Sali&Yosef Chayroom- system characterization document

General description of the system:

As part of a computer communication course, we were asked to build a multi-user instant messaging system based on communication.

The project is divided into two parts, client side and server side.

The server runs from one computer and the client can run different computers that are on the server's local host using an IP address and port.

In the project we used sockets to send messages over TCP

In addition, there is a file download service from the server that is performed using a socket for sending messages over fast reliable UDP.

Every message sent between the client and the server is a string that is encoded and decoded using the "utf-8" encoding except for the file download which is done using data-grams sent in UDP messages that are encoded and decrypted using pickle encoding.

The client and server receive messages, the way they decide what actions to take when certain messages are received is according to the system protocols that will be detailed later in the document.

In this paper we will explain in depth all the system components, protocols, how to run, handling exceptions, testing and Wireshark recordings.

The project is written entirely in the language of Python programming and uses the principles of Computer Communication and Object-Oriented Programming.

Table of Contents

Our System TCP traffic protocol

All the messages send to the server and then the server make decision what to do with the message appending to what the client wrote in the message, when the server receive message to the client, the client make decision what to do with the message appending to what the server wrote in the message the Strings inside ‘< >’ will help to make activities with the server.

To this end, we have prepared a protocol that will translate the messages from the server and clients into the appropriate actions.

TCP Clients Protocol: (TCP messages from a client to the server)

-message: **<~><name> <msg>** (name- the name of the received client, msg- the message)
Will send a private message to client.
For example:
~Daniel Hello Daniel, it’s me dad!
Will send a private message to Daniel: “Hello Daniel, it’s me dad!”

-message: **<show>**
Will receive a message to the client with the names of the other connected clients. (every list starts with word ‘ALL’ (it’s not a client).
For example:
show
Will send message to client: ALL Daniel Dad Mom while the names of the connected clients are “Daniel”, “Dad”, “Mom” and the sender name .

-message: **<get_list_file>**
Will receive a message to the client with the names of the files that you can download from the server.
(every list starts with CHOOSE_FILE, it’s not a file name).
For example:
get_list_file
Will receive a message to client: CHOOSE_FILE file1.txt file2.png file3.mp3

-message: **<download_file> <filename>** (<filename> - the name of the file to download)
Will send a request to download a file from the server, when the server receive this message he opens a UDP socket, sends a TCP message to the client to open a UDP socket and wait for connection. the file will be saved in a directory named “Downloads” in the client side, if the directory not exist, it will be created.
for example:
download_file file3.mp3
will send file to client

-message: **<{quit}>** to disconnect and exit. -
Any other message will send message to all the connected clients as a public message

<u>**TCP Server Protocol:** (TCP messages from the server to the clients)</u>

-message: **<UPD > <msg>** (msg- the message, a part of the file)
When the client requests to download a file from the server, the server divides the file into parts and sends each part of the message in the UDP protocol, the way the client recognizes that it has received a part of a file is when the message starts with the "UDP" string.

-message: **<ALL > <msg>** (msg- the message, connected clients names separated with spaces)
The server sends the names of the clients that connected to the server, and like that the client knows that if the message starts with “ALL”, the following will describe what are the names of the clients that connected to the server.
For example:
ALL Daniel Dad Mom
The names of the connected clients are: “Daniel”, “Dad”, “Mom” and the sender name .

-message: **<<CHOOSE_FILE>> <msg>** (msg- the message, files names separated with spaces)
The server sends the names of the files that prepared to download in “FILES” directory in the server side, and like that the client knows that if the message starts with “<COOSE_FILE>”, the following will describe which files the client is able to download.
For example:
ALL Daniel Dad Mom
The files are: file1.txt file2.png file3.mp3

Our System UDP traffic protocol

When a request is received from a particular client to acknowledge a file, the server searches for a free port between 55000 and 55015, opens a UDP socket and sends a message to the server that will also open a UDP socket at the same port.
These messages will be as follows:

UDP Client Protocol: (UDP messages from a client to the server)

-message: **<part> <i>** (i- index of the next part of the file)
A file is downloaded from the server via UDP messages.
Before starting the download, the server receives the file size, it divides the file into pieces of 1000 bits at most and thus knows how many parts it needs to request from the server.

-message: **<finished>**
The client sends this message when he has finished receiving all the packages he requested in order to download the file in their entirety, after making sure that each package has no information leakage, after sending the message the client closes the UDP socket and when the server receives this message he also closes the UDP socket.

UDP Server Protocol: (UDP messages from the server to the client)

-message: **<(<data>,<data_size)>** (data- the data of the part of the requested file
data_size- size of the data of the part of the requested file.)
After a request from the client for a certain part of the file, the server seeks the file for the beginning of the requested part, saves a maximum of 1000 bits and sends the client the requested part together with the part size, this figure is useful for checking the UDP reliability.

Authors: Yosef Sokolov & Sali Sharfman

Fast Reliable UDP

When downloading a file from the server, we want to make sure that there is no loss of packages because every slightest mistake in the information will result in the file not being able to be opened.

For this reason we must be sure that if the client writes a piece of data to a file he will write the whole piece in its entirety as sent from the server.

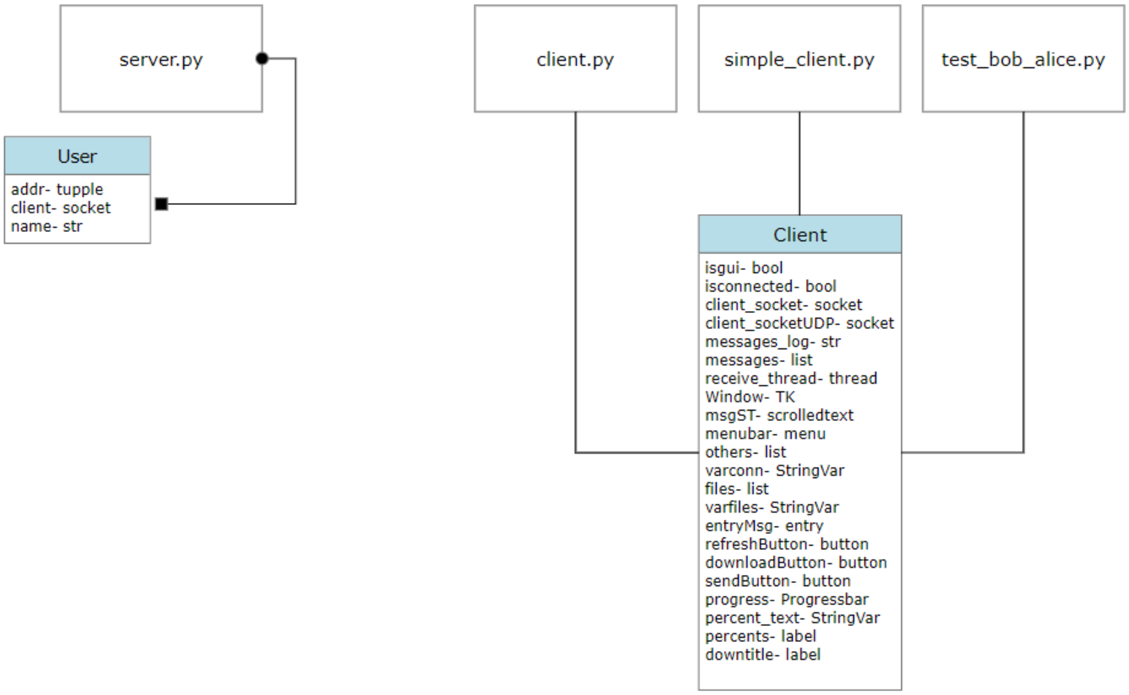
To this end, we implemented a method for RDT over a UDP connection plus Congestion control,

