Blaine McMahon, Jacob Sword, Nicholas Yameen

Network Design

Project Phase 6

5/3/19

**1. Introduction:**

This code completes Phase 6 of the project by using UDP to send images from a client to a server, and vice versa across a channel that has the possibility to introduce bit errors and lost packets. This version utilizes Selective Repear protocol to communicate across the channel. The UDP server runs continuously to accept messages from clients. These messages contain image contents, that the receiver will write to a file, in order. 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 1a. Server Flowchart (Pt. 1)

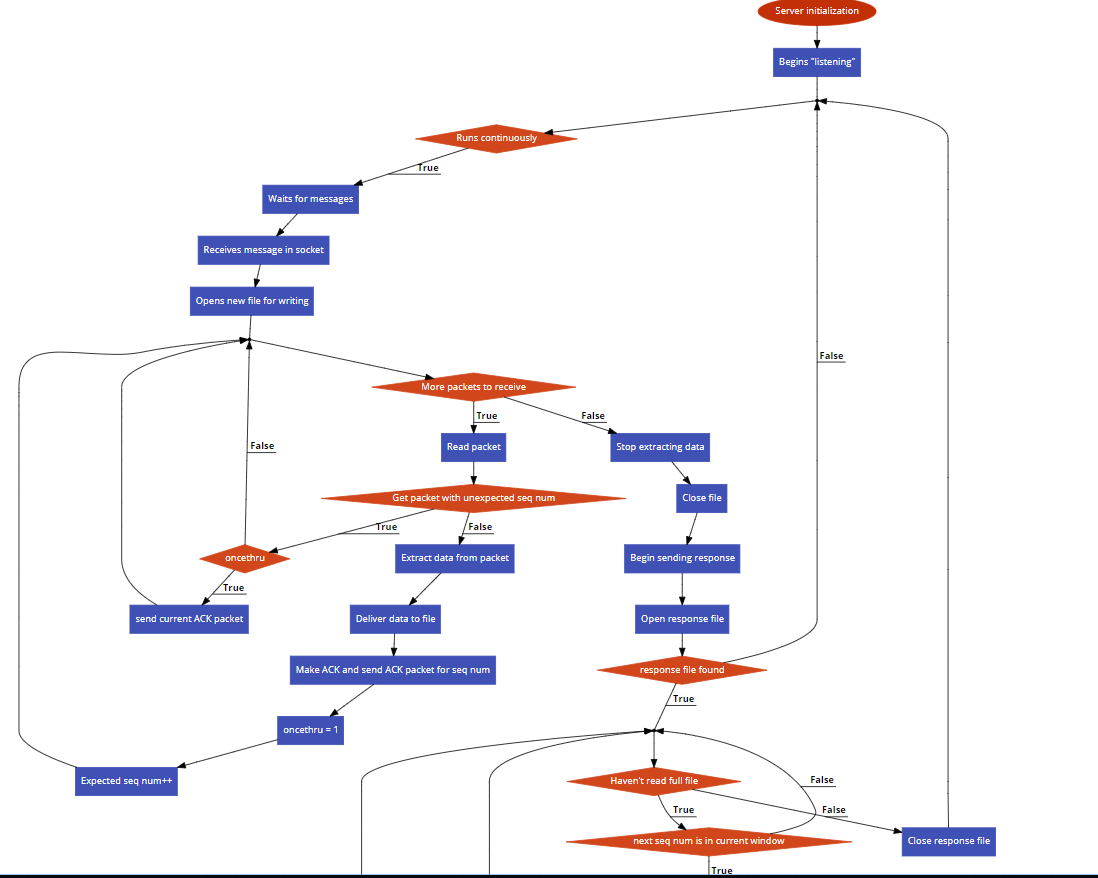


Fig 1b. Server Flowchart (Pt. 2)

