CS 313 Project 2

Group 31: Reliable Blast User Datagram Protocol (RBUDP), Transmission Control Protocol (TCP) and their comparison

Keagan Selwyn Gill: 22804897@sun.ac.za

10/03/2024

Introduction/Overview

For this project, the goal was to implement a basic file transfer protocol, using both Reliable Blast User Datagram Protocol (RBUDP) and the Transmission Control Protocol (TCP) with the ability to transfer files on a local network as well as across different network connections over different machines via a VPN. LogMeIn Hamachi was used as the VPN. This RBUDP protocol used UDP packets to send the data, and a TCP conformation/ack system to check chunks of packets have been sent, making it reliable. Datagram packets were required to contain unique sequence numbers for managing datagram integrity and order. After having created the protocol, experiments were to be done to measure the differences and performance of the protocols.

Unimplemented features

From the set of required features in the specification, all of them have been implemented.

Additional features

- Both the sender and receiver have logic to stay idle while waiting for the mother to connect, and therefore do not crash.
- The ability to send a new file consecutively multiple times without having to restart any program.
- Even though a progress indicator was only required for the receiver of both the RBUDP and TCP programs, I also implemented a progress bar on the GUI of the RBUDP sender and a progress indicator for the TCP sender via text output on the terminal.
- A timer class was implemented for calculating transfer time and throughput.

Description of files

Source Files:

RBUDPSenderWindow class

- This file/class contains the process of sending a file over the network by RBUDP, making use of datagram packets for data transfer and unique sequence numbers for the datagram packets. This file also contains the Java Swing code for displaying and interacting with the Sender's GUI.

RBUDPReceiverWindow class

- This file/class contains the process of receiving a file over the network by RBUDP, making use of datagram packets for data transfer and unique sequence numbers for the datagram packets. This file also contains the Java Swing code for displaying and interacting with the Receiver's GUI.

TCPSenderWindow class

- This file/class contains the process of sending a file over the network by TCP. It reads the data from the file into a byte array and writes it out to the receiver. This file also contains the Java Swing code for displaying and interacting with the Sender's GUI.

TCPReceiverWindow class

- This file/class contains the process of receiving a file over the network by TCP. It reads in the byte array of the file data sent from the sender and writes it to the buffered file output stream to save the file on the machine. This file also contains the Java Swing code for displaying and interacting with the Receiver's GUI.

StartTime class

- This file/class implements a timer using the Calendar and GregorianCalendar libraries so that elapsed time can be computed for calculating transfer time and throughput.

Program description

For file transfer with RBUDP, the RBUDPReceiverWindow class is run to listen for a connection wanting to send a file, a window is opened titled "RBUDP Receiver" with a progress indicator and text displaying "waiting to receive file..." . When a sender has connected to the receiver and a file is sent, all the necessary procedures are executed for receiving the file via RBUDP. The RBUDPReceiverWindow class can receive a new file consecutively in a row without terminating and rerunning either program.

The RBUDPSenderWindow class is then run where a window is opened titled "RBUDP Sender" with two input fields for the host address and the port, progress indicator and a button to browse the file explorer to select a file to send. Once a file has been selected, all the necessary procedures are executed for sending the file via RBUDP. The user can keep selecting new files to send consecutively without terminating and rerunning either program.

For file transfer with TCP, the TCPSenderWindow class is run to listen for a connection wanting to receive a file, a window is opened titled "TCP Sender" with a button to browse the file explorer to select a file to send. Once a receiver has connected to the sender and a file has been selected, all the necessary procedures are executed for sending the file via TCP. The user can keep selecting new files to send consecutively without terminating and rerunning either program.

The TCPReceiverWindow class is then run where a window is opened titled "TCP Receiver" with two input fields for the host address and the port and a button that must be clicked in order to receive each file that is sent by the sender. Once the receiver has connected to the sender, a file has been selected on the sender-side and the 'Receive File' button has been clicked on the

receiver-side, all the necessary procedures are executed for receiving the file via TCP. This process can be repeated consecutively without terminating and rerunning either program.

Experiments

All of the experiments' data were obtained using both code that was written in the source files and bash scripts which made use of the POSIX "time" command, and real (clock) time was used. The data collected was for getting the time to transfer files using varying chunk sizes, file sizes and different rates of packet loss.

Comparing the Throughput of RBUDP and TCP

Experiments comparing RBUDP and TCP throughput with various constant packet sizes. These experiments were conducted on a local machine as well as on the network to test the protocols in different environments.

Expectations:

The expectation for this experiment was that the RBUDP protocol will be faster in nearly all areas. This is because RBUDP will theoretically not need to confirm every packet being sent as it is sent. And so, it was expected to increase the pipe usage while transmitting, allowing for a faster throughput.

