**CS 371 Final Report**

By: James Birch, Luis Contreras, and Nick Yusupov

**Abstract:**

This report features an introduction into what this report is about, which is building a basic network featuring a server and two clients. It includes an introduction that goes into more details about the project, along with a project design and implementation tab featuring pseudocode, descriptions, and rationales behind our design decisions for the project. We also include what experimentation we did along with screenshotted evidence of our network in operation, and then conclude the details towards the end of the report.

**INTRODUCTION:**

This project is the design and implementation of a server-client model in which clients can ask and receive data from either the server or other clients. It was written in python and makes use of socket programming in forming connections and facilitating data transfer between machines, and we made use of a series of acknowledgements between all machines in the network to accomplish this.

In our client-server model, the server’s responsibility is facilitating data transfer between all nodes of the network, either from itself or other clients on the server. There are different implementations for how it facilitates file transfer, with us including four different configurations for file transfer as per the assignment document. The client’s responsibility is just to request files for the most part, but it also includes a thread communicated to the server which can be used to receive any requests from the server. The client also takes into account the server’s different configurations when it forms a connection, allowing the client to dynamically change its configuration based on what the server is using, so long as it is one of the four configurations coded for in the client and server.

The motive behind the project is to test different forms of connections, to see how the distance between client and server would affect the speed of downloads between weird and wireless connections. The following report will explore our findings in the project

**DESIGN AND IMPLEMENTATIONS:**

The Implementation of the Solution:

The Implementation of the project was built by the idea of using threads to send and receive information between servers and client, There are 2 sections to the client, The main thread that communicates with the user and has domain over the server’s dedicated thread and the SR request wich is an open thread that allows the server to request information from the client without interrupting the client’s communication with the server this implementation allows for connection without interruptionsDiagram

Description automatically generated

From the picture to the Left we can see a representation of the connections that are implemented in our solution, the black lines represent creation but hold no communication the Green Lines Represent communication during connections available (as long as the client is connected)

**Server pseudocode**

The Server will loop with instructions:

* if the instruction is “Start”  it will open a thread (going to Transmit) with an available connection  for the client to connect to, you must run this command for however many Clients you want to  connect with
* If the instruction is “Close” it will start shutting down the server (You can not send new instructions), when closing the server it will wait for all the open connections to terminate then it will shut down.
* If the instruction is “print” it will request what you want to print, whether it is the file list, or all of the files in the clients connected to the server
* The instruction “Test” is available however this is a tester function that by default prints how many connections are in the server

    main:

    Command = ""#clear commands

    threadList = list() #Create empty list

    while (Command != "Disconect"):

        Command = input("Server Command: ")

        if (Command == "Start"):

            Create new thread, add it to list and start it on Transmit fn()

        elif (Command == "Close"):

Command = Disconnect

        elif (Command == "Print"):

            go to PrintComand with whatever you wanted to print

        elif (Command == "Test"):

            Open a thread to Test FN

        else:

            Print Invalid command (also go to Invalid)

    Join all threads

    print("The server is now a Graveyard!") (The code has ended)

**The Server** has a total of **13** helper functions, in interest of time there will be a quick summary of what the functions do, what they return and what they call:

**Transmit**:

This function is the one that the Thread coming from the main function of Server, this function will wait indefinitely until a connection to the client is made, then it will request a list of files from the client that is connected to, after it is done it will give control to the client, depending on the commands it will do something different:

* CMD = “END\_” it will call KillSR() then it will confirm the connection closure with the client that it is connected then it will close the connection and the thread will close
* CMD = “GET\_” it will receive the files the client wants from the server, and then it will call sendWithSenarios()
* CMD = “PSH\_” it will send “Pushing to server” to client then call receive()
* CMD is something else, it will print that the command is not found, command and any other data attached to the command along with the connection’s name

Transmit: args : Id, IP, Port

    print("S1, Thread", Id, "| IP", Ip, "| Port", Port)

    Await Connection (to connect use ^^^)

    Turn files in  dir and files in the client’s files into lists

    Handover control to client

    while (Connected):

        Receive (CMD\_”THINGS”)

        if (CMD == 'END\_'):

