CS 354 Project 2: Reliable Blast User Datagram Protocol (RBUDP)

Group 41:

G. Mcgaffin (23565608@sun.ac.za)

J. P. Visser (21553416@sun.ac.za)

19 August 2022

Contents

1 Introduction **4**
- 1.1 Sender Requirements . . . 4
- 1.2 Receiver Requirements . . . 4
- 1.3 Overview . . . 4

2 Additional Features **5**
- 2.1 Sender Cannot Connect to Receiver . . . 5
- 2.2 Transfer Success Status . . . 5

3 File Descriptions **5**
- 3.1 Sender Files . . . 5
  - 3.1.1 `Sender.java` . . . 5
  - 3.1.2 `SenderGUI.java` . . . 5
- 3.2 Receiver Files . . . 5
  - 3.2.1 `Receiver.java` . . . 5
  - 3.2.2 `ReceiverGUI.java` . . . 6

4 Program Description **6**
- 4.1 Sender . . . 6
- 4.2 Receiver . . . 7

5 Experiments **8**
- 5.1 Comparison of TCP and RBUDP Throughput . . . 8
  - 5.1.1 Experiment . . . 8
  - 5.1.2 Hypothesis . . . 8
  - 5.1.3 Test . . . 8
  - 5.1.4 Findings . . . 8
- 5.2 RBUDP Transfer Rate . . . 10
  - 5.2.1 Experiment . . . 10
  - 5.2.2 Hypothesis . . . 10
  - 5.2.3 Test . . . 10
  - 5.2.4 Findings . . . 11
- 5.3 RBUDP Packet Size . . . 12
  - 5.3.1 Experiment . . . 12
  - 5.3.2 Hypothesis . . . 12
  - 5.3.3 Test . . . 12
  - 5.3.4 Findings . . . 12

6 Issues Encountered **14**
- 6.1 RBUDP Sender-Receiver Synchronization . . . 14
- 6.2 Number of Bytes to Write When Using RBUDP . . . 14
- 6.3 Connecting and Transferring over the Hamachi VPN . . . 14

7 Design **14**
- 7.1 The Last RBUDP Packet . . . 14

8 Compilation and Execution **15**
8.1 Dependencies 15
8.2 Receiver 15
8.3 Sender 15

9 Libraries **15**

1 Introduction

We were tasked with developing two programs for transferring files via TCP and RBUDP. The one program must act as the sender and the other as the receiver.

1.1 Sender Requirements

The sender had the following implementation requirements:

1. Must have a simple GUI.
2. Must be able to select a file to transfer.
3. Files must be transferred to the receiver across the local network using RBUDP.
4. Files must be transferred to the receiver across the local network using TCP.
5. RBUDP must make use of datagram packets for data transfer and may use a TCP connection for signalling only.
6. RBUDP datagram packets must contain unique sequence numbers.

1.2 Receiver Requirements

And the receiver had the following implementation requirements:

1. Must have a simple GUI.
2. Must be able to receive files from the sender.
3. Must show the progress of incoming files.
4. Must be able to handle dropped packets and out of order RBUDP datagram packets appropriately.

1.3 Overview

In this document we will provide a complete view of our implementation, by discussing its design (§7), giving a breakdown of the files into which it is organized (§3), and providing a high level description, with more details where appropriate, of the flow of execution of the two programs which it comprises (§4).

By our own evaluation, we are confident that our implementation meets all the requirements, which makes it unnecessary to include a section dedicated to unimplemented features. We will, however, discuss features we have implemented additional to the requirements in §2.

Furthermore, we will also cover issues we encountered during the development process (§6), experiments we have conducted (§5), compilation and execution of the two programs (§8), as well as the libraries we made use of (§9).