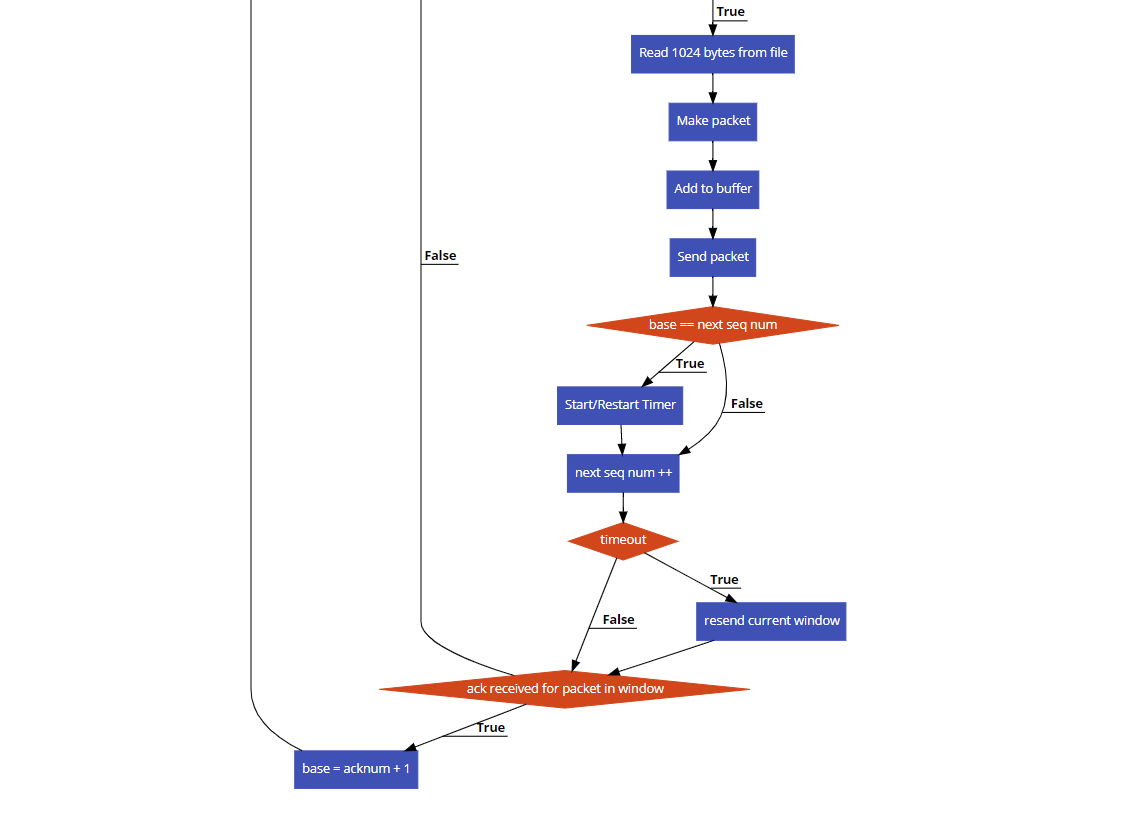
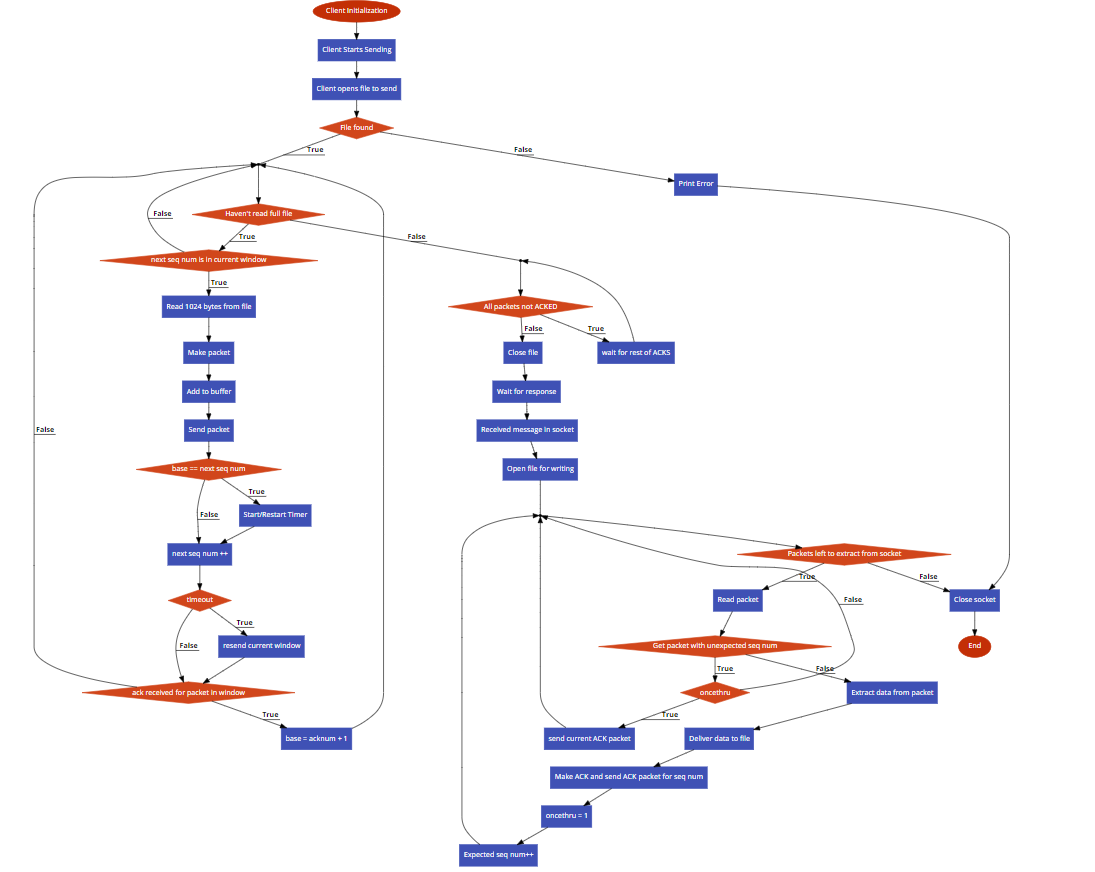
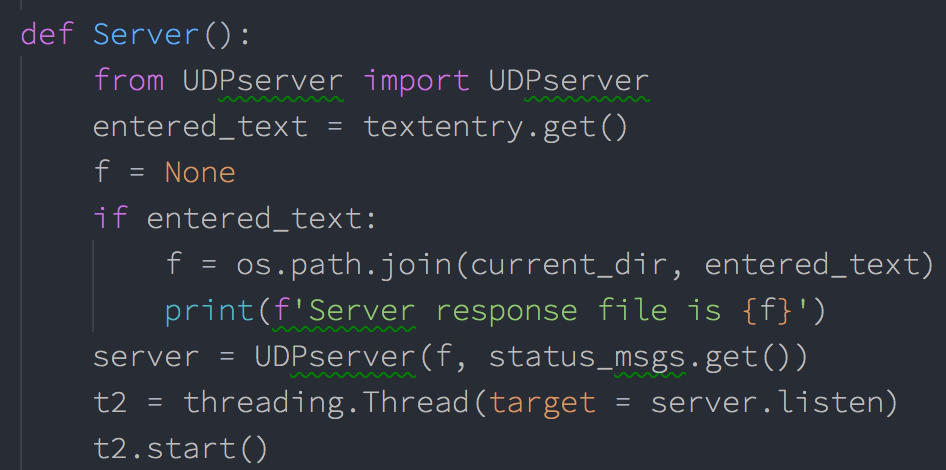


Fig 2. Client Flowchart

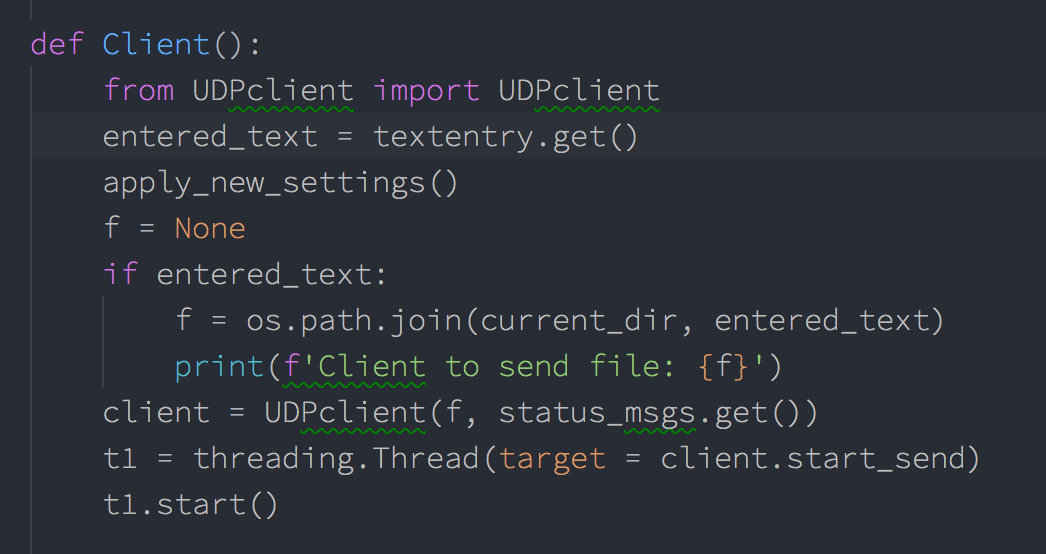


**GUI.py**

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, three input boxes, and two checkboxes. The user will first enter their intended file to send in the first input box, or leave it blank to use our default provided file path. Also, the user can now enter which checkboxes they want which correspond to which printouts are displayed. Checking “Status Msgs” will allow messages such as “Server: Finished writing received file” to be printed to the console. Checking “UDP Err Msgs” will allow messages such as “ACK Packet Dropped!” to be printed to the console. The user then has to click Start Server. This will start the Server function shown below. This function gets the entered filepath, or uses the default, and instantiates a UDPserver class with the noted filepath, and *status\_msgs.get()* which gets the state of the “Status Msgs” checkbox (Boolean variable).



The server class’s “listen” function is then run in a new thread. Next, the user can enter their desired “Corruption Option” and “UDP Error Rate” in the corresponding input boxes of the GUI. Also, the user can uncheck “Loss Recovery” if desired. Then, the user can click the “Start Client” button. This will call *Client* function shown below.



