Design Document 10/30/2023

Go-Back-N

Daniel Maccaline, Nathan Grady

Project Phase 5

**Phase purpose:**

The purpose of this phase is to implement the go-back-n protocol, to create a system that can transfer files more efficiently than RDT over a channel that can drop or corrupt packets

**Code explanation**

**Overview:**

Two Boolean control variables 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 remaining two values are found in Client.py, and are used to have the client send multiple files for testing the time taken. The flag runMultipleTests enables the client to re-send the file, and iterations is the number of times the file is sent. Once complete, the average time for each run is printed to the terminal. See the execution example below for example outputs.

**Client.py**

**Note: Client.py is unchanged from phase 3**

*"""  
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 = []  
  
 while(currentIndex < len(data)):  
  
 #extract just the data from the packet  
 bytesdata=data[currentIndex:currentIndex + packetsize]  
  
 packet.append(bytesdata)  
  
 currentIndex += packetsize  
  
 return packet

Get\_Packets\_Raw functions. Splits the file data into a list of equally sized packets, each of a length specified by packetsize. 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("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()  
  
 start\_time = datetime.datetime.now()  
  
 for i in range (0,len(data)):  
 rdt\_send(clientSocket, serverName, serverPort, data[i])  
 if(printflag): print("sending packet number ",i)  
  
 #send stop bit  
 rdt\_send(clientSocket, serverName, serverPort, b'stop')  
  
 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")

Ty to send the data to the server. First prints the messages indicating the loss chance, then records a start time. The function then loops over every packet in the data variable, and calls rdt\_send (function found in send\_recieve.py, discussed later), followed by a packet indicating to the server that the file transfer is complete. 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 3**

*"""  
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 timerCall():  
 if(printflag): print("Timer expired")  
 global timerExpired  
 timerExpired = True

This is the main timer callback. It flips the global Boolean timerExpired to True to indicate the timer has finished, and prints output if output is enabled. All the remaining timer functionality is in rdt\_send

#using this to track what sequnce number the rdt\_send function attaches to its packet  
sequenceNum=0  
def rdt\_send(clientSocket,serverName,serverPort,data):  
 global sequenceNum  
 global timerExpired  
 flag=True

#make the packet  
 if(printflag): print("Sending packet")  
 sendpkt = make\_pkt(sequenceNum,data)  
#

Beginning of rdt\_send, used by client. sequenceNum is initially 0, and tracks which sequence is sent and expected (0 or 1). Flag is initialized to true, to be used in a loop in the next code block. Sendpkt is initialized to the packet that is to be sent. This packet contains the sequence number and checksum in the header, and is made by make\_pkt, defined in functions.py. The global bool timerExpired is used for the timer interrupt. Additionally this code segment uses make\_pkt (defined in functions.py) to add the checksum and sequence number to the data.

#  
 #Loop until packet recieved (properly acked)  
 goodAck = False  
 while(flag):  
 #udt send packet  
 udt\_send(clientSocket,(serverName,serverPort),sendpkt,corruptPercent\_client\_to\_server,loss\_Percent\_client)  
  
 #Set client socket so recvfrom is not blocking  
 clientSocket.setblocking(0)

NOTE: above udt\_send should be indented. It is indented on the code, but word formatting moves it to the left.

This code block is the beginning of the main sending loop. It will loop until the packet is sent and an appropriate ACK is received, as controlled by the goodAck Boolean. The loop first sends the packet with udt\_send, defined later in the file, which corrupts or drops a percentage of files defined by corruptPercent\_client\_to\_server and loss\_percent\_client. The code block also sets the clientSocket setblocking to 0. This causes the rcvfrom command, which is usually a blocking command that will halt a program until a message is received, to instead return an error message if no data is in the socket. This is required in the case of an ACK being dropped, as otherwise this would cause the program to become stuck.

#Start timer  
 timerExpired = False  
 #arguments (x, f), after x seconds, call function f  
 timer = threading.Timer(.001, timerCall)  
  
 #Start timer  
 timer.start()

This block of code starts the timer. First it sets the global control Boolean timerExpired to false, then calls threading.timer. This starts an asynchronous timer which will call the function timerCall after .001 second or 1 millisecond. This is the timeout time for the client in this case.

