creates the probes to be sent to the server using this.numProbes and this.messageSize, and then sends them off, tracking how long each one takes to be returned by the server. After sending and receiving all the messages it will calculate and print out the measurement corresponding to the one specified in this.measurementType. It does not return anything.
- terminate()
    - This method creates the termination message to send to the server, prints the response message and then closes both the client and server. Does not return anything.
- getIntegerString(input)
    - This method takes in one parameter, which is a BufferedReader, and uses it to get some user input from the keyboard, it then checks that input to ensure it is a valid integer, repeating until it gets one. It returns the gathered input as a string

class echoServer:

Private Attributes:

- serverSocket
  - A ServerSocket variable that is used to create the socket with a given port number in order to connect to the client
- clientSocket
  - A Socket variable that is used to store the connected client socket
- out
  - A PrintWriter variable that is used to write data to the client that the server has connected to.
- in
  - A BufferedReader that is used to read data in from the client that the server has connected to.
- numProbes
  - An int used to store the number of probes expected from the client.
- delay
  - An int used to store the amount of time the server should wait before sending back its messages.

Public Methods:

- main(port)
  - When running the file it accepts one input, a port number, the main method is used to ensure this, as well as the validity of that input, then it calls the start method using that port to start the server. Does not return anything.
- start(port)
  - Initializes the 3 attributes, in, out, and serverSocket, and then uses serverSocket's .accept() method to listen for and connect to a clientSocket. Then it calls the 3 private methods connectionPhase(), measurementPhase(), and terminationPhase() in that order according to standard protocol.
- stop()
  - Closes the serverSocket in order to shut down the server. Does not return anything.

Private Methods:

- connectionPhase()
    - Receives a message from the client, determines its validity as a setup message and then sends a string back to the client letting it know if it was valid. If it is not valid, the server will stop. If it is valid, it will use the message to define this.numProbes and this.delay. Does not return anything.
- measurementPhase()
    - Receives all the probe messages from the client. Ensures the validity of these messages, and that they are in order. If they are valid, it will echo them back to the client, if not it will stop. Does not return anything.
- terminationPhase()
    - Receives a termination message from the client, determines its validity and sends back an appropriate message. After completing this it will stop the server regardless of validity. Returns nothing.

Part 2 Report

Prelude:

All of the tests were performed at my apartment, which is ~15 min walking distance away from campus and ~35 min walking distance away from the CDS building (where I am assuming the csa machines are, if they are on the cloud somewhere then this is largely irrelevant). My PC used to conduct them runs Windows 10 and is connected via ethernet to a fiber optic internet connection. For every experiment 100 probes were sent from the client, which I ran on my home computer, to the server, which was run on csa2.bu.edu.

Experiment 1, RTT:

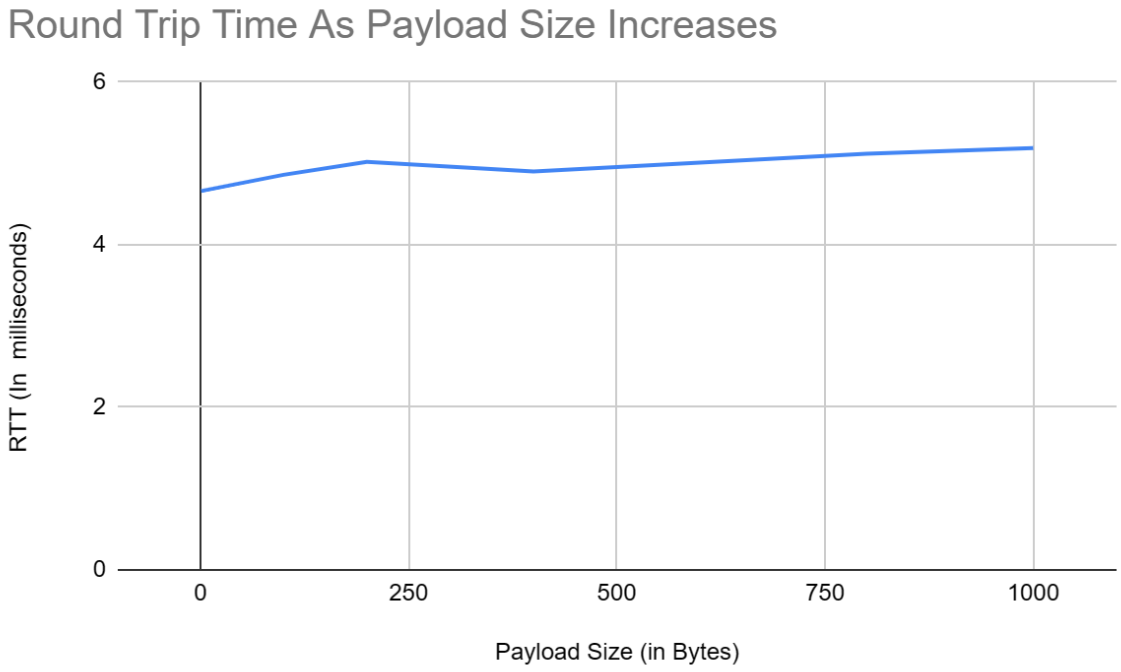
For the first experiment I tested the round trip time for sending packets of varying sizes with 0 artificial delay. The data was calculated by the client by getting the time upon sending the message and again when receiving it and then subtracting the first from the second. Each individual RTT was calculated and then added together to find a total, before being divided by the number of probes in order to find an average. The result was a fairly flat line, where the RTT for each message was roughly the same.