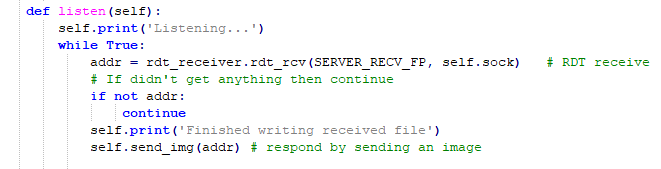
This function operates in a similar fashion to the above-detailed Server function, and the client’s *start\_send* function will run in a new thread. However, before this is done, *apply\_new\_settings()* is called. This function gets the current data from the “Corruption Option” and “UDP error rate” input boxes, in addition to the checkboxes, and passes this data onto our *config.py* file, so that it can be later accessed by functions in the rdt\_sender.py and rdt\_receiver.py files.

**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.

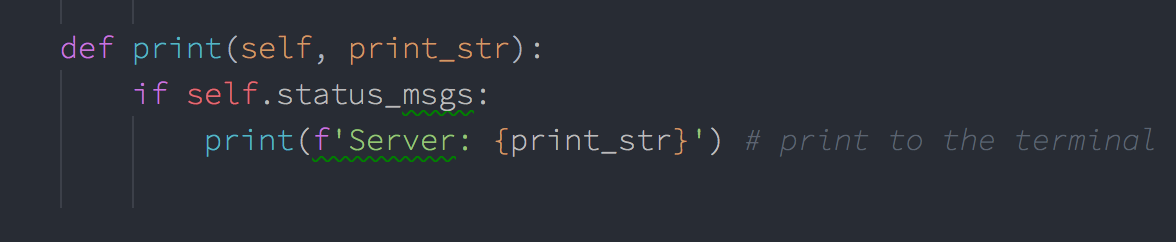


When called, the listen() function will continually run, where it will listen for incoming connections and starts the RDT receive sequence

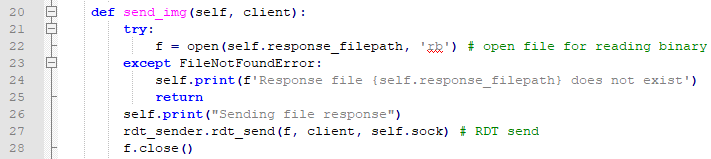


It will then preform a *rdt\_rcv(),* utilizing the rdt\_receiver.py file, where it will continually receive packets from the client and write them to the file until there are no packets left, in a Selective Repeat manner. Once the full image is received it will then close the file and send an image back to the client. (Note: *rdt\_rcv()* is called continuously in a while loop, so if no message is received during this call, (i.e. *if not addr:*) the loop will simply continue to the next iteration, and does not send the response image.

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. However, this is only done if the checkbox for status\_msgs (passed in when server thread was started) was checked.

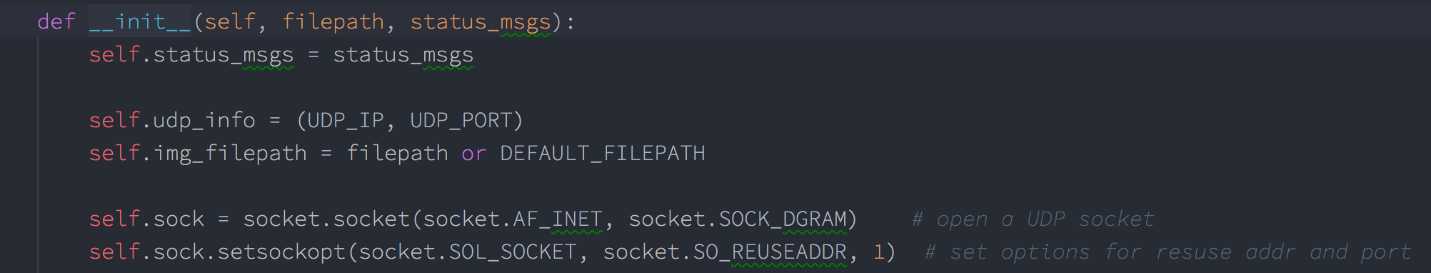


The *send\_img()* function will first open the file specified or the default one for reading binary. Finally, it will send the image using the using the rdt\_sender.py function *rdt\_send()*.

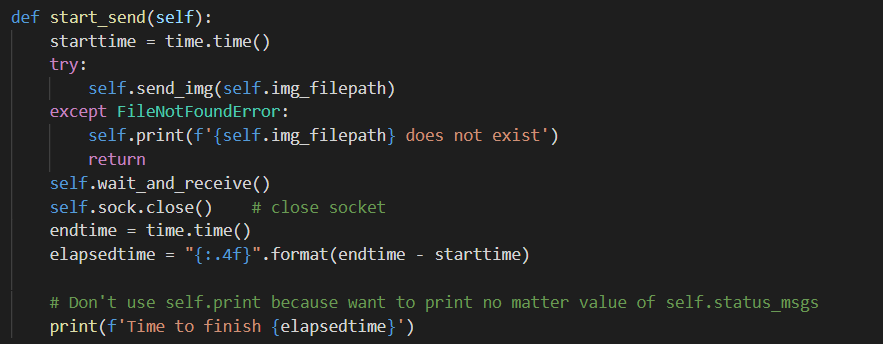


**UDPclient.py**

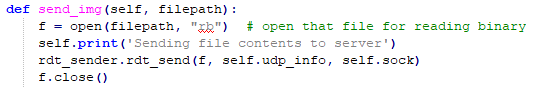
The UDPclient has a class which defines four functions. First, the client will initialize similar to the UDP server. It will connect to the server on the UDP socket, which is made into an instance variable



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 preforms RDT send by calling the *rdt\_send()* function in the rdt\_sender.py file. Once it has finished it will close that file.



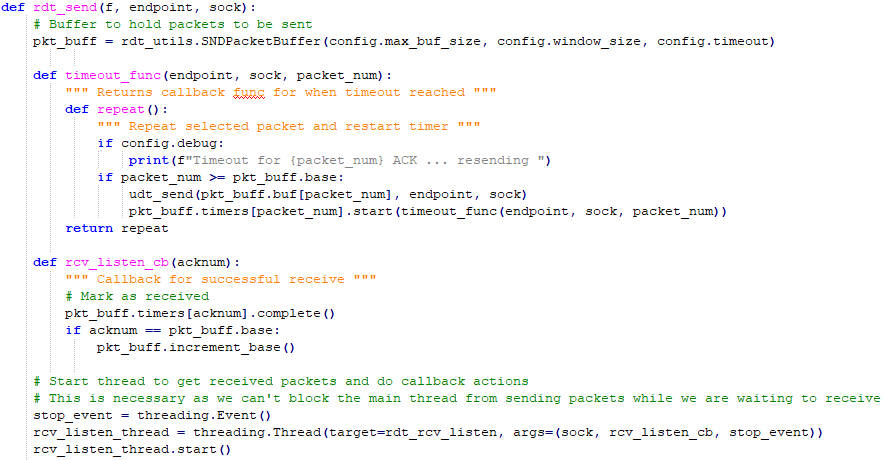
Next, the *start\_send()* will call the *wait\_and\_receive()* function. Here the client will receive the response image name from the server. It will open that file for writing binary. After which it will call *rdt\_rcv()* (from the rdt\_receiver.py file) to receive all the packets of the image. After which, the file will close, and the elapsed time will be printed to the console, which is later used in data collection to constructor our graphs. The client also uses a self.print function to prepend “Client:” to all print strings.

