Sultan Qaboos University
College of Science
Department of Computer Science
Fall 2021

A Client-Server Based Multiplayer Rock, Paper and Scissor Game with In-Game Incentives for Client Sharing of Computing Resources

Research Project

COMP6018 / Distributed Systems

Submitted By:

1. **Ayisha Al Saidi - S128554**
2. **Aviral Goel - S132206**

Submitted to:

Prof. Khalid Day

Table of Contents

Table of Figures

1. Abstract

Distributed Systems (DS) play a crucial role in our life. They have been helping us to get the various information in our hands with one click, particularly enabling us to share resources and communicate with each other easily. Indeed, the challenges behind DSs are still arising to make such systems more powerful, effective, with fewer obstacles and faults, concurrency, and other concepts, but simultaneously it solves many issues that faced humans in the past. However, this project gives an understandable illustration of modern DSs applications that operate via message passing between the clients and the server and it is called a multiplayer rock, paper, and scissor game application. In this study, we will give more descriptions about this application, its concepts and how its work in reality.

2. Introduction

Rapid technology led us to develop and come up with different solutions and applications to play an essential role in the way of making life much easier and simple. This can’t happen without contribution and collaboration from the various fields around us especially, the technology and their effective areas like the network and the Distributed Systems. However, Distributed Systems(DS) have provided cohesive systems that enable us to communicate and share resources in the simple and fastest way through the network which is everywhere around us and easy to access. In this project, we aimed to develop an example of modern distributed applications which is based on Client-Server Systems (CSS), named Multiplayer Rock, Paper, and Scissor Game Application. It allows a two-players or friends to play in any place at any time they want. In the following lines, we will describe more about this game, its definition, how to work, methods used to connect and communicate between client and server, tools used to develop this game, and other details.[1]

2.1 The Idea Description: Multiplayer Rock, Paper and Scissor Game

In the real world, the game of Rock, Paper and Scissor which is called *Rochambeau* indicates a two-player game where the players play an action using hand gestures (either rock, paper or scissor) simultaneously. Based on the rules of the game, a winner is decided amongst the two players. So, the rules of the game state as both players must choose rock, paper, or scissors at the same time, and delayed moves are not allowed. Once the action is played, the rule states that "rock" defeats "scissors" and "scissors" defeats "paper" and "paper" defeats "rock" as shown in Figure 1. [2]

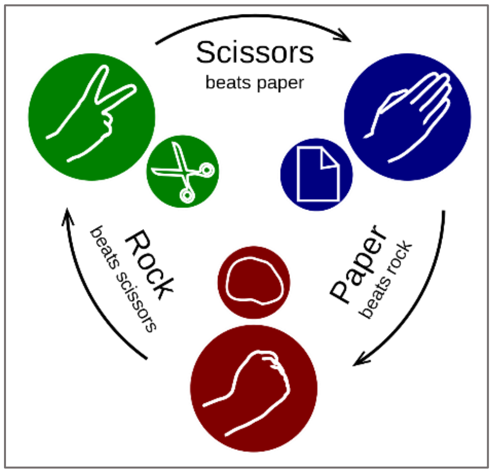


Figure 1. Rules of The Game

The proposed application is a client-server-based architecture where each player is considered as the client and connects to the server of the game. So, the server coordinates (hosts the game session) and moderates (decides the winner) on the game. Moreover, the server is responsible to maintain the state of the game and ensuring each connected player has the most recent version of the state of the game. The server also executes the logic of the game in contrast to the client/players, which are only responsible for connecting to the server and playing their moves. The clients can only connect to the server and make their plays. Lastly, the results show in the client's screen. Accordingly, the clients can start the game a gain or close the application, thus close the connection with the server too. [2]