Table 1:

Packet Size (in Bytes)	RTT (in milliseconds)
1	4.66
100	4.86
200	5.02
400	4.9
800	5.12
1000	5.19

This table shows the raw data recorded during the first experiment

Figure 1:



This graph shows the data from Table 1 using a line graph, it compares the RTT from different messages of varying size.

These results make sense when you consider how the messages are being sent. There is a very slight increase in the time required to send each one on average, but overall the amount of data is very low, with each being less than 1KB. As the connection between client and server is quite fast and stable, these packets can all be sent in roughly the same amount of time. One could expect that if we sent messages with much larger amounts of data the RTT would increase.

Experiment 2, TPUT:

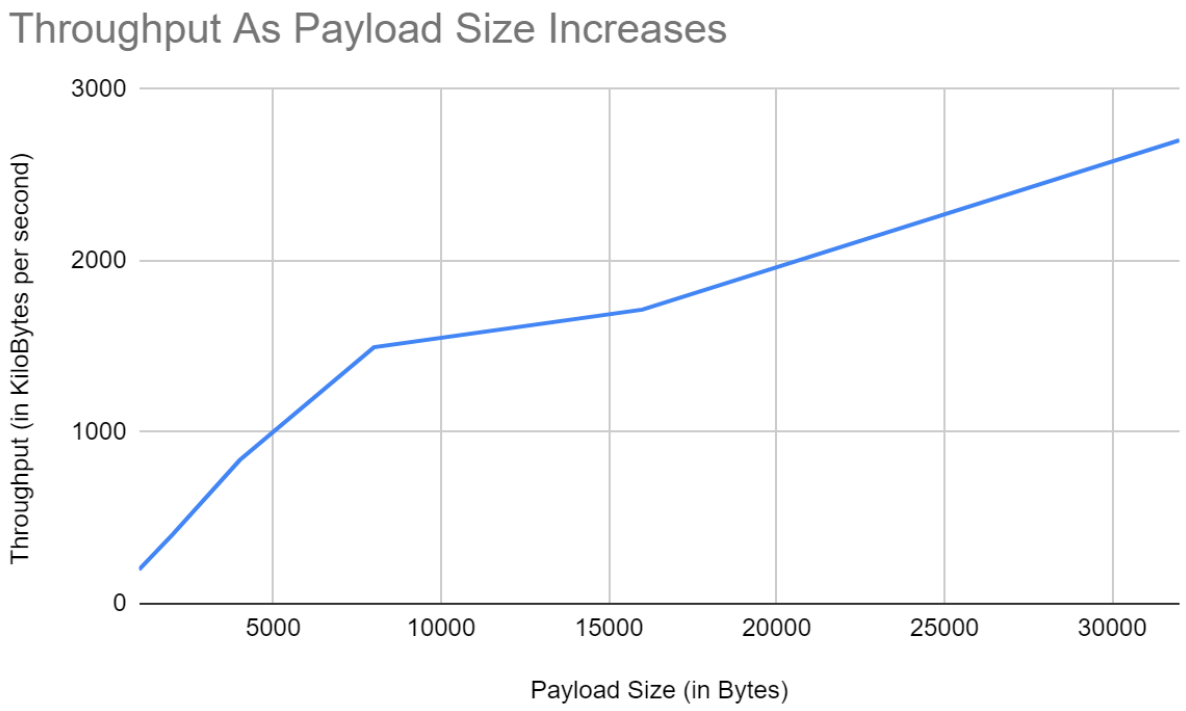
For the second experiment the throughput of sending and receiving different messages was recorded. This was calculated by taking the total amount of bytes sent (only including the payload, and not the header) and then dividing by the RTT for those bytes. This was done individually for each message but the final value was calculated by using the totals. This was because when running the server and client on the same machine you will sometimes get 0 values in the denominator when calculating the throughput on an individual message.