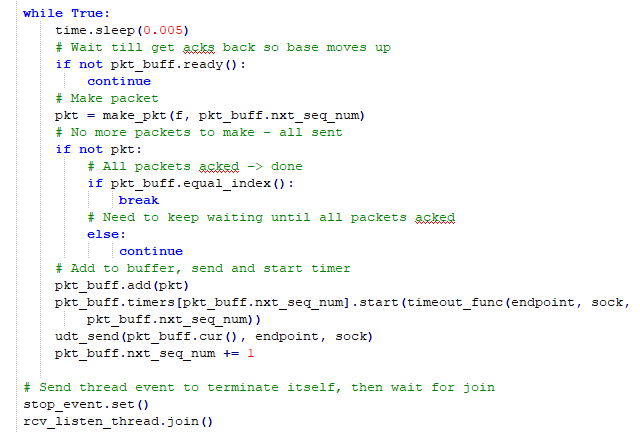
**rtd\_sender.py**

In the file, there are 5 functions defined. The server and client will use these functions to send Selective Repeat messages over a socket. The sender of the UDP interaction will use *rdt\_send()* function. This function will take in a file, endpoint, and the socket. It first starts by making a buffer to hold the packets configured with a set window size and buffer capacity, defined in *config.py*. During this time, it also configures and initializes the timer used in sending. Before the packets can be added to buffer a separate thread is started to ensure the receiver is listening. We have a separate thread running to ensure when packets are waiting to be received, it doesn’t block the main thread which is sending the packets. Also, callback functions are used to define what will happen in the case of a timeout, and when the receiver in the child thread receives a packet.

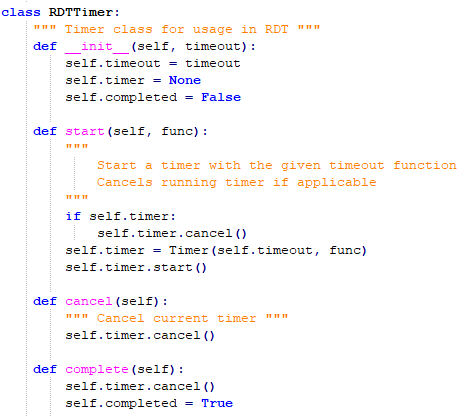
After the initial configurations have been done, the loop condition is checked, that is, if the current packets sequence number is less than the base plus the window size. This is to ensure we are not sending packets that are not in the current window. The packets are then made and added to the buffer. The current window is then specified by the window size. These packets are then sent, and the timer is started. If the timer times out, then all the previous packets that were not ACK’d will be resent using Selective Repeat protocol (using the *repeat()* function). During this processing, if a packet is received that is not corrupt and has a sequence number in the current window, the sender will update its base, restart the timer if the window is complete, and continue working sending packets. If the received packet is corrupt or the wrong sequence, the sender will do nothing. It will continuously make and send packets to the receiver in this fashion, each iteration until the file is completely sent, using the same timer logic each time. Once the file is completely sent, the sender still cannot return, as it must wait for all sent packets to be ACKED. This is handled by a case in the main while loop of this function (*if pkt\_buff.equal\_index(): break else continue)*, as it will continue the while loop indefinitely until all packets are ACKED. Finally, after this happens, the child thread responsible for listening to packets must be cleaned up. This is accomplished by setting an “Event” for the child thread. When this event is set, the child exits from it’s infinite listening loop and finishes. We then join on the child thread. This is shown below.

Additionally, data packet loss is implemented on this side, during sending. As shown below, if the user-defined corruption-option allows for data loss, and the random channel value is lower than the given present corrupt, the packet will not be sent, and will move on through the while loop.



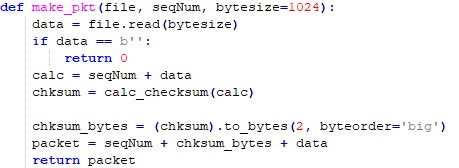


For the timer logic, a timer class RDTTimer is used (from rdt\_utils.py). This timer class allows for “resetting” a timer by canceling it and creating a new one, and passing a callback function to the timer startup, which is called in the case of a timeout. For the callback function, we use the inner-defined *timeout\_func­­* function, which is shown above. This function returns a function the prints a statement to the screen (if the “debug” config option is set, which is controlled by the UDT Error Rate checkbox in the GUI), then resends all un-ACKED packets via *repeat()*. Note, the instantiation of the timer class above uses the timeout value configurated in config.py,this default value is 50ms. The Timer class utilizes Threading.Timer, and is shown below.

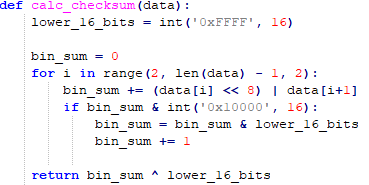


A class needed to be used for this, rather than a simple instantiation of Threading.Timer, because Python does not allow the re-assigning of variables from upper scope within a function. As such, we would not have been able to reassign *timer* to a new timer, which needs to be done after a timeout, and we had to rely on the above shown *RDTTimer.start()* function to do it instead.

To make packets, the *make\_pkt()* function is used to read 1024 bytes at a time. If there is nothing to read, then it will return 0. This is so the *rdt\_send()* has the option to break out of its main while loop after the whole file is read. 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.

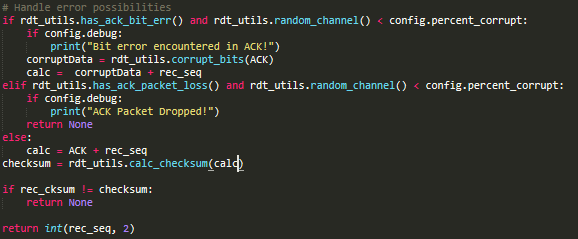


The *udt\_send()­­* function will send the packets to the endpoint over the socket. It returns the number of bytes sent.