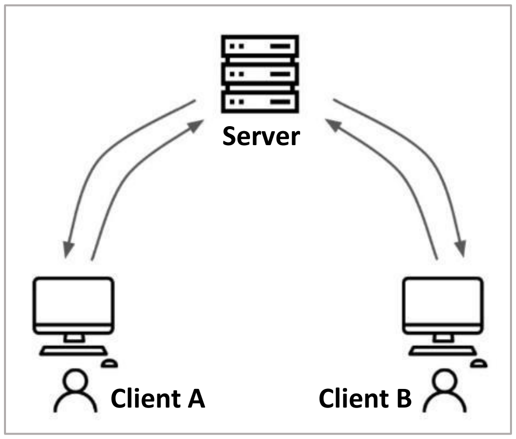


Figure 2. System Architecture

2.2 Client-Server Communication Method: (Sockets)

The network consists of devices (computers) that are either a client or a server. Mostly, servers which can be file servers, print servers, or others are powerful computers or processes that provides some services for the client. While clients which can be PCs or any workstations are requesting some services or resources from the server. Therefore, these two processes communicate with each other by reading from and writing to sockets. Sockets are a form of inter-process communication (IPC) that are used to send messages between two nodes (client-server) through the networks. It is one of the Client-Server communication procedures that are mostly used in different programs or application-level protocols such as FTP, SMTP, and POP3. They use sockets to create a connection between the client and server and then exchange data via using the network. However, a socket can be defined as one endpoint (a combination of an IP address and a port number) of a two-way communication (client-server) link between two programs running on the network. Generally, Sockets can be either connection-oriented which is called Transmission Control Protocol (TCP), or connectionless that is called User Datagram Protocol (UDP). In our project, we will use the connection-oriented (TCP) because of multiple reasons [1][3][4][5]. The followings are some instances:

- **Reliability:** TCP offered a reliable connection between the client and server. Consequently, any packet dropped or lost in the middle of the network for any reason can be detected and handled by using various criteria like checksum, sequence No., timeout, and retransmitted again if it is needed. In the application, we need these offers to reach the packets properly without loss or any related problems because such things will affect the way of playing the game and may not be worked correctly [3][5].
- **In-order data delivery:** TCP can be sure that data will reach its destination in order and we need this such guarantee in our game application.

In contrast, the UDP socket does not offer these concepts which are important in our application because it is unreliable and packets can be lost or reached out of order. Unlike TCP, it is offered a fast connection but no need for such a thing in our application because it is a simple game and just takes a few steps to reach the end [3][5].

2.2.1 How Does the Socket Work?

In the connection-oriented (TCP), the client tries to make a connection with the server using this port number and IP address. So, the server listens and accepts the connection. Once the connection is established, the server can receive a message from the client as well as respond back to a message to the client. To give a clear description, first, the server establishes (binds) an address that enables clients to find the server. When the address is established, the server waits for clients to request a service. Then, the client connects to the server using a socket and sends the request. So, the data (messages) can be exchanged between client and server. The server proceeds with the request of the client and sends the reply to it and when they finished their communication, they closed the connection. Figure 3. shows clearly the typical flow of events in the connection-oriented (TCP) sockets.[3][4][5]

