Blaine McMahon, Jacob Sword, Nicholas Yameen

Network Design

Project Phase 3

3/8/19

**1. Introduction:**

This code completes Phase 3 of the project by using UDP to send images from a client to a server, and vice versa. The UDP server runs continuously to accept messages from clients. These messages are in the form of:

1. A filename of the image, that the client will send

2. The contents of said image.

The server then writes the file. After sending this file, the client waits for a response of the same nature from the server, and then writes it’s transferred file before exiting. The below flowcharts and explanations explain these procedures in more detail.

**2. Flowcharts**

Fig 1. Server Flowchart

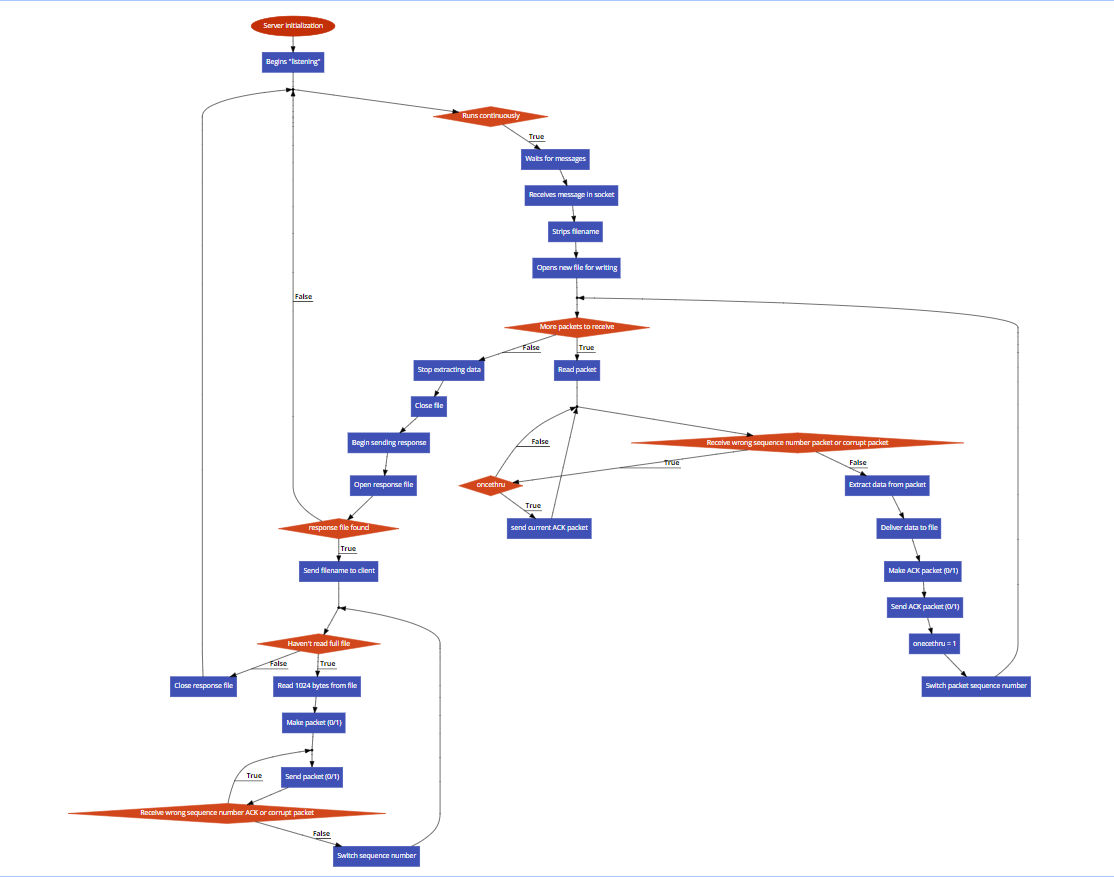
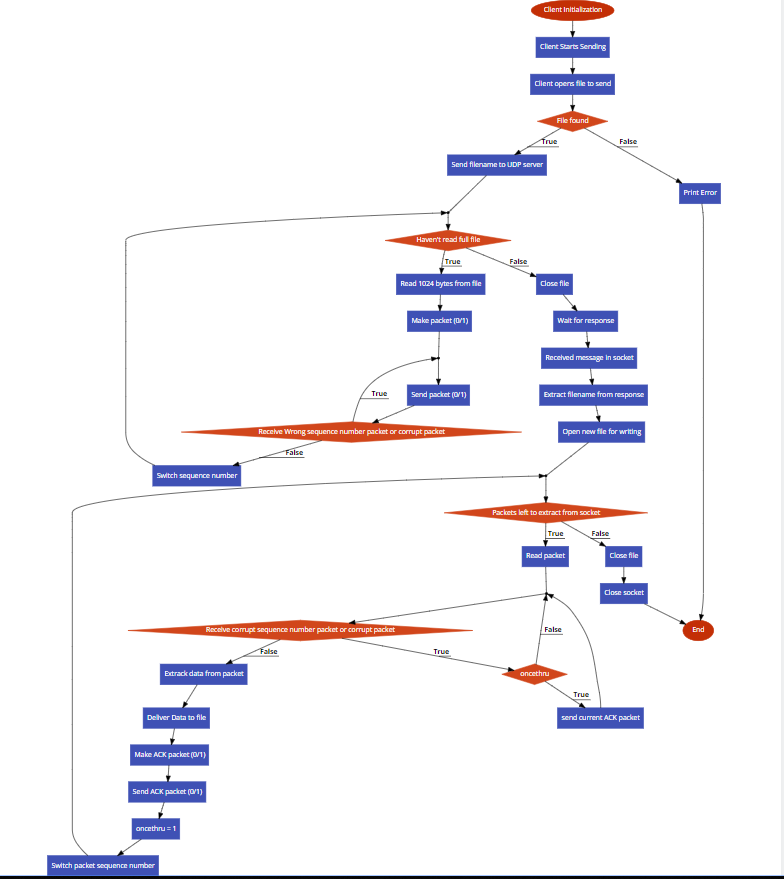
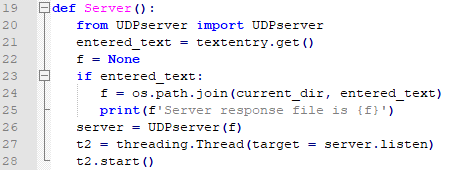
****