Table 2:

Packet Size (in bytes)	TPUT (in kilobytes per second)
1000	198.81
2000	403.26
4000	836.82
8000	1495.33
16000	1714.9
32000	2702.7

This table shows the raw data recorded during the second experiment.

Figure 2:



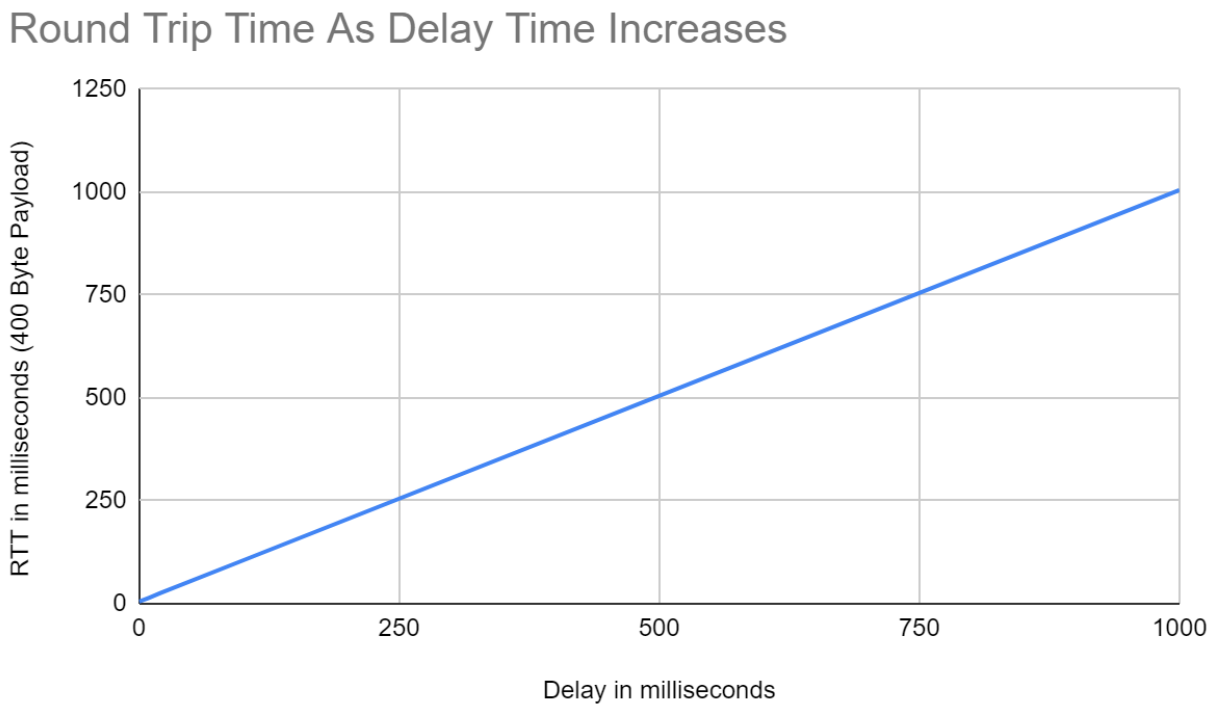
This graph shows the throughput in kilobytes per second increasing as the payload size of the message increases.

