INTRODUCTION TO COMPUTER NETWORKS

"LIVE MEETING AND TIC-TAC-TOE GAME WITH SOCKET PROGRAMMING"

A THESIS

SUBMITTED BY

VENKATA SATYA (CB.EN.U4AIE22005)

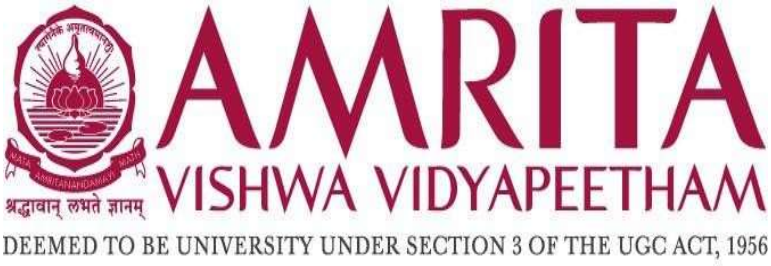
KOLLA LOKESH (CB.EN.U4AIE22027)

GOLI SURYA TEJA (CB.EN.U4AIE22069)

RAMA KRISHNA PRASAD (CB.EN.U4AIE22070)

IN PARTIAL FULFILLMENT FOR THE AWARD OF THE DEGREE OF

BACHELOR OF TECHNOLOGY IN CSE (AI)



CENTRE FOR COMPUTATIONAL ENGINEERING AND NETWORKING

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITAVISHWAVIDYAPEETHAM

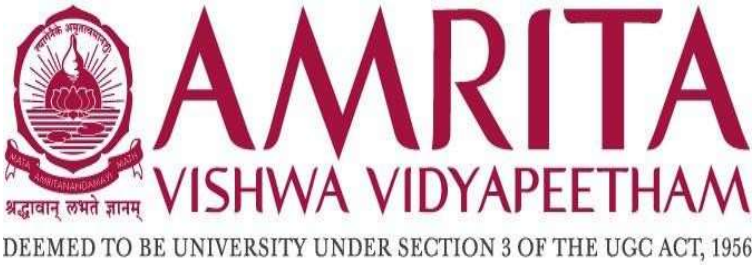
COIMBATORE-641112(INDIA)

DECEMBER-2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE-641112



BONAFIDE CERTIFICATE

This is to certify that the thesis entitled "**LIVE MEETING AND TIC-TAC-TOE GAME WITH SOCKET PROGRAMMING**" submitted by group-12 of batch(A), for the award of the Degree of Bachelor of Technology in the "CSE(AI) " is a bonafide record of the work carried out by us under our faculty guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Mrs.Ganga gowri B

Project Guide

Dr. K.P.Soman

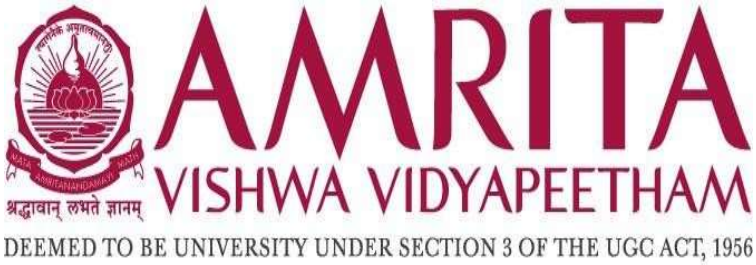
Professor and Head of CEN

Submitted for the university examination held on 19-12-2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE-641112



DECLARATION

We,group-12 of batch(A),hereby declare that this thesis entitled "**LIVE MEETING AND TIC-TAC-TOE GAME WITH SOCKET PROGRAMMING**" is the record of the original workdone by us under the guidance of, Assistant Professor, Centre for Computational Engineering and Networking, Amrita School of Artificial Intelligence, Coimbatore. To the best of my knowledge this work has not formed the basis for the award of any degree/diploma/ associate ship/fellowship/or a similar award to any candidate in any University.

Place:Coimbatore

Date:19-12-2023 **Signature of the Student**

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

We would like to express our special thanks of gratitude to our teacher Mrs Ganga gowri.B ma'am, who gave us the golden opportunity to do this wonderfull project on the topic "**LIVE MEETING AND TIC-TAC-TOE GAME WITH SOCKET PROGRAMMING**",which also helped usin doing a lot of Research and we came to know about so many new things. We are thankful for the opportunity given.We would also like to thank our group members,as with out their cooperation,we would not have been able to complete the project within the prescribed time.

ABSTRACT

This project presents a flexible real-time collaboration platform for Tic-Tac-Toe games and live meetings that runs on Python. It makes use of socket programming to enable smooth communication between users. Client programs power the graphical user interface (GUI), which provides participants with an interactive and intuitive environment.

To provide real-time data exchange during live meetings, the system uses socket programming to create a strong communication channel between the server and clients. By offering an adjustable solution for many meeting contexts and encouraging user engagement and conversation, the initiative seeks to improve remote collaboration.

Concurrently, the project expands its use to multiplayer gaming, showcasing a dynamic Tic-Tac-Toe game that allows players to engage in real-time competition via a network. Scalable and concurrent game sessions are made possible by the use of sockets, which provide effective communication between the server and several client applications. The game logic is managed by the server, which guarantees fair and synchronized gameplay. Low-latency interactions help players of all geographical regions have a seamless and entertaining gaming experience.

This collaborative effort demonstrates how socket programming may be used to develop responsive and interactive programs that promote efficient communication during in-person meetings and provide users with an enjoyable gaming experience.