2 Additional Features

2.1 Sender Cannot Connect to Receiver

If for whatever reason the sender cannot establish a connection with the receiver when the user presses the send button, then an alert is shown to inform the user of this issue.[1]

2.2 Transfer Success Status

In our implementation, once the sender starts sending a file to the receiver, the transfer cannot be stopped via the GUI. However, the user can still terminate the program by closing the window or by sending a kill signal to it via the command-line. Terminating the program will cause the receiver to lose its connection to the sender, and it will consider the transfer complete, even though that cannot be the case.

In the earlier versions of our implementation we simply showed the user an alert to notify them that the transfer had been *completed* (i.e., the sender was sending something, but it is not anymore), without indicating whether the transfer was *successful*.

So in order to provide a better experience to the user running the receiver, we added an extra check that, once a file tranfer has ended (either by completing, or by the sender being terminated), compares the number of bytes written to the byte size of the file that was transferred. Using this check we can inform the user not only about the completion of the transfer, but also whether it was successful.

3 File Descriptions

3.1 Sender Files

3.1.1 `Sender.java`

This file implements the sender's networking logic. It contains a static function for sending files to the receiver, either via TCP or RBUDP. It also contains a private helper function for writing `short int` values to a specified position in a `byte` array.

3.1.2 `SenderGUI.java`

Implements the sender's graphical user interface, and calls the `send` function defined in `Sender.java` to send files to the receiver.

3.2 Receiver Files

3.2.1 `Receiver.java`

Contains one static function for receiving files (via TCP or RBUDP), another for returning the transfer progress as a `double` value between 0 and 1 (inclusive),

[1] We also included various other alerts (in both the sender and receiver) to provide the user with feedback as they interact with the system.

and one for reading a `short int` value from a specified position in a `byte array`.

3.2.2 `ReceiverGUI.java`

This file implements the graphical user interface for the receiver. It repeatedly calls the `receive` and `getProgress` functions defined in `Receiver.java` to receive files and report on the progress of the transfer operation.

4 Program Description

4.1 Sender

1. When `SenderGUI` is run, it processes the command-line arguments passed to it, and sets up the graphical user interface.

2. Key elements of the user interface include a button the user can press to select a file, using the system file chooser; and the send button, which the user can press to initiate the file transfer when they are happy with the file that they have chosen.

3. Upon pressing the send button, certain UI elements are disabled (like the browse button, for instance). The file tranfer is then initiated on a new thread, so that the UI can remain responsive.

4. The file transfer is started by invoking the `send` function, defined in the `Sender.java` file, passing to it the name of the file to send, destination host address, destination port, the protocol to use (TCP or RBUDP), blast size, and the packet size.

5. The sender function will then attempt to connect to the receiver. If successful it will communicate the same information to the receiver via TCP.

6. If the file is being transferred via TCP, the file will be opened for reading, a number of bytes equal to the packet size will be read into a buffer, and sent to the receiver. This will continue until the end of the file is reached.

7. If the file is sent via RBUDP, the process is a bit more complex:

   (a) The sender will first create a number of packets equal to the blast size.

   (b) It creates each packet by writing a sequence number, and data length to the first four bytes (two for the sequence number and two for the data length) of a byte array. Then it read file data into the remaining number of elements in the array.[2]

   (c) The send function then enters a `while` loop, where it waits for the receiver to provide a list of all the missing packets, then sends those packets to the receiver via UDP, and then goes back to waiting for the missing packets list. This goes on until the all of the packets in the current "blast" is completely tranferred.

[2] File data is read as needed, not all at once.

(d) The process then restarts, and continues until the entire file is sent to the receiver.

8. When the file tranfer is complete, the previously disabled UI elements are re-enabled and the user can again select a file to send.

9. Various UI alerts are used throughout the program to give feedback to the user.

10. The user can stop a file transfer before it is finished by terminating the application.

11. Upon termination all resources are closed.

4.2 Receiver

1. When `ReceiverGUI` is run it processes its command-line arguments and sets up the graphical user interface for interacting with the receiver.

2. Key elements of the UI include a buttton that can be pressed in order to select a directory for saving files to; and a button to start listening for incoming files. The selection of the directory is handled by the system file chooser.