Call Kill SR

Disconect

        elif (CMD == 'GET\_'):

            Call sendWithSenarios with (“Things”)

        elif (CMD == 'PSH\_'):

            Push data to server

        else:

            Print “Comand unknown”

            time.sleep(4) # so that you can read it

      End of thread

**sendWithSenarios**:

This function is responsible for sending data to the clients (by extension downloading from other clients to push to clients), this function will clean the names attached to the command received by calling find() and depending on the type of download (Can be set up by changing constant “TYPE\_OF\_DOWNLOAD ”) to ether 1, 2, 3, or 4, each corresponding to their own scenario send the data. Depending on the scenario it will go into its respecting function, for example: TYPE\_OF\_DOWNLOAD == “1” then it will go to S1( )

Sendwithsenarios: args: client, fileARR

Clean FileArr and turn it into a list

    if (TYPE\_OF\_DOWNLOAD == "1"):

        #Communication

        S1(client,FileList)

    elif (TYPE\_OF\_DOWNLOAD == "2"):

        #Merge

        S2(client,FileList)

    elif (TYPE\_OF\_DOWNLOAD == "3"):

        #def (back to back after all are aquired)

        S3(client,FileList)

    elif (TYPE\_OF\_DOWNLOAD == "4"):

        #def (back to back after all are aquired)

        Call S4(Cleint,f ilelist)

    elif (TYPE\_OF\_DOWNLOAD == "0"):

        Default call S3(client,FileList)

**S1**:

This is Scenario 1, it will send the clients the files currently on the server then make the client wait until all the files from the other clients are downloaded and then it will continue to send to the server

 Send files to client - > send client “Wait” -> call SRGET() - > send “Continue” to client - > send files- > send “done”

**S2**:

This is Senario 2, it will make the client wait while the server verifies that it has every file needed and then merge them into a file, after the file has been created it will send it to the Client, after the client has received the file the server will delete the merged file and send Done since it is done sending

Send to client “wait” –> SRGET –> create merged file –> send file –> delete file –> send ”done”

**S3**:

This is Senario 3, it will make the client wait until all the Files are in the server and then it will send all of the files to the client after it is done it will send done

Send to client “wait” -> SRGET ->  send files -> delete file - > send ”done”

**S4**:

This is our 4th scenario wich is just the client asking for the files 1 by 1 on a loop, receives file name then asks the client if it needs more files, then if it is it stops otherwise it keeps sending

–> get file → send to client – > if needs more files –>  end

  ^   ← ←            ←

**SRGET**:

This function will search through the file lists received by the clients to get the data, The thread accomplishes this by using a for loop and iterating through the clients for what it needs, it closes it’s connections every time it needs a new connection, it only breaks out of the loop if it encounters the server as a location.

Setup first conn -> for { if server break; → if prev and new are different make new connection →  receive the files → setup new and prev → if new and prev differ close connection →} → if there is still a connection open, close it

SRGET, args: file ARR

    AddrP=("",int(0))

    AddrN = (fille arr 1 name)

    idx = 0

    for i in range(0, len(fileArr)):

        if(fileArr[i] name == "Server"):

            AddrP = AddrN

            AddrN = Next fileArr item

            break;

        if(AddrN != AddrP):

            Form New connection

        CurrFile = fileArr name

        Send wanted file

        Call Receive

        AddrP = AddrN

        AddrN = Next item in array

        if(data == "Done"):

            print("Done with ", CurrFile)

        if((AddrN != AddrP) or (i == (len(fileArr)-1))):

            Disconnect from old connection

    print("P ",AddrP,"N", AddrN)

    if((AddrN != AddrP)):

Disconnect from old connection

**KillSR**:

This closes the client’s second thread by connecting to mentioned thread and sending it a kill code

Get addres of client → connect to client →  send kill code → confirm → close connection

**receive**:

This function just uses the filename given and creates a new file with that name, then it loops and receives the information from the client until it receives the terminating command, the server, once it is done it closes the file

Open filename → Write data received → Close file

**GetFileNameList**:

Same as find but the output looks like [“client\_”{ip}”-”{port}”.filelist”, {idx of File}]’