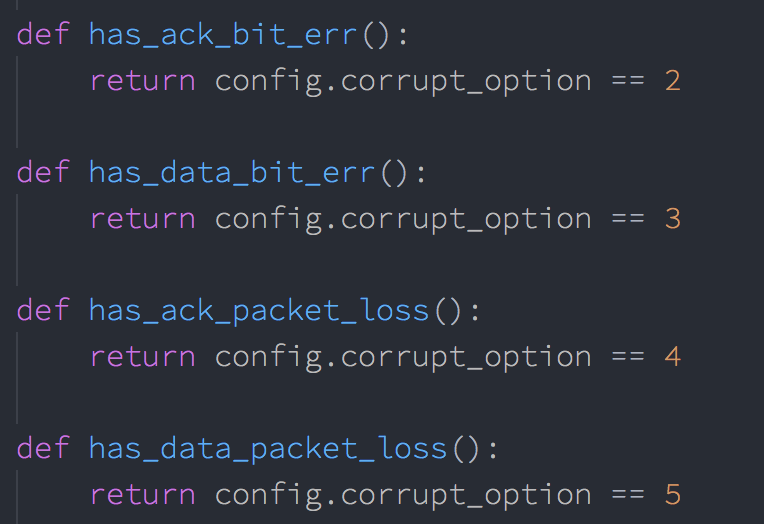
The *rdt\_rcv()* function will receive the acknowledgement from the receiver. It will be called with the socket and sequence number being sent. It will first extract the ACK from the receiver using *extract()*. This function will wait for a packet using extract, if it doesn’t receive something it will timeout. If it did receive a packet it will parse that packet, and parse the checksum from the data using the *parse\_checksum()* function. It then creates its own checksum, this is so the received and calculated checksums can be compared for corruption.

The *rdt\_rcv()* function handles ACK corruption. In the if statement the integer should be a value between 0 (no corruption) to 60 (max corruption). This will have to be changed by the user, as noted above. 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.



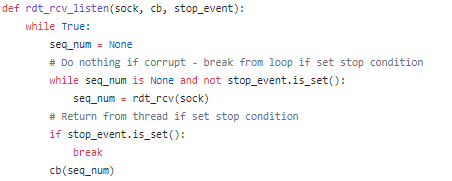


Additionally, packet loss is implemented here using the same logic, if the given corruption open is used. The Boolean value of corruption options states are defined by the given has\*err() functions in rdt\_utils, shown below.



If the data is not corrupt, we return the ACK number that was received.

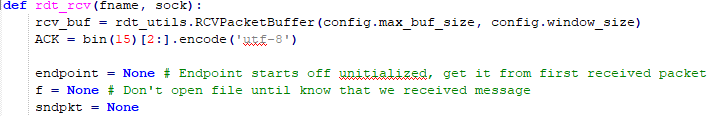
This function is called continuously in the child thread mentioned above via the function *rdt\_rcv\_listen*. This is shown below.



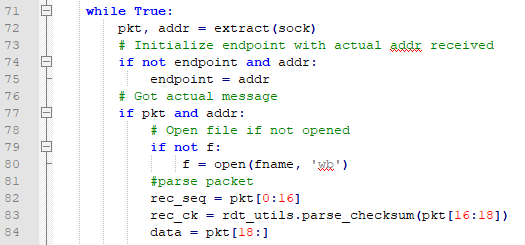
This will run continuously, looking for a valid seq\_num to be returned from *rdt\_rcv*. If it does get a sequence number, it will call the given callback *cb* on the sequence number. The cb function is *rcv\_listen­\_cb*, which was shown earlier. Finally, if this function receives a stop event, it will exit, completing the thread’s work.

**rdt\_receiver.py**

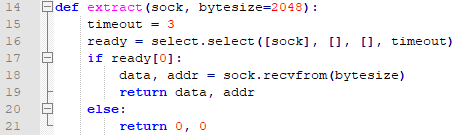
The receiver will utilize the functions inside the rdt\_receiver.py to complete Selective Repeats transactions. The first function is *rdt\_rcv()*, which takes the file to write to, the endpoint, and the socket. This function will continuously run until there is no packet received. It will first setup initial parameters.



The receiver will extract a packet and parse the packet accordingly. If it is the first packet it, it will open a new file for writing binary.



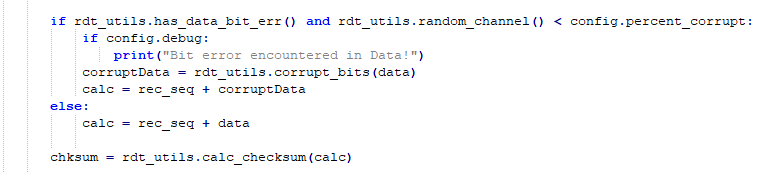
The *extract()* function will receive a packet from the socket. If there is a packet it will return it, otherwise it will return 0 (for both data, and addr).



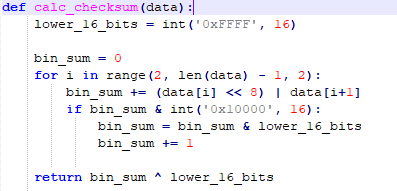
After, the packet is parsed as seen above. The received sequence number and data are stripped from the packet. The received checksum is parsed through the *parse\_checksum()* function.



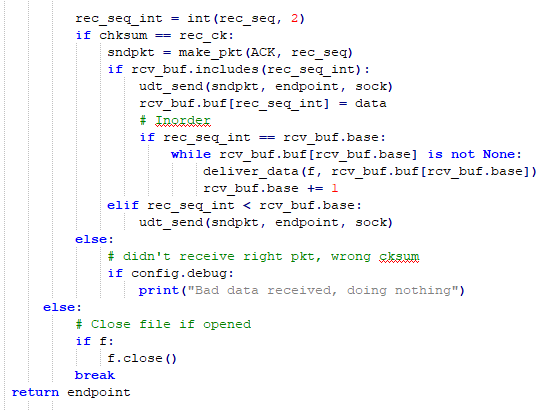
If the user selected some type of corruption, it will notify the user if there is corruption and corrupt the data. Otherwise it will just make its own checksum.



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



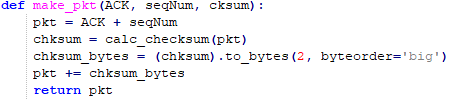
The checksum is compared. If everything is right, it creates an ACK and makes sure the buffer is stored in order. If it is not in order, it will send previous packet. If wrong checksum, it will no do anything.



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.



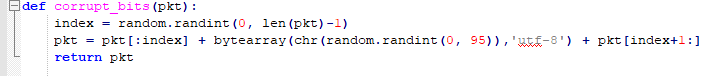
The packet will then be sent over the socket back to the sender.



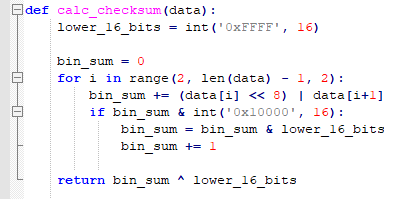
The expected sequence number will then increase and the once through variable will be set. If there was a corrupt packet, or an unexpected sequence number was recieved, the previous ACK packet will be sent again, but only if loss\_recovery is set in the config (corresponding to the “Loss Recovery” checkbox). If the loss­\_recovery Boolean is False, the function will continue sending a new ACK, without dealing with lost packets. Finally, if there are no packets left to receive, the file will close and the *rdt\_rcv()* will return.

**rdt\_utils.py**

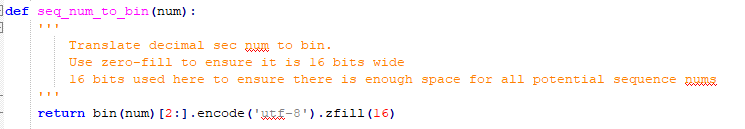
This file holds all the functions that both the receiver and sender use. The first function is to corrupt data. Using some random number, certain bits will be changed.