3. When the receive button is pressed, certain UI elements are disabled to prevent the user from accidently doing something wrong (e.g., pressing the receive button multiple times, thereby starting up multiple threads listening for incoming files). On a new thread, the `receive` function is called and it waits for a connection from the sender.

4. Upon connecting with the sender, the receiver first receives some basic information about the transfer that is going to take place. (See the point 4 of §§4.1.)

5. If the transfer takes place using TCP, then the receiver will simply read the byte data it receives from its socket input stream to disk until it reaches the end of the stream.

6. If the file is being received via RBUDP, the following steps are followed:

   (a) The receiver starts of the conversation by telling the sender that all of the packets are missing. It does this by sending, via TCP, a `boolean` array whose length is equal to the blast size with all of its elements set to `false`.

   (b) The sender then sends all of the packets in the current "blast".

   (c) The receiver tries to receive all `DatagramPacket`s, and takes note of the number of packets, and which packets, it successfully captures.

   (d) It then revises the list of packets the sender needs to retransmit, and sends the updated list to the sender.

   (e) This process goes on until all packets in the "blast" is received. Then the data is extracted from the packets and written to the disk.

(f) The process then restarts, and continues until the entire file is received.

7. While the `receive` function is busy receiving the file on one thread, the `getProgress` function (also from the `Receiver` class) is repeatedly called to determine the percentage of the file that has been written to storage in order to update a progress bar.

8. Various UI alerts are used throughout the program to provide feedback to the user.

9. The user can stop a file transfer before it is finished by terminating the program.

10. Upon termination all resources are closed.

5 Experiments

5.1 Comparison of TCP and RBUDP Throughput

5.1.1 Experiment

Determine which protocol, TCP or RBUDP, has higher throughput.

5.1.2 Hypothesis

During the development process we found RBUDP to reliably be slower than TCP when testing under similar conditions. Therefore, we expect the throughput of TCP to be higher than that of RBUDP.

5.1.3 Test

For both the TCP and RBUDP protocols, capture a set number of packets using the command-line utility `tcpdump`, and compare the average number of bytes per second using Wireshark (see Figure 1 and Figure 2). The experiment is carried out on the local network in order to prevent (or at least mitigate) packet dropping from influencing the results.

5.1.4 Findings

Somewhat surprisingly, we found that the RBUDP protocol has higher throughput than the TCP protocol. Wireshark's data analysis indicated that RBUDP has, on average, a 30.7% higher throughput (on the local network of the machine we used to conduct the experiment).

This contradicts the faster transfer rate we see for TCP. From the data analysis we saw that despite sending 1000 byte packets at a time, both for TCP and RBUDP, TCP transferred packets with a size of 25676 bytes on average. Our suspicion is that, at a lower level, the network architecture collects the bytes written to the socket output stream and only sends them once a packet with a large enough size to meet certain requirements (most likely for the sake of efficiency) can be created.

