Design Document 12/8/2023

RDT 2.0

Daniel Maccaline, Nathan Grady

Project Phase 3

**Phase purpose:**

The purpose of this phase is to implement Go-Back-N protocol, with a system that is able to reliably transfer a file over a connection that can both drop and corrupt packets.

**Code explanation**

**Overview:**

One Boolean control variable and five integer control variables are used to adjust the functionality of the code. The first three are found in functions.py, and are printFlag, corruptionPercent\_server\_to\_client and corruptionsPercent\_client\_to\_server. These printFlag Boolean controls if the debug output information is printed. The corruptionPertcent variables indicate the percent chance of corruption in each direction. In addition, there are two loss\_percent variables that control the chance for the packet to be dropped by the channel. Note that values entered here are in percentages corruptionPercent\_server\_to\_client = 60 is a 60% chance of corruption, or a probability of .6. Note that any time any of these are changed, the server must be restarted for them to take full effect. The next variable is the lossPercent, which is the amount of packets dropped. Note that this variable effects loss in both directions, and is a percentage in the same way the corruption controls are.

The final two values define how the go-back-n protocol functions. The first is window, which determines the window size for go-back-n. The second is the timeout value. The timeout should be given as a decimal in seconds.

**Client.py**

**Note: Client.py**

*"""  
TCP Client  
Authors: Daniel Maccaline and Nathan Grady  
 Based on code from phase 1 (Daniel Maccaline)  
"""*import socket  
import tkinter as tk  
from tkinter import filedialog  
from send\_receive import \*  
import datetime  
import time

Imports and Headers

def Get\_Packets\_Raw(file, packetsize):  
  
 currentIndex = 0  
  
 data = file.read()  
  
 #create packet list  
 packet = [b'']  
  
 while(currentIndex < len(data)):  
  
 #extract just the data from the packet  
 bytesdata=data[currentIndex:currentIndex + packetsize]  
  
 packet.append(bytesdata)  
  
 currentIndex += packetsize  
  
 packet.append(b'stop')  
 return packet

return packet

Get\_Packets\_Raw functions. Splits the file data into a list of equally sized packets, each of a length specified by packetsize. a packet is appended to the end of the packet list that simply says ‘stop’ this will be used to indicate to the server that the file has finished transferring. Note that no headers are applied at this point.

def UDPClient(fileName):  
  
 # packet size in bytes  
 packetSize = 1024  
  
 # read file in here  
 try:  
 file = open(fileName, "rb")  
 except:  
 print("File could not be opened...")  
 return  
  
 if file.closed:  
 print("File could not be opened")  
  
 #raw packets means just the packet with no header or anything yet  
 data = Get\_Packets\_Raw(file, packetSize)  
  
 file.close()  
 # endregion

Attempt file read from filename specified in argument. Note that the variable packetSize is also set at this point. Once opened, create packets using Get\_Packets\_Raw, then close the file.

# set server name and port to expect server at  
 serverName = 'localhost'  
 serverPort = 11000  
 # create UDP Socket  
 clientSocket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

Create UDP Socket for connection to the server

#send packets one at a time  
try:  
 print("PIPELINED DATA TRANSFER WINDOW SIZE: ",window)  
 print("sending ",len(data),"packets")  
 print("corruption rate from the client to the server is: ",corruptPercent\_client\_to\_server,"%")  
 print("corruption rate from the server to the client is: ",corruptPercent\_server\_to\_client,"%")  
 print("loss percent both ways is: ",lossPercent,"%")  
 print("timeout is : ",timeout," seconds")  
 print()  
  
 start\_time = datetime.datetime.now()  
  
 rdt\_send(clientSocket, serverName, serverPort, data)  
  
 #send stop bit  
  
 end\_time=datetime.datetime.now()  
 print()  
 print("finished sending")  
 print("start: ",start\_time," end:",end\_time)  
 time = end\_time-start\_time  
 print("total time: ",(end\_time-start\_time))  
  
except:  
 print("ther server is probably down")  
 return 0, 0

Ty to send the data to the server. First prints the messages indicating the loss chance, then records a start time. Then rdt\_send is called which will transmit all of the packets passed to it. Then records an end time, and prints the statistics for the run.

