COL334 : Assignment 2

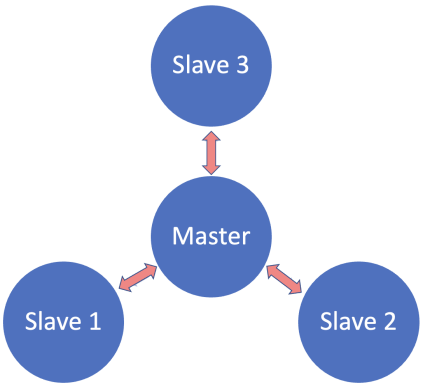
Rajat Golechha, Satyam Kumar, Harshit Gupta, Mani Sarthak

September 8, 2023

Contents

1 Algorithm Description

In order to mimic distributed file transfer, we have established a master-slave design. With one of the devices acting as the master and the rest acting as slaves.



First we start the master which hosts the server for other computers to connect to, once all computers are connected, we use the **Master-Slave START** protocol. All the devices now start **communicating with vayu** and whenever master recieves a new unique line, it's broadcasted to all the slave clients, whenever a slave recieves a new unique line, it forwards the line to master who then broadcasts the same to remaining slave clients. The process keeps on repeating until either the master or any of the slaves finish collecting all 1000 lines.

When the master has 1000 lines, it sends the slaves a **DONE** signal that all 1000 lines are available, and now they can request any specific line they require to complete their data using the **Master-Slave REQUEST** protocol.

When any slave completes 1000 lines, it sends the last line it recieved to master, and sends a **Master-Slave STOP** signal to the master and closes the threads, the master also closes the threads on it's end, when the master recieves a STOP signal from all of the clients, it closes all the threads and exits the program.

2 Protocols

2.1 Communicating with vayu

Client		**Server(Vayu)**
send the **SENDLINE** command	$\rightarrow$	Recieves the Command
recieves the line number and line content	$\leftarrow$	sends the line number and line content
REPEAT		**REPEAT**

2.2 MASTER SLAVE - START

Server(Master)		**Client(Slave)**
Sends the **start** command to all slaves and start protocol of communicating with vayu	$\rightarrow$	Recieves the command and start protocol of communicating with vayu
recieves the data and send to other slaves	$\leftarrow$	get data from vayu server and send to master
get data from vayu server and send to all slaves	$\rightarrow$	recieves the data
REPEAT LAST 2 STEPS		**REPEAT LAST 2 STEPS**

2.3 MASTER SLAVE - REQUEST

Server(Master)		**Client(Slave)**
Sends the **done** command to all slaves when all data is recieved by master	$\rightarrow$	Recieves the command and stop connection with vayu
receives the command	$\leftarrow$	requests the line number which is not yet recieved
send the corresponding line content to the slave	$\rightarrow$	recieves the line content
REPEAT LAST 2 STEPS		**REPEAT LAST 2 STEPS**

2.4 MASTER SLAVE - STOP

Server(Master)		**Client(Slave)**
receives the command and remove the slave from the connection	$\leftarrow$	Sends the **stop** command when all the data is gathered and submitted to vayu

3 CODE

3.1 Master Design

Master is the coordinator of this file-transferring process. Therefore, it creates a TCP socket, and starts listening for any of the clients on the connection_accept_thread.

```
    host = '10.194.20.150'
    port = 13345
    server_socket = socket.socket()
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind((host, port))
    server_socket.listen(10)
    print("Server listening on {}:{}".format(host, port))
    connection_accept_thread = threading.Thread(target=connecting_thread,
     args=(server_socket,))
    connection_accept_thread.start()
```

The function connecting_thread accepts connections from clients and creates new threads for each of them so that they are handled concurrently. We also have the number of max_connections and number of submissions so far as done_count, if all clients have submitted we can close this thread, if some client disconnects and tries to reconnect then it's also accepted and immediately sent a **start** message to begin the program.

```
def connecting_thread(server_socket):
    global max_connection
    global done_count
    count=0
    while done_count < max_connection:
        if done_count==max_connection:
            break
        conn, address = server_socket.accept()
        server_list.append(conn)
        print(f'client connected : {address}')
        client_thread = threading.Thread(target=handling_client, args=(
    conn, address))
        client_thread.start()
        thread_list.append(client_thread)
        count+=1
        if count>max_connection:
            conn.sendall("start".encode())
    server_socket.close()
```

3.1.1 Explanation of starting_client function

So when all clients are connected , the master sends a **start** command to all the clients, and this ensures that the clients and the master are synchronously started and the data being shared amongst them is coherent.

```
    start_command_thread = threading.Thread(target=starting_client)
    start_command_thread.start()
    def starting_client():
        global broadcaasting_started
        while True:
            command = input("Enter 'start' to initiate interaction: \n")
            if command.strip().lower()=='start':
                print("Starting the interaction with client and vayu")
                broadcaasting_started=True
                with broadcast_list_lock:
                    for client_conn in server_list:
                        client_conn.sendall("start".encode())
                break
    start_command_thread.join()
```