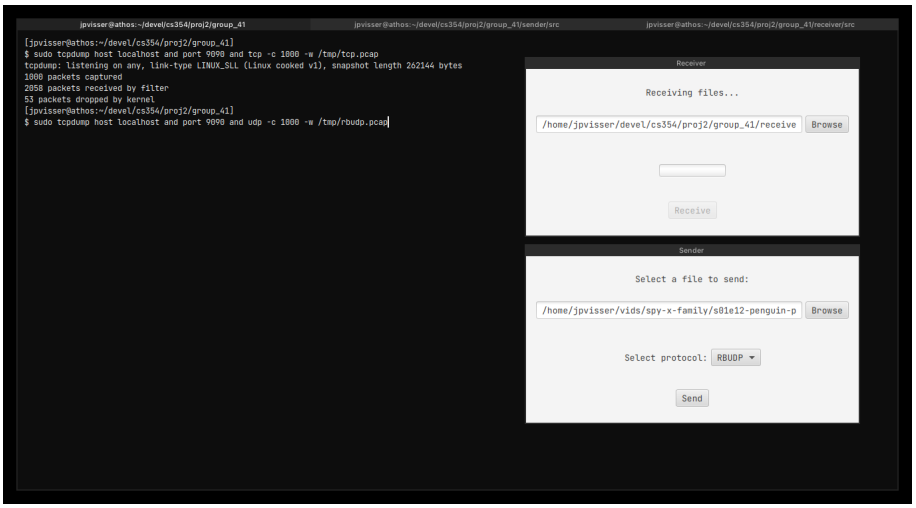

Figure 1: Transferal of a file using RBUDP while capturing 1000 packets using `tcpdump`. (Note that all images in this document have a very high resolution, so the reader may zoom in on them for closer inspection.)

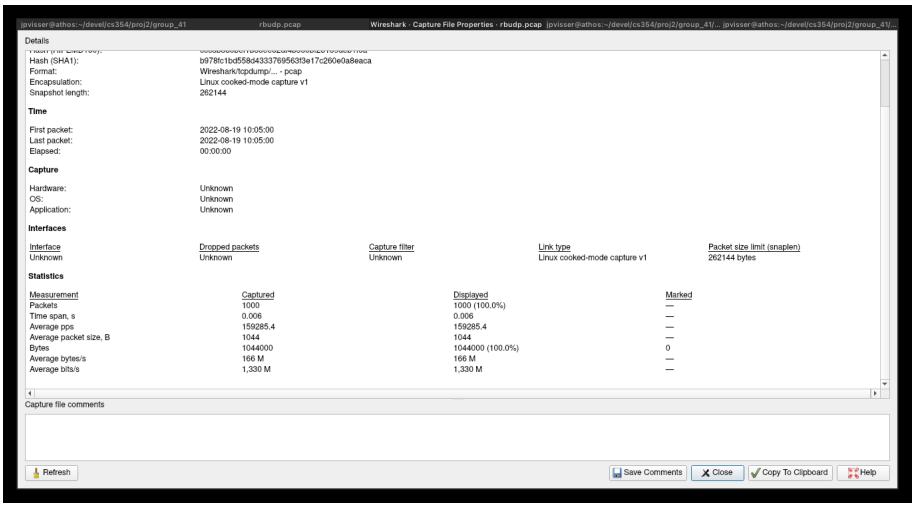

Figure 2: Wireshark's "Capture File Properties" view of the 1000 packets captured using `tcpdump`

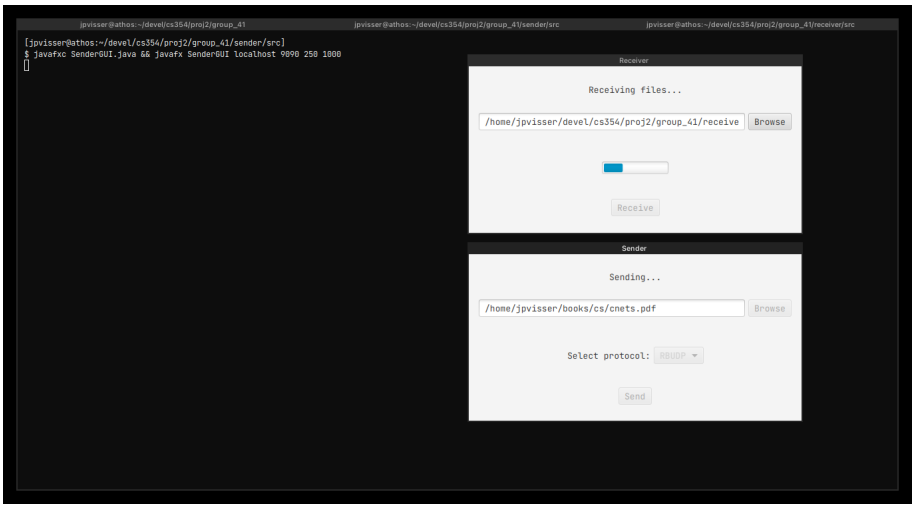

Figure 3: The sender using a blast size of 250 to transfer a file to the receiver.