#  
 #Loop until timer expires  
 while(not timerExpired):  
  
 # Attempt to read from socket. NOTE: With setvlocking(0) set, will throw error if nothing is ready to recieve  
 try:  
 rcvpkt, addr = clientSocket.recvfrom(2048)  
  
 #Packet recieved, extract dadta and check if a good ack  
 data, recieved\_sequence\_num, chksum = extract(rcvpkt)  
 if(printflag): print(" sent sequnce num: ", sequenceNum)  
 if(printflag): print(" recivied seq: ", recieved\_sequence\_num)

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. These are all printed if printFlag is true.

#  
 #if not corrupt and correct sequence number, stop timer, and break from loop (packet successfully sent and acked)  
 if(not (corrupt(rcvpkt)or (not (recieved\_sequence\_num==sequenceNum)))):  
 if(printflag): print("Good ack")  
 #Stop timer and change flags when good ack recieved  
 goodAck = True  
 flag = False  
 timer.cancel()  
  
 #packet send, end function  
 if(printflag): print()  
  
 sequenceNum = (sequenceNum + 1) % 2  
 return  
 else:  
 if(printflag): print("corrupt")

This if else block occurs inside the try of the previous block. This checks if the packet is not corrupt and has the correct sequence number. Note corrupt() is defined in functions.py, and returns true if the packet is corrupt. If the packet is not corrupt and the sequence number is correct, the timer is stopped with timer.cancel, and the flag goodAck is set to true to exit the loop. The sequence number is also updated to equal the next expected sequence number, and the function returns, causing the code to move on to sending the next packet. If the packet is corrupt or has the wrong sequence number, the code prints corrupt if printflag is true. Corrupt packets and back ACKS do not do anything else, and are essentially ignored.

#   
 except:  
 if(printflag and False):  
 print("Waiting for response")  
 #Nothing recieved, do nothing (loop repeats, to try waiting for data again or for timer expiration)

Above is the except block. Note that this only contains an output with and False in the conditional. This is for testing, and does nothing unless manually changed. After this, the code goes back to the start of the loop. If timerExpired is still equal to false, the code looks for input on the socket again. If timerExpired is true, then the timer has expired and the code goes back to the first while loop, re-sending the packet and re-starting the timer.

#send the packets corrupting some of them  
def udt\_send(sendingSocket,destination\_addr,packet,corruptPercent):  
  
 randomNum = random.randint(1, 100)  
  
 # if the random number is less than corrupt percent corrupt the packet  
 if (randomNum <= corruptPercent):  
 packet=coruptPacket(packet)  
  
 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 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.

expected\_sequence\_Num=0  
sndpkt = make\_pkt(1, b'generic response')  
def rdt\_rcv(recievingSocket):  
 global expected\_sequence\_Num  
 global sndpkt  
 flag=True

Beginning of the rdt\_rcv function, which is used by the server to receive packets. Note that the sequence number is initialized here, and separate from the clients initial sequence number. Additionally, a default packet is also created using make\_pkt. This will be explained later in the function.

#true  
 while(flag):  
  
 flag = False  
  
 #get the data  
 rcvPacket, addr=udt\_rcv(recievingSocket)  
  
 # extract the data  
 data,recieved\_sequence\_num,chksum = extract(rcvPacket)  
  
 if(printflag): print(" expecting sequnce num: ",expected\_sequence\_Num)  
 if(printflag): print(" recivied seq: ",recieved\_sequence\_num)

Beginning of loop used to receive packet. Loops until the flag is false. Note that the flag is set to false at the start of the loop, but swapped back to true if the packet received is corrupt or the incorrect sequence number. This part of the loop receives a packet using udt\_rcv (defined later in this file) and extracts the data using the extract function (defined in functions.py).

#Test  
 #for now just read as 'if bad packet' bad=corrupt or wrong sequnce number  
 if(corrupt(rcvPacket) or (not (recieved\_sequence\_num==expected\_sequence\_Num))):  
 #keep previous response do the loop again  
 if(printflag): print(" corrupt")  
 flag=True  
 else:  
 #make good response, exit loop  
 sndpkt=make\_pkt(expected\_sequence\_Num,b'')

If the package is corrupt, as determined by the corrupt function (defined in functions.py, returns true if corrupt or false if not corrupt) or the sequence number does not matches the expected sequence number, set the flag to true to stay in the loop. If not corrupt and the sequence number matches, override sndpkt with a positive acknowledgement, created with make\_pkt.