Findings:

My findings were significantly different to my expectations. The TCP protocol performed much better than the RBUDP protocol in both the local machine environment and over the network (Hamachi). This was unexpected since it was expected that with RBUDP not having to check each packet it would be faster.

Conclusion:

With the surprising data in my findings, I had to change my view on the protocols and my application. One of three conclusions could be drawn. The first, is merely just that my RBUDP protocol got unlucky. That is, just as I tried to test it, the internet or the local machines resources were taken up, leading to a longer wait time. This possibility, however, is unlikely, seeing as both the network and local results show the same outcome. The second possibility is that my implementation of RBUDP is inefficient, and the TCP implementation is not as inefficient. This conclusion is more likely than the first, as there may have been inefficiencies I have overlooked. However, in my implementation, I did try to use the most efficient algorithms. The third possibility sheds some light on the RBUDP protocol itself, highlighting that this protocol is actually not as efficient in the normal setting as I would hope. With the results I have gathered, I cannot fully deduce which of the three conclusions above is true, but the inefficiency of RBUDP in this situation sheds some light on how it may be slower than TCP.

Transfer Rate of RBUDP

Experiments with different file sizes to test the transfer rate of RBUDP using a constant packet size. These experiments will show how effective RBUDP is with different file sizes, which should indicate the overhead caused by the protocol, and therefore the actual transfer rate can be deduced. The experiment will be measuring times for each file size, and then a ratio from file size to time will be taken to see how effective each file size was.

Expectations:

The expectation with the transfer rate with RBUDP is that it will be more effective sending larger files. This is based on the assumption that smaller files will have larger overheads caused by the RBUDP protocol whereas larger files will have less overhead, and so will be more efficient to send.

Findings:

In the attached graph, which summarises my results of transfer rates, I noticed that as the file size increases, the transfer rate increases too. I also noticed that the transfer rate grows increasingly rapidly, until approximately the 59MB/s mark, after which it levels out. Therefore, the transfer rate eventually begins plateau and slow down.

Conclusion:

From the results, I concluded that initially my expectation was true. As the file size increases, the overhead decreases, and so the transfer rate increases. This means that the larger files exhibited a higher transfer rate, as expected. The levelling out of the transfer rate is most probably due to an upper limit of the maximum optimal file size being approached. This upper limit is due to the fact that larger and larger files cannot be sent indefinitely, as the bandwidth is limited. Thus, as the transfer rate approaches this bandwidth cap, the efficiency will slow down and level out.

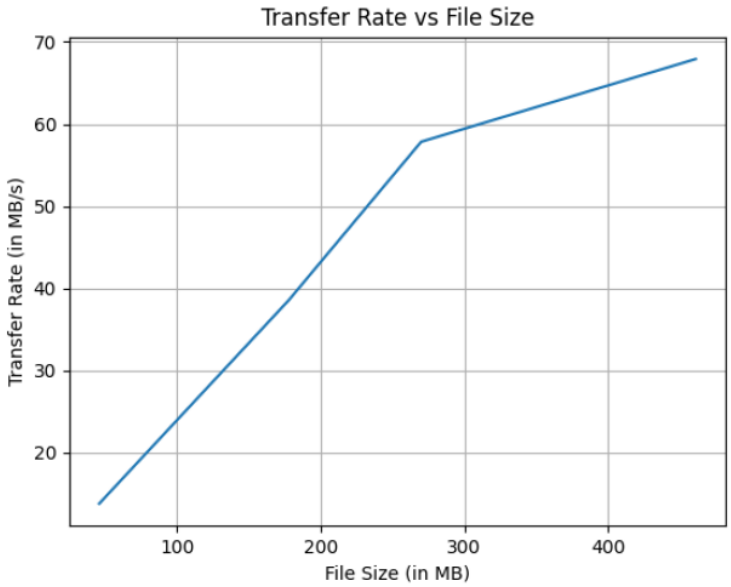

Figure 1: Graph representing the transfer rate of varying file sizes using my RBUDP implementation

Packet size of RBUDP

This experiment was run by testing varying packet sizes for a fixed file size.

Expectations:

I assumed that a larger packet size would be most optimal (anywhere between 32000 – 64000 bytes), as this would mean that big chunks of data would be sent at a given time, meaning that when a packet was sent (and not lost) a large chunk of data would be received, and due to my implementation, this data would just be slotted in to the appropriate slot for that chunk index. This is what I based my assumptions on when I thought bigger a package size equates to better performance (but I assumed that too large has diminishing returns).

Findings:

Most of the informative data came from the "big" test case, a file with size ~500MB, the other tests with smaller file sizes finished too quickly, making the time dependent on the OS, and so they did not provide too much insight. What I found however, was that my initial assumption was incorrect. With the most optimal chunk sizes actually being 2048 bytes (most optimal) and 4096 bytes, during the conclusion I will be going into more detail about why I think this is the reason. In the figure, I notice a missive spike at the 8000 bytes chunk mark, this is owing to an outlier from the dataset, although, even without it, I still see that the trend continues, and that the best chunk size is 2048 bytes, as this has the lowest time average.