The method we wrote is, in every UDP packet sent from the server, the size of the information attached to the packet is also sent, each time the client receives such a message, he enters a loop in which he compares the size of the information sent from the server with the size of the information sent from the client, repeat the loop again until the sizes are equal, when they are equal, we will exit the loop and write the information received to the file.

```
def download(self, port, size, path):
    try:
        with open(path, "wb") as file:
            size = (size // 1000) + 1
            ADDRESS_SERVER = (self.HOST, port)
            CLIENTUDP = socket(AF_INET, SOCK_DGRAM)
            for i in range(size):
                done = False
                while not done:
                    CLIENTUDP.sendto(bytes(f"part {i}", "utf-8"), ADDRESS_SERVER)
                    msg, addr = CLIENTUDP.recvfrom(self.BUFSIZ)
                    # filter messages that are not from our server
                    while (addr != ADDRESS_SERVER):
                        msg, addr = CLIENTUDP.recvfrom(self.BUFSIZ)
                    msg,msg_size =pickle.loads(msg)
                    if type(msg_size) == int and len(msg)==msg_size:
                        done = True
                    else:
                        print("lost a packet, send again.")
                file.write(msg)
                percent=round(i / size * 100, 2)
                print(f"{percent}% downloaded")
                if self.isgui:
                    self.percent_text.set(f"{percent}%")
                    self.progress['value'] = percent
        print("File was downloaded 100%\n")
        if self.isgui:
            self.percent_text.set("100%")
            self.pushMSG(f"{path.split('/')[-1]} downloaded successfully.\n")
            self.progress['value'] =100
            time.sleep(2)
            self.percent_text.set("")
            self.progress['value'] = 0

    finally:
        CLIENTUDP.sendto(bytes("finished", "utf-8"), ADDRESS_SERVER)
        CLIENTUDP.close()
```

 Authors: Yosef Sokolov & Sali Sharfman

UML Diagram


server.py
User
addr- tupple
client- socket
name- str
client.py
simple_client.py
test_bob_alice.py
Client
isgui- bool
isconnected- bool
client_socket- socket
client_socketUDP- socket
messages_log- str
messages- list
receive_thread- thread
Window- TK
msgST- scrolledtext
menubar- menu
others- list
varconn- StringVar
files- list
varfiles- StringVar
entryMsg- entry
refreshButton- button
downloadButton- button
sendButton- button
progress- Progressbar
percent_text- StringVar
percents- label
downtitle- label

Server.py

Upon execution, this file opens a TCP socket for the server, this file also contains a User class whose object type will contain a socket for a client plus an address and name.

The server defines a list of users through which and through the implementation of the server's system protocol, it manages the communication with the clients.

```
py ×   server.py ×
#user class
class User:...
# GLOBAL CONSTANTS
HOST = ''
PORT = 55000
ADDR = (HOST, PORT)
MAX_CONNETIONS = 10
BUFSIZ = 1024
# GLOBAL VARIABLES
users = []
SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)  # set up server
# dict of all ports 55000-55015
# (if port is caught his value = False otherwise True)
ports = {}
for i in range(16):
    ports[55000 + i] = True

def download(client, filename):...

def show_online(client, name):...

def show_online_to_all(name=""):...

def private_msg(msg, sender_name, getter_name):...

def broadcast(msg, name):...

def client_communication(user):...

def wait_for_connection():...

def start_server():...

def stop_server():
    SERVER.close()

if __name__ == "__main__":
    start_server()
```

 Authors: Yosef Sokolov & Sali Sharfman

Server.py run

*Open cmd from “Server_side” folder and write the next command “python server.py “.

<u>The Server’s output after running server.py:</u>

C:\Windows\System32\cmd.exe - python server.py

```
Microsoft Windows [Version 10.0.22000.493]
(c) Microsoft Corporation. All rights reserved.

C:\Users\salis\PycharmProjects\projectYosfSali\Server_Side>python server.py
[STARTED] Waiting for connections...
```

Now the server is up and wait for clients connections.

<u>The Server’s output after chat between 2 clients a download a file:</u>

C:\Windows\System32\cmd.exe - python server.py

```
Microsoft Windows [Version 10.0.22000.493]
(c) Microsoft Corporation. All rights reserved.

C:\Users\salis\PycharmProjects\projectYosfSali\Server_Side>python server.py
[STARTED] Waiting for connections...
[CONNECTION] ('127.0.0.1', 52418) connected to the server at 1646241506.9749155
[CONNECTION] ('127.0.0.1', 52419) connected to the server at 1646241532.5157058
[UDP STARTED] Waiting for connections...
[DISCONNECTED] anotherUser disconnected

[DISCONNECTED] simpleUser disconnected
```


Authors: Yosef Sokolov & Sali Sharfman

client.py

In this file is a Client class, a Client-type object gets a name builder, opens a TCP socket with the server and maintains the client's system protocol.
A Client-type object can be built in two modes, with or without a graphical user interface.
Creating a Client-type object would look like this: Client (<name>, True / False)
*The Boolean argument says whether or not there is a GUI.
Running the file starts the dialog to get the name as input and then builds a Client-type object with the input name and True as values, thus basically running the graphical user interface.