3.1.2 Explanation of handling_client function

In the function to handle client we have made two cases. In the first case we handle the situation when server has not got all lines. And in second case when server has all lines. So when server does not have all the lines, then unique data received from clients are being sent to all other clients so that they could get finish as quickly as possible. To send lines to clients We have applied broadcast_list_lock1. And when any client is done it sends a "STOP" message and we remove that client from our client list.

```
def handling_client(conn,address):
    global done_count
    global max_connection
    global line_num
    global sending_started
    global found
    while True:
        if len(data_dict)<line_num:
            data = recv_input(conn)
            if not data:
                break
            if data=='stop\n':
                with done_count_lock:
                    done_count+=1
                    break
            try:
                lines = data.split("\n")
                line_numb = int(lines[0])
                line_cont = lines[1]
            except Exception as e:
                print(e)
            if line_numb not in data_dict:
                data_dict[line_numb] = line_cont
                with broadcast_list_lock1:
                    for client_conn in server_list:
                        if client_conn != conn:
                            client_conn.sendall(data.encode())
                if len(data_dict)>=line_num:
                    found = True
    with broadcast_list_lock:
        server_list.remove(conn)
        print("Client disconnected:", address)
    conn.close()
```

In the second case when master has all the lines, it sends all clients a **done** message, signalling that data collection from vayu is complete. And now they can ask for specific lines that they are missing, the clients then reply with a line number, and the master sends them the specific line. When the client has all the data it sends a stop message and we increment done_count variable to account for no. of clients which have submitted.

```
    else:
        data = recv_input(conn)
        if not data:
            break
        if data!='stop\n':
            print(data, "from client")
            # print('recieved send command')
            print('sending data')
            lines = data.split("\n")
            lin_num = int(lines[0])
            response = str(lin_num) + '\n' + data_dict[lin_num] + '\n'
```

```
            conn.sendall(response.encode())
        elif data=='stop\n':
            with done_count_lock:
                done_count+=1
                break
```

3.1.3 Explanation of Broadcasting Function

In the Broadcasting thread we establish connection with vayu server and start requesting it for lines. Lines received from vayu are being updated in the data_dict. Whenver we receive new and unique lines we send that line to all other clients connected to the master. To send the lines to all clients we have used lock to avoid any race condition.

```
    broadcast_thread = threading.Thread(target=broadcasting_thread)
    broadcast_thread.start()
    def broadcasting_thread():
    server_address = ("vayu.iitd.ac.in", 9801)
    sendline_command = b"SENDLINE\n"
    global line_num
    global sending_started
    global done_count
    global max_connection
    global found
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(5)
    s.connect(server_address)
    start = time.time()
    while True:
        try:
            while len(data_dict) < line_num:
                print(len(data_dict))
                s.sendall(sendline_command)
                received_data = recv_input(s)
                try:
                    lines = received_data.split("\n")
                    line_number = int(lines[0])
                    line_content = lines[1]
                except Exception as e:
                    print(e)
                if line_number == -1:
                    time.sleep(1e-6)
                    continue
                if line_number not in data_dict:
                    data_dict[line_number] = line_content
                    with broadcast_list_lock2:
                        for ke in server_list:
                            try:
                                ke.sendall(received_data.encode())
                            except Exception as e:
                                continue
                if len(data_dict)>=line_num:
                    found=True
```

And when all thousand line have been received, We set global found variable to true. Then we send done message to all clients so that they can start requesting specific lines they have not received yet, once this is done all devices can submit the compiled data. And when all clients are done, the connection to vayu is closed, and thread is also closed.

```
    if found:
                with broadcast_list_lock:
                    for client_conn in server_list:
                        client_conn.sendall('done\n'.encode())
                    found = False
            if done_count==max_connection:
                submit_command = b"SUBMIT\naseth@col334-672\n" + str(
line_num).encode() + b"\n"
                s.sendall(submit_command)
                for key in range(line_num):
                    with broadcast_list_lock: # is it good idea to use
same lock
                        s.sendall(str(key).encode() + b"\n" + data_dict[
key].encode() + b"\n")
                submission_success = recv_input(s)
                print(submission_success)
                finish = time.time()
                print(f"Time taken: {finish - start}")
                break
        except Exception as e:
            print("An error occurred in broadcasting_thread:", e)
            break
    print('all done broadcasting_thread_close disconnected')

    s.close()
```