Conclusion:

My initial assumption that a larger packet size (up to a threshold) is more optimal was flawed. This flaw being that, if a packet with a large packet size was lost, this is a massive penalty. However, with smaller packet sizes this is not as much as of an issue since it's just a small packet that was lost, and in fact does affect the entire file.

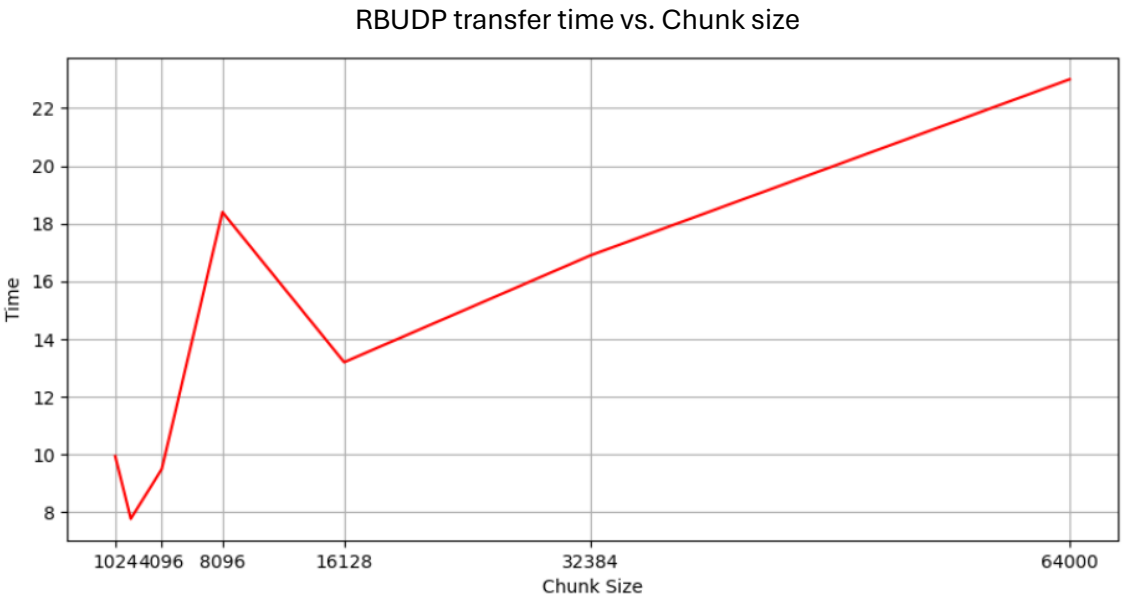

Figure 2: Graph representing the transfer time of varying packet sizes using my RBUDP implementation (~800MB file)

Varying packet loss rates

Experiments were done using packet loss of RBUDP simulated by random drop. The experiments were only conducted on the local machine, so that packets were not (or very rarely) lost by the network. This would make the performance of the program artificially high, but for the sake of experimentation, this was accepted.

Expectations:

From a theoretical point of view, I expected the program to perform worse for a higher packet loss rate. This would be expected to be due to the RBUDP protocol having to resend data. The increase in waiting time was expected to be progressively longer as the packet loss rate increased, being extremely evident at about 80% loss rate.

Findings:

The practical results showed an increase in waiting time for a higher packet loss rate, but drastically less than what was expected. The expected result was to experience a noticeable difference in file transfer times if the loss rate was above a given threshold (say 80%). What was experienced was that the time just increased slightly for each increase in loss rate, even being able to transmit at 90% packet loss rate with a reasonable amount of extra time. This time however, scaled with file size, so the bigger files noticed more of a delay than smaller files.

Conclusion:

The packet loss rate affects the RBUDP protocol less than expected and the protocol (in its implementation in this application) can deal with high packet loss. This being said, packet loss rate still affects the RBUDP protocol, however the protocol can handle this in an elegant way, not wasting too much time because of it. Based on the results seen, a packet loss rate of up to 70% is still manageably efficient. Higher than this is still possible, and will not deteriorate too much, but will be less efficient.