5.2 RBUDP Transfer Rate

5.2.1 Experiment

Determine the correlation of blast size and transfer rate for RBUDP.

5.2.2 Hypothesis

We expect the blast size and transfer rate to be positively correlated, but only up to a certain point, whereafter the correlation will become negative. We also expect this correlation to be influenced by the packet size. Our expectation is based on the logic that a larger blast size will cause data to arrive more rapidly for the receiver to process, which will result in a higher transfer rate. However, once the blast size is increased beyond a certain threshold, a large number of packets will be dropped while the receiver is busy processing a packet, which will result in a greatly decreased transfer rate.

5.2.3 Test

Run the sender program with different blast sizes while keeping the packet size constant. Measure the time it takes to complete a file transfer in each case (using the same file in each case). Then compare the times. Conduct this experiment on a local network to avoid introducing confounding factors.

We used the blast sizes listed in Table 1, and kept the packet size fixed at 1000 bytes. To measure transfer times we used the `stat` command-line utility, part of the `coreutils` package, to get the received file's `Birth` and `Change` times. The transfer time can be found by taking the difference between these two times. See Figure 3 and Figure 4.

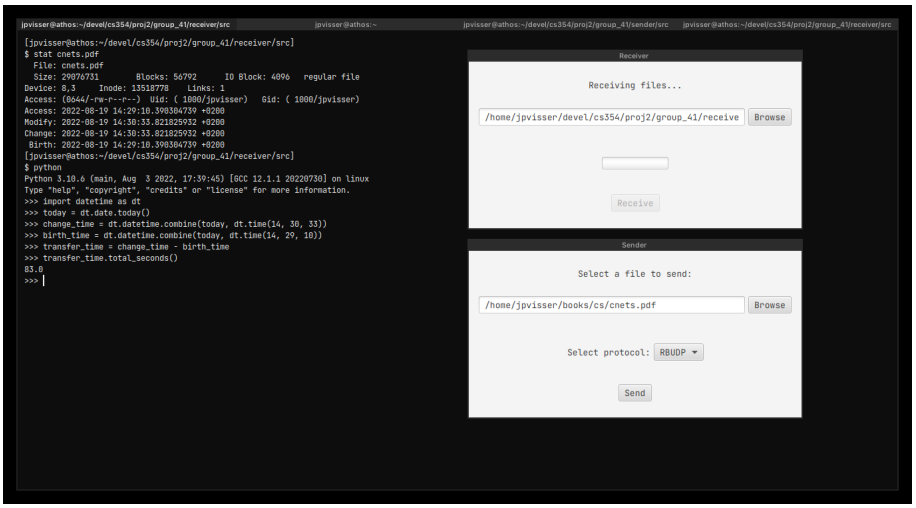

Figure 4: After the file transfer, using a blast size of 250, is successfully completed, the `stat` utility is used to determine the `Birth` and `Change` times. The number of seconds to complete the transfer is then calculated in the Python interpreter.

5.2.4 Findings

Our hypothesis turned out to be correct. Table 1 shows that as the blast size increases exponentially, the transfer time decreases at a similar rate, which indicates that blast size and transfer rate are positively correlated. At a blast size of 128000, the transfer took too long for us to wait until its completion. Therefore, there must be a tipping point, somewhere between 64000 and 128000, where the correlation becomes negative.

Blast Size	Time (s)
1000	70
2000	40
4000	20
8000	17
16000	10
32000	5
64000	7
128000	∞

Table 1: The different blast sizes and their associated transfer times in seconds when transferring a 29 MB file. The transfer time for blast size 128000 is indicated as ∞, because the transfer took too long to complete – in fact, it never even reached the point where the receiver could write data to the file.

