Group 14 Project 2: Reliable Blast UDP

Dylan Callaghan: 21831599@sun.ac.za
Stephen Cochrane: 21748209@sun.ac.za

August 28, 2020

1 Introduction

For this project, the goal was to implement a basic file transfer protocol, using both TCP and the specified RBUDP protocol. This RBUDP protocol used UDP packets to send the data, and a TCP confirmation/ack system to check chunks of packets have been sent, making it reliable. After having created the protocol, experiments were to be done to measure the differences and performance of the protocols.

2 Unimplemented Features

Of the specified features, we did not implement data capturing for the receiver, but implemented everything else.

3 Additional features

- Both the sender and receiver have logic to stay idle while waiting for the other to connect, and so do not crash.
- In Gui mode, the ability to send multiple files after each other without having to restart the application.
- Files larger than 1Gb (or memory), can also be sent, and are done so by dividing the file into segments and sending these separately and building it back up again when received.

4 Description of files

There are 4 main source files for the project:

4.1 Transfer

This is the main class, and contains the logic for reading in and writing out to files. It calls either a TCP or RBUDP connection object to send the file over the network.

4.2 TCP

This file/class contains the process of sending a file over the network by TCP. It takes in the data from the file as a byte array, and on the receiving side returns this array back.

4.3 RBUDP

This file/class contains the process of sending a file over the network by RBUDP. It takes in the data as TCP does above and returns it on the receiving side.

4.4 Gui

This file contains all code related to the GUI. It is called when the program is run in GUI mode and starts a Transfer object as above to do all of the transferring by the specified protocol.

5 Program description

On starting the program, the main method in Transfer is run. From this point, either a Transfer object is immediately created to handle sending of files, or a GUI is opened to ask for user input, and then a Transfer object is created for this input. When both the sender and receiver have be run, the files will start sending over the network using the specified connection type. If the GUI mode was chosen, a progress bar will be displayed to show how much of the file has been transferred already. When the file has been transferred, the program will either close (if in terminal mode), or display sent/saved (if in GUI mode).

6 Experiments

6.1 Experiment Data

All of the experiment data can be found in the results.out.data file, these results where all obtained using the bash scripts: getTimes.sh, down.sh, getTimes2.sh. These scripts made use of the POSIX "time" command, and we use real (clock) time.

- getTimes.sh: Get the times used for certain chunk sizes and files sizes.
- getTimes2.sh: Get the time for different rates of packet loss.

- down.sh: runs a receiver that constantly receives files, so we don't have to keep restarting a receiver for each sender.

Each Section will have an appropriate header for relevant data in results.out.data, it is recommended to run: `$ grep -n <HEADER> results.out.data` to get the starting line number of that sections recorded data.

6.2 Comparing 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. The data for this is found under the header THROUGHPUT.

6.2.1 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.

6.2.2 Findings

Our findings were significanly different to our expectations. The TCP protocol performed much better than the RBDUP 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.

6.2.3 Conclusion

With the surprising data in our findings, we had to change our view on the protocols and our application. One of three conclusions could be drawn. The first, is merely just that our RBUDP protocol got unlucky. That just as we were trying to test it, the internet or the local machines resources were taken up, leading to a longer wait time. This possibility, however, is unlikey, seeing as both the network and local results show the same thing. The second possibility is that our 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 we have overlooked. However, in our implementation, we 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 we would hope. With the results we have gathered, we 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.

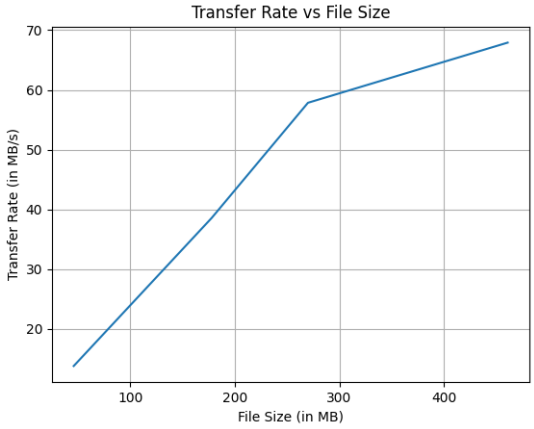

Figure 1: Runtimes of the RBUDP protocol given file sizes

6.3 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. The data for transer can be found under the header TRANSFER.

6.3.1 Expectations

The expectation with the transfer rate with RBUDP is that it will be more effective sending bigger 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.

6.3.2 Findings

In the attached graph, which summarises our results of transfer rates, we noticed that as the file size increases, so does the transfer rate. However this begins to plateau and the transfer rate begins to slow down. We noticed also that the transfer rate grows increasingly rapidly, until the $\sim$ 60MB/s mark, after which it levels out.

6.3.3 Conclusion

From the results, we concluded that our initial expectation was true. As the file size increases, the overhead decreases, and so the transfer rate increases. This

means that the larger files exibited a higher transfer rate, as expected. The leveling out of the transfer rate is most probably due to a 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. And so as the transfer rate approaches this bandwidth cap, the efficiency will slow down and level out.