Fig 2. Client Flowchart

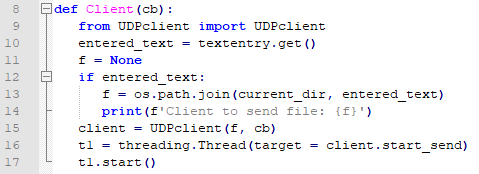


**GUI.py**

This file is the main GUI for the program. It uses tkinter to create the GUI. First, the GUI window will open and display three buttons and a place to enter text. The user can either write the path to the file to be sent or leave it blank to use our default image. The user then has to click Start Server first. This will start the *run\_server()* function. This function will update the gui text to show that the server is listening and then call *Server()*.



This function initializes a Server object and then runs it in a new thread. Next, the user can click Start Client button. This will call *start\_client()* which updates the gui to show that the client is working, then calls the *Client()* function.



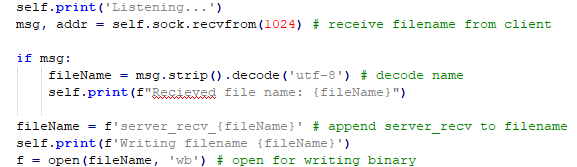
This function initializes a client class with a “callback” parameter. This callback is used within the class to call a function when the client’s work is done. In this case, the function changes the GUI message from “Client working…” to “Client finished.” Client then calls the client object’s start\_send process in a new thread. The two scripts are run as described below.

**UDPserver.py**

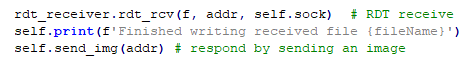
In the UDP server file, there is a class called UDPserver. The init function will first either use the file path given by the user at the start or use the default if not specified by the user. It will then open a UDP socket and bind to it and make it an instance variable.



When called, the listen() function will continually run, where it will listen for incoming connections. The sever will first receive the filename being sent from the client. It will take that name and open the file for writing binary.



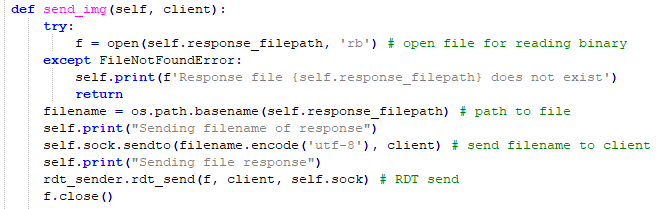
It will then preform a *rdt\_rcv(),* utilizing the rdt\_receiver.py file, where it will continually receive packets from the client and write them to the file until there are no packets left. Once the full image is received it will then close the file and send an image back to the client.



Messages will be printed to the console through this to show the status. All messages printed from the server are run through “self.print()” to prefix it with the “Server:” string for easier reading in the terminal.

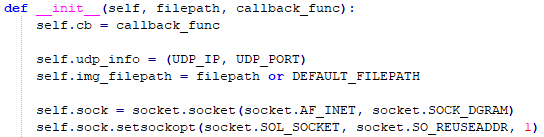


The *send\_img()* function will first open the file specified or the default one for reading binary. It will then send the image name to the client. Finally, it will send the image using the using the rdt\_sender.py function *rdt\_send()*.

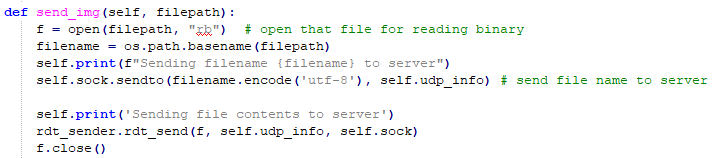


**UDPclient.py**

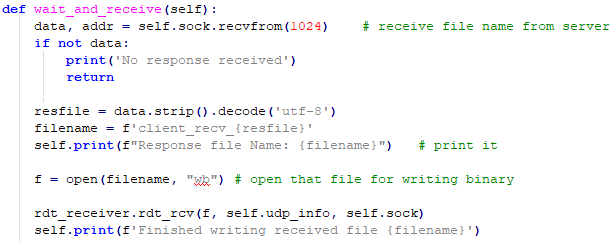
The UDPclient has a class which defines four functions. First, the client will initialize similar to the UDP server. It will connect to the server on the UDP socket, which is made into an instance variable. Then it will go into *start\_send()* function. It will first call *send\_img()* function to send the server an image.