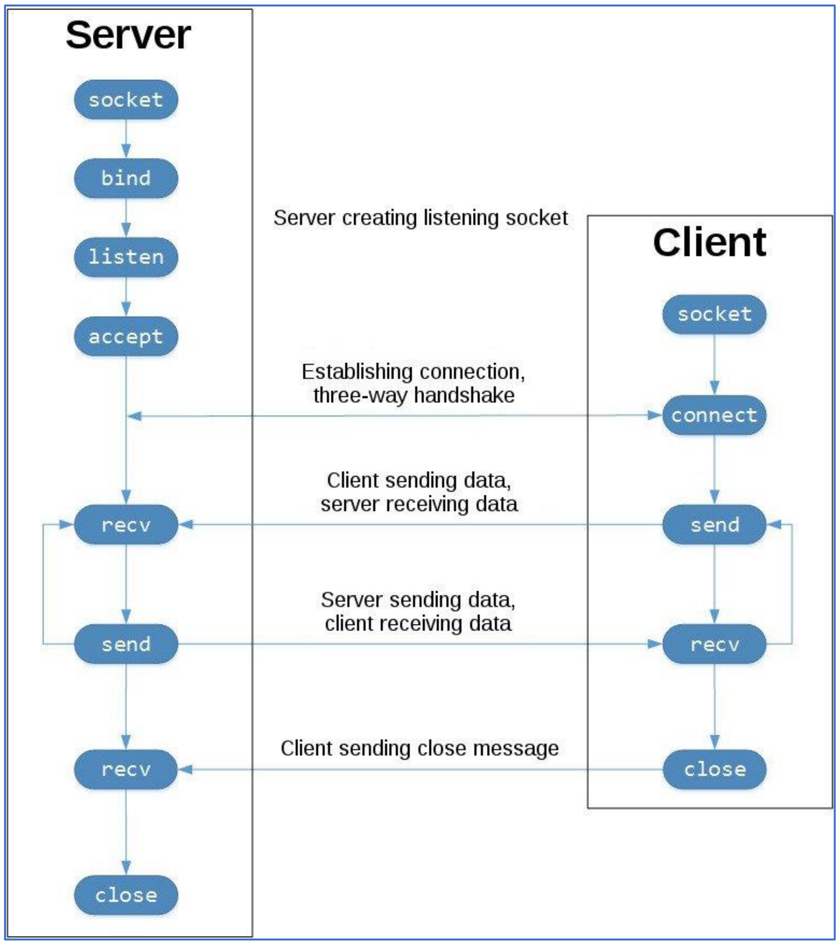


Figure 3. Connection-Oriented TCP Sockets (source:

Basically, sockets used two operations for communication which are *send (write)* and *recv(read)* as shown below[6]:

- **<u>Send:</u>** specified message using specified socket.

  int send(int socket, char *message, int msg_len, int flags)

- **<u>Recv :</u>** receive a message from the specified socket into the specified buffer

  int recv(int scoket, char *buffer, int buf_len, int flags)

The socket programming is started by importing the socket library and making a simple socket like:

import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Briefly, we made a socket instance and passed it two parameters (AF_INET , SOCK_STREAM). AF_INET refers to the address family IPv4. The SOCK_STREAM means connection-oriented TCP protocol. Then, we can connect to a server using this socket [6]. To give a clear illustration:

- **<u>Server-side:</u>**

  The server works with these main operations:

  1. **create() :** Create TCP socket
  2. **bind() :** bind to a specific IP and port to listen to incoming requests on that IP and port.
  3. **listen() :** put the server into a listening mode to listen to incoming connections.
  4. **accept() :** to initiate a connection with the client, so they are ready to transfer data.
  5. **recv():** receive the request from the client and then process data.
  6. **send():** send the data(reply) to the client.
  7. **close():** to close the connection with the client.[3][4]

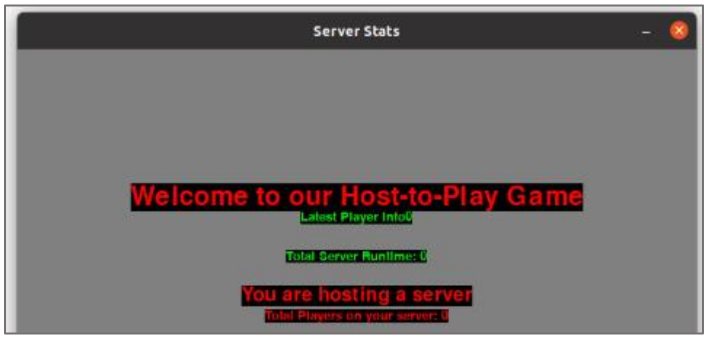
- **Client-side:**

  This is where the server can interact and communicate with. So, on client-side , after we make a socket object, then we can use these methods:

  1. **connect():** connect to localhost on port 12345 (the port on which our server runs).
  2. **Send/write ():** send the request to the server.
  3. **Recv/read():** receive the data from the server.
  4. **close():** close the connection.[3][4]

2.3 Volunteer Based Computing for In-Game Rewards

The typical client-server architecture requires a dedicated server which the manufacturer of the game has to provide. These servers are powerful, capable of running hundreds of instances of the game simultaneously. However, they are expensive to set up and maintain. Due to this reason, typically the servers of old games are shut down, leaving the players no option but to play offline. In our application, we have provided the client an opportunity to act as the server for other anonymous players on the network. This eases the burden of the game manufacturers to cater to increasing numbers of players as well as when the game loses its popularity. In exchange for hosting the server of the game, the user earns rewards purely based on the uptime of the server. These rewards act as an incentive for the users to continue hosting the server when their systems are idle. The following figure shows the server system in our application [2][7].



Welcome to our Host-to-Play Game
Latest Player Info0
Total Server Runtime: 0
You are hosting a server
Total Players on your server: 0

Figure 4. Screenshot of the Server System

2.4 Tools Used for The Application

2.4.1 Python Programming Language:

Python is one of the most popular programming languages used by developers and computer scientists these days. It is used to develop different applications, programs, websites, and games. There are many tools and libraries used in Python to support and develop these uses easily and faster. Additionally, Python can be used by data analysts and other professionals to conduct complex statistical calculations, build machine learning algorithms, analyze data, visualize data, and complete other tasks related to data. However, it is very friendly, powerful, and easy to learn, read and understand programming language. Therefore, it helps to build projects quickly as well as update and improve on them. Nevertheless, many famous companies that used Python include Google, Facebook, Venmo, Dropbox, Spotify, and Netflix. [3][8]

However, Python is a compatible library for sockets that can be used to maintain and establish the connection between the server and clients as well as to facilitate the exchange of information. Accordingly, we are aimed to develop our project using Python. In particular, we aim to use Python libraries like pygame to draw the game on the screen of the players.

2.4.2 Pygame Module:

Pygame module is a Python library that extends the Python language by giving a kind of facility for the programmers to be able to create some graphical objects on the windows. It extends the functionality of Python by providing some methods and features that can be used by developers to draw several objects like buttons, text, and other visual shapes on the screen with ease. In this project, we have used the Pygame module to draw the graphics of our game.[8]

2.5 Problem Statement

This project intends to implement a simple distributed client-server application which is a computer game application of the classic Rock, Paper and Scissor Game using a Python programming language in order to build and program the game. Also, in this application, we aimed to use sockets as a client-server communication method to communicate between players (clients) and the server. The application also extends the typical client-server architecture to a framework where the clients have the ability to become dedicated servers in exchange for incentives.

3. Design

3.1 Client-Server Architecture

Mainly, the aspect of the network in the game has been implemented using the traditional client-server architecture. To be clear, every player runs the client script locally and connects to the server. The server returns to the client global attributes, namely, its unique player ID, unique game session ID and whether the player will be player 1 or player 2 in the game session. The first client to connect to the server has to wait for another client for the game to begin. When two clients connect the server assigns them a common game session and coordinates for the game to begin.[2]

During the gameplay, the clients continuously fetch the latest state of the game for the server and send their local game state (which includes the moves played by the human player). The server receives game states from the client, runs the game logic, updates the game state and sends it back to the clients. Thus, the server is responsible for the game to be played fairly. In case of client disconnection, the server stops the game for both clients.

3.2 Pseudocode of The Game

The following Pseudocode explains briefly how the system works on both sides(client and Server). It gives a clear understanding of how the application will work logically from both the client and server perspectives.