1. LIVE MEETING WITH SOCKET PROGRAMMING

INTRODUCTION:

The need for novel technologies that facilitate real-time interactions has grown significantly in the current context of distant cooperation and communication. This project explores the field of real-time meetings by showcasing an advanced Python graphical user interface (GUI) that is driven by socket programming. The foundation is socket programming, which creates a dependable, immediate communication link between a centralized server and several client applications. This allows for the smooth interchange of data during real-time meetings.

The primary aim of this project is to provide an interactive and intuitive interface that surpasses the constraints of conventional communication instruments. Through the use of socket programming, the system guarantees that users are immersed in a dynamic and fluid interaction space. The key is real-time data exchange, which enables people to interact, share material, and work together without difficulty across geographic boundaries.

The aim of this project is to tackle the difficulties associated with remote collaboration by utilizing socket programming to improve the user experience overall, in addition to offering a real-time meeting platform. The system intends to demonstrate its flexibility to various meeting environments through real deployment and user testing, highlighting the potential of technology to completely transform the field of collaborative communication.

OBJECTIVE:

The main goal of using socket programming to construct live meetings is to provide a reliable and effective platform that enables participants to collaborate and communicate in real-time. Socket programming is a fundamental technique that facilitates dependable connection between a central server and several client applications, allowing for immediate and smooth data transmission.

METHODOLOGY:

Techniques for Using Python Socket Programming to Conduct Live Meetings are mentioned below they are:

1. **DEFINE REQUIREMENTS**: Clearly outline the requirements for real-time communication, video streaming, screen sharing, and audio sharing. Specify the desired features of the graphical user interface (GUI).

2. **SOCKET PROGRAMMING SETUP**: Put server code into action to greet incoming connections, schedule meetings, and enable data sharing.Write client code to manage screen, audio, and video sharing while establishing a connection to the server.

3. **VIDEO STREAMING**: Include a video streaming library or framework (OpenCV, for example) in the client code.Provide the ability to record and share video straight from the user's camera.

4. **VIDEO SCREENING**: Use a screen capture library to record the user's screen (PyAutoGUI, for example).Enable screen sharing so that users can share their displays with others.

5. **AUDIO SHARING:** Include an audio library in the client code (such as PyAudio).Provide the ability to record and broadcast audio directly from the user's microphone**.**

6. **SCALABILITY:** Make sure the system is scalable so that it can manage several meetings going on at once.If you need to divide the load between servers, take into consideration load balancing techniques.

With the help of this approach, you can create a feature-rich live meeting system in Python that supports screen sharing, audio, and video streaming, promoting productive participant interaction and communication.

IMPLEMENTATION IN PYTHON:

1. SERVER CODE:

```
import socket
import threading
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
import cv2
import pickle
import struct
from io import BytesIO
import sounddevice as sd
import numpy as np

SERVER_HOST = '192.168.196.207'
SERVER_PORT_SCREEN = 5051
SERVER_PORT_VIDEO = 9999
SERVER_PORT_AUDIO = 5052

server_socket_screen = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
server_socket_screen.bind((SERVER_HOST, SERVER_PORT_SCREEN))
server_socket_screen.listen()

server_socket_video = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket_video.bind((SERVER_HOST, SERVER_PORT_VIDEO))
server_socket_video.listen()

server_socket_audio = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket_audio.bind((SERVER_HOST, SERVER_PORT_AUDIO))
server_socket_audio.listen()

client_socket_screen, client_address_screen =
server_socket_screen.accept()
client_socket_video, client_address_video =
server_socket_video.accept()
client_socket_audio, client_address_audio =
server_socket_audio.accept()

screen_sharing_enabled = threading.Event()
video_sharing_enabled = threading.Event()
audio_sharing_enabled = threading.Event()

root = None
label_screen = None




AUDIO_CHUNK = 1024
AUDIO_SAMPLE_RATE = 44100
AUDIO_CHANNELS = 2
```

```
def receive_screen():
    global label_screen

    while True:
        try:
            size_data = client_socket_screen.recv(4)
            if not size_data:
                break
            size = int.from_bytes(size_data, byteorder='big')

            received_data = b""
            while len(received_data) < size:
                data_chunk = client_socket_screen.recv(min(size -
len(received_data), 4096))
                if not data_chunk:
                    break
                received_data += data_chunk

            if len(received_data) < size:
                continue

            image = Image.open(BytesIO(received_data))
            photo = ImageTk.PhotoImage(image)

            label_screen.config(image=photo)
            label_screen.image = photo  # keep a reference to the image

        except Exception as e:
            print(e)
            break

def receive_video():
    while True:
        try:
            data_size = client_socket_video.recv(8)
            if not data_size:
                break
            msg_size = struct.unpack("Q", data_size)[0]

            data = b""
            while len(data) < msg_size:
                packet = client_socket_video.recv(min(msg_size -
len(data), 4 * 1024))
                if not packet:
                    break
                data += packet

            if len(data) < msg_size:
                continue

            frame_data = data[:msg_size]
            frame = pickle.loads(frame_data)
            cv2.imshow('RECEIVING VIDEO', frame)
```

```
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
        except Exception as e:
            print(e)
            break


def receive_audio():
    while audio_sharing_enabled.is_set():
        try:
            audio_data_size = client_socket_audio.recv(4)
            if not audio_data_size:
                break
            size = int.from_bytes(audio_data_size, byteorder='big')

            received_data = b""
            while len(received_data) < size:
                data_chunk = client_socket_audio.recv(min(size -
len(received_data), 4096))
                if not data_chunk:
                    break
                received_data += data_chunk

            if len(received_data) < size:
                continue

            sd.play(np.frombuffer(received_data, dtype=np.int16),
samplerate=AUDIO_SAMPLE_RATE, channels=AUDIO_CHANNELS)
            sd.wait()

        except Exception as e:
            print(e)
            break


def toggle_screen_sharing():
    screen_sharing_enabled.set()
    print("Screen Sharing enabled")

def toggle_video_sharing():
    video_sharing_enabled.set()
    print("Video Sharing enabled")

def toggle_audio_sharing():
    audio_sharing_enabled.set()
    print("Audio Sharing enabled")

def stop_screen_sharing():
    screen_sharing_enabled.clear()
    print("Screen Sharing disabled")

def stop_video_sharing():
    video_sharing_enabled.clear()
    print("Video Sharing disabled")

def stop_audio_sharing():
    audio_sharing_enabled.clear()
```

```
    print("Audio Sharing disabled")

def on_closing():
    if messagebox.askokcancel("Quit", "Do you want to quit?"):
        server_socket_screen.close()
        server_socket_video.close()
        server_socket_audio.close()
        root.destroy()

def update_image(event):
    label_screen.image = label_screen.tkinter.PhotoImage(
        label_screen.image)

def start_server():
    global root, label_screen

    root = tk.Tk()
    root.title("Screen, Video, and Audio Sharing Server")
    root.geometry("800x600")

    label_screen = tk.Label(root)
    label_screen.pack(expand="true")

    screen_sharing_button = tk.Button(root, text="Toggle Screen
Sharing", command=toggle_screen_sharing)
    screen_sharing_button.pack(pady=10)

    video_sharing_button = tk.Button(root, text="Toggle Video Sharing",
command=toggle_video_sharing)
    video_sharing_button.pack(pady=10)

    audio_sharing_button = tk.Button(root, text="Toggle Audio Sharing",
command=toggle_audio_sharing)
    audio_sharing_button.pack(pady=10)

    screen_thread = threading.Thread(target=receive_screen)
    screen_thread.start()

    video_thread = threading.Thread(target=receive_video)
    video_thread.start()

    audio_thread = threading.Thread(target=receive_audio)
    audio_thread.start()

    root.protocol("WM_DELETE_WINDOW", on_closing)
    root.bind('<<UpdateImage>>', update_image)
    root.mainloop()


start_server()
```