3.2 Slave Design

In our algorithm, slave is a client which is connected to master and vayu server. The slave/client recieves lines from both vayu and master, any unique lines that the slave recieves from vayu are sent to the master to be forwarded to other slaves and vice-versa. This way using two threads coherent data is being shared amongst the devices.

```
    ClientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    VayuSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    receive_thread = threading.Thread(target=recieve_thread, args=(
ClientSocket,))
    vayu_thread = threading.Thread(target=vayu_thread_broadcast, args=(
VayuSocket,ClientSocket))
    receive_thread.start()
    vayu_thread.start()
```

3.2.1 Explanation of recieve_thread function

Slave keeps on waiting until a start command is sent by the master to begin communication with vayu. When the slave receives a "start" message it starts. Two threads are running concurrently, one with vayu and another with the master. This receive thread work in two phases. First phase is neither the master has all the lines nor the slave. The slave keeps on recieving data from master and vayu until either of the condition is not satisfied when it happens, the flag finalle_started changes value to true, and we move to the second phase.

```
    def recieve_thread(server_socket):
    global started
    global finalle_started
    global started2
    global line_num
    global linenotrecv
```

```
    global linerecv
    qw=False
    while True:
        receive = server_socket.recv(2048).decode('utf-8')
        if receive=='start':
            started=True
            break

        time.sleep(1e-6)
    while True:
        try:
            if finalle_started==False and len(data_dict)<line_num:
                data = recv_input(server_socket)
                if not data:
                    break
                if data == 'done\n':
                    finalle_started=True
                else:
                    try:
                        lines = data.split("\n")
                        line_number = int(lines[0])
                        line_content = lines[1]
                    except Exception as e:
                        continue
                    if line_number not in data_dict:
                        data_dict[line_number] = line_content
                        linerecv.append(line_number)
```

In the second phase of receive_thread if we have all the lines, then we signal the same to the master by sending a **stop** signal, submitting and closing the threads. If not then we make a list of the lines that are missing and start requesting the master for those lines, once we recieve all of them we send a **stop** signal to master and submit and then close the threads.

```
        linenotrecv=list(set(range(0,line_num))-set(linerecv))
        if(len(linenotrecv)==0):
            print("All the data is recieved")
        else:
            snd=str(linenotrecv[0])+'\n'
            server_socket.sendall(snd.encode())
            data = recv_input(server_socket)
            if not data:
                break
            try:
                lines = data.split("\n")
                line_number = int(lines[0])
                line_content = lines[1]
            except Exception as e:
                print(e)
            if line_number not in data_dict:
                data_dict[line_number] = line_content
                linerecv.append(line_number)
        if len(data_dict)>=line_num:
            server_socket.sendall('stop\n'.encode())
            started2=True
            break
```

3.2.2 Explanation of vayu_thread_broadcast function

The vayu thread establishis a connection with vayu server on starting, and waits until a **start** message is recieved from server. Once we recieve a start message from the master,

we start sending **SENDLINE** requests to vayu, if we recieve a new unique line, then we update it in the dictionary and send the line to master also. If we recieve a -1 then we wait for 1 microsecond before sending another request to vayu, if we recieve a repeated line then we continue the loop. The loop breaks when we have 1000 lines, at which we submit all the lines using a for loop to vayu.

```
    def vayu_thread_broadcast(vayu_socket,server_socket):
    server_address = ("vayu.iitd.ac.in", 9801)
    sendline_command = b"SENDLINE\n"
    while True:
        if started==True:
            break
        time.sleep(1e-6)
    vayu_socket.settimeout(5)
    vayu_socket.connect(server_address)
    start = time.time()
    while True:
        try:
            if finalle_started==False and len(data_dict)<line_num:
                vayu_socket.sendall(sendline_command)
                received_data = recv_input(vayu_socket)
                try:
                    lines = received_data.split("\n")
                    line_number = int(lines[0])
                    line_cont = lines[1]
                except Exception as e:
                    continue
                if line_number == -1:
                    time.sleep(1e-6)
                    continue
                if line_number not in data_dict:
                    data_dict[line_number]=line_cont
                    linerecv.append(line_number)
                try:
                    server_socket.sendall(received_data.encode())
                except Exception as e:
                    continue
            else:
                if started2==True or len(data_dict)>=line_num:
                    print("Submitting...")
                    submit_command = b"SUBMIT\naseth@col334-672\n" + str(
line_num).encode() + b"\n"
                    vayu_socket.sendall(submit_command)
                    for key in data_dict.keys():
                        vayu_socket.sendall(str(key).encode() + b"\n" +
data_dict[key].encode() + b"\n")
                    finish = time.time()
                    print(f"Time taken: {finish - start}")
                    submission_success = recv_input(vayu_socket)
                    print(submission_success)
                    break
        except Exception as e:
            print('Error sending:', str(e))
            break
```

