ITU COMPUTER ENGINEERING

2020 FALL COMPUTER PROJECT 1

PROJECT 2 REPORT

06.12.2020

Sercan AYDIN **150170707**

Index

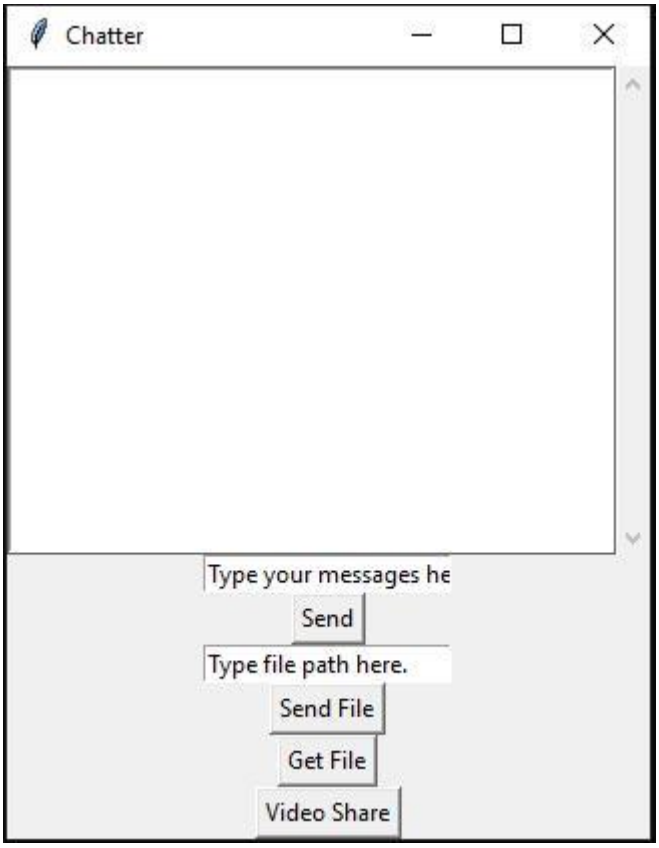
Introduction

In second Project we're expected to develop a software that can people use it to communicate each other in an online platform. This application must have several properties like chatting, file transfering, video sharing, and screen sharing. Also in this project we should consider the privacy laws and when developing we should try not to breach peoples privacy.

In this project we mainly used the python library which is called "Socket" for the communication. In architecture of the project, we used Server layer and Client layer. Socket library helped us to communicate the server and client with its methods like "recv" and "send".

We used a library named "Tkinter" for building a front-end to our project. We can easily manipulate the elements in the front-end with tkinter. We added labels, text areas, buttons and we connected them with our methods.

When building this project, we used some communication apps for pair development. We used Discord and Zoom for this tasks. When we using it we used these applications as a sample to our project. Mainly our project front-end is shown in the figure.



Project Description

We used “tkinter” for our porjects front-end. We implemented it like shown in the figure.

```
top = tkinter.Tk()
top.title("Chatter")

messages_frame = tkinter.Frame(top)
my_msg = tkinter.StringVar()  # For the messages to be sent.
my_msg.set("Type your messages here.")

file_name = tkinter.StringVar()
file_name.set("Type file path here.")

scrollbar = tkinter.Scrollbar(messages_frame)  # To navigate through past messages.
# Following will contain the messages.
msg_list = tkinter.Listbox(messages_frame, height=15, width=50, yscrollcommand=scrollbar.set)
scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)
msg_list.pack(side=tkinter.LEFT, fill=tkinter.BOTH)
msg_list.pack()
messages_frame.pack()

entry_field = tkinter.Entry(top, textvariable=my_msg)
entry_field.bind("<Return>", send)
entry_field.pack()
send_button = tkinter.Button(top, text="Send", command=send)
send_button.pack()

entry_file = tkinter.Entry(top, textvariable=file_name)
entry_file.bind("<Return>", file_name)
entry_file.pack()
send_file_button = tkinter.Button(top, text="Send File", command=sendFile)
send_file_button.pack()

get_file_button = tkinter.Button(top, text="Get File", command=getFile)
get_file_button.pack()

top.protocol("WM_DELETE_WINDOW", on_closing)
```

When we developing the project, we divide the porject into sub tasks. These task are:

1) Chat
2) File transfering
3) Video Sharing
4) Screen Sharing

1) Chat

I created a chat server through which can recieve incoming requests from clients wanting to communicate. For this, I used good ole' sockets and a bit of multithreading. Using frameworks like SocketServer was an option. We will be using TCP sockets for this purpose, and therefore we use AF_INET and SOCK_STREAM flags. We use them over UDP sockets because they're more telephonic, where the recipient has to approve the incoming connection before communication begins, and UDP sockets are more post-mail sort of thing (anyone can send a mail to any recipient whose address s/he knows), so they don't really require an establishment of connection before communication can happen.

```
def accept_incoming_connections():
    """Sets up handling for incoming clients."""
    while True:
        client, client_address = SERVER.accept()
        print("%s:%s has connected." % client_address)
        client.send(bytes("Greetings from the cave! Now type your name and press enter!", "utf8"))
        addresses[client] = client_address
        Thread(target=handle_client, args=(client,)).start()


def handle_client(client):  # Takes client socket as argument.
    """Handles a single client connection."""

    name = client.recv(BUFSIZ).decode("utf8")
    print(name)
    welcome = 'Welcome %s! If you ever want to quit, type {quit} to exit.' % name
    client.send(bytes(welcome, "utf8"))
    msg = "%s has joined the chat!" % name
    broadcast(bytes(msg, "utf8"))
    clients[client] = name
```

Clearly, TCP suit more to our purpose than UDP sockets, therefore we use them. We break our task of serving into accepting new connections, broadcasting messages and handling particular clients. after we send the new client the welcoming message, it will reply with the name s/he wants to use for further communication. In the handle_client() function, the first task we do is we save this name, and then send another message to the client, regarding further instructions. After this comes the main loop for communication: here we recieve further messages from the client and if a message doesn't contain instructions to quit, we simply broadcast the messsage to other connected clients (we'll be defining the broadcast method in a moment).

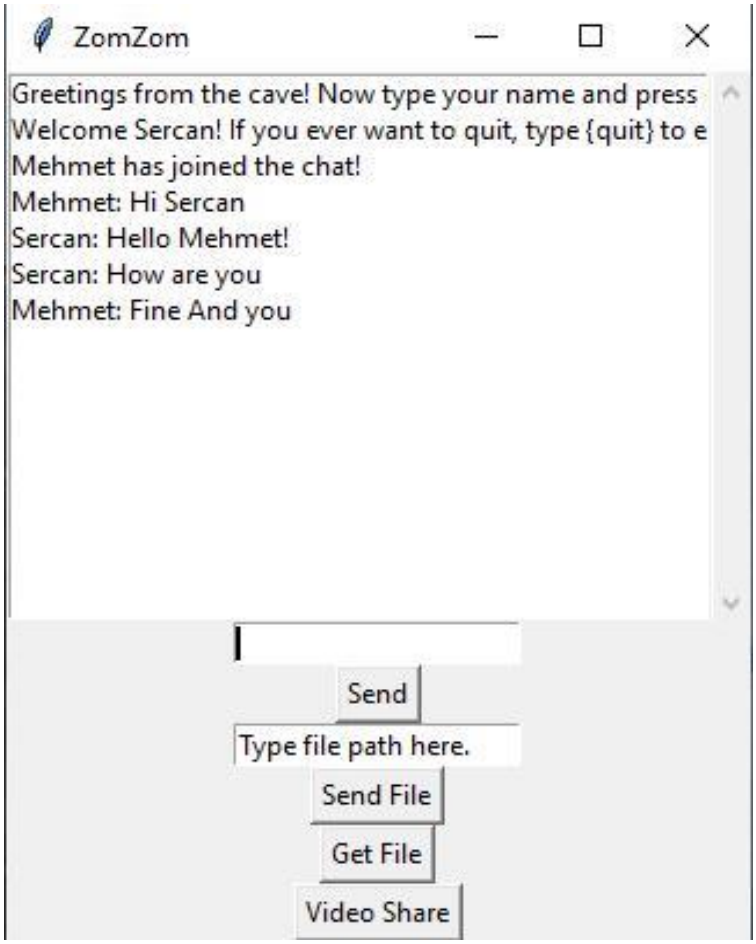
If we do encounter a message with exit instructions (i.e., the client sends a {quit}), we echo back the same message to the client (it triggers close action on the client side) and then we close the connection socket for it.

```
def receive():
    """Handles receiving of messages."""
    while True:
        try:
            msg = client_socket.recv(BUFSIZ).decode("utf8")
            msg_list.insert(tkinter.END, msg)
        except OSError:  # Possibly client has left the chat.
            break


def send(event=None):  # event is passed by binders.
    """Handles sending of messages."""
    msg = my_msg.get()
    my_msg.set("")  # Clears input field.
    client_socket.send(bytes(msg, "utf8"))
    if msg == "{quit}":
        client_socket.close()
        top.quit()
```