When it comes to throughput, the amount of data per second will increase as you increase the amount of data, up until it reaches a bottleneck where the connection can no longer send more data any faster. In the case of this experiment it seems evident that we did not reach that bottleneck, otherwise you would see a plateau in the graph.

Experiment 3 Changing the Delay:

In both of the previous experiments the delay parameter of the connection message was set to 0. When this number is increased the server will wait that long before sending the echo back to the client. For RTT the effect is very straightforward, the amount of delay you input is simply added to the RTT. This is because the packet will still be sent to and from at the same speed, the only difference being how long the server waits before re-sending it. This is also backed up by the experiment performed, shown in Figure 3, where the message size was fixed at 400 bytes and the delay was increased instead of the payload.

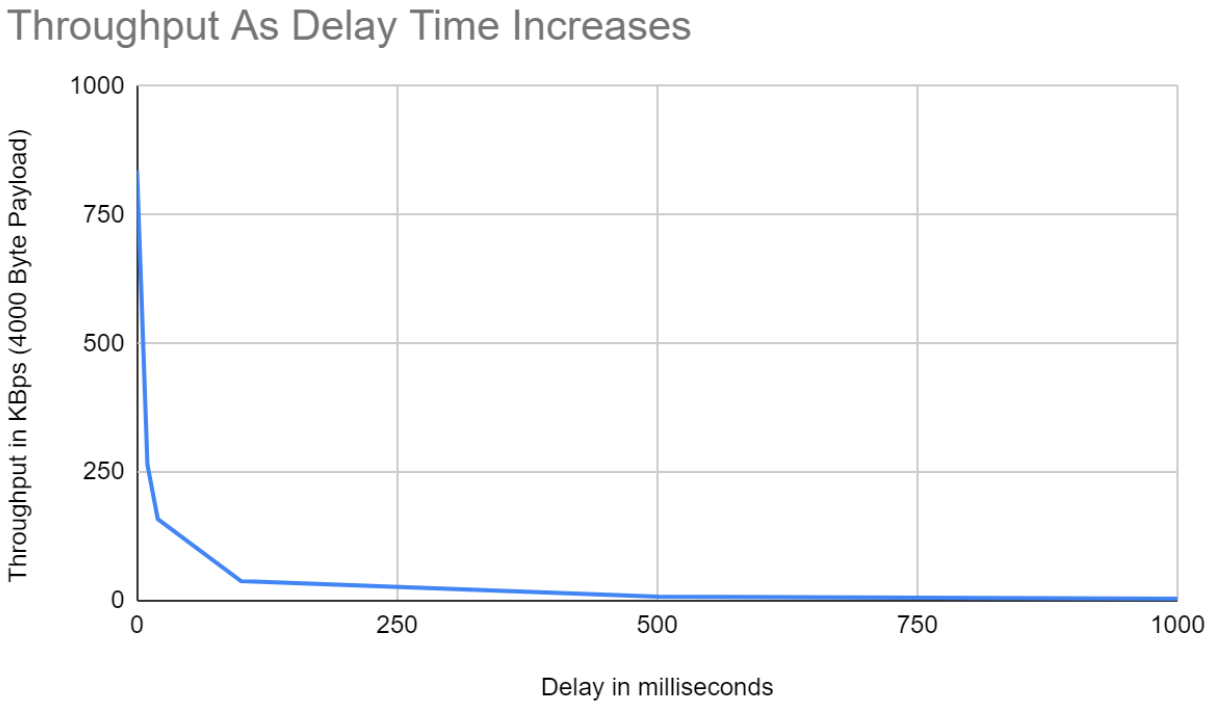
Figure 3:



This graph shows the RTT at a fixed payload size increasing directly proportional to the delay being added.

For throughput, the effect is still easily explained, but slightly less intuitive. The calculation is the amount of data divided by the RTT. Since the amount of data does not change, the only difference is the change in RTT which was explained above. In this case, the throughput has an inverse relationship to the RTT, where as the RTT increases the throughput will decrease. This can be seen demonstrated in Figure 4, the experiment was done with a fixed payload size of 4000 bytes.

Figure 4:



This graph shows the throughput as it changes due to an increase in delay, causing an increase in RTT.