If the try fails, then the server did not respond, and an error is printed.

clientSocket.close()  
 #Return time taken in microseconds and seconds  
 return time.microseconds, time.seconds

Close the socket connection, convert time to a microseconds and seconds value, and return the values to main

#Main, used to start TCPClient and send name of passed file  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 #Variables used for automatic tests, Iterations -> Number of tests, runMultipleTests bool used to control if tests are done  
 iterations = 3  
 runMultipleTests = False  
  
 root = tk.Tk()  
 root.withdraw()  
  
 file\_path = filedialog.askopenfilename()  
  
 #check if input argument provided  
 if len(file\_path) <= 1:  
 #output error if no input file provided  
 print("Error: No input file specified")

Main function. Sets values for running multiple tests, and opens dialog box prompting the user for a filepath. Then checks if a filepath was selected.

else:  
  
 #if not running multiple, just call UDP client and disregard returns  
 if not runMultipleTests:  
 #pass input file name to client  
 UDPClient(file\_path)

This else statement is connected to the if statement in the previous set of code. If not running multiple tests, the code simply calls the UPDClient function with the filepath as an argument.

#Test  
 else:  
 #Store number of iterations for denominator of average calc  
 avgDen = iterations  
 #Stores time returned by function  
 timeR = 0  
 #iterate for (iterations) times  
 while(iterations > 0):  
 print("\n\n\nStarting run " + str(avgDen-iterations))  
 iterations = iterations - 1  
 #Function returns time spent in microseconds and seconds  
 micros, sec = UDPClient(file\_path)  
 #add time to timeR, dividing micros to adjust  
 timeR = timeR + micros/1000000 + sec  
 #sleep before next call (avg time goes up without this sleep)  
 time.sleep(1)  
 #print average results  
 print("Average results: ", str(timeR/avgDen))

This if is connected to the if in the previous code block. If running multiple tests, store the number of iterations, and set timeR to 0 to store the returned times. Then iterate in a loop for iterations times, storing the timed result of each run to timeR. Then print the average, by dividing timeR by avgDen.

**Server.py**

**Note: Server.py is unchanged from phase 4**

*"""  
TCP Server  
Authors: Daniel Maccaline and Nathan Grady  
 Based on code from phase 1 (Daniel Maccaline)  
"""*import socket  
from send\_receive import \*  
import os

Header and import statements.

def UDPServer():  
  
 #string of bytes to hold passed file  
 frame=b''  
  
 #Define server port number  
 serverPort = 11000  
  
 #Create UDP Socket  
 serverSocket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  
 #Bind socket to port mumber  
 serverSocket.bind(('', serverPort))  
 #output message indicating ready to recieve  
 print('The server is ready to recieve')  
  
 count=0

Setup server and initialize variables to store the received data (frame, stores in bytes) and which packet the server is receiving (count, only used for debug output).

#Loop forever, continually read messages sent to socket  
 while True:  
  
 rcvPacket, addr =rdt\_rcv(serverSocket)  
 if(printflag): print("recieved packet: ", count)  
 if(printflag): print()  
 count+=1

Loop, continually receiving packets indefinetly. Uses the function rdt\_rcv to receive packets (function defined in send\_recieve.py, discussed later).

#Test  
 #if passed sentence = stop code  
 if(rcvPacket==b'stop'):  
 count=0  
 #store created output to bmp file and open the file  
  
 f = open("temp.jpg", "wb")  
 f.write(frame)  
 f.close()  
 os.startfile("temp.jpg")  
  
 #clear the frame  
 frame = b''  
 #Output completion statement  
 print("Finished recieving file\nFile opened in seperate window")  
  
 else:  
 #if not at end of file, concatenate sentence to frame  
 frame+=(rcvPacket)

If the packet sent has the stop code, then stop the server is at the end of the file. Reset count for the next file, write the data to a file (temp) and clear the frame. Finally, output message indicating completion. Otherwise, append the received packet to the frame.

