Group 6 CS313 Project 2: File Sharing

Title

Our application is called Send and Receive. The group members are:

Ruben Bosma (25857606@sun.ac.za) (Frontend)

Brandon Spiver (22701168@sun.ac.za) (Backend)

Christopher Hill (26045532@sun.ac.za) (Documentation, report, and testing)

Introduction / Overview

The purpose of the project was to familiarize ourselves with networks via implementing data transfer protocols. This was done specifically through the TCP (Transmission Control Protocol) and RBUDP (Reliable Burst User Datagram Protocol) protocols. The two were compared and analysed to find their specific differences, advantages and disadvantages. Sender and receiver classes were used to handle data transfer, and GUI implementations for both were included.

Features

All required features were included in the project.

File Descriptions

Sender.java

The “backend” sender side of the file transfer system. Can send files over RBUDP or TCP.

Receiver.java

The “backend” receiver side of the file transfer system. Can receive files via TCP or RBUDP.

GUI_Sender.java

The GUI for users to send files.

GUI_Receiver.java

The GUI for users to receive files. Contains a progress bar for users to see how much of the file has been transferred.

Program Description

The Sender.java class is responsible for sending files from the sender user’s system to the receiver’s. It can handle this using both TCP (established via the connectToReceiver method) and RBUDP (established via the setupUDP method) protocols. It communicates directly with the GUI using its guiSender instance variable. The sendTCPMethod and sendRBUDPMethod methods handle the actual file sending using the two protocols. This class also handles any errors that could arise from file sending.

The Receiver.java class manages the reception of files sent from a sender. This class manages the TCP connection socket with the sender, stores the port information for UDP connection, and also manages the UDP socket for receiving datagrams via RBUDP. It communicates directly with its respective GUI using its guiReceiver instance variable. Its TCPReceiveFile method handles receiving a file over TCP, saving it to a temp directory and updating the GUI. The RBUDPReceiveFile method receives a file over RBUDP and assembles the data it receives into a temporary file. Packets are received and constructed into a file via the receivePackets method, and the progress bar is updated as needed. This method also handles all errors stemming from the receiver process.

GUI_Sender and GUI_Receiver both handle GUIs of their respective processes. They provide an interactable interface for users to send and receive files. The GUI_Sender allows users to input address and port data to establish connections with the receiver. The GUI_Receiver only needs users to input a port number. The GUI_Receiver also includes a progress bar for users to see how many packets have been sent. Both classes initialize the connection processes that are further handled by their respective “backend” classes.

Experiments

Experiment 1: Data Transfer, TCP vs RBUDP (Throughput of TCP and RBUDP)

Question:

Which protocol transfers the same file quicker: TCP or RBUDP?

Hypothesis: RBUDP will transfer the same file in a shorter timeframe.

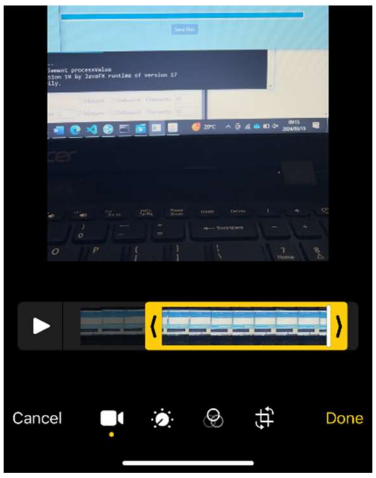
Independent variable: The protocol used, i.e. TCP or RBUDP.

Dependent variable: The time taken to transfer the file.

Control variable(s): File size, network conditions, computer architectures (i.e. the same architectures for sender and receiver were used both times).

Process:

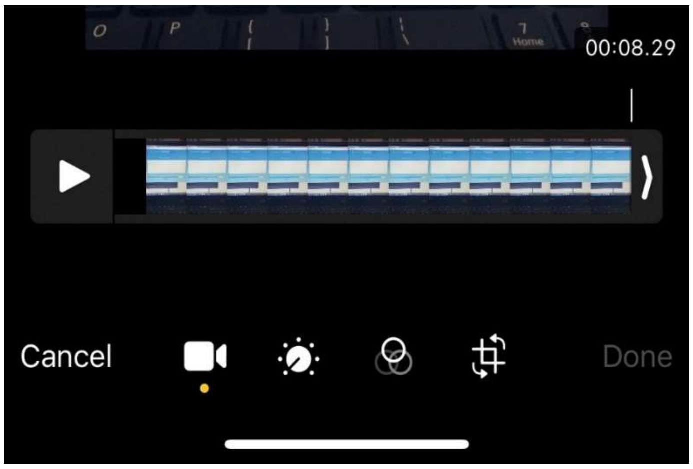
A video entitled 20200512_131046.mp4 is sent over TCP and then RBUDP. The size of the video is 265412 KB. A larger file size is chosen to make discerning time differences easier, as a small file size is almost instantaneously received. Videos of both downloads are taken on the receiver end. The frame data of both videos are analysed, and the exact timeframe of the send is determined. The first frame of the video is the moment the progress bar starts filling up, and the final frame of the video is the exact moment that the file shows up for download on the receiver side (i.e. when the progress bar is filled). This experiment can be fully recreated without changing any code in the main files.