**find:**

Creates an array with the filenames and then it Searches for them in the file lists and in the server then sorts them by IP, this means that all of the ones on server will be last. Then returns an array of the form [[String(IP), int(Port), String(Filename)], …, [String(IP), int(Port), String(Filename)]]

Turn string to array → use regex to find the client lists → search for the items in clients → Search in server dir → format array → return Array

**send**:

This function sends data of a file in the server, it accomplishes the send by sending the file name first and then sending the data with a tag to differentiate them,

Send filename → {send data until no more data} → send “done sending”

**PrintCommand**:

Just prints the command you requested on main (Not important)

**Invalid**:

Prints that you imputed an invalid command and it prints the Available options (Not important)

**Client pseudocode:**

**Uptimes**

**Code**

UPtimes(files, start, end): # Function for handling the calculation of upload times

    send = end - start # Calculate the time it takes between the start and end of an upload

    sz = 0 # total size of the files upload

    for i in files: # iterate through the files list

        fsize = os.path.getsize(i) # get the size of the file that we retrieved

        upload\_times[i] = send # Insert into a dictionary for upload times

        upload\_speed[i] = (fsize / send) # Insert into a dictionary for upload speed

        sz = sz + fsize # Size of all files together

    # Print the details of the upload

    print("Rate of transfer for files:", files, "is", sz, "bytes per second over", send, "seconds.")

**rationale**

This is what we used for calculating the upload time of files, and we decided to do the average time and rate across the start and end points in our function for consistency.

**DNtimes**

**Code**

DNTimes(files, start, end): # Function for handling the calculation of download times

    send = end - start # Calculate the time it takes between the start and end of a download

    sz = 0 # total size of the files download

    for i in files:  # iterate through the files list

        fsize = os.path.getsize(i) # get the size of the file that we retrieved

        download\_times[i] = send # Insert into a dictionary for download times

        download\_speed[i] = (fsize / send) # Calculate and insert into a dictionary for download speed

        sz = sz + fsize # Size of all files together

    # Print details for the download.

    print("Rate of transfer for files:", files, "is", sz, "bytes per second over", send, "seconds.")

**Rationale**

This is what we used for calculating the download time of files, and we decided to do the average time and rate across the start and end points in our function for consistency.

**Merged Split**

**Code**

# filename is defined on an agreed upon merge text file

merge\_split(filename):

file = open(filename, ‘r’) —> Open the received merged file in read only

while filesplit is false: # While the file is not split.

for line in filename: # For each line in the merged file

i ← i.split(“\n”) # split by \n character, as each line ends with \n

i ← i.pop(0) # get the data from the previous split

if i[0:4] == “<T>\_”: # Check the first four lines of data

write to i[4:BUFF\_SIZE] file # Start writing a file

if i[0:4] == “<D>\_”: # Check the first four lines of data

line = i[4:SIZE] + “\n” # line to write the data line to the file

new\_file.write(line) # Writes line to file

if i[0:4] == “\n”: # empty spaces are interpreted as \n at end of file

new\_file.write(“\n”) #write a new line character

new\_file.close()  #close the file

filesplit = true # break out of loop

**Rationale**

We designed this function this way to split our merged.txt file as we merge it at the server level

**upload**

**code**

upload(client, cmd, filename): # Upload function, takes client connection, a cmd, and filename

    client.send(cmd.encode(FORMAT)) # Send CMD to the client

    print("Waiting for ACK") # We are waiting for a response back

    msg = client.recv(SIZE).decode(FORMAT) # Response from server

    print(f"[SERVER]: {msg}") # Print the response from the server

    if filename in recDict: # Add +1 to the file upload count in our dictionary for our files

        recDict[filename] += 1 # add to count

    else:

        recDict[filename] = 1 # add to dictionary and set it to one.

    if (msg != ''): # So long as the server is not sending nothing

        file = open(filename, "r") # open file for upload

        data = file.read(SIZE-4) # read in initial piece of data

        print(data) # print the data we got for validation

        while data != "": # while we have data to send

            cmd = "SED\_" + data # build send data to the server, SED\_ is a codeword for this

            client.send(cmd.encode(FORMAT)) # send the data with the tag

            data = file.read(SIZE-4) # read in more of the file

        print("Waiting for ACK") # when we finish and server receives end of our file

        msg = client.recv(SIZE).decode(FORMAT) # decode the message from server

        print(f"[SERVER]: {msg}") # print out the server message

        if (msg != ''): # so long as message isn’t nothing, or get an acknowledgement

            file.close() # close the file

            end = time.time() # get the end time point

            client.send(("SOV\_connect").encode(FORMAT)) # send to the server that we’re done