The *send\_img()* function will first either use the path given by the user or the default one. If there is an error finding the given filename, it will be caught above, and the client will stop execution. If the file is successfully found, it will open that image for reading binary. The client will send the server the image name. Then it preforms RDT send by calling the *rdt\_send()* function in the rdt\_sender.py file. Once it has finished it will close that file.

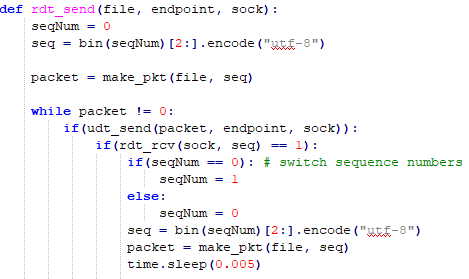


Next, the *start\_send()* will call the *wait\_and\_receive()* function. Here the client will receive the response image name from the server. It will open that file for writing binary. After which it will call *rdt\_rcv()* (from the rdt\_receiver.py file) to receive all the packets of the image. After which, the file will close, and the callback function will be called. This function just prints it has finished. The client will then close the socket and be finished. The client also uses a self.print function to prepend “Client:” to all print strings.

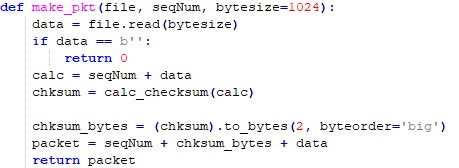


**rtd\_sender.py**

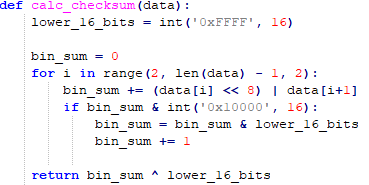
In the file, there are seven functions defined. The server and client will use these functions to send RTD 2.2 over a socket. The sender will use *rdt\_send()* function. This function will take in a file, endpoint, and the socket. It first starts by making a packet. Then it will send the first packet to the receiver. If the ACK received is not corrupt it will switch sequence numbers. If it is corrupt it won’t switch numbers, it will send the old packet. It will continuously make and send packets to the receiver, switching the sequence number each iteration until the file is completely sent, checking for corruption each time.



It will make packets using the *make\_pkt()* function by reading 1024 bytes at a time. If there is nothing to read, then it will return 0. This is so the *rdt\_send()* can break out of the while loop. It takes the data read and the sequence number to calculate a checksum number. Finally, it creates the packet to be sent.



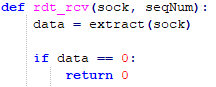
To calculate the checksum, the function *calc\_checksum()­* will be called. This preformsthe 1’s complement of wraparound 16-bit sum.

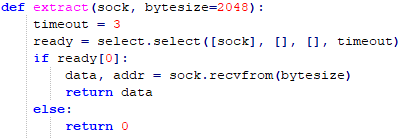


The *udt\_send()­­* function will send the packets to the endpoint over the socket. It returns the number of bytes sent.

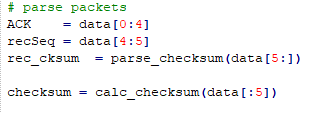


The *rdt\_rcv()* function will receive the acknowledgement from the receiver. It will be called with the socket and sequence number being sent. It will first extract the ACK from the receiver using *extract()*. This function will wait for a packet, if it doesn’t receive something it will timeout.





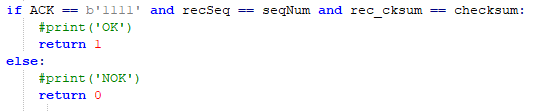
If it did receive a packet it will parse that packet.



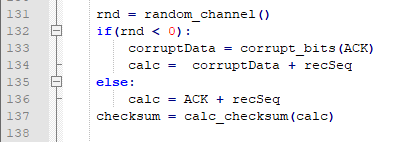
It will parse the checksum from the data using the *parse\_checksum()* function.



It then creates it own checksum, this is so the received and calculated checksums can be compared for corruption. Finally, it will check to see if the parsed packet isn’t corrupt, if not it returns 1.



To corrupt the data on the sender side, the function *rdt\_rcv()* will corrupt the ACK. In the if statement the integer should be a value between 0 (no corruption) to 60 (max corruption). This will have to changed by the user.

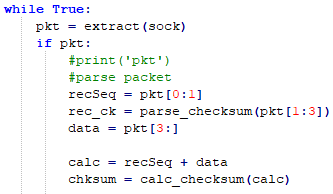


There are two functions to pick a number between 0-100 called *random\_channel()*. This number determines the percentage. The data will be corrupt in the *corrupt\_bits()* function.

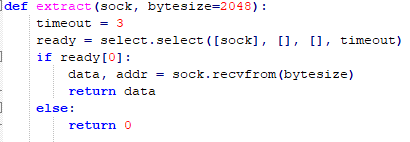


**rdt\_receiver.py**

The receiver will utilize the functions inside the rdt\_receiver.py to complete RDT 2.2 transactions. The first function is *rdt\_rcv()*, which takes the file to write to, the endpoint, and the socket. This function will continuously run until there is no packet received. It will first extract a packet and parse the packet accordingly.



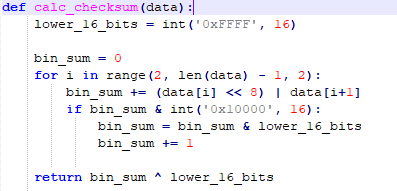
The *extract()* function will receive a packet from the socket. If there is a packet it will return it, otherwise it will return 0.