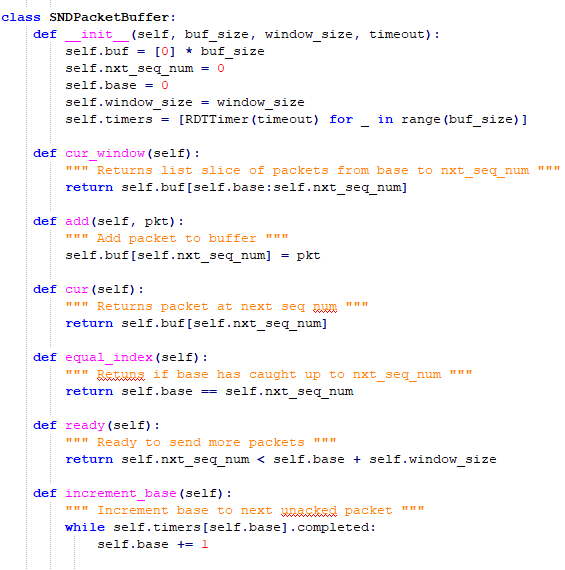
Next is the *calc\_checksum* function. This function was shown in the previous sections. However, it preforms the 1’s compliment addition to create a checksum.



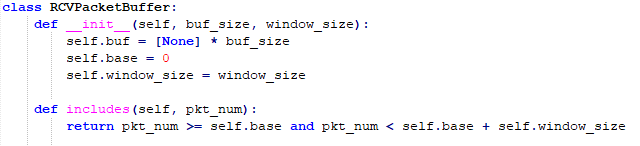
The next function if a function used to translate a decimal number to binary, called *seq\_num\_to\_bin()*.



New to Phase 6 is the SNDPacketBuffer class and RCVPacketBuffer. The SNDPacketBuffer class defines 6 functions. The init function will set the next sequence number, base, and window size. The rest of the function defines what to do for the buffer window. First, cur\_*window()* returns all packets from base to next sequence number, this is used in the *go­\_back\_n* function shown earlier. The *add()* function adds new packets to the window, at the index of *nxt\_seq\_num*. The *cur()*function will return packets at next sequence number. *Equal\_index()* function will return when base is equal to next sequence number, which is used to determine whether to start a new timer, and whether all packets are ACKED (corresponding to different conditions detailed earlier). Lastly, *ready()* returns when the sender is ready to send more packets (i.e. has a window with unsent packets).

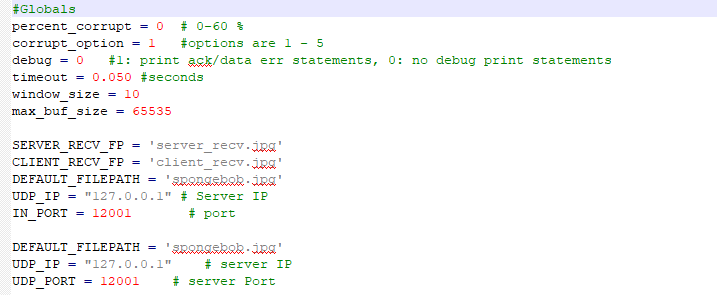


Lastly, RCVPacketBuffer has 2 functions defined. The *init()*function initializes the buffer, base, and window size of the receiver. The *includes()* function will return true if packet number is greater or equal to the base AND if packet number is less than base + window.

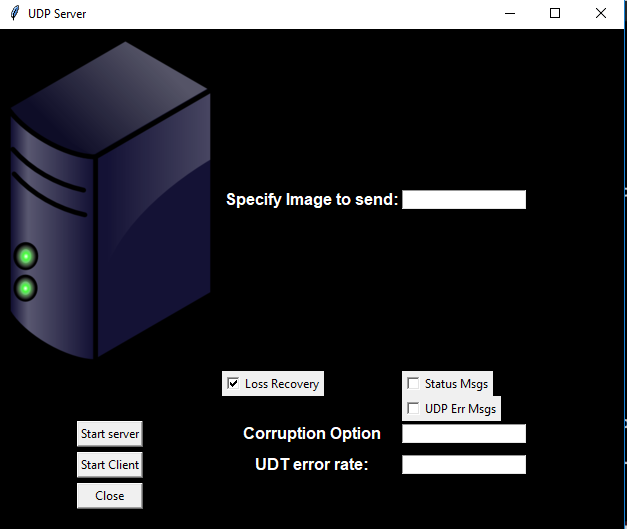


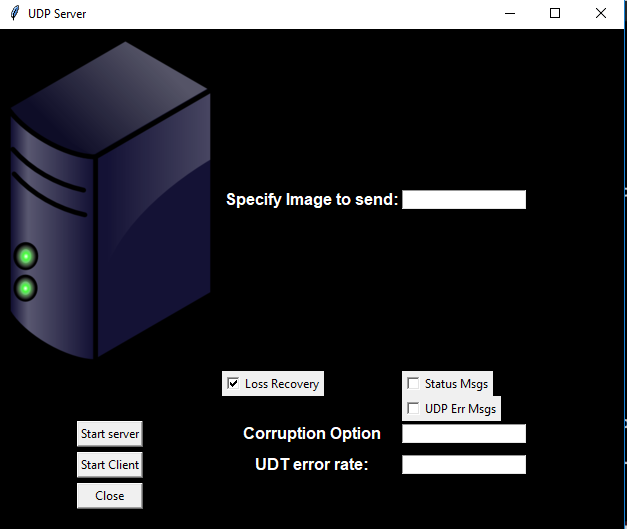
**Config.py**

The last file is our config file. This file holds all of our configurations to preforms Selective Repeat and handle error rates. The contents of this file are below.