**rationale**

We built out upload as such to upload files to the way our server is designed to receive files

**download**

**code**

download(client, filename): # Call download, give connection to server and filename(s)

    cmd = filename # cmd is filename and a command, built to communicate with server

    client.send(cmd.encode(FORMAT)) # send cmd to the server

    data = client.recv(SIZE).decode(FORMAT) # receive filename and command from server

    print("File name Recieved", data) # print filename we are going to build from server

    filename= filename[4:len(filename)] # get the filename from the received command

    print(filename) # print filename for validation

    filename.split() # split in case there are multiple

    print(filename) # print filename once more for validation

    if data in receivedDict: # add it to files we downloaded dictionary

        receivedDict[filename] += 1 # add one if in dictionary

    else:

        receivedDict[filename] = 1 # add it to dictionary with key value of 1

    while (data[0:4] != "DON\_"): # while server is not done sending

        data = client.recv(SIZE).decode(FORMAT) # receive data

        if (data[0:4] == "SNF\_"): # if the data is SNF, codeword for send

            print("Open File") # validate we are in loop

            f = open(data[4:SIZE], "w") # start writing a file

            start = time.time() # start the timer for file download

            print("file opened", data[4:SIZE]) # validate we have opened a file

            data = '' # data is not yet started, so set to blank.

            while data[0:4] != "SOV\_": # while the server is not sending file data

                print("dowloading...") # validate and output we are downloading

                data = client.recv(SIZE).decode(FORMAT) # draw in data

                data2 = data[4:SIZE] # grab actual data as a separate unit

                if data[0:4] == "SOV\_": # check the codeword

                   data2 = '' # if it is SOV, or server not sending, we ignore any data to write

                print('datalin=', (data[0:4])) # print the codeword

                print('data=', (data2)) # print the data we will have written

                f.write(data2) # write data to the file

            f.close() #close the file after we received the data

            print("File downloaded!") # confirm we printed

            client.send(("Done").encode(FORMAT)) # send to server we are done

        if (data[0:4] == "WIT\_"): # if code is WIT, we need to wait for files to get onto server

            while (data[0:4] != "RED\_"): # wait until we get RED\_ command from server

                data = client.recv(SIZE).decode(FORMAT) # read in server communication

        print(data) # print for validation

**rationale**

We designed our code to work with our server using a series of acknowledgements to trigger different states based on the servers requests while we download files

**direct**

**code**

direct(): # direct function to list of directory and downloaded file details

    print("\nPrinting all files in directory and details") # state what we’re printing

    for i in recDict.keys(): # for every file in recDict, or the directory

        # print off details for each file in directory

        print("Filename:", i, "| Size:", os.path.getsize(i), "bytes | Creation date:", time.ctime(os.path.getctime(i)),

              "| Number of downloads by server:", recDict[i])

    print("\nPrinting all files downloaded from network") # state what we’re printing

    for i in receivedDict.keys():

        # print off details for each downloaded file

        print("Filename:", i, "| Size:", os.path.getsize(i), "bytes | Creation date:", time.ctime(os.path.getctime(i)),

              "| Number of downloads by client:", receivedDict[i])

    # print off dictionaries for all infographic elements

    print(recDict,

        receivedDict,

        upload\_times,

        download\_times,

        upload\_speed,

        download\_speed)

**rationale**

This function was built to simply display the files in our directory and their details, in addition to files we have downloaded and how many times we have downloaded them

**terminate**

**code**

terminate(client): # function to terminate our connection

    cmd = "END\_" # codeword for termination

    client.send(cmd.encode(FORMAT)) # format and send to the client

    conf = client.recv(SIZE).decode(FORMAT) # receive code from client to properly terminate

    if conf == "CONFIRM": # we receive the acknowledgement

        client.close() # close the connection

        print("Disconnecting from Server... ") # notify client we have disconnected.