# reply to the data with either "good" repsonse or the previous response  
 udt\_send(recievingSocket, addr, sndpkt,corruptPercent\_server\_to\_client)

Send acknowledgement. Always sends what is stored in sndpkt. If the received packet was good, the previous code block set sndpkt to acknowledge with the current sequence number. If the received packet was bad, then it sends what was stored in sndpkt previously, which is an acknowledgement with the previous sequence number.

#deliver the data  
return data, addr

Deliver data back to server function.

**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 = 0  
corruptPercent\_server\_to\_client= 0  
loss\_Percent\_server=10  
loss\_Percent\_client=10

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.

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)  
 if(seq == 1):  
 seq = int.to\_bytes(255, 1, "big")  
 else:  
 seq = int.to\_bytes(seq, 1, "big")  
  
 #seq=int.to\_bytes(seq, 1, "big")  
 packet=seq+data  
 chksum = GetCheckSum(packet)  
 packet=chksum+packet  
  
 return packet

Entirety of make\_pkt. First overrides the sequence number “1” to 255. This is so the 1 is sent as 11111111 instead of 00000001 (adds redundancy, as need 8 simultaneous bitflips to swap from 1 to 0 or 0 to 1 with this system). Then 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 byte, and data as 4th byte and up.

def extract(rcvpkt):  
  
 chksum = rcvpkt[0:2] #chksum is 1st and 2nd Byte  
  
 seq = rcvpkt[2:3] # sequence num should be 3rd Byte  
  
 seqinteger = int.from\_bytes(seq, "big")  
  
 if(seqinteger == 255):  
 seqinteger = 1  
  
 pckt=rcvpkt[3:] #packet starts at the 4th Byte  
  
 return pckt, seqinteger, chksum

Entirety of extract packet. Extracts the checksum, data, and sequence number and returns them as separate values. Note it also changes 255 back to 1 for the sequence number.

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.

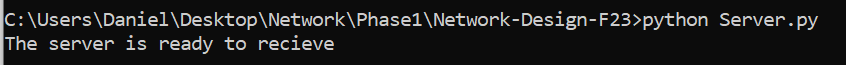
def coruptPacket(packet):  
 corruptpacket =packet+b'corrupt'  
 return corruptpacket

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.

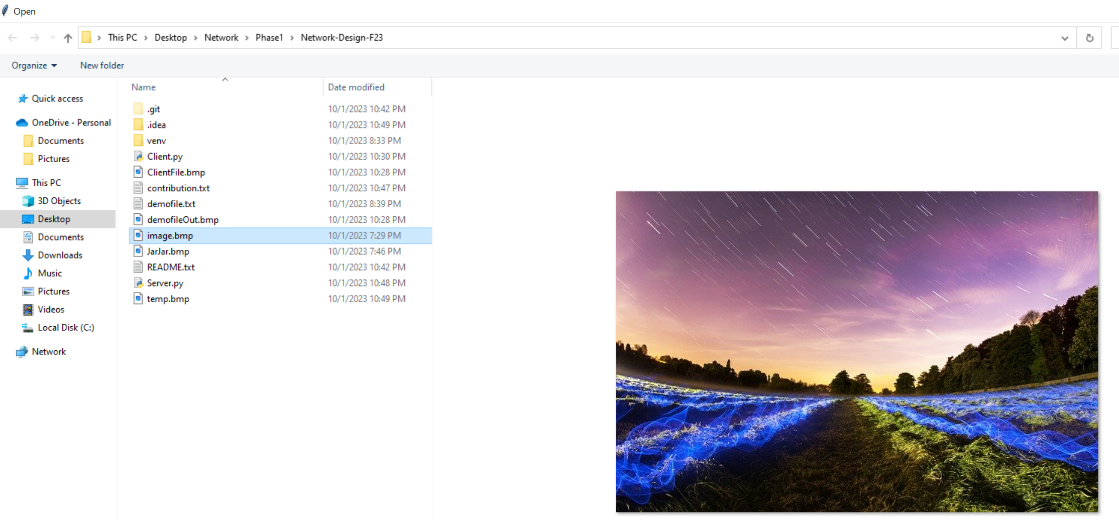
**Execution Example**



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.