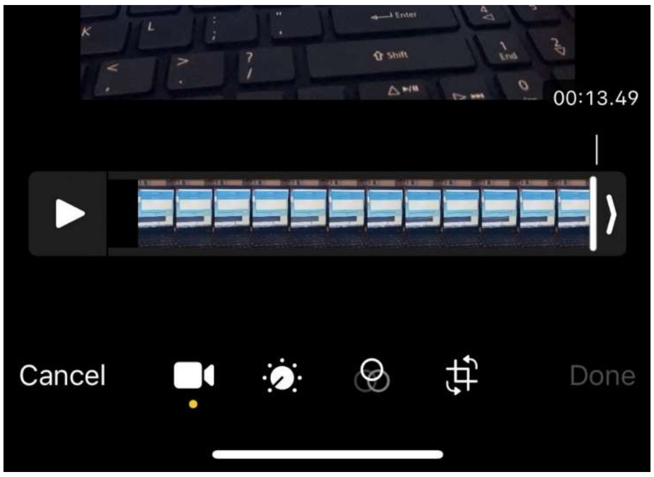


A screenshot of the editing process is shown.

Results:

The time to send the file over TCP was exactly 8.49 seconds. The time to send the file over RBUDP was exactly 13.49 seconds. Both are pictured in order.





Conclusion:

File transfer is faster using TCP. Our hypothesis was incorrect.

Experiment 2: Transfer Rate (of RBUDP)

Question:

What is the average transfer rate of RBUDP?

Hypothesis: The transfer rate of RBUDP is 64KBps.

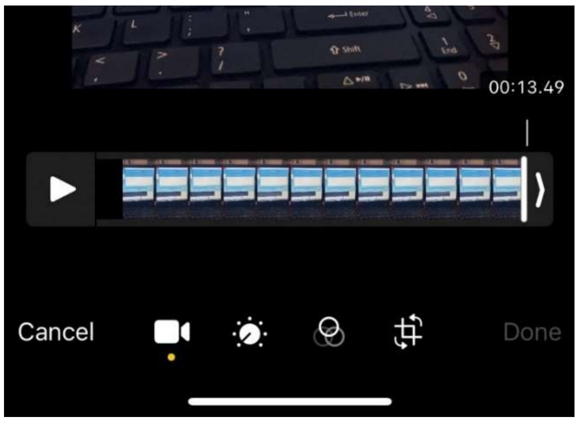
Independent variable: File size.

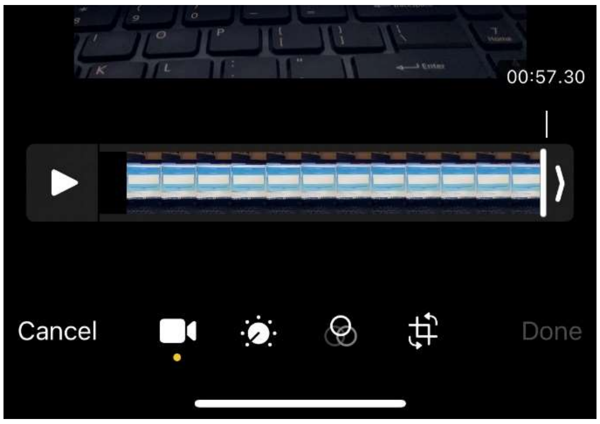
Dependent variable: Rate of transfer in bits per second.

Control variable(s): Packet size (8192 bytes), protocol (RBUDP)

Process: A similar process was used to the experiment above. A video of the file send was taken, frame data was analysed, and the time to transmit the file was noted. Two different files were sent: 20200512_131046.mp4, which had a file size of 265412 KB, and 20200605_125934.mp4, which had a file size of 613939 KB. In each iteration, the file size was noted and then used to calculate the bytes transferred per second for both files. Then, an average was taken to find the average transfer rate of RBUDP.