2. CLIENT CODE:

```
import socket
import threading
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk, ImageGrab
import cv2
import pickle
import struct
from io import BytesIO
import sounddevice as sd
import numpy as np

SERVER_HOST = '192.168.196.207'
SERVER_PORT_SCREEN = 5051
SERVER_PORT_VIDEO = 9999
SERVER_PORT_AUDIO = 5052



AUDIO_CHUNK = 1024
AUDIO_SAMPLE_RATE = 44100
AUDIO_CHANNELS = 2



client_socket_screen = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
client_socket_screen.connect((SERVER_HOST, SERVER_PORT_SCREEN))

client_socket_video = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
client_socket_video.connect((SERVER_HOST, SERVER_PORT_VIDEO))

client_socket_audio = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
client_socket_audio.connect((SERVER_HOST, SERVER_PORT_AUDIO))

screen_sharing_enabled = False
video_sharing_enabled = False
audio_sharing_enabled = False

root = None
label = None

def start_screen_sharing():
    global screen_sharing_enabled
    screen_sharing_enabled = True
    while screen_sharing_enabled:
        try:

            screen = ImageGrab.grab()
            screen = screen.resize((200, 150))
            photo = ImageTk.PhotoImage(screen)
            label.config(image=photo)
            label.image = photo
```

```
            photo_data = BytesIO()
            screen.save(photo_data, format='JPEG')
            photo_data = photo_data.getvalue()
            size = len(photo_data).to_bytes(4, byteorder='big')
            client_socket_screen.sendall(size + photo_data)
        except Exception as e:
            print(e)
            break

def start_video_sharing():
    global video_sharing_enabled
    video_sharing_enabled = True
    vid = cv2.VideoCapture(0)
    while video_sharing_enabled:
        try:

            _, frame = vid.read()
            data = pickle.dumps(frame)
            message = struct.pack("Q", len(data)) + data
            client_socket_video.sendall(message)
        except Exception as e:
            print(e)
            break

def start_audio_sharing():
    global audio_sharing_enabled
    audio_sharing_enabled = True
    with sd.OutputStream(samplerate=AUDIO_SAMPLE_RATE,
channels=AUDIO_CHANNELS, dtype=np.int16) as stream:
        while audio_sharing_enabled:
            try:
                audio_chunk, overflowed = stream.read(AUDIO_CHUNK)
                audio_data = audio_chunk.tobytes()
                size = len(audio_data).to_bytes(4, byteorder='big')
                client_socket_audio.sendall(size + audio_data)
                print(f"Sent audio data size: {len(size +
audio_data)}")
            except Exception as e:
                print(f"Audio send error: {e}")
                break

def receive_video():
    global label
    while video_sharing_enabled:
        try:

            data_size = client_socket_video.recv(8)
            if not data_size:
                break
            msg_size = struct.unpack("Q", data_size)[0]


            data = b""
            while len(data) < msg_size:
                packet = client_socket_video.recv(min(msg_size -
len(data), 4 * 1024))
                if not packet:
```

```
                    break
                data += packet

            if len(data) < msg_size:
                continue


            frame_data = data[:msg_size]
            frame = pickle.loads(frame_data)
            cv2.imshow("Received Video", frame)

            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

        except Exception as e:
            print(e)
            break

def toggle_screen_sharing():
    threading.Thread(target=start_screen_sharing).start()

def toggle_video_sharing():
    threading.Thread(target=start_video_sharing).start()
    threading.Thread(target=receive_video).start()

def toggle_audio_sharing():
    threading.Thread(target=start_audio_sharing).start()

def stop_screen_sharing():
    global screen_sharing_enabled
    screen_sharing_enabled = False

def stop_video_sharing():
    global video_sharing_enabled
    video_sharing_enabled = False

def stop_audio_sharing():
    global audio_sharing_enabled
    audio_sharing_enabled = False

def on_closing():
    if messagebox.askokcancel("Quit", "Do you want to quit?"):
        client_socket_screen.close()
        client_socket_video.close()
        client_socket_audio.close()
        root.destroy()


root = tk.Tk()
root.title("Screen, Video, and Audio Sharing Client")
root.geometry("800x600")

label = tk.Label(root)
label.pack(expand="true")


start_screen_sharing_button = tk.Button(root, text="Start Screen
```

```
Sharing", command=toggle_screen_sharing)
start_screen_sharing_button.pack(pady=10)


stop_screen_sharing_button = tk.Button(root, text="Stop Screen
Sharing", command=stop_screen_sharing)
stop_screen_sharing_button.pack(pady=10)


start_video_sharing_button = tk.Button(root, text="Start Video
Sharing", command=toggle_video_sharing)
start_video_sharing_button.pack(pady=10)


stop_video_sharing_button = tk.Button(root, text="Stop Video
Sharing", command=stop_video_sharing)
stop_video_sharing_button.pack(pady=10)


start_audio_sharing_button = tk.Button(root, text="Start Audio
Sharing", command=toggle_audio_sharing)
start_audio_sharing_button.pack(pady=10)


stop_audio_sharing_button = tk.Button(root, text="Stop Audio
Sharing", command=stop_audio_sharing)
stop_audio_sharing_button.pack(pady=10)

root.protocol("WM_DELETE_WINDOW", on_closing)
root.mainloop()
```