We then do some cleanup by deleting the entry for the client, and finally give a shoutout to other connected people that this particular person has left the conversation. Broadcast function simply sends the msg to all the connected clients, and prepends an optional prefix if necessary. We do pass a prefix to broadcast() in our handle_client() function, and we do it so that people can see exactly who is the sender of a particular message.

We're using event as an argument because it is implicitly passed by Tkinter when the send button on the GUI is pressed. my_msg is the input field on the GUI, and therefore we extract the message to be sent usin g msg = my_msg.get(). After that, we clear the input field and then send the message to the server, which, as we've seen before, broadcasts this message to all the clients (if it's not an exit message). If it is an exit message, we close the socket and then the GUI app (via top.close()).We define one more function, which will be called when we choose to close the GUI window. It is a sort of cleanup-before-close function and shall close the socket connection before the GUI closes: we create the input field for the user to input their message, and bind it to the string variable defined above.

We also bind it to the send() function so that whenever the user presses return, the message is sent to the server. Next, we create the send button if the user wishes to send their messages by clicking on it. Again, we bind the clicking of this button to the send() function. And yes, we also pack all this stuff we created just now. Furthermore, don't forget to make use of the cleanup function on_closing() which should be called when the user wishes to close the GUI window



2) File Transfer

In this part we also used “Socket” tool of python. When a user connected and pressed the according buttons which is send file or get file in the front-end he/she can send or recieve files with the application after typing the name of the sending or recieving file into the file path area.

In the client part we used sendFile and GetFile methods. These methods are responsible for transferring the files and get connection with server side.

```
def getFile(event=None):
    gettingfileName = "getFile" + file_name.get()
    client_socket.send(bytes(gettingfileName,"utf-8"))
    print("sended")
    confirmation = client_socket.recv(1024)
    print("confirmed",confirmation.decode())
    if confirmation.decode() == "file-doesn't-exist":
        print("File doesn't exist on server.")


    else:
        write_name = 'from_server '+ gettingfileName
        if os.path.exists(write_name): os.remove(write_name)

        f = open(write_name,"w")
        f.write(confirmation.decode())
        f.close()

        #with open(write_name,'wb') as file:
        #    print("writing")
        #    file.write(confirmation.decode())

        #print(gettingfileName,'successfully downloaded.')
```

We need to encode and the code the messages to transfer it from byte to string and write it to the file.

In the server part we implemented it like shown in the figure.

```
while True:
    msg = client.recv(BUFSIZ)
    print(msg)
    if msg.decode()[0:7] == "getFile":
        msg=msg.decode()[7:]
        print("MSG:",msg)
        print("getFile")
        if not os.path.exists(msg):
            print("doesnt exist file")
            client.send("file-doesn't-exist".encode())

        else:
            print("file Exists")
            client.send("file-exists".encode())
            print('Sending',msg)
            if msg != '':
                file = open(msg,'rb')
                dosyaici = file.read(1024)
                print("Dosya ici:", dosyaici, type(dosyaici))
                while dosyaici:
                    client.send(dosyaici)
                    dosyaici = file.read(1024)

                client.close()
```