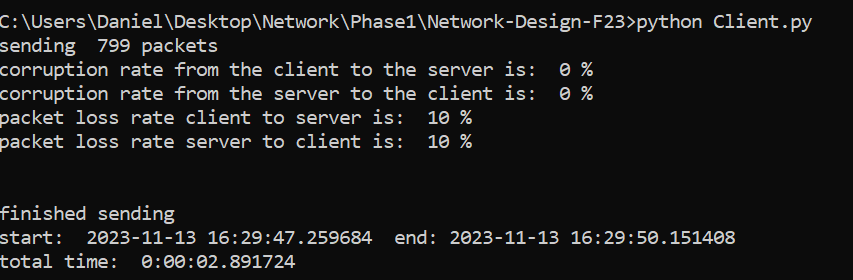
Command line command used to start the client code. In this case, the Client.py file is in the folder Network-Design-F23. Note after the above is run, a standard file select window will open



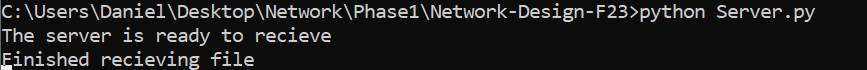
File Select window opened when client.py is run

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)

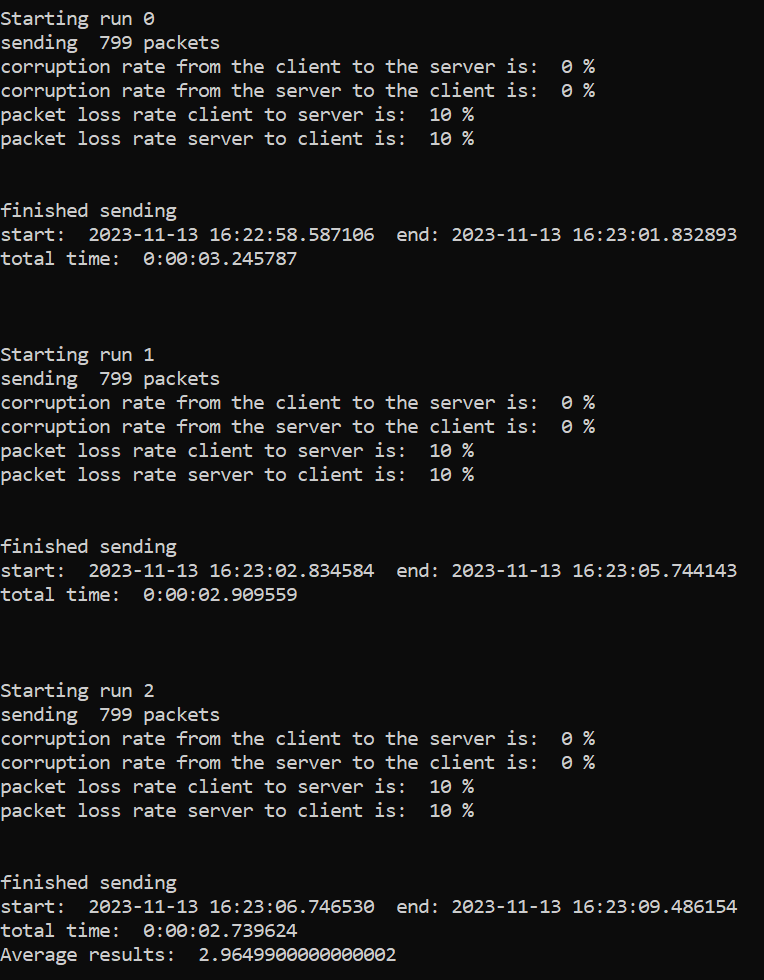


Output produced by the client with both flags set to false. Note the client to server and server to client corruption rate is shown. Additionally, the start and end time, as well as the total time are printed at the end.