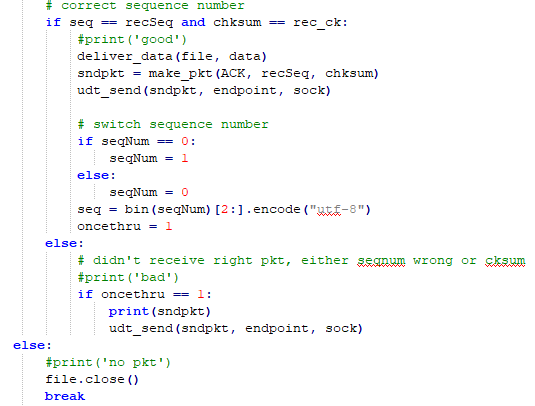
After, the packet is parsed as seen above. The received sequence number and data are stripped from the packet. The received checksum is parsed through the *parse\_checksum()* function.



To verify the data is not corrupt, the data and sequence number received will be made into a checksum using the *calc\_checksum()* function.



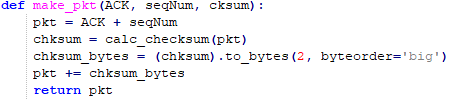
The expect sequence number, received sequence number, calculated checksum and received checksum are compared.



If the everything matches it will deliver the data using the *deliver\_data()* function. This function writes the data to the file.



Then it will make an acknowledge packet using the *make\_pkt()* function. This function will make a new checksum and create the ACK packet.

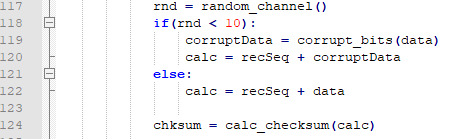


The packet will then be sent over the socket back to the sender.



The expected sequence number will then switch and the once through variable will be set. If there was a corrupt packet, the previous ACK packet will be sent again. Finally, if there was no packet received, the file will close and the *rdt\_rcv()* will return.

On the receiver side, *rdt\_rcv()*can corrupt the data bits, similar to as before.

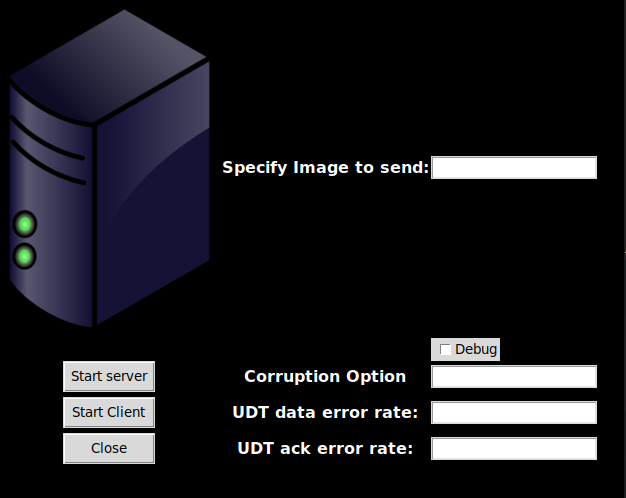
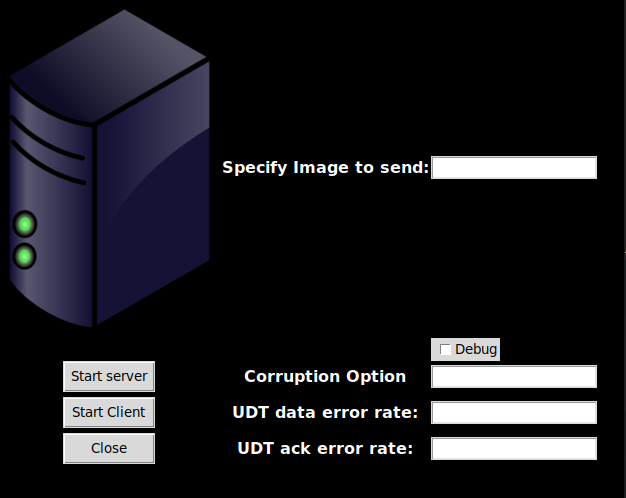


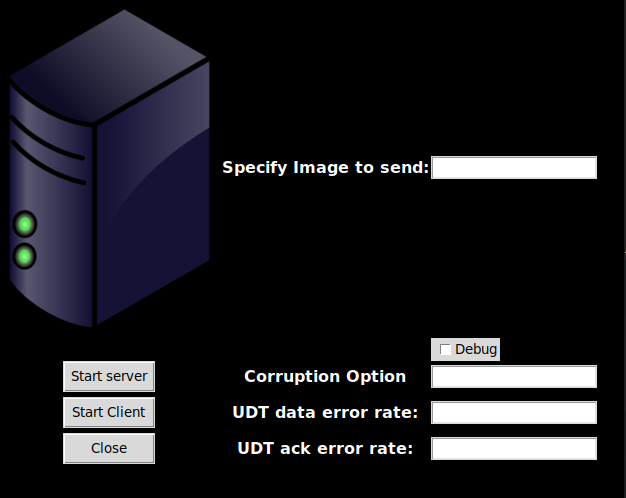
As said before, the integer value will have to change to a number between 0 – 60. In the image above it is 10% corruption.