**rationale**

The function simply terminates our connection to the server through a series of codewords and acknowledgements

**SRequest**

**code**

SRequest(bl, CURRENT\_PORT): # call for opening thread to CURRENT\_PORT

    sleep(1) # sleep for one second to let the connection build first and give server details

    # print thread info

    print(" \nS1, Thread", "| IP", IP, "| Port", CURRENT\_PORT)

    # Setup for connection

    ADDR = (IP, CURRENT\_PORT)

    server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    server.bind(ADDR)

    Connected = True # while we want a connection to client

    while (Connected): # while we are wanting to be connected to the client

        server.listen(1) # listen for client

        conn, addr = server.accept() # accept connection

        print("connection accepted") # acknowledge we’ve accepted a connection

        conn.send("CONFIRM".encode(FORMAT)) # send confirm that we have connected thread

        socket\_connect = True # connection to socket established

        print("We're connected") # additional acknowledgement to client user

        while (socket\_connect): #while we are connected to client with the socket

            intent = conn.recv(SIZE).decode(FORMAT) # receive the intent of client

            if (intent[0:4] == "SRD\_"): # if it is SRD,  or command to send

                filename = intent[4:SIZE] # draw filename from the intent in this case

                print("The File to send was recived: ", filename) # confirm what we received

                cmd = "PSH\_" + filename # combine push and filename

                upload(conn, cmd, filename) # send the details to upload

            if (intent[0:4] == "SRT\_"): # if the intent received to terminate socket

                socket\_connect = False # terminate connection process

            if (intent[0:4] == "TER\_"): # if the intent received is to terminate the connection

                socket\_connect = False # terminate socket connection

                Connected = False # terminate connection process

        # Added indentation from Fn to While

        conn.send("TERMINATING".encode(FORMAT)) # send confirmation we are terminating

        terminate(conn) # terminate

**rationale**

This is the code for the additional thread that will handle file requests from the server to our client through a series of codewords and acknowledgements.

**connect**

**code**

connect(conn, CONNECT\_IP, PORT): # call connect to connect to an IP and PORT

    # connect to server

    ADDR = (CONNECT\_IP, PORT)

    client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    client.connect(ADDR)

    # Print conformation of connection

    print(f"[CONNECTED] Client connected to server at {CONNECT\_IP}:{PORT}")

    # Create File with list of files

    FilesInFolder = listdir() # get the details

    # Set file name with IP and PORT

    filename = "Client\_" + IP + "-" + str(SREQUESTPORT) + ".FileList" # create filename to hand to server of our files

    file = open(filename, "w") # open file to write directory contents

    for i in FilesInFolder: # for every file in our directory

        file.write(i) # write to the file

        file.write("\n") # write a new line at the end to separate them

    file.close() # close after finished

    # Set Command to SND(Send file)

    cmd = "SND\_" + filename

    # Upload File List

    upload(client, cmd, filename)

    cmd = "" # set up wait for acknowledgement

    while (cmd == ""): # while cmd remains empty until a command

        cmd = client.recv(SIZE).decode(FORMAT) # receive input from server

    act = cmd # set act to cmd

    # loop as long connection is true

    Connection = True

    while Connection: #while we want a connection

        msg = input("Enter Instruction: ") # get instruction from the client

        # Set input to array

        files = msg.split() # split it by empty spaces

        # First Part of Message for if statements (Desired Instruction)

        instruction = files[0] # get instruction from the client on intent

        if (instruction == "Upload"): # Upload [filename]

            # Set command to push(force download on server)

            for i in range(1, len(files)): # for each file in the files array from command+1 to end

                filename = files[i] #set filename

                cmd = "PSH\_" + filename #combine command and filename

                start = time.time() #start the timer

                upload(client, cmd, filename) # send it

                end = time.time() #end the timer

                UPtimes(files, start, end) #calculate the time for upload

        elif (instruction == "Download"): # Download [filename]