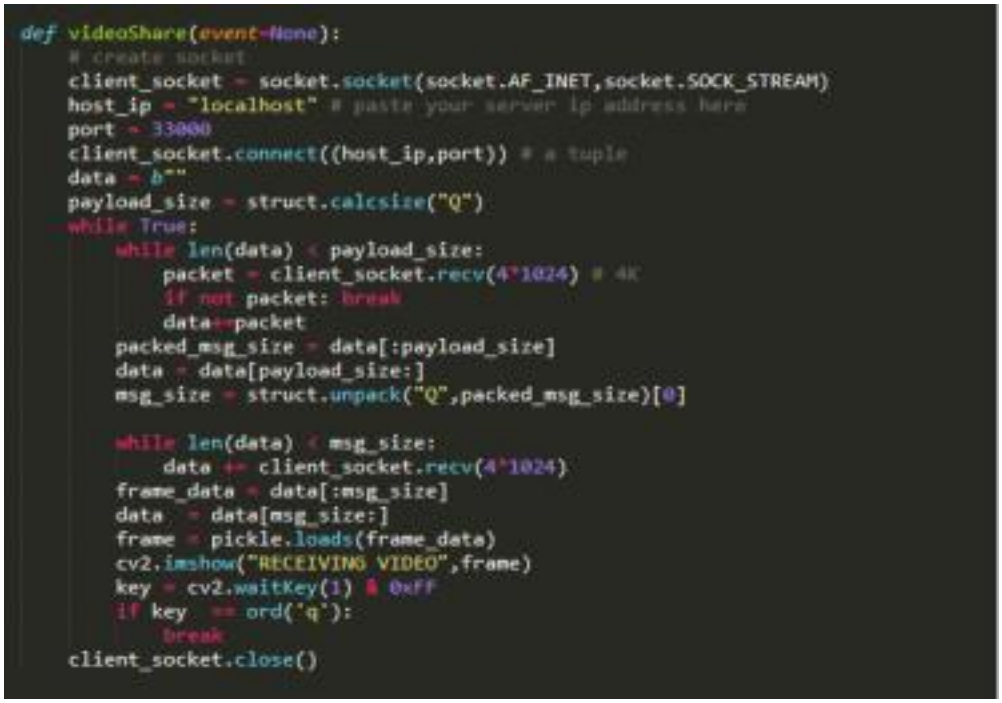
We controlled it if the message is file or not. If is it file we started the sending file procedures. In the end a file named “fromserverMytext.txt” file downloaded from server to local folder which is working directory of the client.

3)Video Sharing

We will run python codes for server and by using opencv we will extract video of the server's webcam and then send it to the client. The server and client modules can either run on the same computer or to separate computers that are connected to the wi-fi. In the standard internet protocols a socket address is the combination of an ip address and a port number much like a telephone address which is the combination of a phone number and a particular extension.

```
# Socket Create
server_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
host_name  = socket.gethostname()
host_ip = "localhost"
print('HOST IP:',host_ip)
port = 33000
socket_address = (host_ip,port)

# Socket Bind
server_socket.bind(socket_address)

# Socket Listen
server_socket.listen(5)

# Socket Accept
while True:
    client_socket,addr = server_socket.accept()
    print('GOT CONNECTION FROM:',addr)
    if client_socket:
        vid = cv2.VideoCapture(0)

        while(vid.isOpened()):
            img,frame = vid.read()
            frame = imutils.resize(frame,width=320)
            a = pickle.dumps(frame)
            message = struct.pack("Q",len(a))+a
            client_socket.sendall(message)

            cv2.imshow('TRANSMITTING VIDEO',frame)
            key = cv2.waitKey(1) & 0xFF
            if key ==ord('q'):
                client_socket.close()
```

There are several types of internet sockets such as connectionless based on user datagram protocol or UDP. In UDP each packet is sent and received individually without a specific sequence therefore the order and reliability are not guaranteed. On the other hand in connection oriented sockets such as in transmission control protocol or TCP the packets are sent in a sequence or stream therefore we call them stream sockets the order and reliability is also guaranteed.