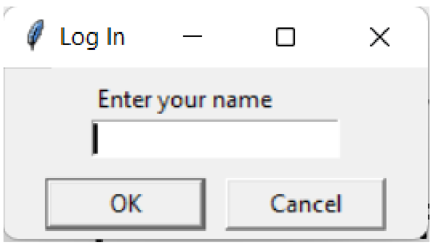
```python
class Client:
    """
    for communication with server
    """
    if len(sys.argv) > 1:
        HOST = sys.argv[1]
    else:
        HOST = '127.0.0.1'
    PORT = 55000
    ADDR = (HOST, PORT)
    BUFSIZ = 1024
    def __init__(self,name:str,isgui=True):...

    def Gui_Constractor(self):...

    def send_from_GUI(self):...

    # set the pull down of the connected users
    def setOthers(self):...

    # set the pull down of the files
    def setFiles(self):...

    def download(self, port, size, path):...

    def download_btn(self):...

    def receive_messages(self):...

    def pushMSG(self,msg):...

    def send_message(self, msg):...

    def get_messages(self):...

    #saving the chat to a text file
    def save_log(self):...

    def renameClient(self):...
```

```python
    def renameClient(self):...

    def disconnect(self):...

    #after disconnecting from gui you can back to chat
    def back_to_the_chat(self):...

    #open help.pdf on a web browser
    def about(self):...

def legal(name):...

#create a dialog to input name
def ask_for_username():...

def run():
    name=ask_for_username()
    Client(name)


if __name__=='__main__':
    run()
```

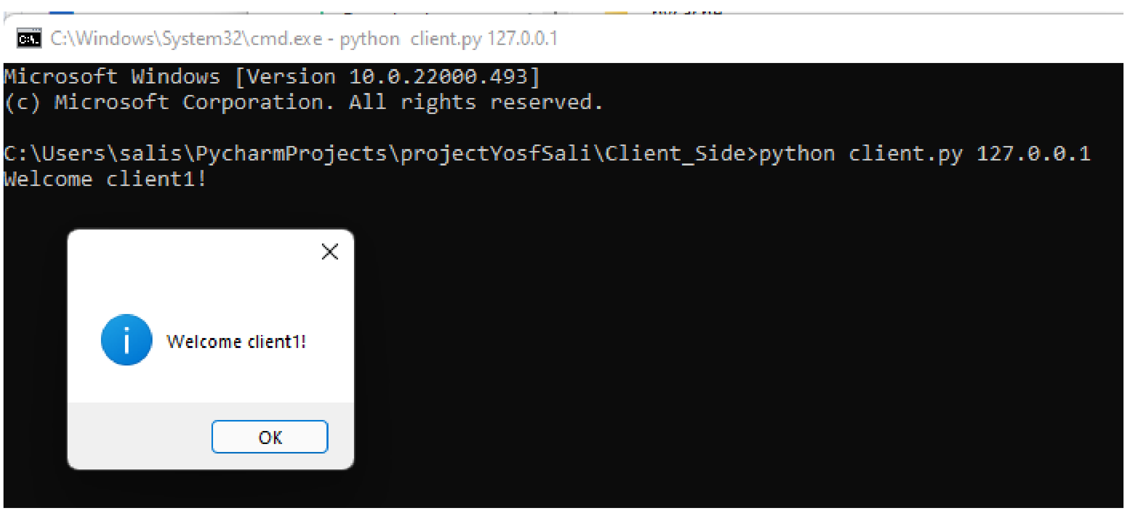
client.py- Graphic User Interface run

after running the server,
*Open cmd from “Client_side” folder.
*Write the next command “python simple_client.py <IPv4>”
<IPv4>- the IPv4 address of the server that already run.
for example: python test_bob_alice.py 127.0.0.1 (this IP address was set default you are not have to write it)

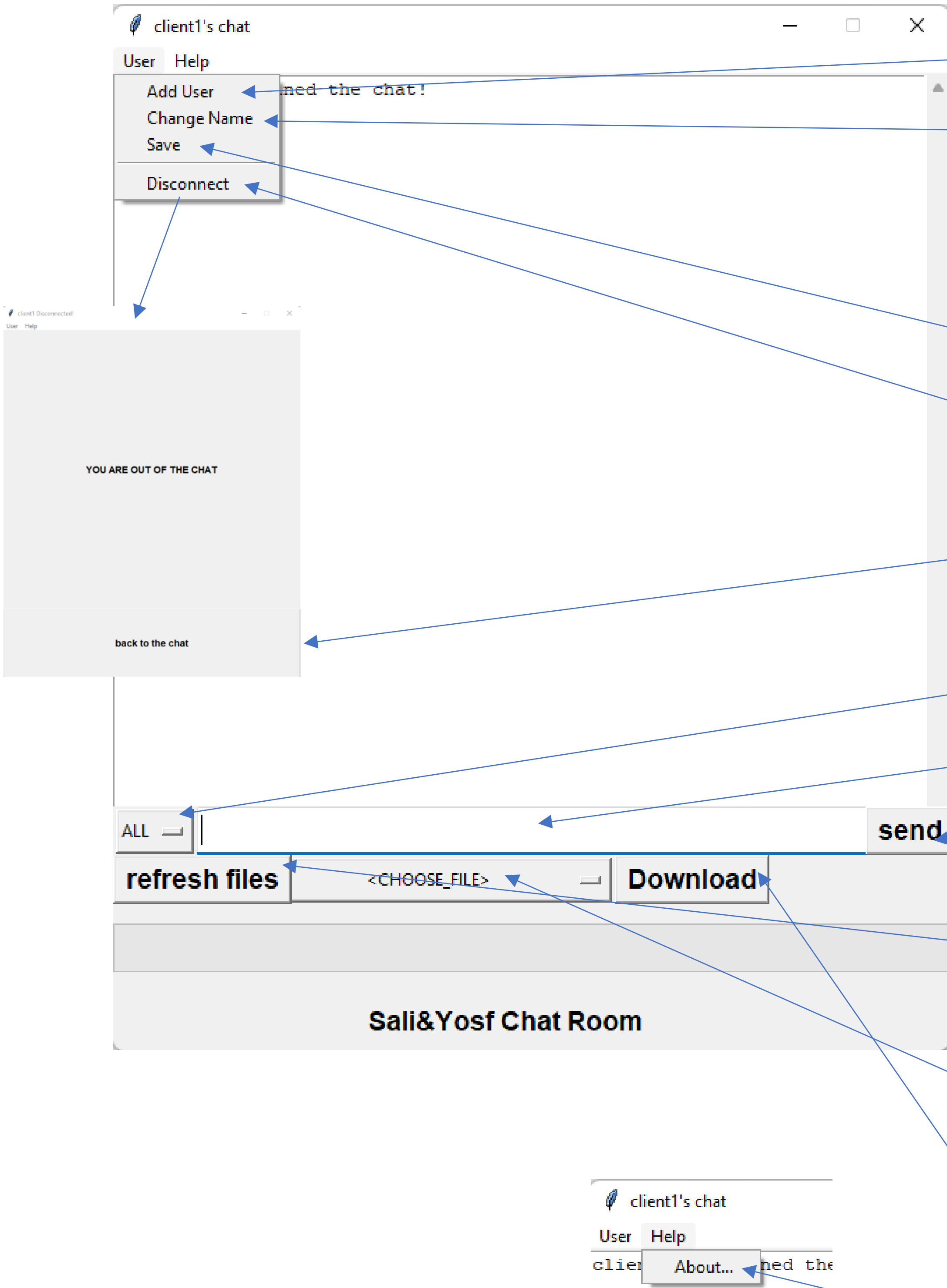
*Write your name and press OK



*Press OK and you are connected and you can start chatting with the Graphic User Interface.