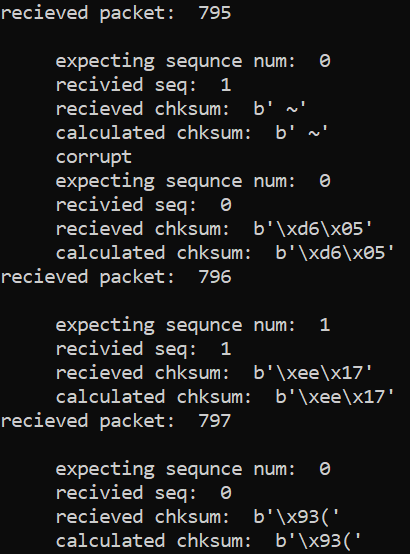
Output produced by the server

Below is the output produced by the client with the runMultipleTests flag set to true, with iterations set to 3 (3 runs are performed then averaged). Note that the output of the server is unchanged between each runs, and as such the server produces the same output as if the client were run multiple times, and as such is not shown below.

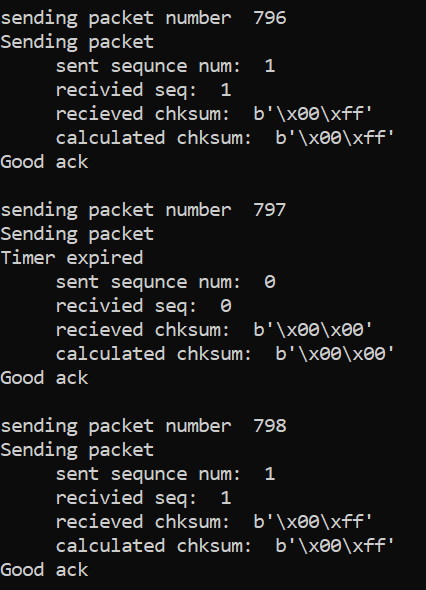


Output produced by the client when running multiple runs. Note the only changes are the run # being printed out before the run, and the average across all the runs printed at the end of all the runs.

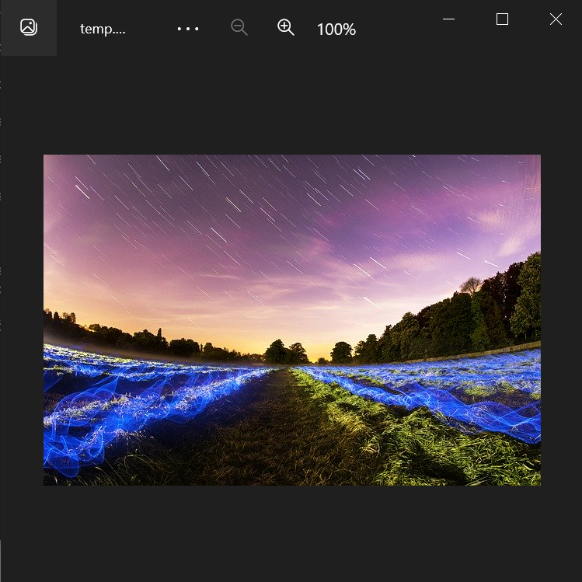
Below is the output produced by the client and server when the print flag is true. For the below output the runMultipleTests flag is false.



Excerpt of output produced by the Server with print True. Note that after receiving packet 795, the server receives a packet with an incorrect sequence number. This indicates the Acknowledgement for packet 795 was dropped, causing the client to re-send the packet, resulting in the server receiving a packet with the incorrect sequence number. This packet is listed as “corrupt,” due to the sequence number being incorrect. After this the next packet is the correct sequence number, indicating the client received the new ack 1, allowing the process to continue.



Output produced by the Client with print True. Notice the “Timer expired” line under packet 797. This indicates that either no ack was received, or a negative acknowledgement was received. As the corrupt chance in this run was 0%, either the packet or ack can be assumed to have been dropped. The timer expired line indicates that a duplicate packet 979 was sent. As can be seen by the results below this, the packet was correctly sent, and the proper acknowledgement was received for the re-sent 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