#Main method used to start server  
if \_\_name\_\_ == "\_\_main\_\_":  
 UDPServer()

Main function. Used to simply start the server function.

**send\_and\_recieve.py**

from functions import \*  
import threading  
import time  
  
timerExpired = False

Import functions. Functions is defined in functions.py, discussed later in this document. Additionally declares the global Boolean timerExpired, which is used to allow the timer to signal when it has run out of time.

def rdt\_send(clientSocket,serverName,serverPort,file):  
 global TimerExpired,window,fail  
 nextSeqNum = 1  
 base = 1

. Beginning of RDT Send. Imports needed global variables and initializes the next SeqNum and Base to 1.

#sndpkt = [b'']  
while (base<len(file)-1):

Loop until all packets have been sent. Initalize the packet buffer

if not (TimerExpired):  
 # if the next sequence number is between the base and the window but not out of the end of the file  
 if (nextSeqNum < base + window and nextSeqNum<len(file)):  
  
 # make the packet and add it to the buffer  
 if (printflag): print("sent: ", nextSeqNum, " base: ", base)  
  
 sndpkt.append(make\_pkt(nextSeqNum, file[nextSeqNum]))  
  
 udt\_send(clientSocket, (serverName, serverPort), sndpkt[nextSeqNum], corruptPercent\_client\_to\_server)  
 sends+=1  
  
 if (base == nextSeqNum):  
 # start timer  
 starttimer()  
 nextSeqNum += 1

This block of code makes a packet and adds it to the packet buffer then it transmits the next packet in the buffer if it is within the window size. If a packet is transmitted the nexSeqnum is ticked up 1. If the packet is the first packet in the given window at the time it is transmitted start the timer.

## recieve the data  
clientSocket.setblocking(0)  
try:  
 rcvPacket, addr = clientSocket.recvfrom(2048)  
 fail = 0  
  
 if (not corrupt(rcvPacket)):  
  
 data, recieved\_sequence\_num, chksum = extract(rcvPacket)  
  
 if (printflag): print("recieved: ", recieved\_sequence\_num, " new base: ", recieved\_sequence\_num + 1)  
  
 base = recieved\_sequence\_num + 1  
  
 # if the whole window was recieved then pause the timer  
 if (base == nextSeqNum):  
 stopTimer()  
  
 # if we the recieved sequence number is equal to or above the base then reset the timer  
 elif (recieved\_sequence\_num >= base):  
 starttimer()  
except:  
 pass

The above block is the main code used to wait for a response. As recvfrom will return an error if no data is in the socket, a try except statement is used. In the try statement, we try to read input using recvfrom, and if it succeeds. The function extract, defined in functions.py, is called to get the data, sequence number, and checksum. The base is updated to the sequence number of the packet we received. Then the timer is reset.

## if the timer is expired resart the timer then retransmit base up to nextseqnum-1  
if (TimerExpired):  
  
 starttimer()  
 for i in range(base, nextSeqNum):  
 if (printflag): print("resending : ", i, " base: ", base)  
 udt\_send(clientSocket, (serverName, serverPort), sndpkt[i], corruptPercent\_client\_to\_server)

If the timer expires retransmit the base up to nextseq-1 and restart the timer

#send the packets corrupting some of them  
def udt\_send(sendingSocket,destination\_addr,packet,corruptPercent):  
 global lossPercent  
  
 randomNumC = random.randint(1, 100) #for corrupting  
 randomNumL = random.randint(0, 99) #for losing packets  
  
 # if the random number is less than corrupt percent corrupt the packet  
 if (randomNumC <= corruptPercent):  
 packet=coruptPacket(packet)  
  
 if(randomNumL<(100-lossPercent)):  
 sendingSocket.sendto(packet, destination\_addr)