The window of the chat room:



Action:

To add another different client

To change your name.

To save a log with all your messaging in this chatroom, the log will be saved in "Client_Side" directory in "logs".

To disconnect the chat.

To return to the chat room, Continues the chat in the same mode before disconnecting.

To choose who will get your message (ALL will send to everyone).

Write your message here.

To send the message.

To refresh the list of the files the server has for download.

To view the list of the files the server has for download, and choosing one.

To download the file that you chose.

To open the system characterization document.

simple_client.py

This is an executable file that allows you to connect to the server through the command line. First the user asked to enter a name as input, then the system protocol is printed on the screen so that the user knows how to communicate with the server, then, a client object is created without a graphical interface with the name entered and then the loop begins, every message the client receives is printed on the screen and every input is sent as a message Follow the array protocol to communicate with the server optimally.

```python
Thread(target=update_messages).start()
print(f"--------------welcome{name}!--------------\n")
print("The input you will type will send message to the server\n")
print("The next inputs (inside the <>) will help you to make some activities with the server:\n")
print("input:<~><name> <msg> to send a private message to client. (name- the name of the received client, msg- your message)\n")
print("input: <show> to get the names of the connected clients. (every list starts with ALL, its not a client)\n")
print("input: <get_list_file> to get the names of the files that you can download from the server.(every list starts with CHOOSE_FILE, its not a file name)\n")
print("input: <download_file> <filename> to download a file from the server. (<filename>- the name of the file that you want to download)\n")
print("input: <{quit}> to disconnect and exit.")
print("any other input will be sent from you to all the connected clients as a public message.\n")
print("--------------sali&yosef chatroom--------------\n")


while True:
    msg=input()
    if msg == '{quit}':
        c1.disconnect()
        sys.exit()
    else:
        c1.send_message(msg)
```

simple_client.py run

after running the server,
*Open cmd from “Client_side” folder.
*Write the next command “python simple_client.py <IPv4>”
<IPv4>- the IPv4 address of the server that already run.
for example: python test_bob_alice.py 127.0.0.1 (this IP address was set default you are not have to write it)

```
C:\Windows\System32\cmd.exe - python  simple_client.py 127.0.0.1
Microsoft Windows [Version 10.0.22000.493]
(c) Microsoft Corporation. All rights reserved.

C:\Users\salis\PycharmProjects\projectYosfSali\Client_Side>python simple_client.py 127.0.0.1
Enter your name:
simpleUser
client simpleUser received message:simpleUser has joined the chat!
client simpleUser received message:ALL
--------------welcome simpleUser!--------------

The input you will type will send message to the server

The next inputs (inside the <>) will help you to make some activities with the server:

input:<~><name> <msg> to send a private message to client. (name- the name of the received client, msg- your message)

input: <show> to get the names of the connected clients. (every list starts with ALL, its not a client)

input: <get_list_file> to get the names of the files that you can download from the server.(every list starts with CHOOSE_FILE, its not a file name)

input: <download_file> <filename> to download a file from the server. (<filename>- the name of the file that you want to download)

input: <{quit}> to disconnect and exit.
any other input will be sent from you to all the connected clients as a public message.

--------------sali&yosef chatroom--------------
```

An example for chatting from simple_client.py:

```
C:\Windows\System32\cmd.exe - python  simple_client.py
--------------sali&yosef chatroom--------------

hey alll
client simpleUser received message:simpleUser: hey alll
show
client simpleUser received message:ALL
client simpleUser received message:anotherUser has joined the chat!
client simpleUser received message:ALL anotherUser
~anotherUser hey another user how are you?
client simpleUser received message:from you to anotherUser: (PRIVATE)  hey another user how are you?
client simpleUser received message:anotherUser: (PRIVATE)  im good how are you?
~anotherUser im good thanks
client simpleUser received message:from you to anotherUser: (PRIVATE)  im good thanks
client simpleUser received message:anotherUser: (PRIVATE)  have you already downloaded a file from the server?
~anotherUser no, how to do it?
client simpleUser received message:from you to anotherUser: (PRIVATE)  no, how to do it?
client simpleUser received message:anotherUser: (PRIVATE)  asked the server!!
~anotherUser ok
client simpleUser received message:from you to anotherUser: (PRIVATE)  ok
get_list_file
client simpleUser received message:CHOOSE_FILE alice.jpg bob&alice_Wireshark.pcapng bob.jpg Ex2.jar hello.c Pokemon
download_file hello.c
client simpleUser received message:UDP 55000 0 hello.c
0.0% downloaded
File was downloaded 100%

~anotherUser wow!! I have downloaded a C file right now
client simpleUser received message:from you to anotherUser: (PRIVATE)  wow!! I have downloaded a C file right now
~anotherUser thank you
client simpleUser received message:from you to anotherUser: (PRIVATE)  thank you
client simpleUser received message:anotherUser: (PRIVATE)  my plesure, bye bye
~anotherUser bye
client simpleUser received message:from you to anotherUser: (PRIVATE)  bye
client simpleUser received message:anotherUser left the chat...
client simpleUser received message:ALL
{quit}
```


Authors: Yosef Sokolov & Sali Sharfman

bob_alice_test.py

In addition to the regular executable files we added a unittest file named "bob_alice_test.py".

This file shows the use of chat by two users, Bob and Alice, while running the file, two client-type objects are opened, and they activate all communication functions with the server according to the system protocol, every received message append to one of 2 lists, list for bob's received messages and list for Alice's, after each Bob and Alice operation the test system will compare the expected messages with the messages actually entering the lists..