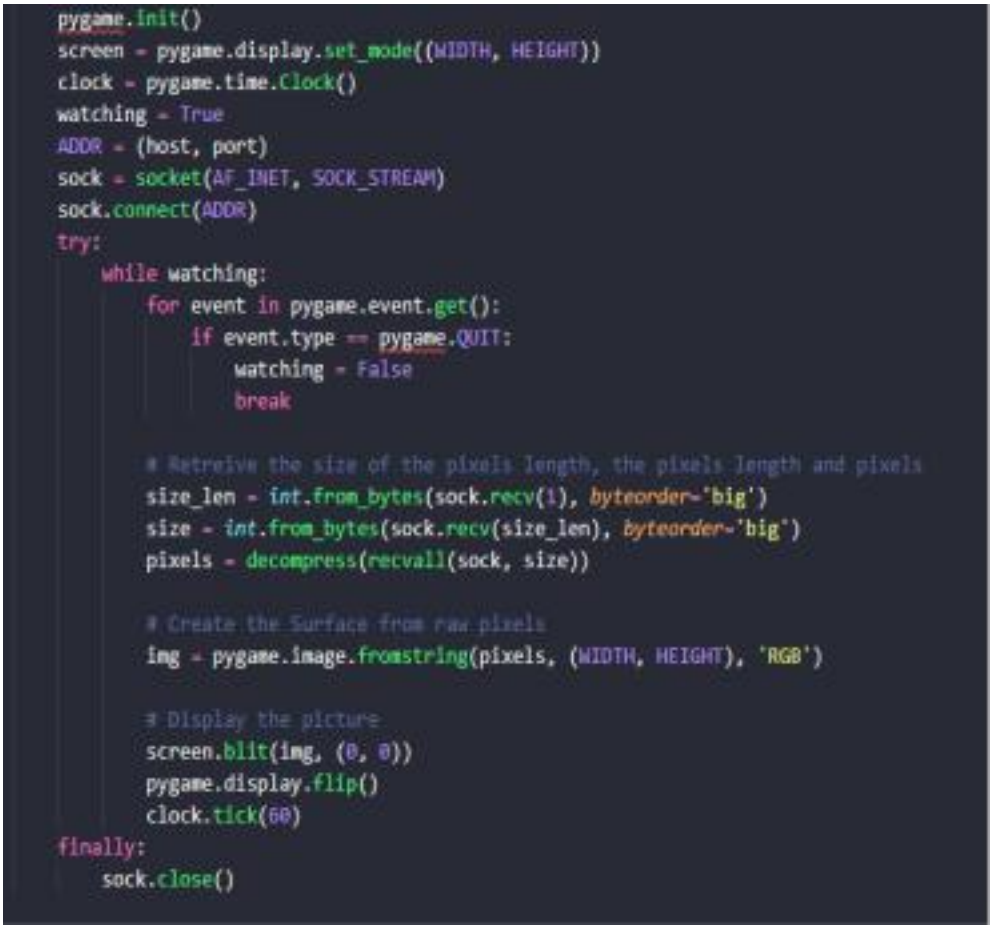


```
def videoShare(event=None):
    # create socket
    client_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    host_ip = "localhost" # paste your server ip address here
    port = 33000
    client_socket.connect((host_ip,port)) # a tuple
    data = b""
    payload_size = struct.calcsize("Q")
    while True:
        while len(data) < payload_size:
            packet = client_socket.recv(4*1024) # 4K
            if not packet: break
            data+=packet
        packed_msg_size = data[:payload_size]
        data = data[payload_size:]
        msg_size = struct.unpack("Q",packed_msg_size)[0]

        while len(data) < msg_size:
            data += client_socket.recv(4*1024)
        frame_data = data[:msg_size]
        data  = data[msg_size:]
        frame = pickle.loads(frame_data)
        cv2.imshow("RECEIVING VIDEO",frame)
        key = cv2.waitKey(1) & 0xFF
        if key  == ord('q'):
            break
    client_socket.close()
```

We use the stream sockets now let's come to the client server communication model the server initially creates a socket it then goes to the listening mode at the beginning it does not require ip address of the client however the client must know the ip address and the port number of the server to start communication.

4)Screen Sharing

In this sub-task we are implemented the screen share part like shown in the next figure.



```
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
clock = pygame.time.Clock()
watching = True
ADDR = (host, port)
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(ADDR)
try:
    while watching:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                watching = False
                break

        # Retreive the size of the pixels length, the pixels length and pixels
        size_len = int.from_bytes(sock.recv(1), byteorder='big')
        size = int.from_bytes(sock.recv(size_len), byteorder='big')
        pixels = decompress(recvall(sock, size))

        # Create the Surface from raw pixels
        img = pygame.image.fromstring(pixels, (WIDTH, HEIGHT), 'RGB')

        # Display the picture
        screen.blit(img, (0, 0))
        pygame.display.flip()
        clock.tick(60)
finally:
    sock.close()
```

We used decompress library tool to decompressing the pixels. After the program runned we can share screens in the given resolution rates. For this example we used 800x600 pixe

CONCLUSION

Thanks to this project, we understanded how to use the main functions and methods in Python's socket module to write your own client - server applications. This includes showing us how to use a custom class to send messages and data between endpoints that you can build upon and utilize for your own applications. We looked at the low-level socket API in Python's socket module and saw how it can be used to create client-server applications. We also created our own custom class and used it as an application-layer protocol to exchange messages and data between endpoints. This project has given us the information, examples, and inspiration needed to start you on your sockets development journey.