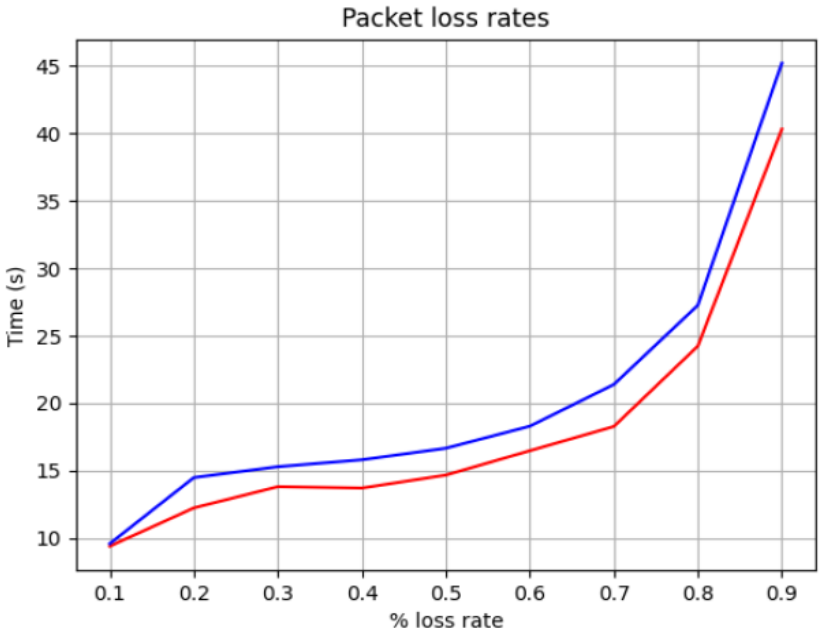

Figure 3: Graph representing the time taken to transfer for varying packet loss rates (~800MB file)

Issues encountered

The main issue I had was implementing the ability to send a new file consecutively multiple times without having to restart any program. I resorted to using a while loop to keep listening for incoming connections and file transfers but since the while loop iterates so incredibly fast the program could not receive the instruction to interrupt the loop in order to execute the necessary procedures for file transfer. Therefore, I used the Thread library to call the sleep method to cause the current thread to suspend execution for 100 milliseconds. I am aware that this is not optimal nor recommended, however at least it helps my program do what I need it to do.

Design

My design for the RBUDP and TCP file transfer programs was to have separate classes for each sender and receiver as well as a unique GUI for each.

The GUI for the RBUDP sender was designed to have two input fields to enter the host address and port, a progress bar, and a select file button. The GUI for the RBUDP receiver was designed to have a progress bar, a text field to display "Waiting to receive file..." to the user, and a text field that becomes visible once a new file has been sent which says, "File received!".

The GUI for the TCP sender was designed to just have a button to select a file. The GUI for the TCP receiver was designed to have two fields to enter the host address and port, a progress bar, and a button to receive a file.

Both the RBUDP and TCP file transfer programs were designed to be able to send a new file consecutively multiple times without restarting any program.

Compilation and execution

Since this project was built using the Apache Netbeans 14 IDE, it cannot be compiled and run from the command-line/terminal. The project must be first opened and then compiled and ran from within the Apache Netbeans IDE. To do so, follow these steps:

1) You must have the Apache Netbeans 14 IDE installed on your computer.
2) Open the IDE.
3) Move your cursor to the top left of the application and click on "File" and then "Open Project" (or use the keyboard shortcut Ctrl+Shift+O).
4) Navigate to the directory of your desired project that you want to open.
5) Select the project folder that contains the folders generated by Netbeans upon creating a project, namely "nbproject" and "src". In this case the project folder is called "RBUDP and TCP" and it should have a little coffee-cup-on-a-saucer icon to the left of it indicating that it is a Java project. Once selected click on "Open Project" at the bottom right of the window.
6) To run the RBUDP program, first click on the RBUDPReceiverWindow.java file to open it, click on any line of code in the file (so that Netbeans knows that this file is the file you want to run), then either left click to open a drop-down list and click on "Run File" or use the keyboard shortcut Shift+F6. Now the built-in terminal will be displayed, and you should see the text, "Waiting to receive next file...". The GUI window for the Receiver will also be displayed.

    a. Now we run the Sender, so click on the RBUDPSenderWindow.java file to open it, now repeat the process in step 6 above that is underlined. Now the built-in terminal will be displayed, and you should see the text, “Listening for input...” repeating. The GUI window for the Sender will also be displayed.

7) To run the TCP program, first click on the TCPSenderWindow.java file to open it and then repeat the process in step 6 above that is underlined. The GUI for the Sender will then be displayed.
    a. Now we run the Receiver, so click on the TCPReceiverWindow.java file to open it and then repeat the process in step 6 above that is underlined. The GUI for the Receiver will then be displayed.

Libraries used

- java.io.File
- java.io.FileOutputStream
- java.io.FileInputStream
- java.io.IOException
- java.net.DatagramPacket
- java.net.DatagramSocket
- java.net.InetAddress
- java.nio.ByteBuffer
- java.net.SocketTimeoutException
- java.util.Vector
- javax.swing.JFileChooser
- java.io.BufferedInputStream
- java.io.BufferedOutputStream
- java.io.DataInputStream
- java.io.DataOutputStream
- java.io.InputStream
- java.io.OutputStream
- java.net.Socket
- java.net.ServerSocket