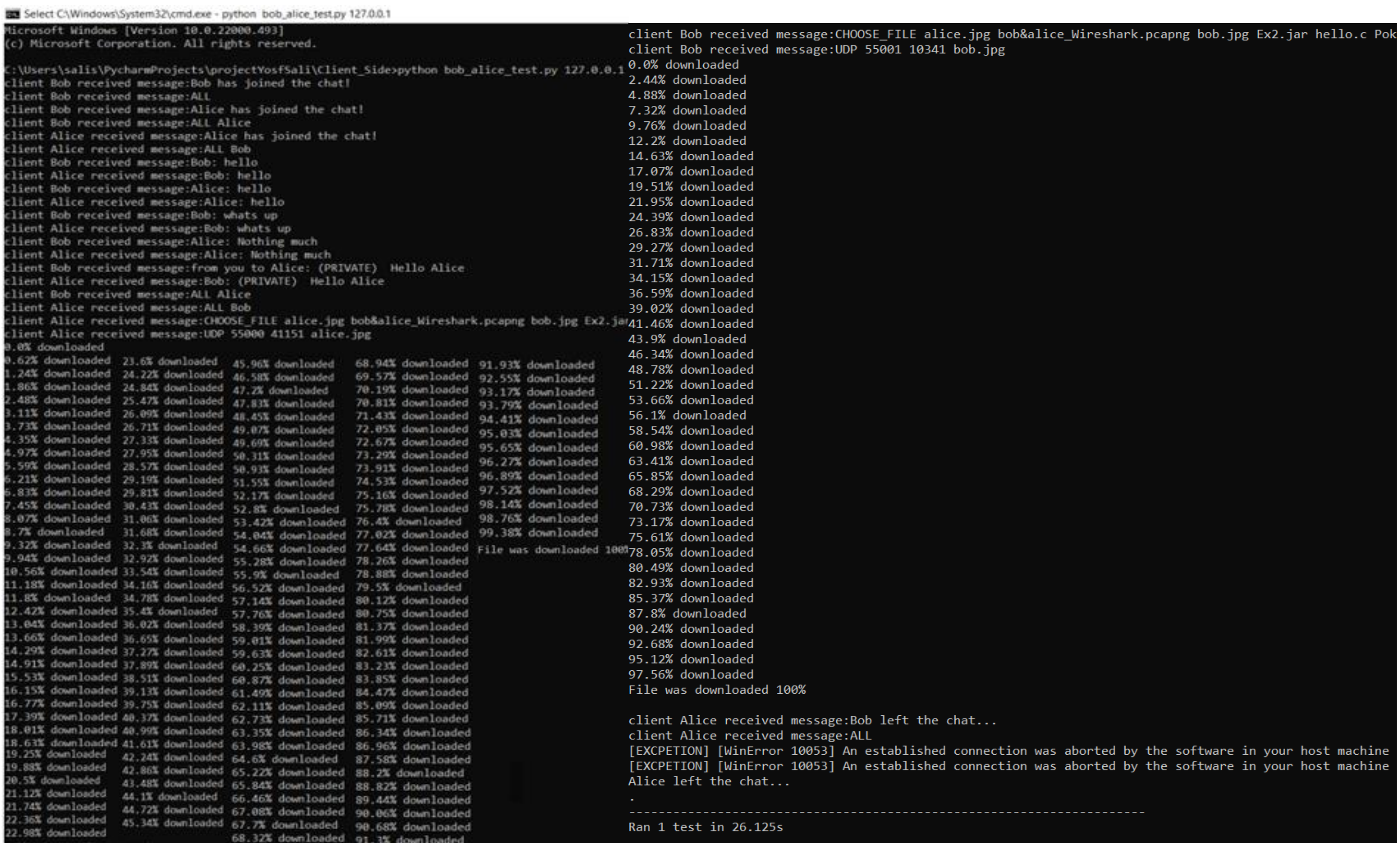
```python
# create client for bob without gui
bob = Client("Bob", False)
# create client for bob without gui
alice = Client("Alice", False)

bobmsg=[]
alicemsg=[]

def update_messages():...

class MyTestCase(unittest.TestCase):
    def setUp(self) -> None:
        # start thread for receive messages
        Thread(target=update_messages).start()

    def test_bob_alice(self):
        self.assertEqual('Bob has joined the chat!',bobmsg[0])
        self.assertEqual('Alice has joined the chat!', alicemsg[0])

        # Bob sends message to all
        bob.send_message("hello")
        time.sleep(2)
        self.assertEqual("Bob: hello", bobmsg[-1])
        self.assertEqual("Bob: hello", alicemsg[-1])

        # Alice sends message to all
        alice.send_message("hello")
        time.sleep(2)
        self.assertEqual("Alice: hello", alicemsg[-1])
        self.assertEqual("Alice: hello", bobmsg[-1])

        # Bob sends message to all
        bob.send_message("whats up")
        time.sleep(2)
        self.assertEqual("Bob: whats up", bobmsg[-1])
        self.assertEqual("Bob: whats up", alicemsg[-1])
```

```python
# Alice sends message to all
alice.send_message("Nothing much")
time.sleep(2)
self.assertEqual("Alice: Nothing much", alicemsg[-1])
self.assertEqual("Alice: Nothing much", bobmsg[-1])

# Bob sends message privately to Alice
bob.send_message("~Alice Hello Alice")
time.sleep(2)
self.assertEqual("from you to Alice: (PRIVATE)  Hello Alice", bobmsg[-1])
self.assertEqual("Bob: (PRIVATE)  Hello Alice", alicemsg[-1])

# Bob wants to know who is connected to the server
bob.send_message("show")
time.sleep(2)
self.assertEqual('ALL Alice ', bobmsg[-1])

# Alice wants to know who is connected to the server
alice.send_message("show")
time.sleep(2)
self.assertEqual('ALL Bob ', alicemsg[-1])

# Alice wants to know which files the server has for download
alice.send_message("get_list_file")
time.sleep(2)
self.assertEqual('CHOOSE_FILE', alicemsg[-1].split()[0])

# Alice download a file
alice.send_message("download_file alice.jpg")
time.sleep(3)
self.assertEqual('UDP', alicemsg[-1].split()[0])

# Bob wants to know which files the server has for download
bob.send_message("get_list_file")
time.sleep(2)
self.assertEqual('CHOOSE_FILE', bobmsg[-1].split()[0])
 # Bob download a file
 bob.send_message("download_file bob.jpg")
 time.sleep(3)
 self.assertEqual('UDP', bobmsg[-1].split()[0])


 # Bob disconnect
 bob.disconnect()
 time.sleep(2)
 self.assertEqual('Bob left the chat...', alicemsg[-2])
 self.assertEqual('ALL ',alicemsg[-1])


 # Alice disconnect
 alice.disconnect()
 print('Alice left the chat...')
```

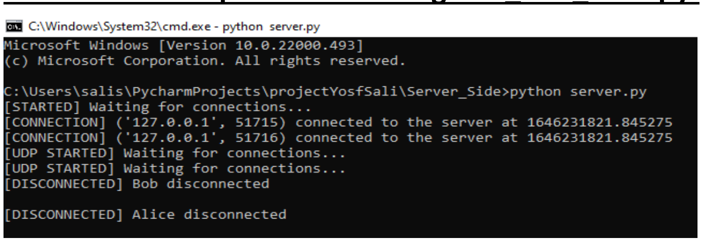
As you can see in the code comments, first Bob and Alice connect to the server, then Bob sends a public message "hello", Alice sends a public message "hello".
Bob sends a public message "whats up", Alice sends a public message "Nothing much", Bob Sends private message to Alice "Hello Alice", Bob sends "show" to get the list of connected users, Alice sends "show" to get the list of connected users, Alice sends "get_list_file" to get the list of files from the server, Alice sends "download_file alice .jpg "To download her image from the server, Bob sends" get_list_file "to get the list of files from the server, Bob sends" download_file bob.jpg "to download her image from the server, Bob sends" {quit} "to disconnect from the server and finally Alice sends {"quit"} to disconnect from the server.