**Phase 5**

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

Using a default window size of 10, timeout of 1 ms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Chance of corruption/loss chance | Data Corruption | Data Loss | Ack Corruption | Ack Loss |
| 0 | .2789 s | .2789 s | .2749 s | .2789 s |
| 5 | .7383 s | .7563 | .2822 s | .2942 s |
| 10 | 1.349 s | 1.493 s | .2876 s | .2743 s |
| 15 | 2.071 s | 2.271 s | .3131 s | .3025 s |
| 20 | 2.988 s | 3.031 s | .3430 s | .3319 s |
| 25 | 4.079 s | 3.962 s | .3938 s | .4012 s |
| 30 | 5.38 s | 5.323 s | .4494 s | .4398 s |
| 35 | 6.916 s | 7.008 s | .4785 s | .4734 s |
| 40 | 8.336 s | 8.106 s | .5855 s | .5752 s |
| 45 | 10.29 s | 10.47 s | .6859 s | .6733 s |
| 50 | 11.76 s | 12.99 s | 1.131 s | 1.034 s |
| 55 | 14.85 s | 15.46 s | 1.509 s | 1.514 s |
| 60 | 18.80 s | 16.47 s | 1.792 s | 1.801 s |
| 65 | 22.50 s | 22.55 s | 2.253 s | 2.261 s |
| 70 | 29.71 s | 28.57 s | 3.323 s | 3.353 s |

Note that in the above chart, Ack Corruption is behind the Ack loss, but cannot be seen well due to them overlapping. As can be seen above, the data corruption and data loss lines math very closely, while the ack corruption and ack loss lines also line up. This is because the protocol does not respond to duplicate Nacks, and as such the sender will wait for timeouts regardless of whether the packet is lost or corrupted. Additionally, the Acks are significantly less impacted by loss or corruption. This is because the duplicate ack adds redundancy. For example, if Data 3 is lost, then the packet has to be resent. However if Ack 3 is lost, as long as a future ack such as ack 4 arrives, then ack 3 does not need to be re-sent. As such, the lines for corrupted or lost acks increases much more slowly than the line for corrupted or lost data packets.

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

Window size is 10

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Timeout | No Loss/Corruption | 20% Data Corruption | 20% Data Loss | 20% Ack Corruption | 20% Ack Loss |
| 10 ms (.01 s) | .3077 | 2.899 | 3.391 | .3042 | .2757 |
| 20 ms (.02 s) | .2847 | 5.246 | 5.389 | .2827 | .2832 |
| 30 ms (.03 s) | .2942 | 6.917 | 6.591 | .2847 | .2687 |
| 40 ms (.04 s) | .2953 | 9.635 | 9.224 | .2782 | .2757 |
| 50 ms (.05 s) | .2941 | 10.23 | 10.41 | .2822 | .2717 |
| 60 ms (.06 s) | .2962 | 11.87 | 12.24 | .2787 | .2762 |
| 70 ms (.07 s) | .3011 | 14.203 | 15.51 | .2772 | .2777 |
| 80 ms (.08 s) | .2814 | 18.64 | 18.71 | .5214 | .2807 |
| 90 ms (.09 s) | .2914 | 21.21 | 22.45 | .2767 | .2821 |
| 100 ms (.1 s) | .2932 | 24.76 | 24.77 | .3107 | .2912 |

The effect of a varying timeout is a linear increase for the data packet loss or corruption, This result makes sense, as if the data packet is lost or corrupted it must be re-sent, and with no re-transmission based on duplicate acks, the sender will wait for a timeout to occur before transmission. As such, with a slower timeout, the sender takes longer to recover from data errors.

The effect of timeout when Ack loss is implemented is similar to the first chart. This is again due to the increased redundancy for ack losses in the protocol preventing a heavy impact from occurring when acks are lost.

With a slower computer or network, it may be the case that a timeout triggers too quickly, and causes un-necessary retransmission, which would cause the line too start going up on the left side. 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 | No Loss/Corruption | 20% Data Corruption | 20% Data Loss | 20% Ack Corruption | 20% Ack Loss |
| 1 | .4113 s | 3.223 s | 2.846 s | 3.195 s | 2.879 s |
| 2 | .2827 s | 2.952 s | 3.006 s | .5499 s | .5615 s |
| 5 | .2912 s | 3.11 s | 3.048 s | .2997 s | .3148 s |
| 10 | .3087 s | 2.983 s | 3.121 s | .3087 s | .3203 s |
| 20 | .3374 s | 2.796 s | 3.055 s | .3348 s | .3223 s |
| 30 | .3848 s | 2.893 s | 3.114 s | .3968 s | .3348 s |
| 40 | .4454 s | 3.090 s | 3.121 s | .4348 s | .3698 s |
| 50 | .5124 s | 2.865 s | 2.985 s | .4829 s | .3963 s |

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. On a slower network, the optimal window size would likely be larger.

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.