4 Observation and Results

4.1 Plots

Figure 1: Plot of Number of Clients vs. Time Taken

Time Taken	No. of Clients
80 sec	1
36 sec	2
26 sec	3
20 sec	4

Table 1: The following data is the average of several runs

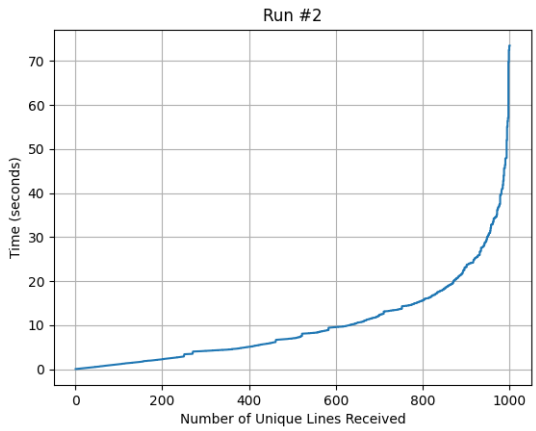


Figure 2: For a single device

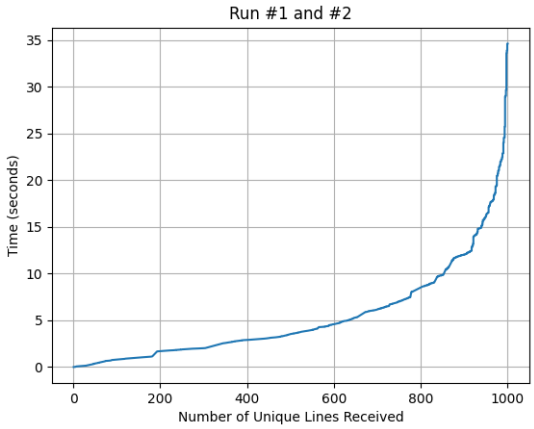


Figure 3: For two devices

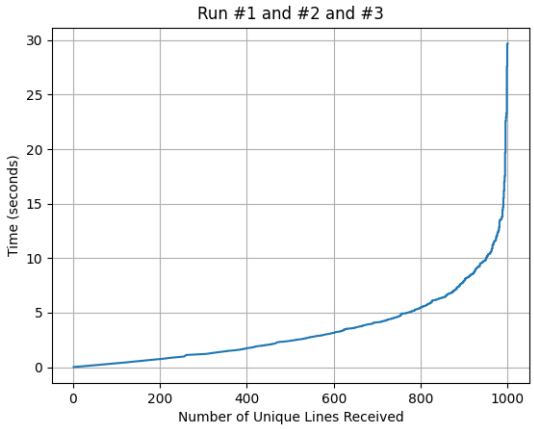


Figure 4: For three devices

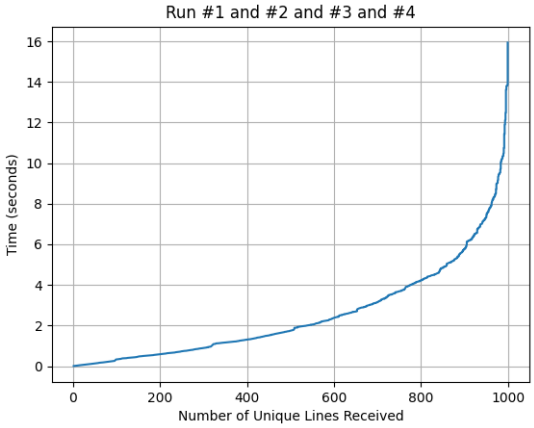


Figure 5: For four devices

4.2 Inference

1. Initially, as We add more clients to the network, We observe a relatively linear reduction in file sharing time. This means that if as we double the number of clients, we have roughly half the time it takes to download a file because there are more sources available for parallel downloads.

2. However, as the number of clients continues to increase, the reduction in file sharing time tends to slow down. This is due to several factors, including network overhead, limitations in individual clients' upload bandwidth, and the law of diminishing marginal returns. At some point, adding more clients may not lead to a proportional decrease in file sharing time.

3. Thus the reduction in file sharing time in our distributed file transfer system as the number of clients increases follows a non-linear pattern.

5 Exception Handling

We have handled various exceptions in our project.

5.1 Client Disconnects & reconnects

The code is designed in a manner that maintains a list of clients, and whenever someone disconnects and rejoins after start, immediately on their joining they get the START message and if the other clients finish earlier, then it gets a DONE message and can start asking the specific lines that are missing in it.

5.2 Master Crashes

In the event of a master server crashing, the code is designed to still keep the clients running individually with vayu.

5.3 Client Crashes

In the event of crash of any of the client, the code is designed to keep the other clients and the master running.

6 Repository :

Github-repository : FILES