Entirety of udt\_send. Used by the client and server to send packets, this function is responsible for corrupting files that are sent, using the passed corruptPercent, using a random value from random.ranint to determine what to corrupt. The packet is corrupted using the corruptPacket function, defined in functions.py. The function randomly decides whether or not to drop the packet as well. The function ends by sending the packet.

def udt\_rcv(recievingSocket):  
 while True:  
 #recieve the data as a byte array  
 data, addr = recievingSocket.recvfrom(2048) # buffer size is 2048 bytes  
  
 return data, addr

Entirety of udt\_rcv. Simple receives the data and returns it to the calling function.

sndpkt = make\_pkt(0, b'generic response')  
expected\_sequence\_Num=1  
def rdt\_rcv(recievingSocket):  
 global expected\_sequence\_Num,sndpkt  
 # get the data  
 flag=True  
  
 while(flag):  
 rcvPacket, addr = udt\_rcv(recievingSocket)  
  
 # extract the data  
 data, recieved\_sequence\_num, chksum = extract(rcvPacket)  
 if(printflag):print("recieved: ", recieved\_sequence\_num," expecting: ",expected\_sequence\_Num)  
  
 #if the data is not corrupt and it has the correct sequence number update the response  
 if( not corrupt(rcvPacket)and recieved\_sequence\_num==expected\_sequence\_Num):  
 sndpkt = make\_pkt(expected\_sequence\_Num, b'')  
 expected\_sequence\_Num += 1  
 flag=False  
 if(corrupt(rcvPacket)):  
 if(printflag):print("corrupt")  
  
 #respond to the data  
 if printflag: print("sent: ",expected\_sequence\_Num)  
 udt\_send(recievingSocket, addr, sndpkt, corruptPercent\_server\_to\_client)  
  
 return data, addr

entirety of rdt\_recieve. Receive a packet. If the packet is not corrupt and has the correct sequence then update the response packet and switch the flag so as to exit the main while loop. Then transmit the response packet. If we have exited the while loop then deliver the data.

def Timer():  
 if(printflag):print("Timer Expired")  
 global TimerExpired,fail  
 fail=0  
 TimerExpired=True  
  
timerThread= threading.Timer(timeout, Timer)  
  
  
#if a timer thread is running stop it, start another  
def starttimer():  
 global timerThread,TimerExpired,timeout  
 if(printflag):print("starting timer")  
 TimerExpired=False  
 timerThread.cancel()  
 timerThread = threading.Timer(timeout, Timer)  
 timerThread.start()  
  
  
#if a timer thread is running stop it  
def stopTimer():  
 global timerThread,timercount,TimerExpired  
 if(printflag):print("stop timer")  
 timerThread.cancel()  
 TimerExpired=False

entirety of timer functions. When starttimer() is called it will create a thread that will call the timer function after a set amount of time. If a timer thread is already running it will cancel that timer as well. Stoptimer() will just stop the timer thread if one is running. When the Timer() function is ever called it will changed the global TimerExpired variable to true. The timer Expired variable is used in rdt\_send as explained earlier.

**functions.py**

import random  
printflag=False  
#NOTE: Below values are in % (60 -> 60% chance of corruption or probablility of .6 to corrupt)  
corruptPercent\_client\_to\_server = 20  
corruptPercent\_server\_to\_client= 0  
lossPercent=0  
window = 3  
timeout=1e-6

Imports and control flags. Printflag controls debug output, corruptPercent controls the chance of corruption, as explained in the overview. Loss\_percent controls the chance for a packet to be dropped by the channel. Timeout controls how long in seconds the timer used in rdt\_send will wait before it times out

def make\_pkt(seq,data):  
 #packet format = seq,chksum,data  
 #Note: Sequence number 0 is encoded as 00000000 and Sequence number 1 is encoded as 11111111 (255)  
 seq = int.to\_bytes(seq, 2, "big")  
  