CODE OVERVIEW:

1. **SERVER CODE**: Using socket programming, the included Python script implements screen sharing, video streaming, and audio sharing features as a server for a live meeting system. To handle screen, video, and audio data, three distinct sockets are initialized, each tied to a different port. The matching client sockets and addresses are retrieved upon client connection. The script runs routines that continually accept data from connected clients for screen, video, and audio sharing in parallel using threading. The Pillow library is used to handle the receiving screen pictures before they are shown in a tkinter label. The sounddevice library is used to play back audio data, while OpenCV is used to handle video frames. Screen, video, and audio sharing may be toggled using buttons in the tkinter GUI, and the server responds to user input with grace. attempts to shut server sockets and ask for confirmation in order to close the GUI.

2. **CLIENT CODE**: The included Python script serves as a client application with screen sharing, video streaming, and audio sharing features for a live meeting system. In order to connect to a server at a certain IP address (SERVER_HOST) and port numbers (SERVER_PORT_SCREEN, SERVER_PORT_VIDEO, SERVER_PORT_AUDIO), the script uses socket programming. A graphical user interface (GUI) that enables users to start and stop screen sharing, video streaming, and audio sharing is created by the client program using the tkinter library. For concurrent execution, distinct threads are used for each sharing function. The ImageGrab module is used by the screen-sharing function to take a picture of the user's screen, resize it, and send the screen data back to the server repeatedly. The user's webcam is used by the video sharing feature to record frames, which are then serialized and sent to the server via pickle. The sounddevice library is used by the audio sharing function to record audio from the user's microphone. The audio is then delivered in segments to the server. In addition, the client uses OpenCV to show the video frames it gets from the server in a separate thread. The GUI gives participants in a live meeting an easy-to-use interface by having buttons to start and stop each sharing

feature. The GUI and graceful connection termination are handled by the script. It's crucial to remember that this client-side code works in tandem with the server-side implementation to produce a full live meeting system that allows users to actively participate in screen sharing, audio sharing, and video streaming during group projects.

OUTPUT:

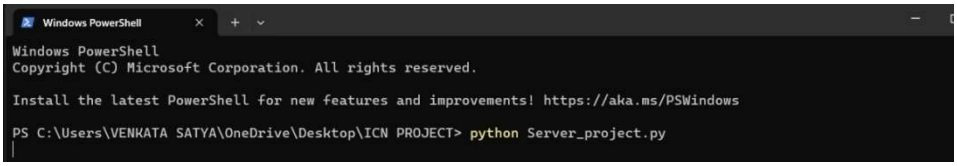


Figure 1:RUNNING SERVER

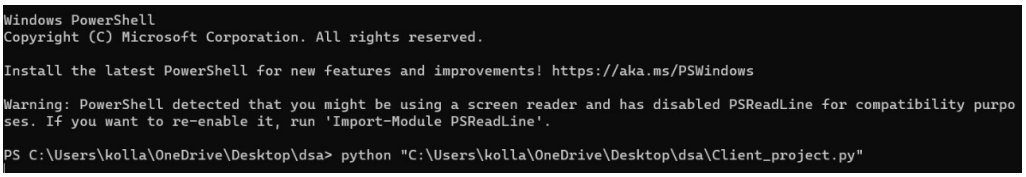


Figure 2:RUNNING CLIENT

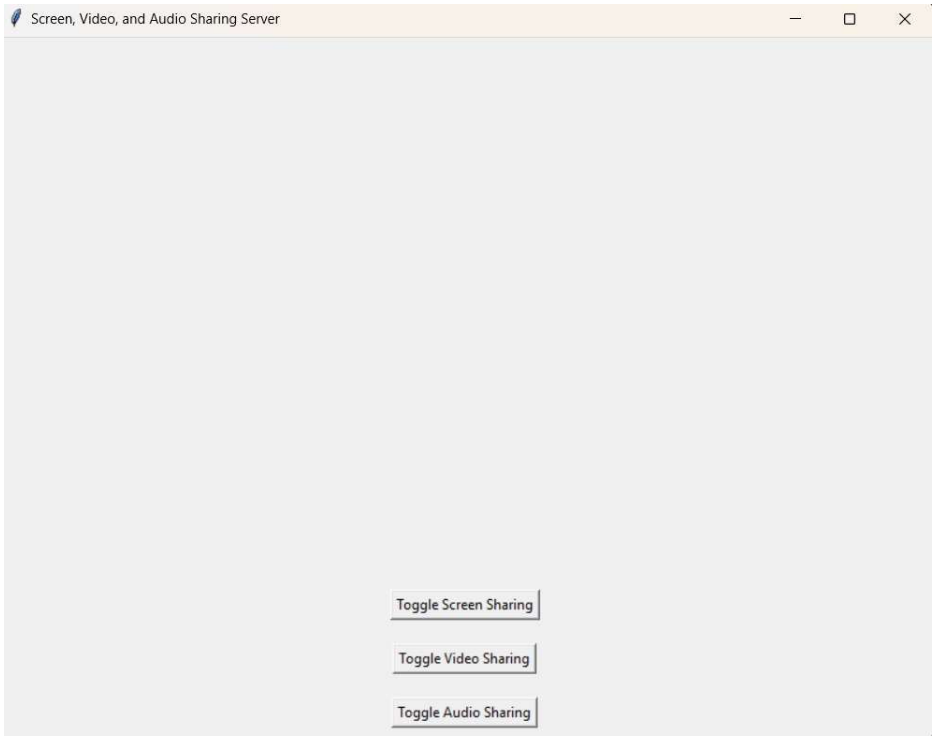


Figure 3:SERVER PORTAL

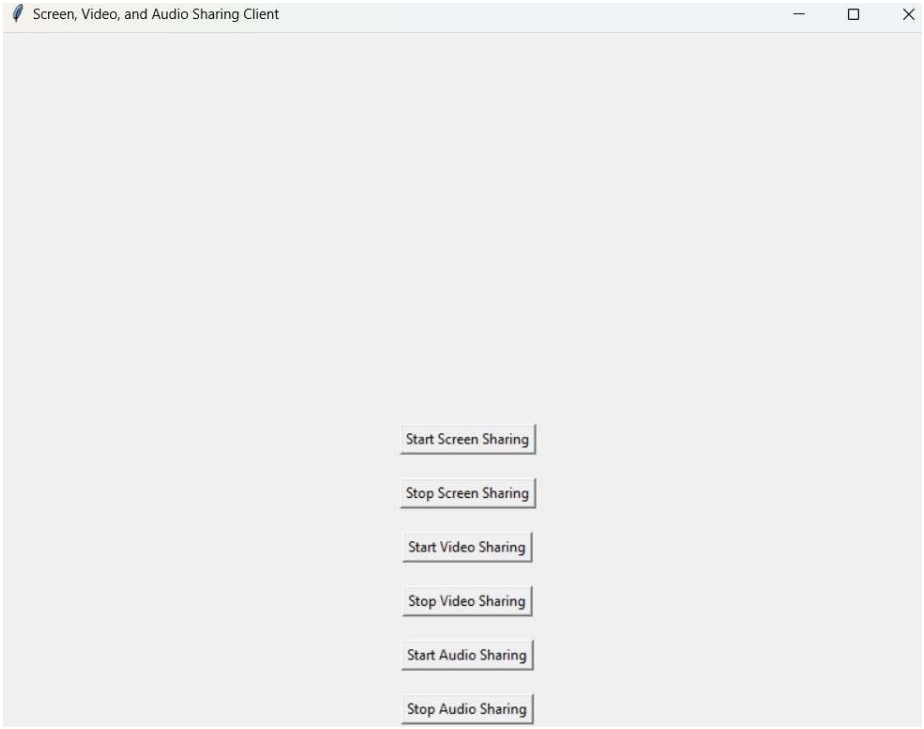


Figure 4:CLIENT PORTAL

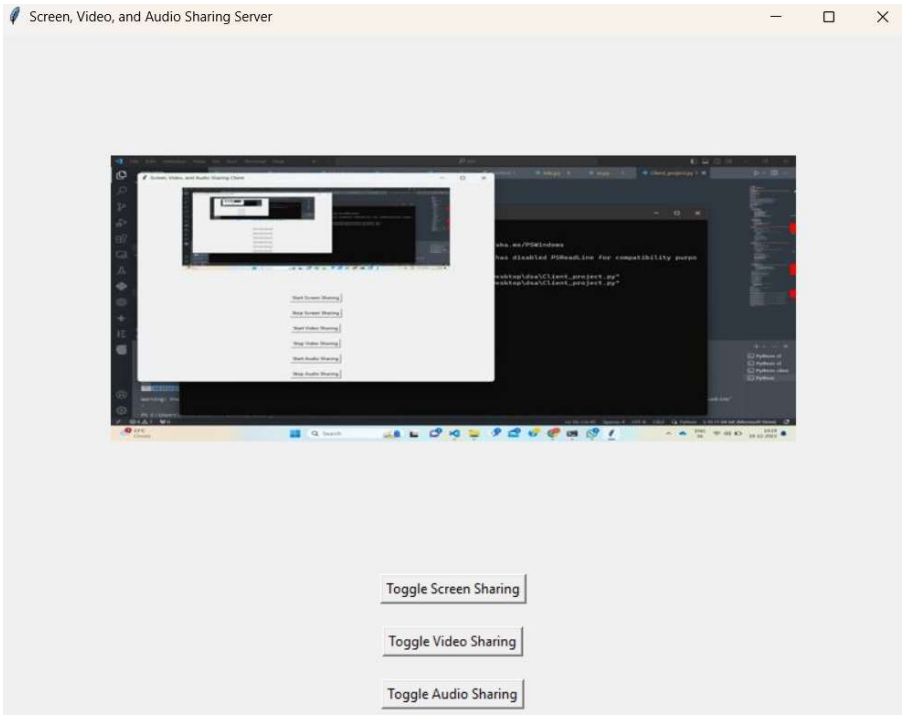


Figure 5:SCREEN SHARING

Figure 6:RECIEVED VIDEO

3. TICTACTOE GAME WITH SOCKET PROGRAMMING

INTRODUCTION:

This project presents a multiplayer version of the traditional Tic-Tac-Toe game that has been improved by the use of socket programming. The straightforward but strategic game of tic tac toe is turned into a vibrant, interactive environment where players may engage in competitive play via a network. Socket programming allows the game server to communicate with several clients in a seamless manner, allowing players to engage in real time even if they are located in different places.

This implementation's main objective is to bring the classic Tic-Tac-Toe gaming into a multiplayer environment, encouraging competitive play and interpersonal relationships. The foundation of the game's connectivity is socket programming, which enables real-time information sharing, game state synchronization, and player movement. This project demonstrates how networking elements may be included into a traditional game. supplying a platform where users may engage in multiplayer gaming and showcasing socket programming's adaptability in creating interactive applications.

OBJECTIVE:

The initiative aims to offer a dynamic and interactive platform that goes beyond the constraints of single-player gaming, encouraging user competitiveness and social involvement. By incorporating socket programming, the system guarantees low-latency communication, enabling participants to play the game together regardless of where they are physically located. The goal is to demonstrate how networking principles may be easily included into a traditional game, demonstrating how socket programming can be used to create flexible environments for multiplayer, real-time gaming.

METHODOLOGY:

1. **DEFINE GAME LOGIC:** Define the Tic-Tac-Toe game's rules and logic in detail, making sure to include player turns, victory conditions, and board representation.

2. **SOCKET INTIALIZATION**: In order to receive incoming connections from clients, implement server-side socket initialization.Create socket connections on the client side to communicate with the game server.

3. **MULTIPLAYER GAME MANAGEMENT**: Provide server logic that can handle many gaming sessions running simultaneously, giving each session a distinct identification.Put in place systems to manage player movements, verify moves, and update the status of the game.

4. **REAL-TIME COMMUNICATION**: To provide real-time communication between the server and clients, use socket programming.Establish message formats for alerts at the conclusion of the game, player movements, and game updates.

IMPLEMENTATION IN PYTHON:

1. PLAYER 1 CODE:

```
import socket
import threading

class TicTacToe:
    def __init__(self):
        self.board = [["", "", ""], ["", "", ""], ["", "", ""]]
        self.turn = "X"
        self.you = "X"
        self.opponent = "O"
        self.winner = None
        self.game_over = False
        self.counter = 0

    def host_game(self, host, port):
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind((host, port))
        server.listen(1)
        client, addr = server.accept()
        self.you = "X"
        self.opponent = "O"
        threading.Thread(target=self.handle_connection,
args=(client,)).start()
        server.close()

    def connect_to_game(self, host, port):
        try:
            client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client.connect((host, port))
            print(f"Connected to the game server at {host}:{port}")
            self.you = "O"
            self.opponent = "X"
            threading.Thread(target=self.handle_connection,
args=(client,)).start()
        except ConnectionRefusedError:
            print(f"Connection to {host}:{port} refused. Make sure the
server is running.")
        except Exception as e:
            print(f"Error connecting to the server: {e}")
        finally:
            # Close the client socket if an exception occurs
            client.close()

    def handle_connection(self, client):
        try:
            while not self.game_over:
                if self.turn == self.you:
                    move = input("Enter a move (row, column): ")
                    if self.check_valid_move(move.split(',')):
                        client.send(move.encode('utf-8'))
                        self.apply_move(move.split(','), self.you)
                        self.turn = self.opponent
```

```
                    else:
                        print("Invalid move! The space is already taken.")
                else:
                    data = client.recv(1024)
                    if not data:
                        client.close()
                        break
                    else:
                        self.apply_move(data.decode('utf-8').split(','),
self.opponent)
                        self.turn = self.you  # Change the turn back to
'you'

            print("Game over!")
        except Exception as e:
            print(f"An error occurred: {e}")
        finally:
            client.close()

    def apply_move(self, move, player):
        if self.game_over:
            return
        row, col = int(move[0]), int(move[1])
        if self.board[row][col] == "":
            self.counter += 1
            self.board[row][col] = player
            self.print_board()
            if self.check_if_won():
                if self.winner == self.you:
                    print("You win!")
                elif self.winner == self.opponent:
                    print("You lose!")
                self.game_over = True
            elif self.counter == 9:
                print("It is a tie!")
                self.game_over = True
        else:
            print("Invalid move! The space is already taken.")

    def check_valid_move(self, move):
        return len(move) == 2 and move[0].isdigit() and move[1].isdigit()
and \
               0 <= int(move[0]) < 3 and 0 <= int(move[1]) < 3 and
self.board[int(move[0])][int(move[1])] == ""

    def check_if_won(self):
        for row in range(3):
            if self.board[row][0] == self.board[row][1] ==
self.board[row][2] != "":
                self.winner = self.board[row][0]
                return True

        for col in range(3):
            if self.board[0][col] == self.board[1][col] ==
self.board[2][col] != "":
                self.winner = self.board[0][col]
                return True
```

```
        if self.board[0][0] == self.board[1][1] == self.board[2][2] != "":
            self.winner = self.board[0][0]
            return True

        if self.board[0][2] == self.board[1][1] == self.board[2][0] != "":
            self.winner = self.board[0][2]
            return True

        return False

    def print_board(self):
        for row in range(3):
            for col in range(3):
                print(f" {self.board[row][col]}", end="")
                if col < 2:
                    print(" |", end="")
            print()
            if row < 2:
                print("-----------")

# Example usage
game = TicTacToe()
game.host_game("localhost", 9999)
```