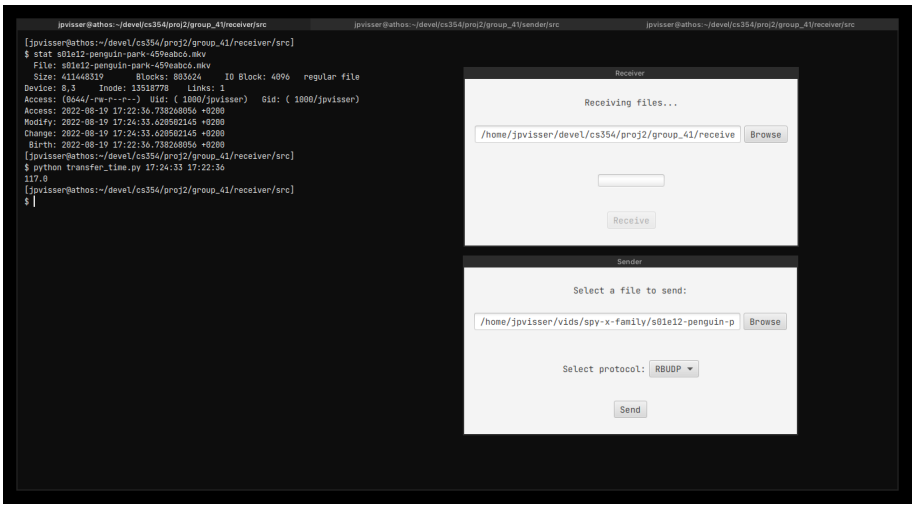

Figure 5: The result of running the sender using a packet size of 10000.

5.3 RBUDP Packet Size

5.3.1 Experiment

Determine the correlation between packet size and transfer rate for RBUDP.

5.3.2 Hypothesis

We expect results similar to those seen in §§5.2. We expect the transfer rate to increase as the packet size increases, and to then again decrease beyond a certain threshold.

5.3.3 Test

Run the sender multiple times using a different packet size for each run, while keeping the blast size constant. Measure the time it takes for each file transfer to finish, and determine the correlation, if any. Be sure to use a local network to ensure that packet dropping does not influence the results.

We conducted the experiment using a constant blast size of 1000 packets, and the different packet sizes shown in Table 2. Figure 5 shows the result obtained by running the sender with a packet size of 10000.

5.3.4 Findings

Our tests partially confirm our hypothesis. In Table 2 it can clearly be seen that as the packet size doubles, the transfer time gets cut in half, almost exactly. This indicates a positive correlation between packet size and transfer rate.

We were, however, not able to test whether there exists a tipping point where the quantities become negatively correlated: When we attempted to test the sender with a packet size of 80000, a `SocketException` was thrown by the sender (see Figure 6).

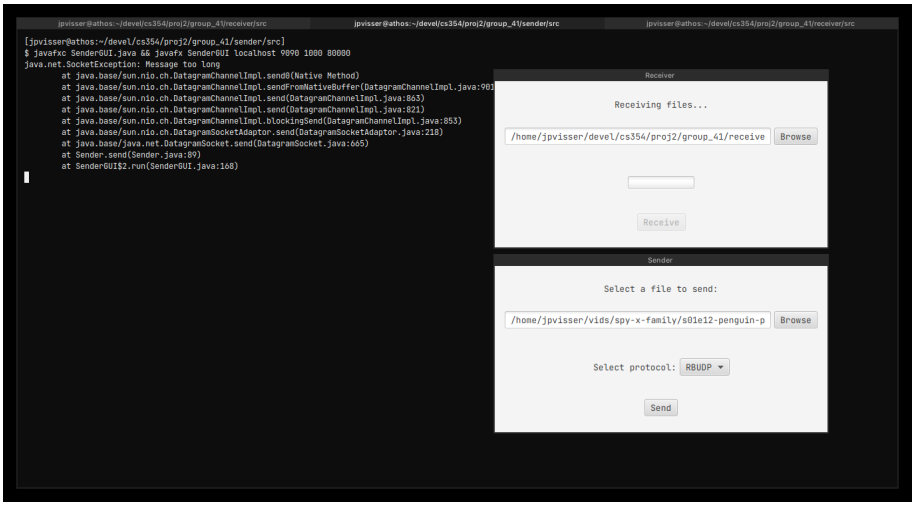

Figure 6: `SocketException` thrown when using a packet size of 80000.