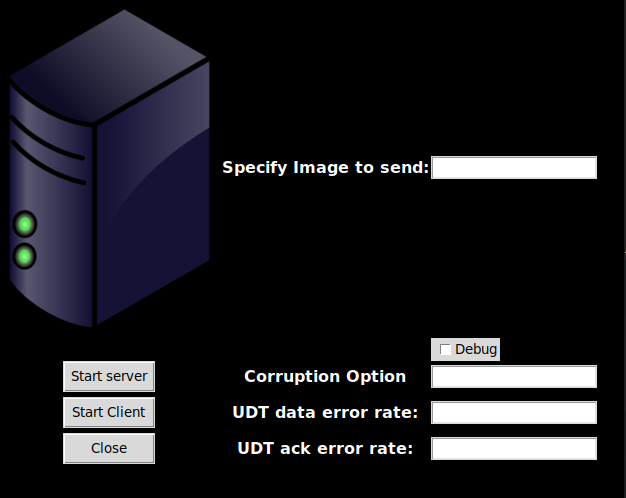
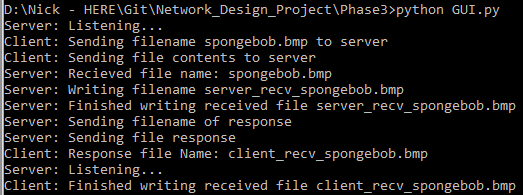
There are two functions to pick a number between 0-100 called *random\_channel()*. This number determines the percentage. The data will be corrupt in the *corrupt\_bits()* function.



**How to Run**

1. Make sure all files laid out in the ReadMe.txt are present.
2. Issue the command “python GUI.py” on the command line. The following GUI will appear.  
   
3. Leave file to send blank to use default or specify path to your own picture. Click server  
   
4. Click “Debug” if you would like to see error status messages like "Bit error encountered in Data!". Enter the desired corruption option (1-3). Entering 1 denotes no bit errors, 2 denotes bit error in ACK packets, and 3 denotes bit error in Data packets. These options will be used by both client and server.