2. PLAYER 2 CODE:

```
import socket
import threading

class TicTacToe:
    def __init__(self):
        self.board = [["", "", ""], ["", "", ""], ["", "", ""]]
        self.turn = "X"
        self.you = "X"
        self.opponent = "O"
        self.winner = None
        self.game_over = False
        self.counter = 0

    def host_game(self, host, port):
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind((host, port))
        server.listen(1)
        client, addr = server.accept()
        self.you = "X"
        self.opponent = "O"
        threading.Thread(target=self.handle_connection,
args=(client,)).start()
        server.close()

    def connect_to_game(self, host, port):
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client.connect((host, port))
        self.you = "O"
        self.opponent = "X"
        threading.Thread(target=self.handle_connection,
args=(client,)).start()

    def handle_connection(self, client):
        try:
            while not self.game_over:
                if self.turn == self.you:
                    move = input("Enter a move (row, column): ")
                    if self.check_valid_move(move.split(',')):
                        client.send(move.encode('utf-8'))
                        self.apply_move(move.split(','), self.you)
                        self.turn = self.opponent
                    else:
                        print("Invalid move! The space is already taken.")
                else:
                    data = client.recv(1024)
                    if not data:
                        client.close()
                        break
                    else:
                        self.apply_move(data.decode('utf-8').split(','),
self.opponent)
                        self.turn = self.you  # Change the turn back to
'you'

            print("Game over!")
```

```
        except Exception as e:
            print(f"An error occurred: {e}")
        finally:
            client.close()

    def apply_move(self, move, player):
        if self.game_over:
            return
        row, col = int(move[0]), int(move[1])
        if self.board[row][col] == "":
            self.counter += 1
            self.board[row][col] = player
            self.print_board()
            if self.check_if_won():
                if self.winner == self.you:
                    print("You win!")
                elif self.winner == self.opponent:
                    print("You lose!")
                self.game_over = True
            elif self.counter == 9:
                print("It is a tie!")
                self.game_over = True
        else:
            print("Invalid move! The space is already taken.")

    def check_valid_move(self, move):
        return len(move) == 2 and move[0].isdigit() and move[1].isdigit()
and \
               0 <= int(move[0]) < 3 and 0 <= int(move[1]) < 3 and
self.board[int(move[0])][int(move[1])] == ""

    def check_if_won(self):
        for row in range(3):
            if self.board[row][0] == self.board[row][1] ==
self.board[row][2] != "":
                self.winner = self.board[row][0]
                return True

        for col in range(3):
            if self.board[0][col] == self.board[1][col] ==
self.board[2][col] != "":
                self.winner = self.board[0][col]
                return True

        if self.board[0][0] == self.board[1][1] == self.board[2][2] != "":
            self.winner = self.board[0][0]
            return True

        if self.board[0][2] == self.board[1][1] == self.board[2][0] != "":
            self.winner = self.board[0][2]
            return True

        return False

    def print_board(self):
        for row in range(3):
            for col in range(3):
```

```
                print(f" {self.board[row][col]}", end="")
                if col < 2:
                    print(" |", end="")
            print()
            if row < 2:
                print("-----------")

# Example usage
game = TicTacToe()
game.connect_to_game("localhost", 9999)
```

CODE OVERVIEW:

1. **PLAYER 1 CODE**: The "player 1" code segment initiates the hosting of a Tic-Tac-Toe game by creating an instance of the TicTacToe class. It sets the player's symbol as "X" and the opponent's symbol as "O." The host_game method is invoked, establishing a server socket on the localhost at port 9999 and listening for incoming connections. Upon receiving a connection from "player 2" (the client), a new thread is spawned to execute the handle_connection method. Within this method, a while loop runs continuously, facilitating the game flow until the game_over flag is set to true. During the player's turn (if it's "player 1's" turn, denoted by the self.turn variable), the code prompts "player 1" to input their move as coordinates (row, column). The move is validated using the check_valid_move method, and if valid, it is sent to "player 2" over the network via the established socket connection. The apply_move method updates the game board and checks for a winner or a tie. The game continues in this loop until a conclusive outcome is reached, and the server socket is closed, marking the end of the game session. Overall, this code segment orchestrates the hosting of a Tic-Tac-Toe game and the management of player interactions during the course of the game