  
 #seq=int.to\_bytes(seq, 1, "big")  
 packet=seq+data  
 chksum = GetCheckSum(packet)  
 packet=chksum+packet  
  
 return packet

Entirety of make\_pkt. Concatenates the data to the sequence number. Then calculates checksum, and concatenates the checksum and packet. Finally returns packet. Note that header format is checksum as 1st and 2nd byte, sequence as 3rd and 4th byte, and data as 5th byte and up. ­

def extract(rcvpkt):  
  
 chksum = rcvpkt[0:2] #chksum is 1st and 2nd Byte  
  
 seq = rcvpkt[2:4] # sequence num should be 3rd and 4th Byte  
  
 seqinteger = int.from\_bytes(seq, "big")  
  
 pckt=rcvpkt[4:] #packet starts at the 5th Byte  
  
 return pckt, seqinteger, chksum

Entirety of extract packet. Extracts the checksum, data, and sequence number and returns them as separate values.

def GetCheckSum(packet):  
 chksum=0  
 currentIndex = 0  
 #go though packet grouping bytes in groups of 2  
 while(currentIndex < len(packet)):  
 byteslice=packet[currentIndex:currentIndex + 2]  
  
 #convert the byte slice to int  
 intslice=int.from\_bytes(byteslice, "big")  
  
 #add the integer to the pre existing checksum  
 chksum+=intslice  
  
 #if the chksum is greater than 65536 than subtrack 65535 (this is the decimal equivlent of  
 #getting rid of the 17th 1 in binary and the adding 1 on the lsb  
 if(chksum>=65536):  
 chksum-=65535  
 currentIndex += 2  
 # convert integer back to byte slice  
 chksum\_byte = int.to\_bytes(chksum, 2, "big")  
  
 return chksum\_byte

Entirety of GetCheckSum, which calculates checksum of a packet. Does this by slicing packets into 2-byte pieces, and adding them together. Returns checksum.

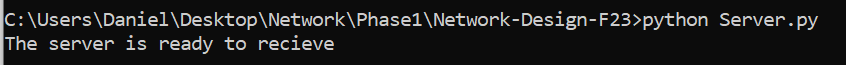
def coruptPacket(packet):  
 corruptpacket =packet+b'corrupt'  
 return corruptpacket

Packet corrupter. Corrupts packets by adding bytes to the packet.

#tells you if a packet is corrupt or not  
def corrupt(rcvPacket):  
 recieved\_chksum=rcvPacket[0:2] #checksum is the first two bits  
  
 packet=rcvPacket[2:] #take the chksum off of the front of the packet  
  
 calculated\_chksum=GetCheckSum(packet)#calculate the actual chksum of the packet so you can compare it  
 #to the recieved one  
  
 if(printflag): print(" recieved chksum: ",recieved\_chksum)  
 if(printflag): print(" calculated chksum: ", calculated\_chksum)  
  
 return (not(recieved\_chksum==calculated\_chksum))

Entirety of corrupt function, used to determine if a packet is corrupt. Done by extracting the checksum, then calculating the checksum for the packet, and comparing the results. Returns true if corrupt, false if not corrupt.