test_bob_alice.py run

after running the server,
*Open cmd from “Client_side” folder.
*Write the next command “python test_bob_alice.py <IPv4>”
<IPv4>- the IPv4 address of the server that already run.
for example: python test_bob_alice.py 127.0.0.1 (this IP address was set default you are not have to write it)



The Server’s output after running test_bob_alice.py:

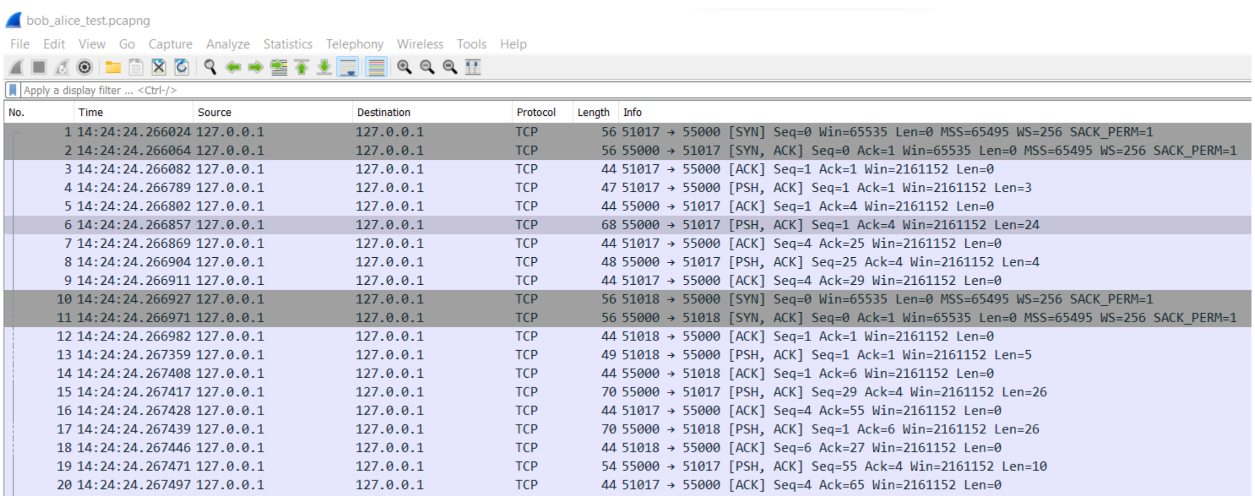


test_bob_alice.pcapng- Wireshark Record

While running the " test_bob_alice.py" program we will run Wireshark to capture the facts that are sent and received between the clients and the server.

You can find the full recording file named " test_bob_alice.pcapng" in the project files.
<u>Representative screenshots of selected parts of the recording file:</u>

<u>TCP traffic:</u>
No. 1- Bob (port 51017) send a connection request to the server (port 55000).
No. 2- server (port 55000) send ACK message to Bob (port 51017).
No. 3-9- TCP messages and ACK between server and BOB.
No. 10- Alice (port 51018) send a connection request to the server (port 55000).
No. 11- server (port 55000) send ACK message to Alice (port 51018).
No. 12-68- messages and ACK between server and Alice or server and BOB.


bob_alice_test.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	14:24:24.266024	127.0.0.1	127.0.0.1	TCP	56	51017 → 55000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
2	14:24:24.266064	127.0.0.1	127.0.0.1	TCP	56	55000 → 51017 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
3	14:24:24.266082	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
4	14:24:24.266789	127.0.0.1	127.0.0.1	TCP	47	51017 → 55000 [PSH, ACK] Seq=1 Ack=1 Win=2161152 Len=3
5	14:24:24.266802	127.0.0.1	127.0.0.1	TCP	44	55000 → 51017 [ACK] Seq=1 Ack=4 Win=2161152 Len=0
6	14:24:24.266857	127.0.0.1	127.0.0.1	TCP	68	55000 → 51017 [PSH, ACK] Seq=1 Ack=4 Win=2161152 Len=24
7	14:24:24.266869	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [ACK] Seq=4 Ack=25 Win=2161152 Len=0
8	14:24:24.266904	127.0.0.1	127.0.0.1	TCP	48	55000 → 51017 [PSH, ACK] Seq=25 Ack=4 Win=2161152 Len=4
9	14:24:24.266911	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [ACK] Seq=4 Ack=29 Win=2161152 Len=0
10	14:24:24.266927	127.0.0.1	127.0.0.1	TCP	56	51018 → 55000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
11	14:24:24.266971	127.0.0.1	127.0.0.1	TCP	56	55000 → 51018 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
12	14:24:24.266982	127.0.0.1	127.0.0.1	TCP	44	51018 → 55000 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
13	14:24:24.267359	127.0.0.1	127.0.0.1	TCP	49	51018 → 55000 [PSH, ACK] Seq=1 Ack=1 Win=2161152 Len=5
14	14:24:24.267408	127.0.0.1	127.0.0.1	TCP	44	55000 → 51018 [ACK] Seq=1 Ack=6 Win=2161152 Len=0
15	14:24:24.267417	127.0.0.1	127.0.0.1	TCP	70	55000 → 51017 [PSH, ACK] Seq=29 Ack=4 Win=2161152 Len=26
16	14:24:24.267428	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [ACK] Seq=4 Ack=55 Win=2161152 Len=0
17	14:24:24.267439	127.0.0.1	127.0.0.1	TCP	70	55000 → 51018 [PSH, ACK] Seq=1 Ack=6 Win=2161152 Len=26
18	14:24:24.267446	127.0.0.1	127.0.0.1	TCP	44	51018 → 55000 [ACK] Seq=6 Ack=27 Win=2161152 Len=0
19	14:24:24.267471	127.0.0.1	127.0.0.1	TCP	54	55000 → 51017 [PSH, ACK] Seq=55 Ack=4 Win=2161152 Len=10
20	14:24:24.267497	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [ACK] Seq=4 Ack=65 Win=2161152 Len=0

No. 69-1035- UDP traffic between Alice (port 59120) and server (port 55000) for download an image.