- **Client-side:**
    - Create() TCP client socket.
    - Establish the connection with the server using (3-way handshake)
    - After connection, the server will send a player ID to get started, so the client will get player information
    - The clients get a game session to start playing
    - Now, the game gets started and the 1st player can move with(Rock, Scissors, Paper) & the opponent on the 2nd window do same.
    - For 2 or 3 rounds, then the clients get the result (win or lost)
    - Close the session.

- **Server-side:**
    - Create() TCP socket
    - Using bind(), bind the socket to server address(Ip address & port number).
    - Listen() and wait for the client to ask for a connection.
    - Accept(), now the connection is established between the client and server.
    - Now, when the client connected with the server , the server sends ID No for the player (client)
    - Create a new game session for players or add another player for the existing session.
    - Start a parallel thread to handle requests from the client.
    - Server receives the game status from the player & send the game status to the player
    - Show the results for each client on their screens.
    - Close the connection.

3.3 A flowchart of The Game

Figure 5, demonstrates a flowchart representation of the game for client-server communication. It describes the system step by step from the beginning when establishing the connection to the end of the game. Therefore, it shows the process of the application and how it works logically for both the client and server perspectives.

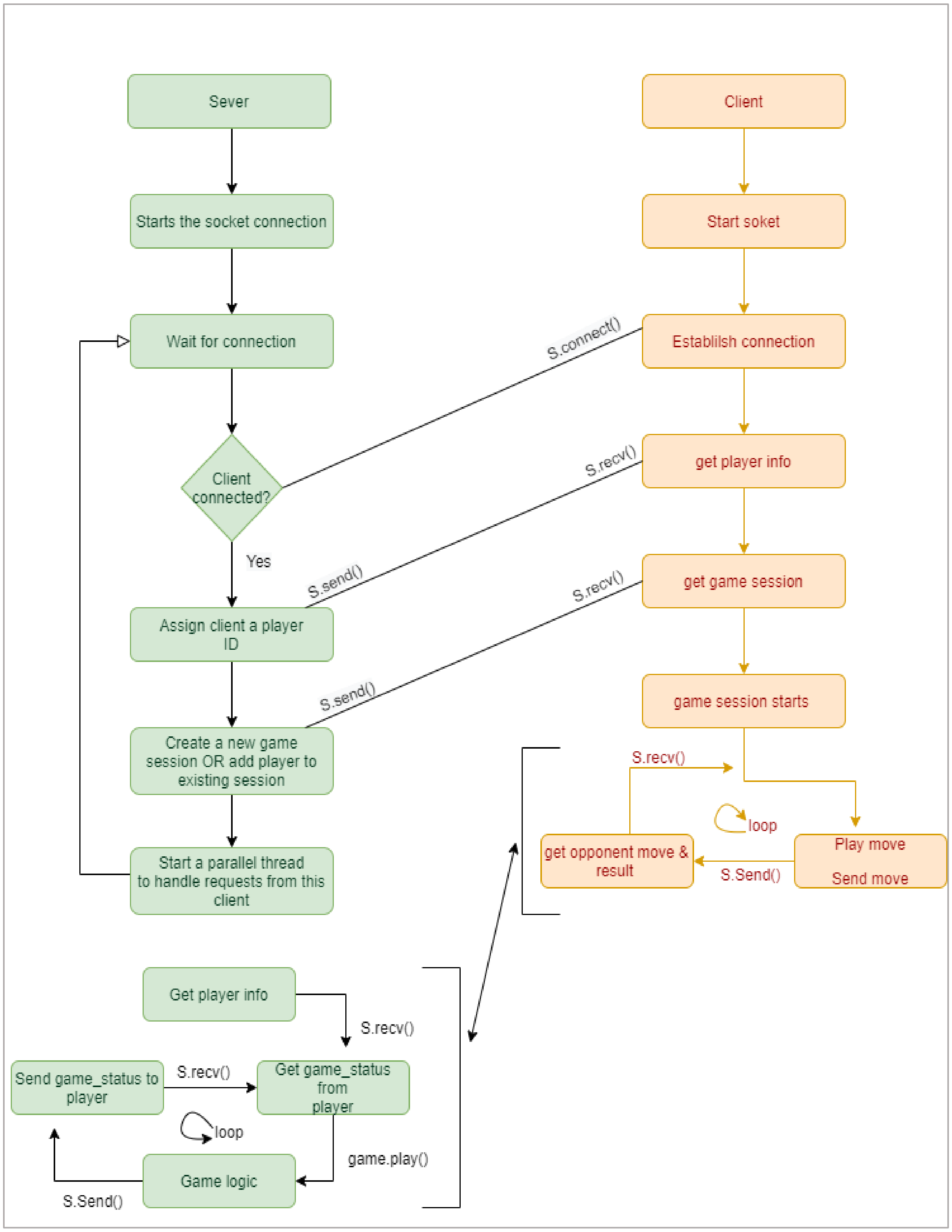


Figure 5. A Descriptive Flowchart of the Application

3.4 Sample Designs of the Game

The following figure illustrate sampled designs of how the game application looks like for the users. Basically, the application begins with the welcome screen and presents the system running with two choices either to act as a server or be a player to play the game as shown in figure6. Then, the figure.8 illustrates a sample interface for the game application and how it looks like in reality.