2. **PLAYER 2 CODE**: The "player 2" code initiates the connection to a Tic-Tac-Toe game hosted by "player 1." It creates an instance of the TicTacToe class, setting the player's symbol as "O" and the opponent's symbol as "X." The connect_to_game method establishes a socket connection to the specified host and port where the game is hosted. Upon successful connection, a new thread is launched to execute the handle_connection method. Within the handle_connection method, a continuous loop runs until the game_over flag is set to true. During "player 2's" turn (indicated by the self.turn variable), the code prompts the user to input their move as coordinates (row, column). The move is validated using the check_valid_move method, and if valid, it is sent to "player 1" over the network through the established socket connection. The apply_move method updates the game board and checks for a winner or a tie. The game continues in this loop until a conclusive outcome is reached, and the client socket is closed, marking the end of the game session. Overall, this code segment orchestrates the connection to a Tic-Tac-Toe game hosted by "player 1" and manages player interactions during the course of the game.

OUTPUT:

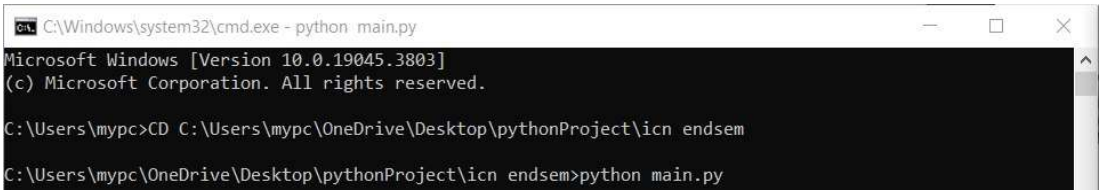


Figure 7:RUNNING PLAYER 1

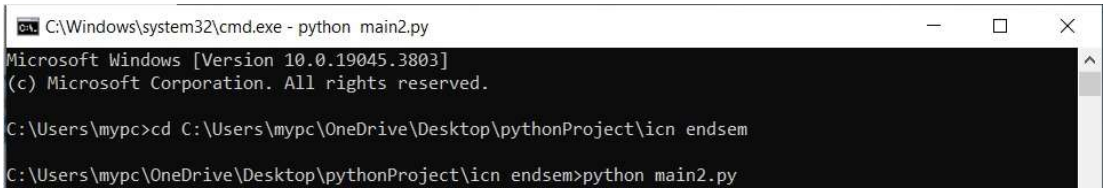


Figure 8:RUNNING PLAYER 2

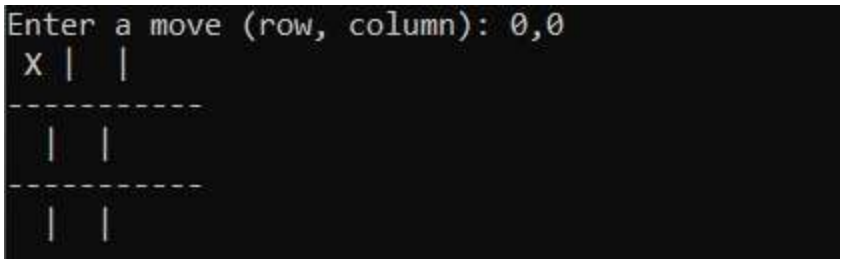


Figure 9:TURN ON PLAYER 1

Figure 10:TURN OF PLAYER 2

Figure 11:PLAYER 1 WIN

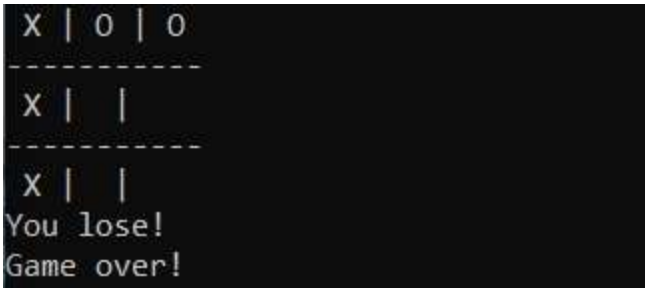


Figure 12:PLAYER 2 LOSE

3 . CONCLUSION

In summary, the integration of socket programming in both the live meeting platform and the Tic-Tac-Toe game demonstrates the versatility and transformative potential of this technology. The live meeting platform, developed in Python, establishes a robust communication channel between a central server and clients, fostering real-time collaboration with an intuitive graphical user interface. This project showcases the adaptability of socket programming to redefine remote communication, making meetings more dynamic and interactive.

Similarly, the multiplayer Tic-Tac-Toe game harnesses the power of sockets to create an engaging and real-time gaming experience. The project successfully extends the classic game into a multiplayer setting, allowing players to compete across a network. Socket programming enables seamless communication between the game server and clients, facilitating instantaneous exchange of moves and game states. Together, these projects highlight the broad applicability of socket programming, from collaborative meeting platforms to multiplayer gaming scenarios, showcasing its ability to enhance user interactions in both professional and recreational contexts.