No.	Time	Source	Destination	Protocol	Length	Info
61	14:24:38.347331	127.0.0.1	127.0.0.1	TCP	57	51018 → 55000 [PSH, ACK] Seq=27 Ack=124 Win=2161152 Len=13
62	14:24:38.347405	127.0.0.1	127.0.0.1	TCP	44	55000 → 51018 [ACK] Seq=124 Ack=40 Win=2161152 Len=0
63	14:24:38.347888	127.0.0.1	127.0.0.1	TCP	164	55000 → 51018 [PSH, ACK] Seq=124 Ack=40 Win=2161152 Len=120
64	14:24:38.347945	127.0.0.1	127.0.0.1	TCP	44	51018 → 55000 [ACK] Seq=40 Ack=244 Win=2160896 Len=0
65	14:24:40.352229	127.0.0.1	127.0.0.1	TCP	67	51018 → 55000 [PSH, ACK] Seq=40 Ack=244 Win=2160896 Len=23
66	14:24:40.352295	127.0.0.1	127.0.0.1	TCP	44	55000 → 51018 [ACK] Seq=244 Ack=63 Win=2161152 Len=0
67	14:24:40.352988	127.0.0.1	127.0.0.1	TCP	69	55000 → 51018 [PSH, ACK] Seq=244 Ack=63 Win=2161152 Len=25
68	14:24:40.353047	127.0.0.1	127.0.0.1	TCP	44	51018 → 55000 [ACK] Seq=63 Ack=269 Win=2160896 Len=0
69	14:24:40.354796	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
70	14:24:40.356845	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
71	14:24:40.357089	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
72	14:24:40.357334	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
73	14:24:40.357539	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
74	14:24:40.357744	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
75	14:24:40.358020	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
76	14:24:40.358212	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
77	14:24:40.358396	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
78	14:24:40.358589	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
79	14:24:40.358770	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
80	14:24:40.358965	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
81	14:24:40.359207	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
82	14:24:40.359400	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
83	14:24:40.359586	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
84	14:24:40.359779	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
85	14:24:40.359962	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
86	14:24:40.360155	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
87	14:24:40.360395	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
88	14:24:40.360587	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
89	14:24:40.360773	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
90	14:24:40.360965	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
91	14:24:40.361150	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
92	14:24:40.361341	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018
93	14:24:40.361581	127.0.0.1	127.0.0.1	UDP	38	59120 → 55000 Len=6
94	14:24:40.361776	127.0.0.1	127.0.0.1	UDP	1050	55000 → 59120 Len=1018

No. 1036-1043- TCP messages and ACK between server and BOB.

1035	14:24:40.392064	127.0.0.1	127.0.0.1	UDP	40	59120 → 55000 Len=8
1036	14:24:43.367150	127.0.0.1	127.0.0.1	TCP	57	51017 → 55000 [PSH, ACK] Seq=39 Ack=170 Win=2161152 Len=13
1037	14:24:43.367240	127.0.0.1	127.0.0.1	TCP	44	55000 → 51017 [ACK] Seq=170 Ack=52 Win=2161152 Len=0
1038	14:24:43.367721	127.0.0.1	127.0.0.1	TCP	164	55000 → 51017 [PSH, ACK] Seq=170 Ack=52 Win=2161152 Len=120
1039	14:24:43.367768	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [ACK] Seq=52 Ack=290 Win=2160896 Len=0
1040	14:24:45.372639	127.0.0.1	127.0.0.1	TCP	65	51017 → 55000 [PSH, ACK] Seq=52 Ack=290 Win=2160896 Len=21
1041	14:24:45.372727	127.0.0.1	127.0.0.1	TCP	44	55000 → 51017 [ACK] Seq=290 Ack=73 Win=2161152 Len=0
1042	14:24:45.373435	127.0.0.1	127.0.0.1	TCP	67	55000 → 51017 [PSH, ACK] Seq=290 Ack=73 Win=2161152 Len=23
1043	14:24:45.373498	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [ACK] Seq=73 Ack=313 Win=2160896 Len=0
1044	14:24:45.375031	127.0.0.1	127.0.0.1	UDP	38	59121 → 55001 Len=6

No. 1044-1290- UDP traffic between Bob (port 59121) and server (port 55001) for download an image.

No. 1291-1292- TCP message and ACK from Bob to server

No. 1293-1295- TCP messages between Bob and server, Bob is disconnecting.

No. 1296-1301- TCP message and ACK from Alice to server

No. 1302- TCP message from Alice to server, Alice is disconnecting

1289	14:24:45.392504	127.0.0.1	127.0.0.1	UDP	47	55001 → 59121 Len=15
1290	14:24:45.392651	127.0.0.1	127.0.0.1	UDP	40	59121 → 55001 Len=8
1291	14:24:48.387567	127.0.0.1	127.0.0.1	TCP	50	51017 → 55000 [PSH, ACK] Seq=73 Ack=313 Win=2160896 Len=6
1292	14:24:48.387638	127.0.0.1	127.0.0.1	TCP	44	55000 → 51017 [ACK] Seq=313 Ack=79 Win=2161152 Len=0
1293	14:24:48.387720	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [RST, ACK] Seq=79 Ack=313 Win=0 Len=0
1294	14:24:48.387726	127.0.0.1	127.0.0.1	TCP	44	55000 → 51017 [FIN, ACK] Seq=313 Ack=79 Win=2161152 Len=0
1295	14:24:48.387763	127.0.0.1	127.0.0.1	TCP	44	51017 → 55000 [RST] Seq=79 Win=0 Len=0
1296	14:24:48.387854	127.0.0.1	127.0.0.1	TCP	64	55000 → 51018 [PSH, ACK] Seq=269 Ack=63 Win=2161152 Len=20
1297	14:24:48.387896	127.0.0.1	127.0.0.1	TCP	44	51018 → 55000 [ACK] Seq=63 Ack=289 Win=2160896 Len=0
1298	14:24:48.387988	127.0.0.1	127.0.0.1	TCP	48	55000 → 51018 [PSH, ACK] Seq=289 Ack=63 Win=2161152 Len=4
1299	14:24:48.388011	127.0.0.1	127.0.0.1	TCP	44	51018 → 55000 [ACK] Seq=63 Ack=293 Win=2160896 Len=0
1300	14:24:50.392998	127.0.0.1	127.0.0.1	TCP	50	51018 → 55000 [PSH, ACK] Seq=63 Ack=293 Win=2160896 Len=6
1301	14:24:50.393040	127.0.0.1	127.0.0.1	TCP	44	55000 → 51018 [ACK] Seq=293 Ack=69 Win=2161152 Len=0
1302	14:24:50.393082	127.0.0.1	127.0.0.1	TCP	44	51018 → 55000 [RST, ACK] Seq=69 Ack=293 Win=0 Len=0

Exception handling

Name validity:

A name can not contain characters that do not count letters or _. - or the word ALL so that a private message can be sent to each input given as a name.

File list:

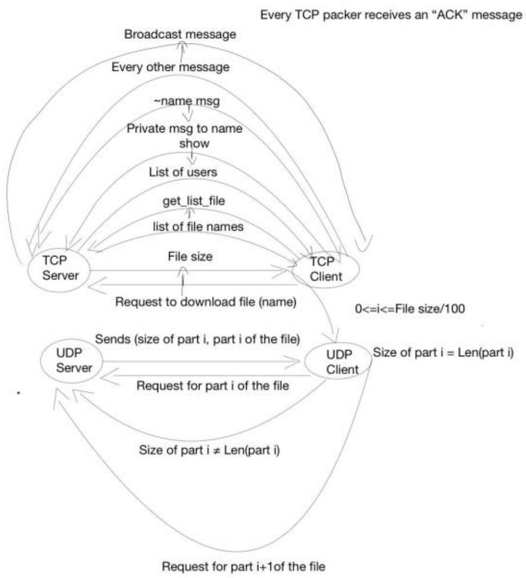
The first part of the file list and the keyword on the server that the client knows is a file list is <CHOOSE_FILE> We used characters <> intentionally because these characters can not appear in a file name and this string will never be a file name.