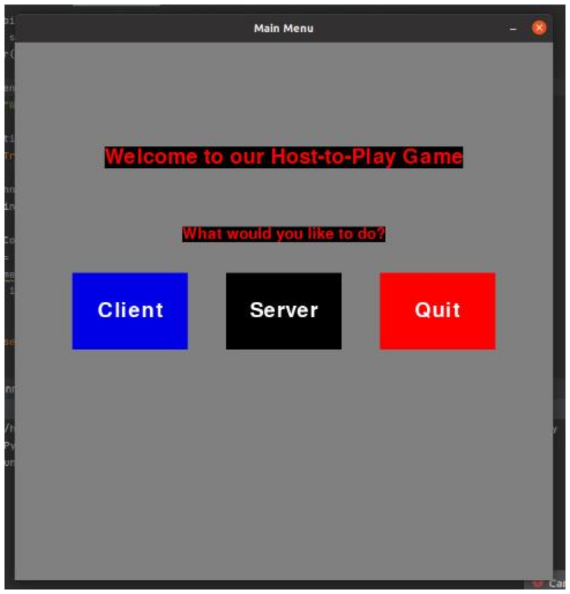


Figure 6. Welcome Screen

4 Implementation

4.4 Program Code

▪ main.py

```
import os

import pygame

from button import Button


width = 700

height = 700

win = pygame.display.set_mode((width, height))  # start the pygame game window render service

pygame.display.set_caption("Main Menu")

pygame.font.init()

run = True;


def drawWindow():

    win.fill((128, 128, 128))

    font = pygame.font.SysFont("comicsans", 40)

    text = font.render("Welcome to our Host-to-Play Game", 1, (255, 0, 0), True)

    win.blit(text, (width / 2 - text.get_width() / 2, -200 + height / 2 - text.get_height() / 2))

    font = pygame.font.SysFont("comicsans", 30)

    text2 = font.render("What would you like to do?", 1, (255, 0, 0), True)

    win.blit(text2, (width / 2 - text2.get_width() / 2, -100 + height / 2 - text2.get_height() / 2))

    for btn in btns:

        btn.draw(win)

    pygame.display.update()


btns = [Button("Client", 75, 300, (0, 0, 230)), Button("Server", 275, 300, (0, 0, 0)),

        Button("Quit", 475, 300, (255, 0, 0))]
```

```
def userInput():


    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            run = False
            exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
            run = False


        if event.type == pygame.MOUSEBUTTONDOWN:
            pos = pygame.mouse.get_pos()
            for btn in btns:
                if btn.click(pos):
                    if btn.text == "Client":
                        pygame.quit()
                        run = False
                        os.system('python client.py')
                        exit()
                    elif btn.text == "Server":
                        pygame.quit()
                        os.system('python server.py')
                        exit()
                        run = False
if __name__ == "__main__":
    clock = pygame.time.Clock()
    while run:
        clock.tick(60)
    drawWindow()
    userInput()
```