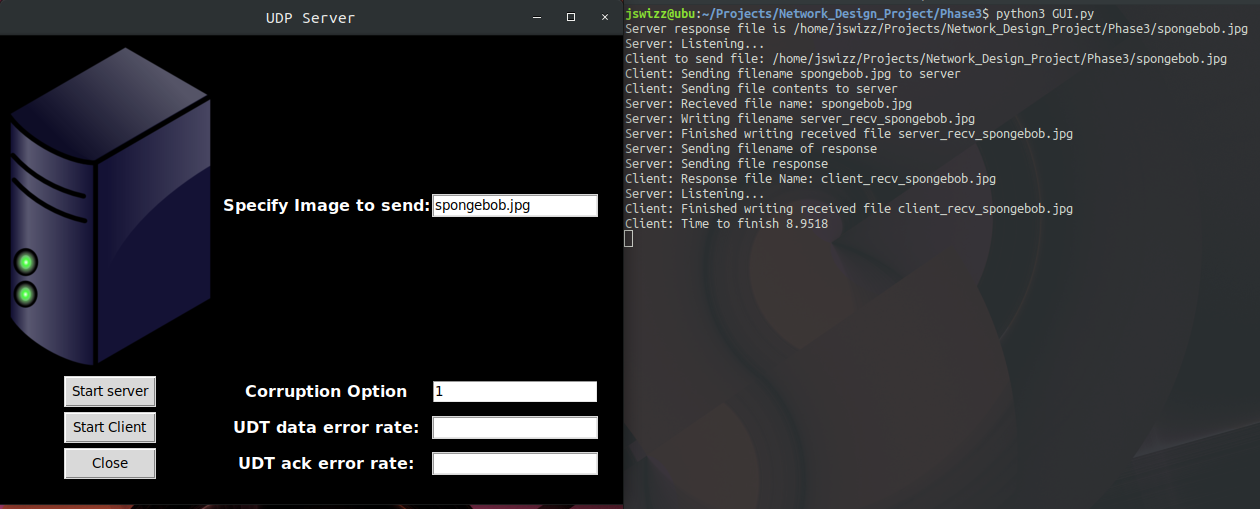


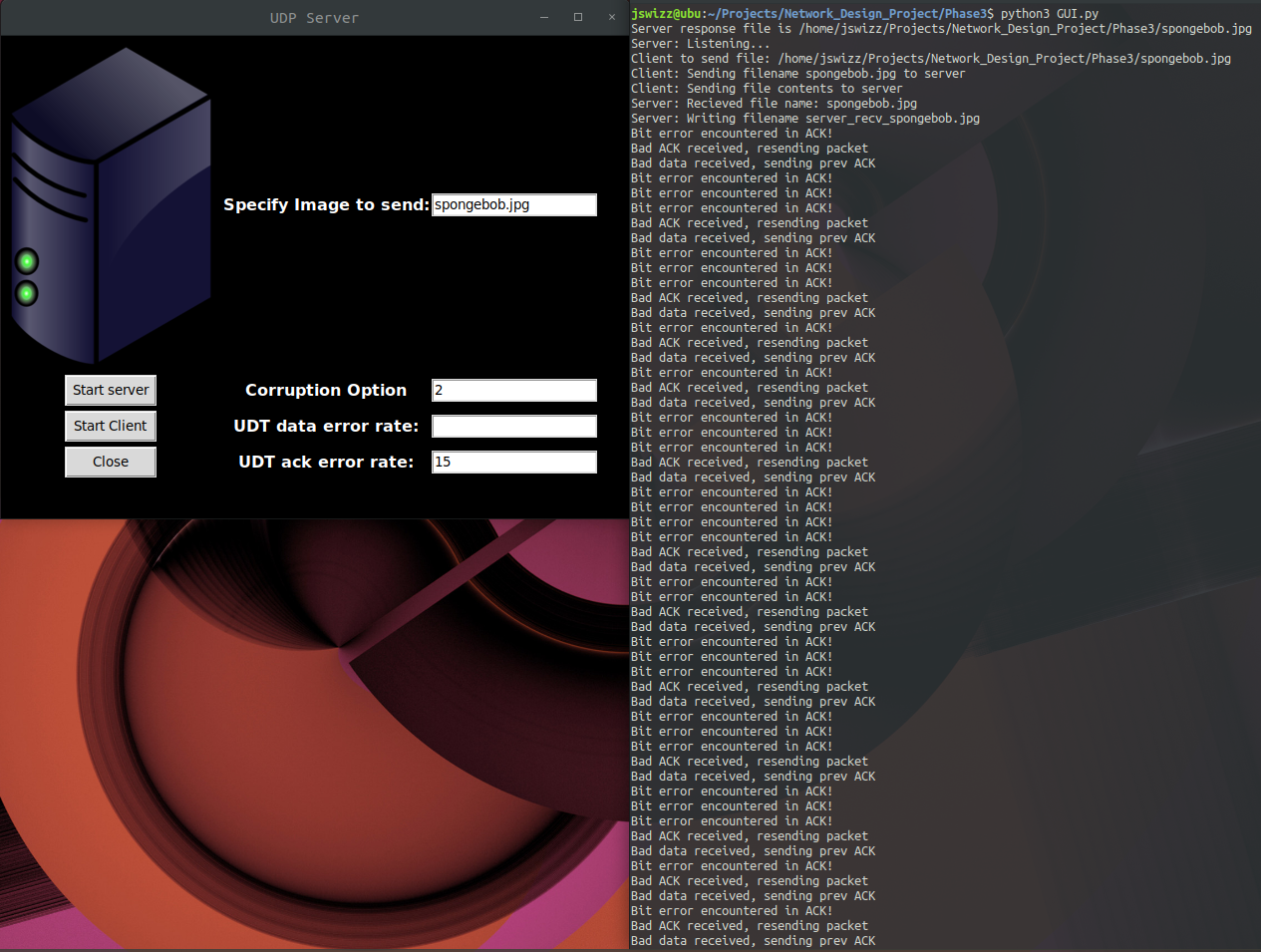
1. Again, leave image blank to use default, or specify a different image, this time for the client to send. Then click client.  
   
2. You will see messages on the GUI, as seen in the image above. In the terminal messages will appear of what is happening as seen below.  
   
3. New files will be prefixed with “client\_recv\_{name}” and “server\_recv{name}” depending on which entity they were received by.
4. Click client again to re-run the client or click close to exit. Entering a new image to send, or corruption/debug options will be applied when client is run again.

**Results:**

The below images show test runs of the program using a ~1MB file (spongebob.jpg, included) at each corruption option, 20% was used for error rate where applicable. Note, these were run with debugging statements enabled.

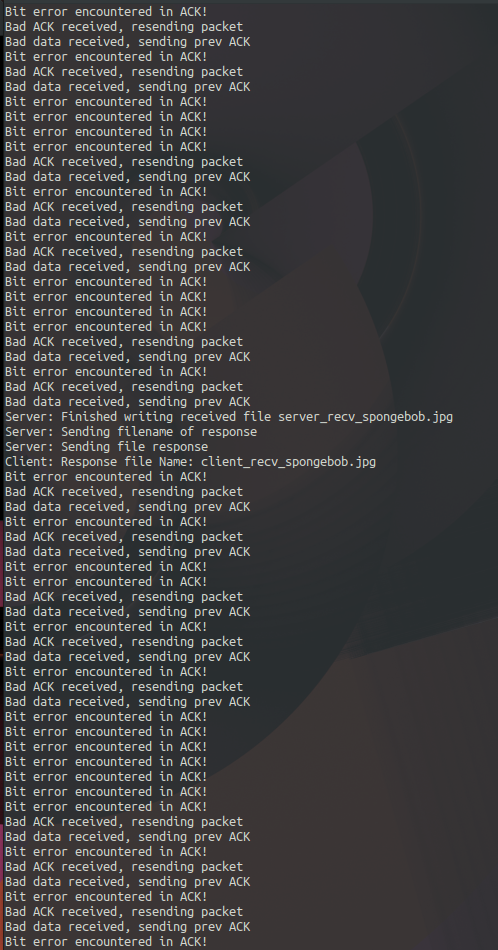
Corruption Option 1 (No Bit Errors)