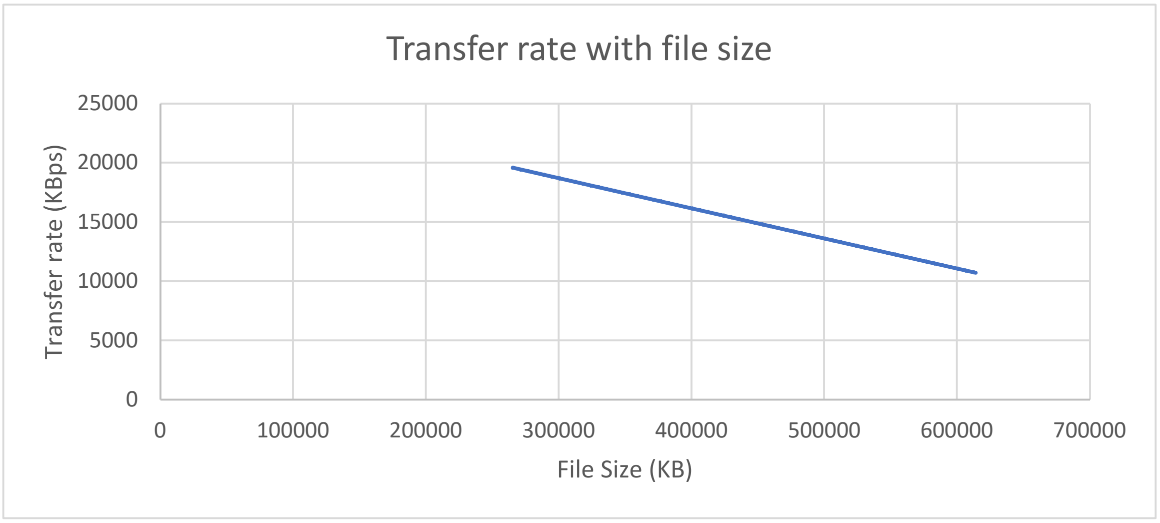
Results:





As pictured above, the transfer time for 20200512_131046.mp4 was 13.49 seconds and the transfer time for 20200605_125934.mp4 was 57.30 seconds.

This leads to a transfer rate of:
(265412KB/13.49sec) = 19574 KB/sec for 20200512_131046.mp4
(613939KB/57.30sec) = 10714 KB/sec for 20200605_125934.mp4



And an average transfer rate of:

(19574 KB/sec + 10714 KB/sec)/2 = 15 144 KB/sec.

Conclusion:

Our hypothesis is incorrect, as the transfer rate is much larger than what was imagined. Interestingly, a larger file size seems to drastically decrease the rate of transfer, as an increase in file size of ~3 times leads to a relative transfer rate of about 2/3. This could be due to the packet sequence assembling protocols, which would take much longer with a larger file. This process is also able to be recreated without altering any code.

Experiment 3: Packet Size (in RBUDP)

Question: How does the speed of RBUDP change as packet size changes?

Hypothesis: The speed of RBUDP file transfer increases as packet size increases.

Independent variable: Packet size

Dependent variable: Speed of file transfer

Control variable(s): Size of file, transfer protocol (RBUDP)

Process:

The Calculus Early Transcendentals textbook was sent to the receiver via RBUDP. The textbook has a size of 117 907 KB. The packet size was changed using the following lines of code:

```java
private int packetSize = 8192;
```

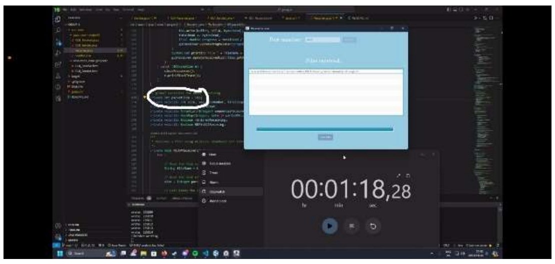
This line is on line 156 of the Sender.java class.

```java
private int packetSize = 8192;
```

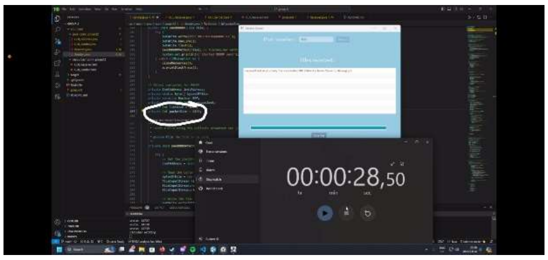
This line is on line 169 of the Receiver.java class. Both sender and receiver must have the same packet size.