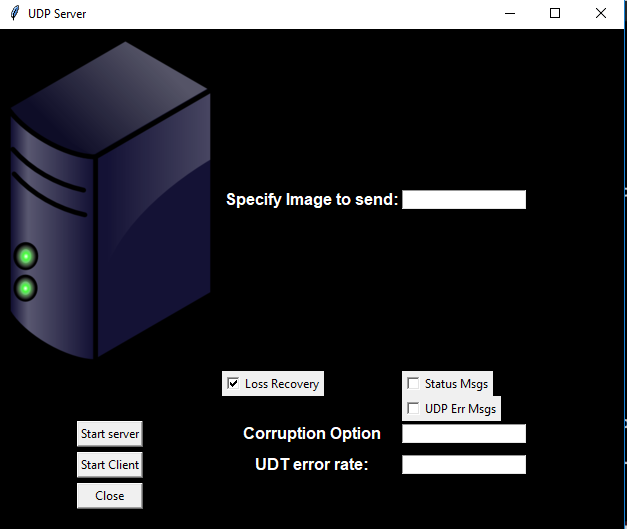


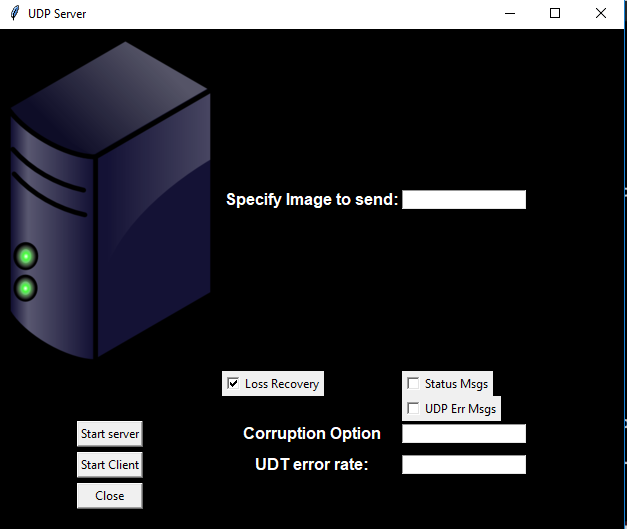
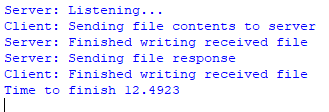
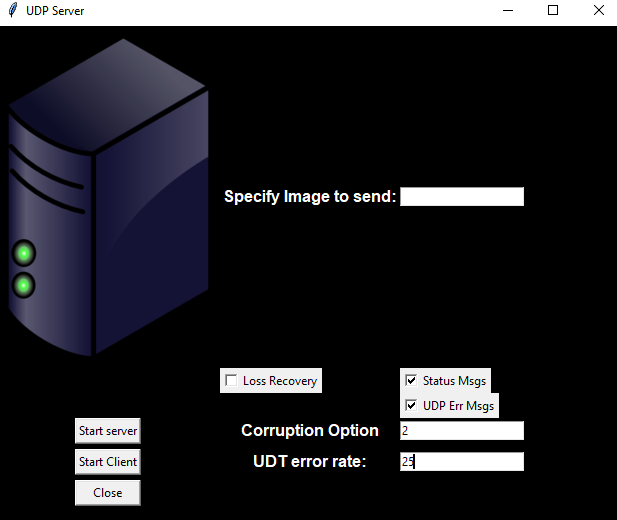
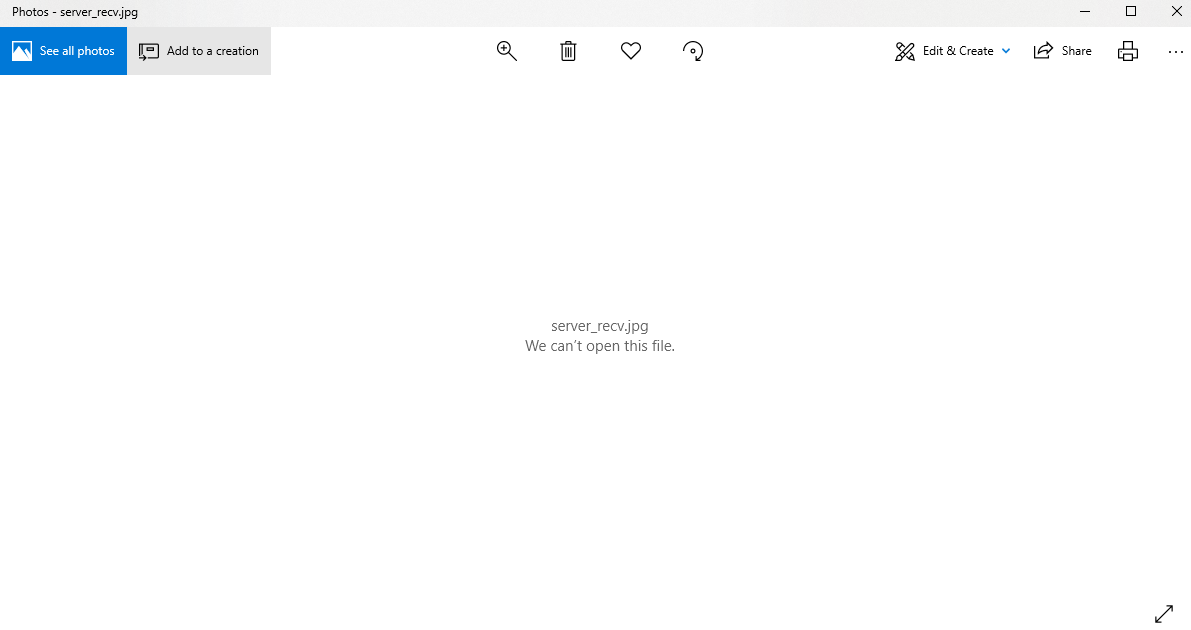
**How to Run**

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. Click Status Msgs, if you would like to see messages such as “Client: Sending file contents to server“, and click “UDP Err Msgs” if you would like to see error status messages like "Data Packet Dropped!". Enter the desired corruption option (1-5). Entering 1 denotes no errors, 2 denotes bit error in ACK packets, 3 denotes bit error in Data packets, 4 denotes loss of ACK packets, and 5 denotes loss of Data packets. These options will be used by both client and server.



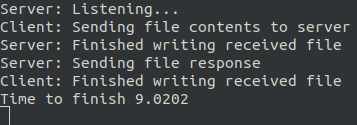
1. Leave file to send blank to use default or specify path to your own picture. Click server



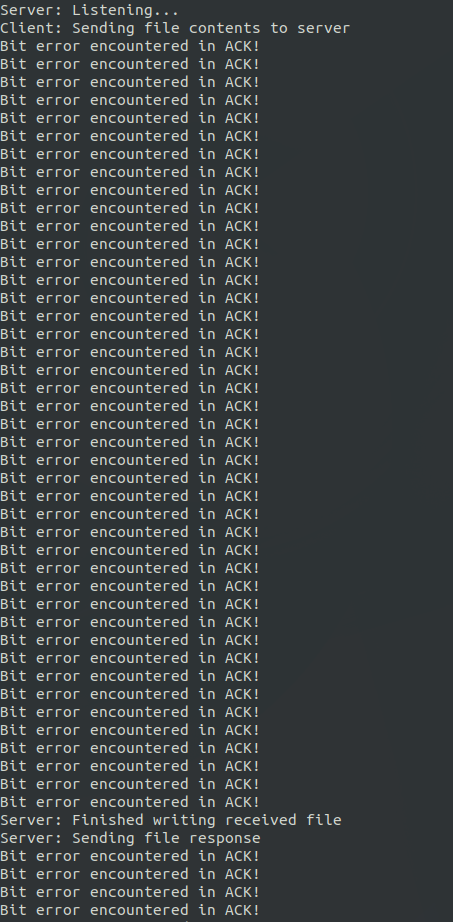
1. Again, leave image blank to use default, or specify a different image, this time for the client to send. Then click client.  
   
2. the terminal messages will appear (if relevant checkboxes are clicked) of what is happening as seen below.   
   
3. At the end of the bidirectional transfer, the client will print out the completion time for the transfer. This printout happens whether or not the checkboxes are clicked.
4. New files will be named “client\_recv.jpg” and “server\_recv.jpg” depending on which entity they were received by.
5. 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.
6. To see what would happen if there was no loss recovery, uncheck the Loss recovery box and run the server and client with errors of some type. An example is below.  
     