****

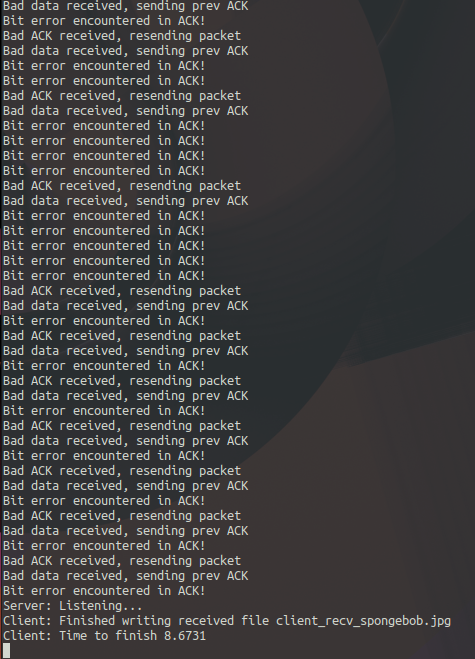
Corruption Option 2 (ACK Bit Errors)

\\

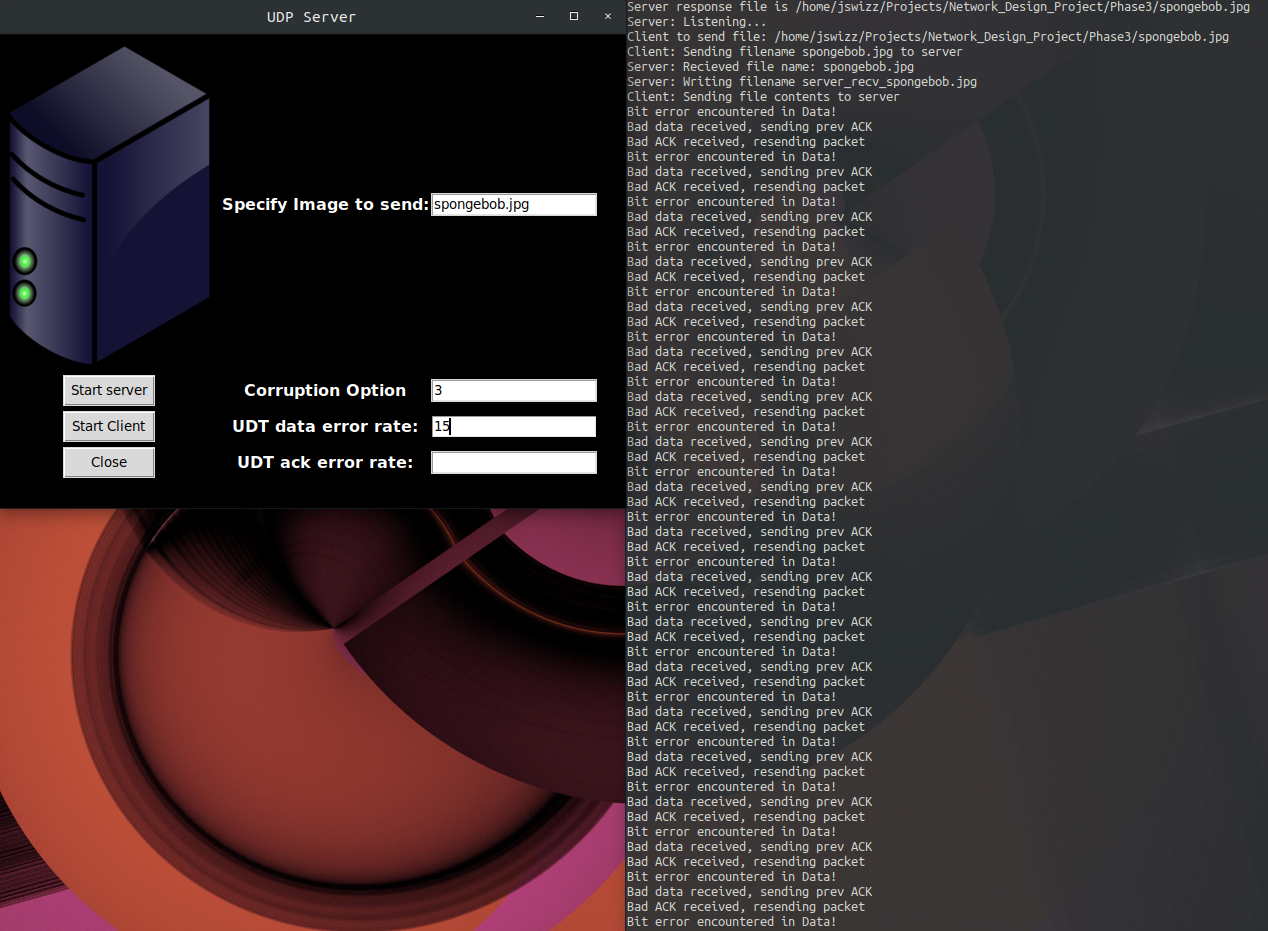
Corruption Option 2 (ACK Bit Errors) [Cont. 1]



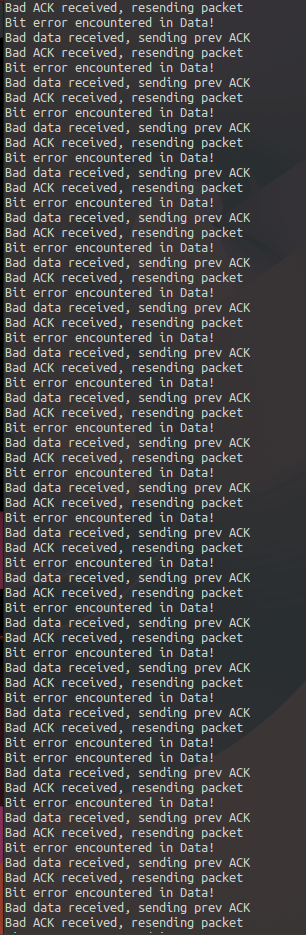
Corruption Option 2 (ACK Bit Errors) [Cont. 2]