Open Folders Downloads and Logs:

On the server side we write files to folders named Downloads and Logs so before writing the files, if the required folder does not exist the system opens folders in these
names

Part B - Questions&Answers

1. Situation diagram



2. How the system overcomes lost packages?

When downloading a file from the server, we want to make sure that there is no loss of packages because every slightest mistake in the information will result in the file not being able to be opened. For this reason we must be sure that if the client writes a piece of data to a file he will write the whole piece in its entirety as sent from the server. To this end, we implemented a method for RDT over a UDP connection plus Congestion control, The method we wrote is, in every UDP packet sent from the server, the size of the information attached to the packet is also sent, each time the client receives such a message, he enters a loop in which he compares the size of the information sent from the server with the size of the information sent from the client, repeat the loop again until the sizes are equal, when they are equal, we will exit the loop and write the information received to the file.

3. How the system overcomes latency issues

The system overcomes latency problems in that each client opens a separate process for receiving messages and so in case a particular customer has a problem receiving messages he does not delay the whole system, in addition, each file download opens a separate process and it happens on a separate port and socket and so files can be downloaded simultaneously in several Customers and while a customer is downloading Correspondence can be made from different customers at this time and even from the customer who downloads himself.

 Authors: Yosef Sokolov & Sali Sharfman

Part C – Questions & Answers

1. Given a new computer connecting to the network please describe all the

messages that pass from the initial connection to the switch until the message is received on the other side of the chat.

So computer 'Alice' needs MAC address of computer 'Bob' in order to send him message. First of all computer 'Alice' will check if MAC address of computer 'Bob' exists in its internal cache, if it is – excellent, if isn't computer 'Alice' will send ARP message to the internal network with MAC address as public. Once a computer is connected to the switch, it sends ARP message in order to create connection with other side (second computer). APR – is a broadcast message that sends to all end units under the same broadcast domain.

In the message Alice' asks to find MAC address of IP of 'Bob'. All computers in the network will see the message. Computer 'Bob' will see the message and response to it with his MAC address and IP address through ARP protocol. computer 'Alice' receives the message in its internal cache.

If the target is not under the same Broadcast Domain(network).

The message is sent under the same Broadcast Domain and if the end unit is not under the same network it will forward the message to a third layer component used as the Default Gateway in order to route the message to the appropriate destination.

Once the router recognizes the leg of the target's network, it will enter the leg's MAC address as the Source MAC so that the target can return the answer and it will forward the message to the source according to the Source IP in the Unicast message.

2. Explain what C is

CRC- cyclic redundancy check- is an additive part added to the packet to check that there are no changes in the packet. For example checksum: simple test to check the sum of the packets to see that the sum is the same sum as we expected.

3. What are the differences between http 1.0, http 1.1, http 2.0, QUIC?

In **HTTP 1.0** you had to open a new connection for each request/response pair. And after each response the connection would be closed. This lead to some big efficiency problems because of TCP Slow Start.

HTTP 1.1 allows you to have persistent connections which means that you can have more than one request/response on the same HTTP connection. Also HTTP 1.1 loads resources one after the other, so if one resource cannot be loaded, it blocks all the other resources behind it.

HTTP 2.0 is able to use a single TCP connection to send multiple streams of data at once so that no one resource blocks any other resource so it aims to be a faster.

QUIC - transport layer network protocol developed by Google. QUIC runs over connectionless UDP instead of the connection-oriented TCP.

 Authors: Yosef Sokolov & Sali Sharfman

4. Why do you need port numbers?

A port number is a way to identify a specific process to which an internet or other network message is to be forwarded when it arrives at a server. All network-connected devices come equipped with standardized ports that have an assigned number. A port is simply a channel of communication which is numbered between 1 and 65000. HTTP traffic will pass through port number 80. The ports are divided into ranges in each range defined what is in it 0-1023 it is for well-known ports, 1024-49151 for registered ports and the rest up to 65545 for private or dynamic

5. What is a subnet and why do you need it?

When we are in an internal network there is a sub-network. What the subnet gives us is the option to save ip addresses because there are especially final addresses in ipv4 that have a large amount but not enough so it is a solution to overcome the shortage by dividing into subnets and each subnet will have ip addresses, so there may be two people in different networks but their internal ip address will be the same.

6. Why do you need mac addresses Why is it not enough to work with IP addresses?

MAC Addresses handle the physical connection from computer to computer while IP Addresses handle the logical routeable connection from both computer to computer AND network to network. THE local IP address will dynamically change when you are disconnected from the WIFI network or ethernet. (sometimes it's not changing because that anyone else is not connected to the network to reserve your recent IP). so MAC address is a unique address for your network card. We can uniquely identify your device from it. It cannot be changed but nowadays also can change it.

7. What are the differences between Router Switch Nat

Switch - A hardware component that knows up to the link layer that knows how to map what is connected to it and transmit information according to what is predefined, the component isn’t smart

Router - is a smart component that recognizes up to the third layer whose function is to route the information and transmit the facts in the smoothest and best way, the router works even in varying conditions in the field.

Nat - A router that allows dynamic address assignment to computers under it on the network. The NAT will store information about the computers that gives them a temporary address so that the information can return to the right place

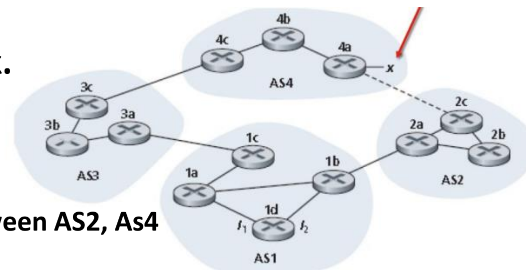
8. What are the methods to overcome the lack of IPv4? detail

To overcome the lack of ipv4 there is the CIDR protocol that allows division of the network into subnets so that it will be possible to assign different ranges to each subnet.

In addition, there is the NAT - which allows dynamic assignments, the assignment of temporary IP addresses that are changed in order to allow re-use of the same address.

In addition, we can move to ipv6 - longer IP addresses

9. Given the following network.



a.AS2, AS3 run OSPF
b.AS1, AS4 run RIP
c. Between the Ass runs BGP
d. There is no physical connection between AS2, As4

e. Using what protocol the router 3c learns about x?
Using the **BGP** protocol the router **3c** learns about the subnet x

f. Using what protocol the router 3a learns about x?
Using the **OSPF** protocol the router **3a** learns about the subnet x

g. Using what protocol the router 1c learns about x?
Using the **BGP** protocol the router **1c** learns about the subnet x

h. Using what protocol the router 2c learns about x?
Using the **OSPF** protocol the router **2c** learns about the subnet x