Packet Size	Time (s)
10000	117
20000	61
40000	32
80000	NA

Table 2: The different packet sizes and their associated transfer times in seconds when transferring a 393 MB file. The transfer time for packet size 80000 is indicated as NA, because using a packet size this large caused a `SocketException` to be thrown.

6 Issues Encountered

6.1 RBUDP Sender-Receiver Synchronization

Designing the algorithms used by the sender and receiver to transfer data via the RBUDP protocol was difficult, because our aim was to make our implementation efficient, but also elegant and easy to understand. In the end, we believe the extra time spent on designing our algorithms paid off, as we ended up with a succinct implementation that incurs very little overhead.[3]

The bugs we had to deal with in this area were also particularly insidious.

6.2 Number of Bytes to Write When Using RBUDP

When transferring data via RBUDP, extracting and writing the bytes received to disk is mostly an easy task. But once the last part of the file is reached, and there is not enough data left to completely fill the last packet, it gives rise to the following problem: How should the receiver be informed of how many bytes in the last packet actually contains file data that needs to be written?

We discuss how we handled this problem in §§7.1, but we also thought it worth mentioning here, since it was a difficult design decision to make. This is also one of the areas where we believe our implementation can be improved.

6.3 Connecting and Transferring over the Hamachi VPN

We had considerable difficulty with connecting and transferring files over the Hamachi VPN, even though our programs performed well when we tested them locally.

7 Design

7.1 The Last RBUDP Packet

A simple approach to transferring data via RBUDP is to:[4]

1. Select a set number of packets (blast size) and a set packet size.
2. Send this these quantities to the receiver.
3. Incrementally read a number of bytes equal to the packet size from the file into a byte array. This will constitute a packet.
4. Once the number of packets created is equal to the blast size, send all of the packets to the receiver.
5. The receiver gets each packet and extracts and writes the same set number (communicated to it in step 2) of bytes to the new file.

[3]This is not to say our implementation has no shortcomings. In fact, we are aware of several ways in which it can be refined, but due to time constraints we opted to keep it as it currently is.

[4]The approach presented here is simplified so that we may focus on the problem of extracting and writing the data from a byte array.

This works well enough until reaching the end of the file. As explained in §§6.2, when reaching the end of the file, there is no longer enough bytes left to create a packet with the same size as all of the other packets that were sent before it. We will also not have the same number of packets to "blast".

If this problem goes unaddressed (and we assume the sender will still send the same number of packets, containing the same number of bytes) the receiver will end up almost always writing null bytes or junk data to the end of the file (because the receiver always extracts a set number of bytes from a set number of packets).

There are a number of ways to address this problem, but we chose to (in addition to sequence numbers) include the number of data bytes at the start of the packet (after the sequence number). This way the receiver will always know exactly how many bytes of those it receives in a packet to write to disk.

Our solution obviously has some (maybe even significant) overhead associated with it, but we settled on it due to the simplicity of the code required to implement it, as well as time constraints.

8 Compilation and Execution

It is assumed that the project will be run on Linux from a Bash shell.

8.1 Dependencies

In order to compile and run the sender and receiver the following dependencies must be installed and available on the `PATH`:

- OpenJDK 18.0.2
- OpenJFX 18.0.2.u2-1
- Gradle 7.5

8.2 Receiver

The receiver can be compiled and executed by following the next steps: From the project root directory `cd` into the `receiver` directory. Execute the script `run.sh`.

8.3 Sender

From another terminal window, follow the same steps for the receiver, but in step 1 `cd` into the `sender` directory, and, when executing `run.sh`, supply as argument the IP address or hostname of the machine or network the receiver is running on.

9 Libraries

We used only the following libraries for this project:

- Java Standard API
- JavaFX