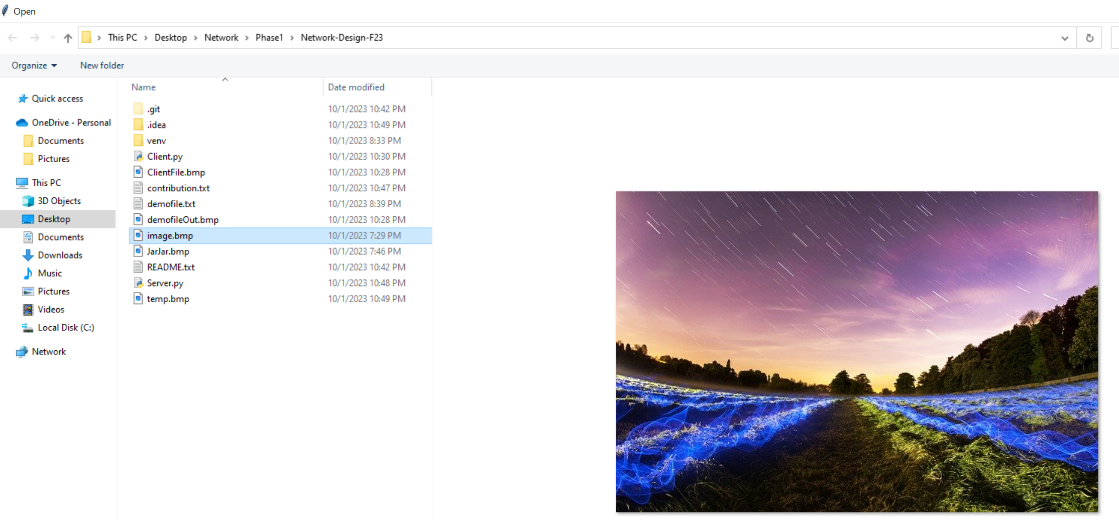
**Execution Example**



Command line command used to start the server code. In this case, the Server.py file is in the folder Network-Design-F23. In the above, you can also see the output provided by the server before the client is run.



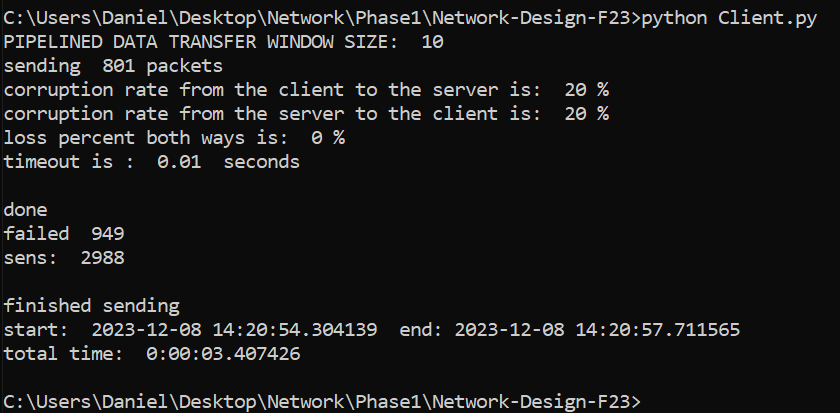
Command line command used to start the client code. In this case, the Client.py file is in the folder Network-Design-F23. Note after the above is run, a standard file select window will open



File Select window opened when client.py is run

As noted in the code explanation section, the output produced depends on two separate Boolean flags, printflag (found in functions.py) and runMultipleTests (found on line 98 of Client.py). As such, the different outputs for each combination of these Booleans are shown in this section. Note the information in this section above are not affected by these Boolean values.

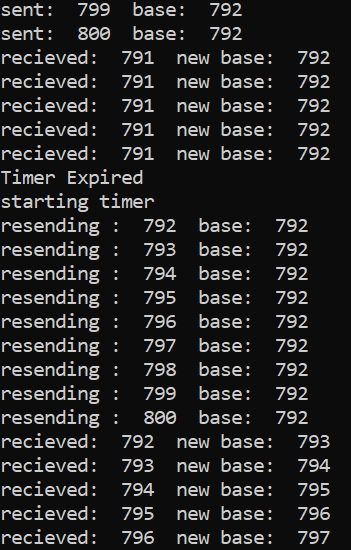
Below is the output produced by the client and server with both flags set to False (no output, only one run)



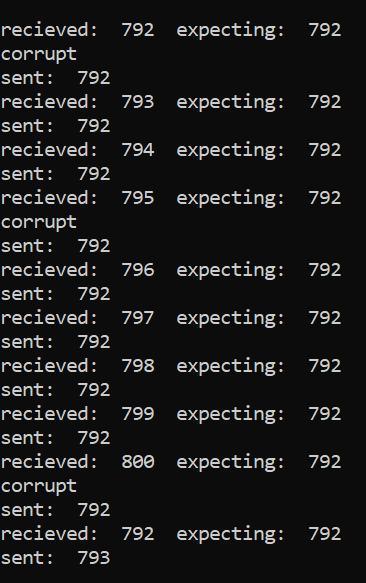
Output produced by the client with print flag set to False. Note the client to server and server to client corruption rate, loss rate, window size, and timeout is shown. Additionally, the start and end time, as well as the total time are printed at the end.