     
     
   Server received (file is corrupt and unable to open):  
     
   Client received (file is missing/has garbled data):  
   

**Results:**

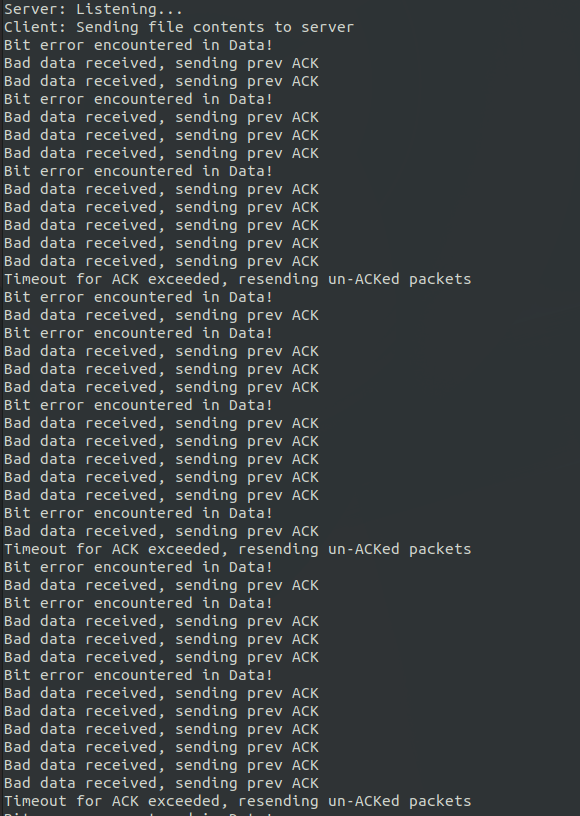
The below images show test output of the program using a ~1MB file (spongebob.jpg, included), and corruption options 1, 2, 3, and 4 and 5 respectively with 20% error rate where applicable. “Status Msgs” and “UDP Err Msgs” output was enabled for these images. (Note: full output to completion of transfer was not included for options 2-5 as it was many pages long).



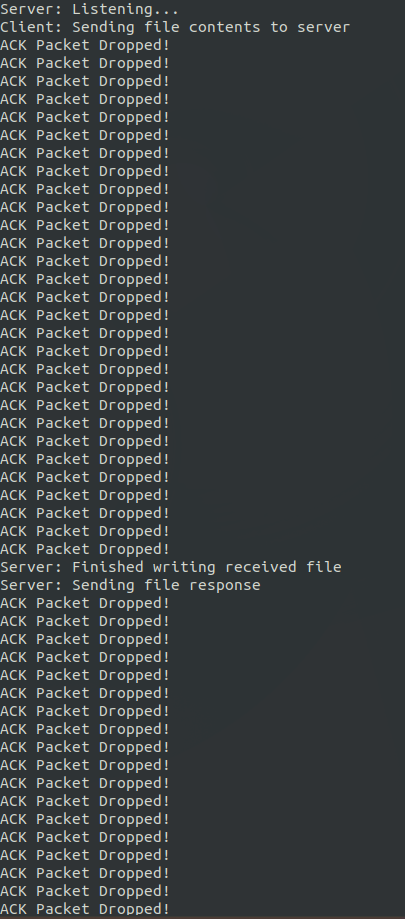
Test Output, Corruption Option 1



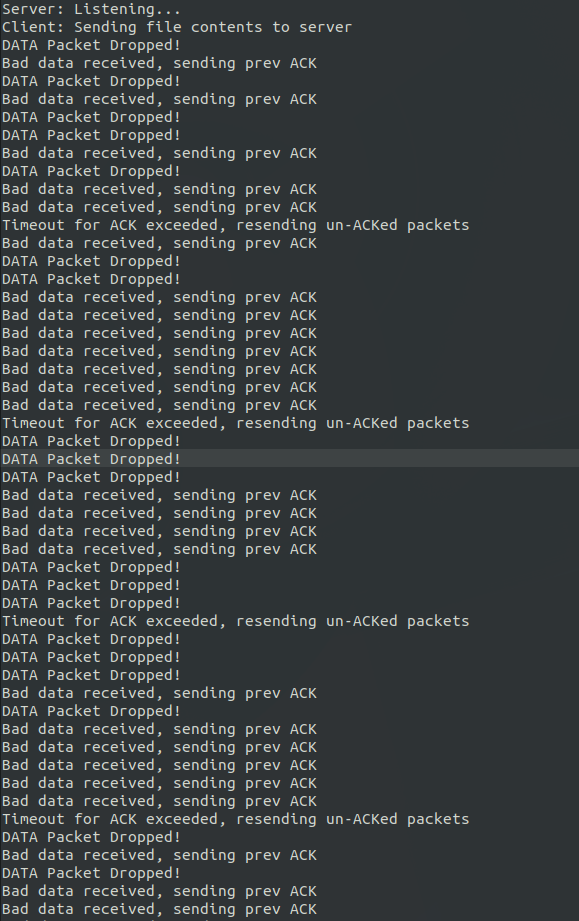
Test Output, Corruption Option 2 with 20% Error Rate



Test Output, Corruption Option 3 with 20% Error Rate



Test Output, Corruption Option 4 with 20% Error Rate



Test Output, Corruption Option 5 with 20% Error Rate

Finally, to test the result of different corruption options and error percentages, tests were run to compare time-to-finish the bidirectional image transfer at increments of 5% error rate for corruption options 2 through 5, 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 all debugging print statements were turned off. Also, the default timeout value of 50ms, and window size of 10 (defined in config.py) was used. 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 see that Go-Back-N handles No Corruption, Bit Errors in the ACK, and ACK Packet Loss the best. For Option 1, Option 2, and Option 4, the image appeared to be sent in about 12s for each percentile. The slope is nearly zero for each. This is because Go-Back-N handles ACK packet errors best because of the collective ACK. Wrong/dropped ACK packets are handled in most cases by collective-ack, as an ACK for a later sequence number makes it through. This mean the sender doesn’t need to handle making new packets. However, for Data Loss (option 5) and Data Error (option 3), Go-Back-N seems to not handle it well. It took more time to complete the longer it took. The sender would have to send the bit-errored/dropped packet again as well as any left in the window. This is what made the time so long. The slope for Option 3 was 1.57 and Option 5 slope was 1.28. To conclude, Go-Back-N handles ACK loss the best due to the fact it is a collective ACK.

The next graph is the Time to Complete with a fixed loss rate of 10% while varying the window size. Again the trial was run 3 times each. Below is the results.

Here we see a small window size (less than 2) has varying results. However, looking at the data, each packet roughly took the same time to transfer given its Option. When comparing it to graph 1, the times at 10% Loss/Error are roughly the same throughout different window sizes. This shows that a varfyed window does not affect its outcome. The data here suggests that optimal window size can be any size greater that 2.

The last chart is comapring Phase 3, Phase 4, and Phase 5.

Here we see that Phase 4 has the more linear graph, averaging around 15s to complete. However, when it comes to ACK Loss or Error, Phase 5 completes its transfer faster by about 2 seconds. Phase 3 appears to be somewhere between 4 and 5. To conclude, Phase 5 is better for handling ACK Loss or Errors, while Phase 4 is better for Data loss.