The file is then sent after the packet size is changed and the program is recompiled. The time taken to transfer the file is determined with a stopwatch. The packet sizes used were 512, 4096 and 8192 bytes.

Results:

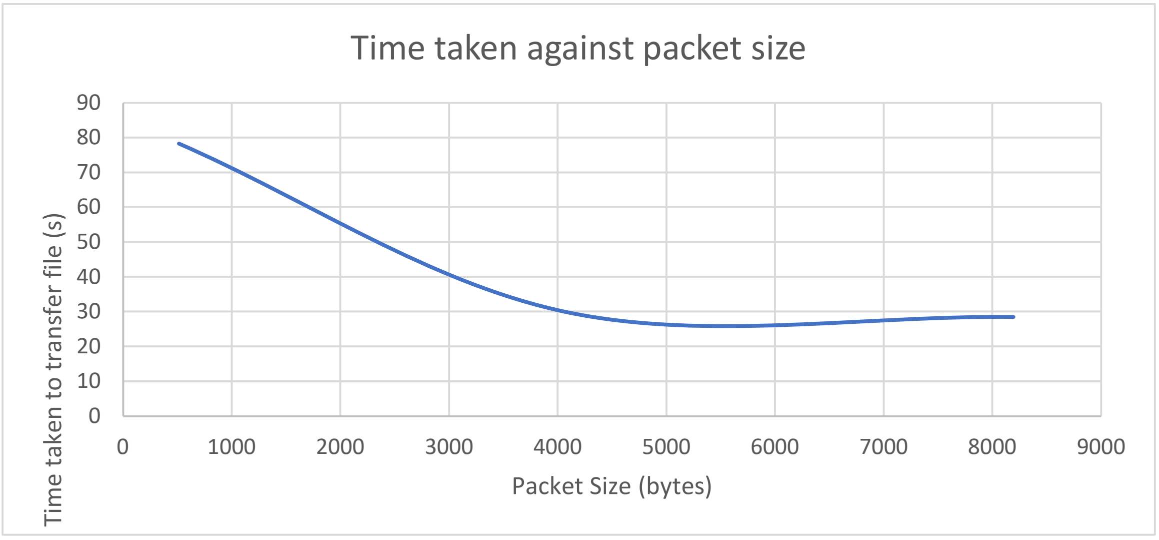


00:01:18,28



00:00:28,50

Pictured are the times taken to transfer the file, as well as the packet size used (in bytes). The table and graph below show the data more clearly:



Packet size (B)	Time taken (s)
512	78.28
4096	29.83
8192	28.50

Conclusion: A larger packet size will result in a faster transfer time up to a certain point (around 4000 bytes). Thereafter, the rate plateaus and speed does not increase further. Our hypothesis was partially correct, but missed the detail about the rate plateau.

Experiment 4: Packet Loss (compare TCP and RBUDP)

Question: How does the application handle artificial packet loss in both TCP and RBUDP?

Hypothesis: The application can identify and resend lost packets.

Independent variable: Number of packets sent

Dependent variable: Received packets: quality and number of packets

Control variable(s): Computer architecture.

Process:

For RBUDP, the

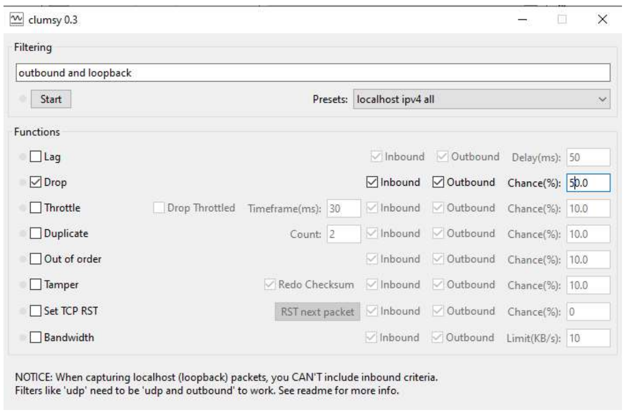
```
sendPacket(i, sequenceNr);
```

line in the sendRBUDPMethod(File file) method was bracketed with the following if statement:

```java
if (sequenceNr % 2 == 0) {
                    sendPacket(i, sequenceNr);
                }
```

This ensured that only every second packet was sent. The original line is on line 200 of the Sender.java class.