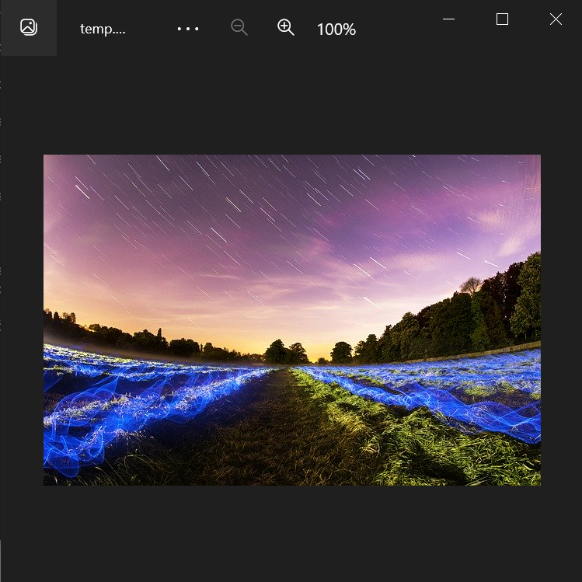
Output produced by the server. Note that even though the server remains running after the file is received, the server should be restarted before the client is run again.



Above is a section of output produced by the client with the print flag set to true. At the start of this snippet the base of the window is 792, with a window size of 10. As such, up to packet 801 can be sent. As can be seen in the above section of code, packet 800 is the last packet sent before the client begins to receive acknowledgements. However, the packet 792 was corrupted, resulting in the server responding with 791, or a Nack. This occurs until the timer expires, at which point the timer is restarted, and the packets are transmitted again, as can be seen in the output. The second time however, the packets arrive successfully, resulting in the client receiving packet 792, and adjusting the base upwards to 793, as seen towards the bottom of the image.



Above is the output produced by the server, which corresponds to the client output already examined. As can be seen in the first two lines, packet 792 arrives successfully, but is corrupted, resulting in a Nack to be sent. When 793 arrives, the server still wants 792, and as such again sends a nack. This repeats until the last two lines of the output, at which time the client timeout has occurred, and the client re-sent packet 792, which arrives successfully. At this point, the Ack is sent and the Server can continue waiting for the next packet.



Window opened by the server containing the received image.

**Performance Plots:**

Basic considerations:

* File size was 799kB
* All measurements taken on the same pc (Daniel Maccaline’s Desktop)
* All Times taken as an average over 10 runs
* All print commands that occur between the timer being started and stopped were disabled
  + The only print commands were the print statement indicating the program was starting and the time display at the end
* Note: After the results from Phase 5, Phase 4 results are included for comparison, as several of the Phase 5 analysis sections compare the results of Phase 5 to Phase 4

**Phase 5**

**Chart 1: Loss (0%-70%)**

Using a default window size of 10, timeout of .001 seconds (5ms)

Note loss in this table is both ways

|  |  |
| --- | --- |
| Loss chance | Transfer Time (s) |
| 0 | .274 |
| 5 | .625 |
| 10 | 1.27 |
| 15 | 1.96 |
| 20 | 2.81 |
| 25 | 4.27 |
| 30 | 4.80 |
| 35 | 6.70 |
| 40 | 7.56 |
| 45 | 10.03 |
| 50 | 11.21 |
| 55 | 14.77 |
| 60 | 17.00 |
| 65 | 23.39 |
| 70 | 26.07 |