            # Scen 2-1

            if (act == "1"): # if the server predefined configuration is scenario 2-1

                files.pop(0) # pop the 0th element, or the command

                CMD = "GET\_" #set command to GET\_

                for i in files: # for each file in files being requested

                    CMD = CMD + i + " " # combine command and filename and combine to one string

                start = time.time() # start timer for download

                download(client, CMD) # request to download the file

                end = time.time() # end the timer for download

                DNTimes(files, start, end) # calculate time for download

            # Scen 2-2

            elif (act == "2"): # If server intent is predefined configuration for scenario 2-2

                files.pop(0) # pop the command

                CMD = "GET\_" # set command to GET\_

                for i in files: # for each file in files being requested

                    CMD = CMD + i + " " # combine command and filename and combine to one string

                start = time.time() # start timer for download

                download(client, CMD) # download the data

                mergesplit(MRG\_TITLE) # split the merged file into separate files

                end = time.time() # stop timer for download

                DNTimes(files, start, end) # calculate time for download

            # Scen 2-3

            elif (act == "3"): # if server intent is predefined configuration for scenario 2-3

                files.pop(0) # pop the command

                CMD = "GET\_" # set command to GET\_

                for i in files: # foreach file in files being requested

                    CMD = CMD + i + " " # combine command and filename and combine to one string

                start = time.time() # start timer for download

                download(client, CMD) # download the file

                end = time.time() # stop timer for download

                DNTimes(files, start, end) # calculate time for download

            # Scen 2-4

            elif (act == "4"):  # if server intent is predefined configuration for scenario 2-4, our own

                files.pop(0) # pop the command

                start = time.time() # start timer for downloading in series

                for i in files: # for each file in our request, we download one at a time

                    CMD = "GET\_" # command to get a file

                    CMD = CMD + i # combine the command

                    download(client, CMD) # download the command

                client.send(("END\_").encode(FORMAT)) # once finished, notify server we are done

                end = time.time() # stop the timer

                DNTimes(files, start, end) # calculate time for download

        # Dir

        elif (msg == "Dir"): # if client wants to use Dir to look at file details

            direct() # call direct

        # Terminate

        elif (msg == "Terminate"): # if client wants to terminate

            terminate(client) # call terminate function with client

            Connection = False # set connection is false to break out of loop

            for i in recDict.keys(): # for each file in recDict

                print("File Name:", i, "Times Uploaded to Server:", recDict[i]) # print uploaded times to server

            for i in receivedDict.keys(): # for each file in receivedDict, or downloaded

                print("File Name:", i, "Times Downloaded from Server:", receivedDict[i]) # print times downloaded

            print("Disconnected") # notify user we have disconnected

        else: # if invalid command is chosen

            print("Invalid Instruction! Try ", connectList) # give user list of commands

**rationale**

Our connect function was the main function we would use to communicate our requests from a client to the server, so we added in key functions that we felt would be able to do so as designed in each code. There is an intent that the server will send to let us know which scenario we are simulating. Once the client is done, the connection to the server ends, and the connection is disassembled.

**main**

**code**

def main(): # main loop function

    # Set threadList and initialize message to empty space

    for i in listdir(): # for every time in directory

        recDict[i] = 0 # initialize the directory dictionary keys

    threadList = list() # list of threads we are running

    message = "" # message to trigger loop

    while (message != "Exit"): # while message is not Exit

        message = input("Client Message: ")  # client user inputs message

        # Connect [IP] [PORT]

        if (message.split()[0] == "Connect"):

            # Get IP and PORT form input

            CONNECT\_IP = message.split()[1]

            PORT = int(message.split()[2])

            # open thread to connect and SRequests

            connecting\_thread = threading.Thread(target=connect, args=(len(threadList), CONNECT\_IP, PORT))

            SRequest\_thread = threading.Thread(target=SRequest, args=(len(threadList), SREQUESTPORT))

            # add threads to list

            threadList.append(connecting\_thread)

            threadList.append(SRequest\_thread)

            # Start Threads

            SRequest\_thread.start()

            connecting\_thread.start()

        # Print valid commands

        elif (message == "Help"):

            print(mainList)

        # Delete file

        elif (message == "Delete"):

            for file in receivedDict.keys():

                path = os.path.join(os.getcwd(), file)

                os.remove(path)

        # Exit loop

        elif (message == "Terminate"):

            message = "Exit"

            print("Exiting...")