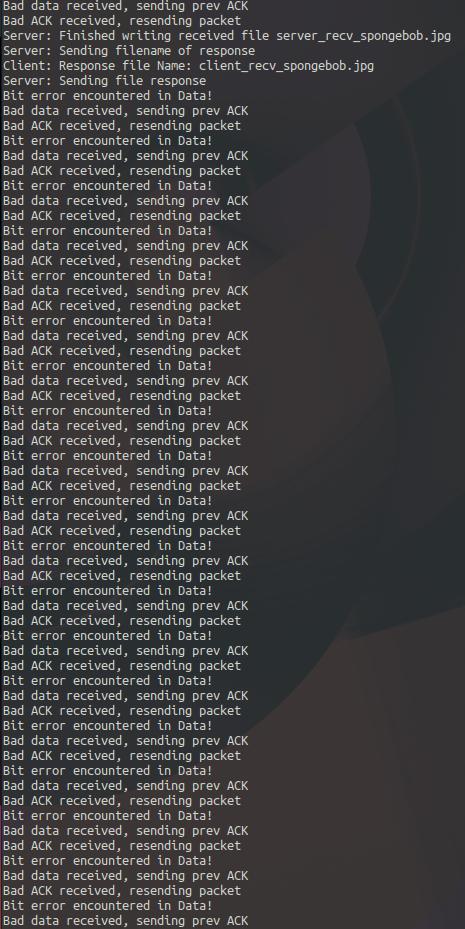
Corruption Option 3 (Data Bit Errors)



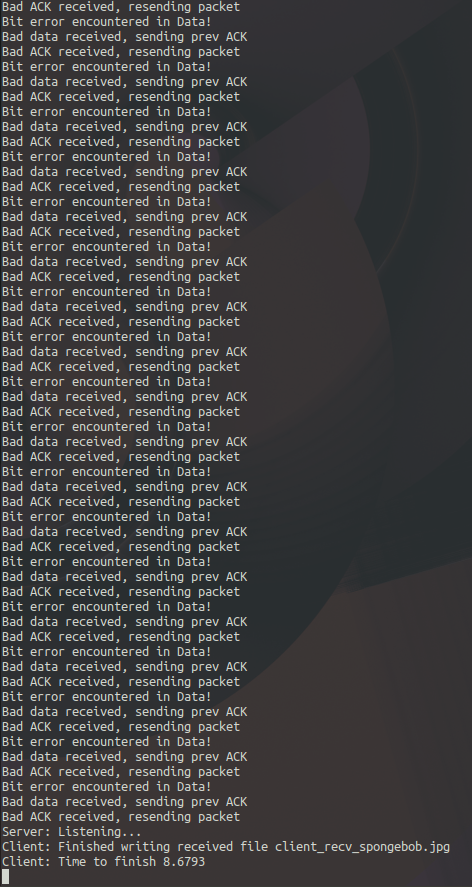
Corruption Option 3 (Data Bit Errors) [Cont. 1]



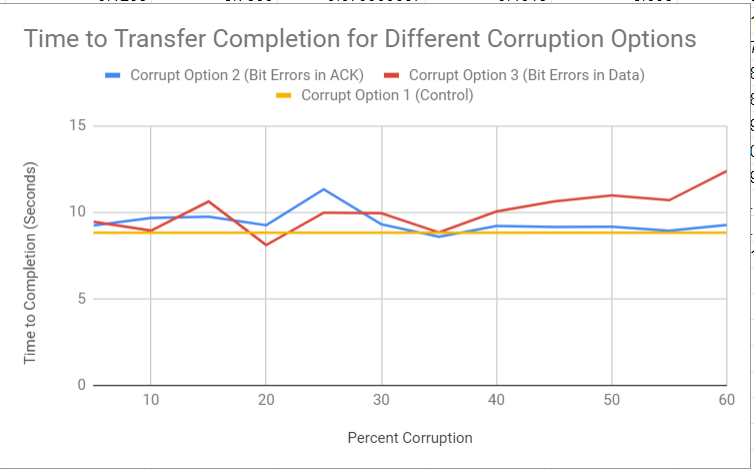
Corruption Option 3 (ACK Bit Errors) [Cont. 2]



Corruption Option 3 (ACK Bit Errors) [Cont. 3]



Finally, to test the result of different corruption options and bit error percentages, tests were run to compare time-to-finish the bidirectional image transfer at increments of 5% bit error rate for both corruption options 2 and 3, and compared with a control timing that was conducted with corruption option 1. For these tests, the same ~1MB jpg file noted above was used, and debugging print statements were turned off. From these results, a plot was created. Each data-point in the plot represents the average of 3 trial runs at the given bit error rate. This plot can be seen below.



From this plot, we could tell that for the most part, every error rate used for both corruption options 2 and 3 took longer to complete than the control. However, the blue line denoting corruption option 2 did not present a clear trend upward or downward as percentage of errors was increased, and tests were fairly sporadic in results. The red line denoting corruption option 2 did however pose a relative upward trend especially at higher error rates, but returned sporadic timing results at lower levels.