**Chart 2: timeout 10ms to 100ms**

Using 20% loss, 20% corruption both ways. Window size is 10

|  |  |
| --- | --- |
| Timeout | Transfer Time (s) |
| 10 ms (.01 s) | 7.249212 |
| 20 ms (.02 s) | 13.57017 |
| 30 ms (.03 s) | 14.27938 |
| 40 ms (.04 s) | 21.62018 |
| 50 ms (.05 s) | 27.60207 |
| 60 ms (.06 s) | 29.56862 |
| 70 ms (.07 s) | 33.01379 |
| 80 ms (.08 s) | 43.65607 |
| 90 ms (.09 s) | 45.8975 |
| 100 ms (.1 s) | 48.30063 |

The results of the changing time is a linear increase in transmission time relative to the timeout value. This makes sense, as the number of timeouts that occur does not change, so the time taken increases linearly with the timeout length. With a slower computer or network, it may be the case that a timeout triggers too quickly, and causes un-necessary retransmission, however this is not the case here, as evident by the linear nature, and also due to the fact that chart 1 used a significantly lower timeout and had no issues with unnecessary re-transmission.

From this chart, the optimal timeout was 10 ms (.01 s). However it is likely an optimal time would be less than this value

**Chart 3: 20% loss, variable window size**

Using 20% loss, 20% corruption both ways, timeout of 10 ms

|  |  |
| --- | --- |
| Window size | Transfer Time (s) |
| 1 | 14.90 |
| 2 | 8.10 |
| 5 | 6.98 |
| 10 | 6.76 |
| 20 | 6.93 |
| 30 | 6.94 |
| 40 | 6.91 |
| 50 | 7.20 |

The results for this chart start with a drastic drop in time from a window size of 1 to 2. The reason for this is likely due to the cumulative acks adding redundancy. If packet 1 is received, but the ack 1 is dropped or corrupted, then packet 1 has to be retransmitted. However, with a window size of 2, if packets 1 and 2 are both received, ack 1 is corrupted or lost, but ack 2 gets back to the sender, then no re-transmission occurs. After this the time continues to decrease, however there are diminishing returns. This is likely due to the single PC nature of the network, which causes the acknowledgements to return before all the packets are sent, meaning the full window is likely never in flight at the same time for the larger window sizes

From this chart, the optimal window size is 10.

**Chart 4, Phase 3, 4, and 5, with optimal window and timeout (from chart 2 and 3)**

Phase 3:

Using 20% corruption chance in both directions:

Time for file transfer (P3): .455 s

Phase 4:

Using 20 % corruption chance in both directions, and .01 s timeout

Time for file transfer (P4): 6.453 s

Phase 5:

Using 20% corruption chance in both directions, a .01s timeout, and window size of 10

Time for file transfer (P5): 3.087 s

The results for this comparison are not entirely surprising. Phase 3 has a significantly lower time than Phase 4, due to Phase 3 not considering packet loss. As Phase 3 has not packet loss, the negative acknowledgement received from the server triggers an immediate re-transmission, with no timeouts. Phase 4 on the other hand always waits for either an Ack or a timeout, and as such takes much longer to retransmit when a retransmission is necessary, and can also retransmit too early and send more information than may be necessary.

Phase 5 is faster than phase 4, due to the redundancy added by the cumulative acks significantly reducing the number of retransmissions that occur due to lost or corrupt acks. Additionally phase 5 can multiply packets in the case that the ack takes longer to return. However, it is still slower than phase 3, because phase 5 still has to wait for a timeout in the case that the file is corrupted on the way to the server. Additionally, the main advantage of Phase 5, being able to have multiple packets in flight, is reduced due to the simulated network being nearly instant on a single pc, such as the one these tests were run on. Over an actual network, the one-by-one packet nature would slow phase 3 down and potentially make it slower than phase 5.

Regarding the window size and timeout for phase 5. For this particular pc and network, a window size of 10 and timeout of .01 seconds was the best performers in as shown in the previous charts. However, in an actual network where packet transmission takes more time, a larger window size would likely improve performance, and require a longer timeout. As such, these results do demonstrate the importance of adjusting these values based on the state of the network.