        # File Directory

        elif (message == "Dir"):

            direct()

        else:

            # No valid command, print valid commands

            print("Invalid Command! Valid commands:", mainList)

        # join threads

        for index, CurrThread in enumerate(threadList):

            CurrThread.join()

**rationale**

This was our main function built to allow the client to do what is needed as per the project requirements,

**Experiments:**

To test and experiment our project, we used 3 given scenarios.

The first scenario we tested was a client requesting a file from the server.  If the server has it, it will send it to the client. If the server doesn’t have the file, the server will request the temporary  file from a second client, send it to the first client, and delete the temporary file. This was done by a client requesting a.txt, the server had this file and sent it. The client then asked for file file1.txt, the server didn’t have this and connected to a thread with client 2 and forced it to upload the file to the server, then sent it to client 1. The file sizes were similar and had similar completion times as seen in the screenshots below. The file not on the server, however, did complete slightly faster at 2.81 seconds  in the second screen shot vs 2.86 seconds for the file on the server in the first screenshot. Note: the file size in the second image is larger, yet it is still faster than directly to the server

Text

Description automatically generatedText

Description automatically generated

In the second scenario, client 1 would ask for multiple files of the same type from the server. This was divided into four separate strategies. In strategy one, the server checks to see if it has one of the requested files, it will send the files it has, then request for the files it doesn’t have from client 2. 2 sends the files to server, server sends client 2 files to client1. To test this scenario, client 1 requests a file the server had and a client that client 2 had, those being z.txt(server) and f.txt(client2). The screenshot below each strategy description are the results. Strategies were declared by the server and received by the client as 1,2,3, or 4. The client would request downloads differently in if statements. Time was kept using os time declared at the start and end of each strategy

Text

Description automatically generated

In strategy 2, client requests file, server asks client 2 for files it doesn’t have and combines them with files it has into a merged file, it then sends the merged file to client 1, and client 1 decodes the tags of the merge file and writes files based on it.

Text

Description automatically generated

Merged file

Graphical user interface, text

Description automatically generated

Decoded files (f.txt, z.txt)

Text

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidence

In strategy 3, same as strat 2, but doesn’t merge. Server waits for files from client 2, then sends them and the files it has to client 1 all at once.

Text

Description automatically generated

In strat 4, client requests files 1 at a time, and server checks if it has it, if it does it  sends it, if not it gets it from client 2 and sends it, then client requests another file. This was a custom strategy we made to compare.

Text

Description automatically generated with low confidence

Results of Scenario 2: Strategy 2 was the fastest strat with a completion time of 5.22 seconds. Merging all the files, sending it once, and decoding the files all resulted in a faster time. The other times are 1(8.81), 3(7.77), 4(7.5). Strat 1 was the slower where the client would need to send an acknowledgement. This makes some sense as there was only one send between client and server vs the several for each file that could be delayed by acknowledgements and sleep functions.

The last scenario 3, The server is moved to a new location to see if compilation times change. This was completed by both clients being on one side of the Willy T. Library. The following screenshots are the results, the first being next to each other, and the second being at a different location. This is the result of client 1 requesting 2 files that client 2 had. The result is, when the server was in a different location, the load time was faster. 6.83(next to) seconds vs 6.76 seconds(away). It is a surprising result, but it's what we got. We were still on the same ip address, but the server was in a different location. It was just surprisingly faster away from the clients

Text

Description automatically generatedText

Description automatically generated

**Conclusion:**

In conclusion, the results for this project were surprising. In Scenario 1, the completion time was faster when receiving a file from another client. We expected direct connection to be better, however we were proven wrong. In Scenario 2, Strategy 2 was the fastest, merging files with only one upload to client 1 was quicker with less acknowledgements and sleep commands. Strategy 1 was the slowest, being 3 seconds slower. This was due to required acknowledgements and sleep commands at every download and upload.  In scenario 3, the server in a different location was actually faster. That surprised us a lot, though the distance wasn’t huge, we expected location being closer would be faster, but we were proven wrong, not by a lot but still wrong. Maybe it was a fluke or just had a clearer connection, but the results speak for themself.