For TCP, the network was artificially congested using the Clumsy application for Windows. The application allows for "dropping" packets. The settings used are shown:



clumsy 0.3

Filtering

outbound and loopback

Start   Presets: localhost ipv4 all

Functions

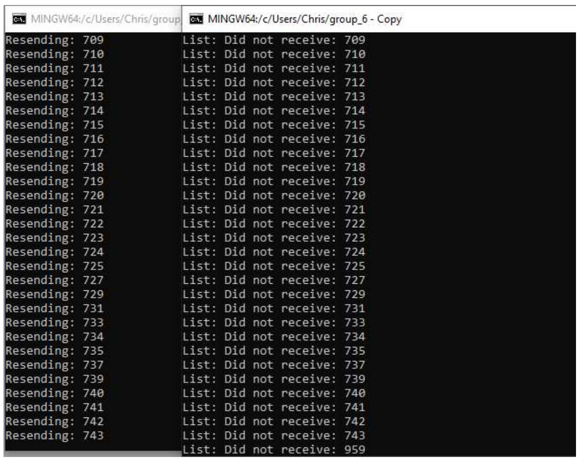
☐ Lag		✓ Inbound	✓ Outbound	Delay(ms):	50
☑ Drop		☑ Inbound	☑ Outbound	Chance(%):	50.0
☐ Throttle	☐ Drop Throttled Timeframe(ms): 30	✓ Inbound	✓ Outbound	Chance(%):	10.0
☐ Duplicate	Count: 2	✓ Inbound	✓ Outbound	Chance(%):	10.0
☐ Out of order		✓ Inbound	✓ Outbound	Chance(%):	10.0
☐ Tamper	✓ Redo Checksum	✓ Inbound	✓ Outbound	Chance(%):	10.0
☐ Set TCP RST	RST next packet	✓ Inbound	✓ Outbound	Chance(%):	0
☐ Bandwidth		✓ Inbound	✓ Outbound	Limit(KB/s):	10

NOTICE: When capturing localhost (loopback) packets, you CAN'T include inbound criteria.
Filters like 'udp' need to be 'udp and outbound' to work. See readme for more info.

This also allowed for 50% of packets to not be sent.

<u>Results:</u>

For RBUDP, all the dropped packets are requested to be resent by the receiver. This is shown in both the receiver and sender in the console:



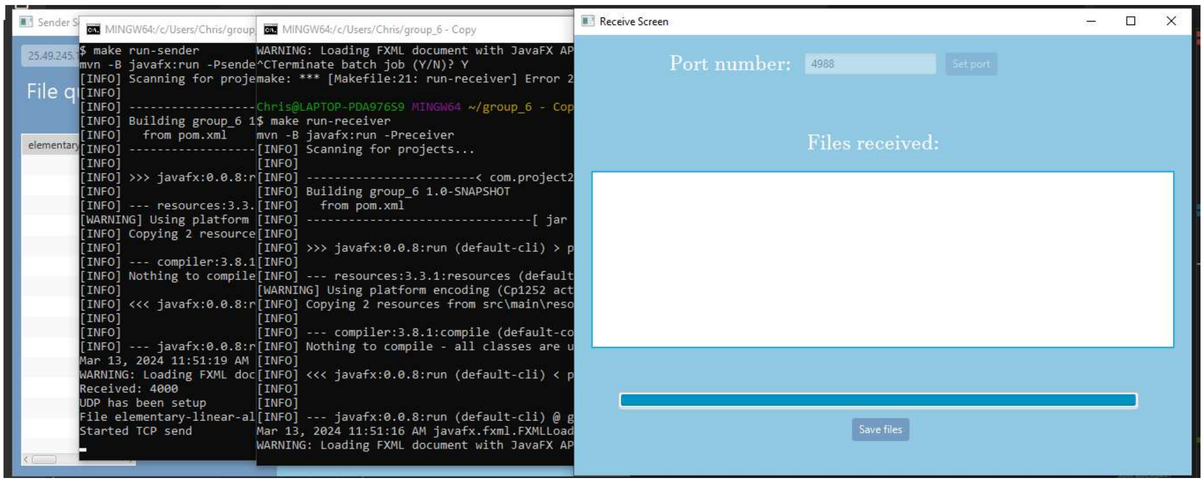
MINGW64:/c/Users/Chris/group

```
Resending: 709
Resending: 710
Resending: 711
Resending: 712
Resending: 713
Resending: 714
Resending: 715
Resending: 716
Resending: 717
Resending: 718
Resending: 719
Resending: 720
Resending: 721
Resending: 722
Resending: 723
Resending: 724
Resending: 725
Resending: 727
Resending: 729
Resending: 731
Resending: 733
Resending: 734
Resending: 735
Resending: 737
Resending: 739
Resending: 740
Resending: 741
Resending: 742
Resending: 743
```