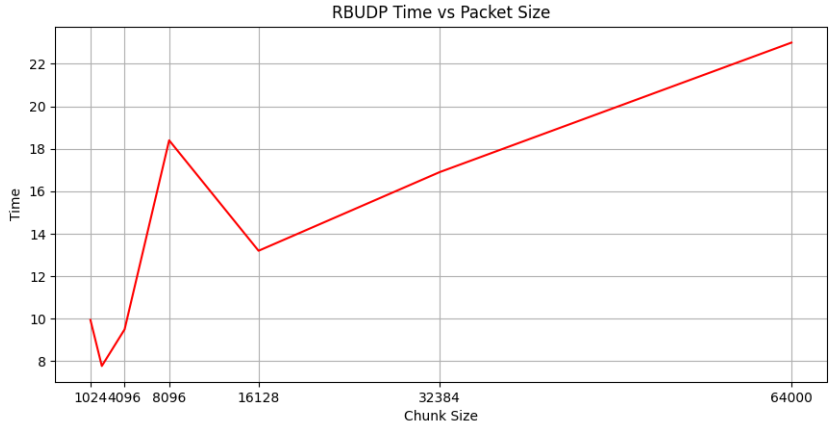

Figure 2: Varying packet loss rates and the time taken (∼800MB file)

6.4 Packet Size of RBUDP

The data for this section is found under the header: PACSIZE First we will talk about local data, that is data being sent on a local network. What we noticed when testing different packet sizes and a fixed file size, was that a packet size of 2048 (bytes) was optimal (not by a large margin however).

6.4.1 Expectations

We assumed that a larger packet size would be most optimal (anywhere between $32000 - 64000$), 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 our implementation, this "data" would just be slotted in to the appropriate slot for that chunk index. This is what we based our assumptions on when we thought bigger package size = better performance (but we assumed too large is not necessarily good).

6.4.2 Findings

Most of the informative data came from the "big" test case, a file with size $\sim$ 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 we found however, was that our initial assumption was incorrect. With the most optimal chunk sizes actually being: 2048 (most optimal) and 4096, during the conclusion we will be going into more detail about why we think this is the reason. In the figure, we notice a missive spike at the 8000 chunk mark, this is owing to an outlier from the dataset, although, even without it, we still see that the trend continues, and that the best chunk size is 2048, as this has the lowest time average.

6.4.3 Conclusion

The conclusion we drew for this occurrence actually intuitively made sense, so our initial assumption that "big chunk size = good" was actually flawed. This flaw being that , *if* a packet with a large packet size was lost, this is a massive penalty. But with smaller packet sizes this is not as much as of an issue since its just a small packet that was lost, and in fact does affect the entire file.

6.5 Varying Packet loss rate

The data for this section is found under the header: PACKLOSS Experiments 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 notably would make the performance of the program artificially high, but for the sake of experimentation, this was accepted.

6.5.1 Expectations

From a theoretical point of view, we 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, using the Reliable TCP part of RBUDP. 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.

6.5.2 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 loss rate 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.

6.5.3 Conclusion

The conclusion from these results is that the packet loss rate affects the RBUDP protocol less than expected, and that the protocol (in it's implementation in this application) can deal with high packet loss. This being said, packet loss rate still affects the RBUDP protocol, but 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 effecient. Higher than this is still possible, and will not deteriorate too much, but will be less effecient.

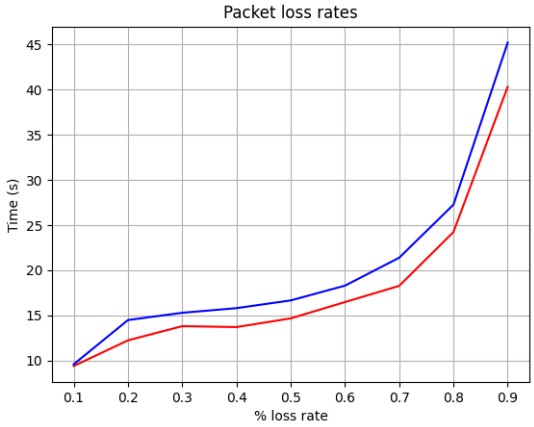

Figure 3: Varying packet loss rates and the time taken (800MB file)

7 Issues encountered

The only issues we encountered was using hamachi. We often would have to restart hamachi as, as it would "stall" and refuse to send/ was rather slow. Our pings would take long/ would time out and the only thing that would fix this was restarting hamachi. This was achieved by logging out and back in again.

8 Design

When designing the structure of our application, we decided to go with making an abstract Connection object, which represented a TCP or RBUDP connection. This proved very useful later on, as implementing both TCP and RBUDP together was trivial. All that we had to do was call methods from the Connection object and rely on these Objects to send the information via their protocols respectively.

9 Compilation

A full description of compilation for this project is described in the README file in the gitlab repository. A short summary is given below.

9.1 Compiling

Simply run,

```
$ make
```

9.2 Running the Tests

Simply run,

```
$ make test
```

10 Execution

A full description of the execution of this project is described in the README file in the gitlab repository.

10.1 Running the GUI

Simply run,

```
$ java -cp src Transer 2
```

10.2 Running TUI

Simply run,

```
$ -cp src Transfer <mode> <protocol> <port> <file> <hostname> <chunk>
```

With

- mode = 0 ⇒ Receive
- mode = 1 ⇒ Send
- protocol = 0 ⇒ TCP
- protocol = 1 ⇒ RBUDP
- file ⇒ Send/Save file

- hostname ⇒ host to connect too
- chunk ⇒ chunk sizes to send (in bytes, only for Sender)

11 Libraries used

- java.net.*;
- java.io.*;
- java.util.Scanner;
- java.nio.file.Files;
- java.nio.file.Files;
- java.nio.ByteBuffer;
- javax.swing.*;
- java.awt.Color.*;