MINGW64:/c/Users/Chris/group_6 - Copy

```
List: Did not receive: 709
List: Did not receive: 710
List: Did not receive: 711
List: Did not receive: 712
List: Did not receive: 713
List: Did not receive: 714
List: Did not receive: 715
List: Did not receive: 716
List: Did not receive: 717
List: Did not receive: 718
List: Did not receive: 719
List: Did not receive: 720
List: Did not receive: 721
List: Did not receive: 722
List: Did not receive: 723
List: Did not receive: 724
List: Did not receive: 725
List: Did not receive: 727
List: Did not receive: 729
List: Did not receive: 731
List: Did not receive: 733
List: Did not receive: 734
List: Did not receive: 735
List: Did not receive: 737
List: Did not receive: 739
List: Did not receive: 740
List: Did not receive: 741
List: Did not receive: 742
List: Did not receive: 743
List: Did not receive: 959
```

For TCP, the progress bar fills (so every packet is sent) but it seems that the file never truly sends. This could be attributed to the interaction of Clumsy with the network and our application, as there are procedures in place to handle dropped packets in the TCP portion of our application.



```
$ make run-sender           WARNING: Loading FXML document with JavaFX AP
mvn -B javafx:run -Psende^CTerminate batch job (Y/N)? Y
[INFO] Scanning for projemake: *** [Makefile:21: run-receiver] Error 2
[INFO]
[INFO] -----------------Chris@LAPTOP-PDA976S9 MINGW64 ~/group_6 - Cop
[INFO] Building group_6 1$ make run-receiver
[INFO]   from pom.xml    mvn -B javafx:run -Preceiver
[INFO] -----------------[INFO] Scanning for projects...
[INFO]                  [INFO]
[INFO] >>> javafx:0.0.8:r[INFO] -----------------------< com.project2
[INFO]                  [INFO] Building group_6 1.0-SNAPSHOT
[INFO] --- resources:3.3.[INFO]   from pom.xml
[WARNING] Using platform [INFO] --------------------------------[ jar
[INFO] Copying 2 resource[INFO]
[INFO]                  [INFO] >>> javafx:0.0.8:run (default-cli) > p
[INFO] --- compiler:3.8.1[INFO]
[INFO] Nothing to compile[INFO] --- resources:3.3.1:resources (default
[INFO]                  [WARNING] Using platform encoding (Cp1252 act
[INFO] <<< javafx:0.0.8:r[INFO] Copying 2 resources from src\main\reso
[INFO]                  [INFO]
[INFO]                  [INFO] --- compiler:3.8.1:compile (default-co
[INFO] --- javafx:0.0.8:r[INFO] Nothing to compile - all classes are u
Mar 13, 2024 11:51:19 AM [INFO]
WARNING: Loading FXML doc[INFO] <<< javafx:0.0.8:run (default-cli) < p
Received: 4000          [INFO]
UDP has been setup      [INFO]
File elementary-linear-al[INFO] --- javafx:0.0.8:run (default-cli) @ g
Started TCP send        Mar 13, 2024 11:51:16 AM javafx.fxml.FXMLLoad
                        WARNING: Loading FXML document with JavaFX AP
```

Conclusion:

Our hypothesis was correct for RBUDP implementation and inconclusive for TCP.

Issues Encountered

RBUDP was extremely difficult to implement for group members. There was particular difficulty with the list system used to track packets sent and to handle for packet loss.

Design

A log of files sent was implemented in the sender, as well as a log of files received on the receiver side. The interactable list of files to send was another major aspect of our design and allowed specific files to be sent at specific times, instead of just “dumping” every uploaded file to send onto the receiver. The choice to send files over TCP or RBUDP is done via the click of one button and can be freely interchanged between each respective file transfer. In other words, a file that has already been uploaded can be sent via either protocol, based on what the user wants. The same file can be sent multiple times, while only needing to be uploaded once. This is done via the interactable file queue list in the GUI.