- **server.py**

```
import socket
from _thread import *
import pickle
from game import Game
import time
import pygame

width = 700
height = 700
server = "192.168.48.129"
port = 5677

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
win = pygame.display.set_mode((width, height))  # start the pygame game window render service
pygame.display.set_caption("Server Stats")
pygame.font.init()

connected = set()
games = {}
idCount = 0
def threaded_client(conn, p, gameId):
    global idCount
    conn.send(str.encode(str(p)))

    reply = ""
```

```
while True:
    try:
        data = conn.recv(4096).decode()

        if gameId in games:
            game = games[gameId]

            if not data:
                break
            else:
                if data == "reset":
                    game.resetWent()
                elif data != "get":
                    game.play(p, data)

                conn.sendall(pickle.dumps(game))
        else:
            break
    except:
        break

print("Lost connection")
try:
    del games[gameId]
    print("Closing Game", gameId)
except:
    pass
idCount -= 1
conn.close()
```

```
def updateStats(currentPlayers, currentTime, currentPlayerIP):

    win.fill((128, 128, 128))
    font = pygame.font.SysFont("comicsans", 40)
    text = font.render("Welcome to our Host-to-Play Game", 1, (255, 0, 0), True)
    win.blit(text, (width / 2 - text.get_width() / 2, -200 + height / 2 - text.get_height() / 2))
    font = pygame.font.SysFont("comicsans", 30)
    text2 = font.render("You are hosting a server", 1, (255, 0, 0), True)
    win.blit(text2, (width / 2 - text2.get_width() / 2, -100 + height / 2 - text2.get_height() / 2))
    font = pygame.font.SysFont("comicsans", 20)
    text2 = font.render("Total Players on your server: " + str(currentPlayers), 1, (255, 0, 0), True)
    win.blit(text2, (width / 2 - text2.get_width() / 2, -80 + height / 2 - text2.get_height() / 2))
    text2 = font.render("Total Server Runtime: " + str(currentTime), 1, (0, 235, 0), True)
    win.blit(text2, (width / 2 - text2.get_width() / 2, -140 + height / 2 - text2.get_height() / 2))
    text2 = font.render(("Latest Player Info" + str(currentPlayerIP)), 1, (0, 235, 0), True)
    win.blit(text2, (width / 2 - text2.get_width() / 2, -180 + height / 2 - text2.get_height() / 2))
    pygame.display.update()



def checkForConnection():
    global idCount
    try:
        s.bind((server, port))
    except socket.error as e:
        str(e)

    s.listen(2)
    print("Waiting for a connection, Server Started")
    start_time = time.time()
    while True:

        conn, addr = s.accept()
        print("Player with IP address: ",addr, " connected")

        idCount += 1
        p = 0
        gameId = (idCount - 1)//2
        if idCount % 2 == 1:
            games[gameId] = Game(gameId)
            print("Creating a new game...")
        else:
            games[gameId].ready = True
            p = 1
        updateStats(idCount, (time.time() - start_time), addr)
        start_new_thread(threaded_client, (conn, p, gameId))


updateStats(0, 0, 0)
checkForConnection()
```

- **client.py**

```
import pygame
from network import Network
from button import Button
import pickle
pygame.font.init()  # start the pygame text render services
```

```
width = 700

height = 700

win = pygame.display.set_mode((width, height))  # start the pygame game window render service

pygame.display.set_caption("Client")


# the screen renderer method for the game this method runs about 60 times per second (60FPS)
and draws/refreshes the

# game on the screen it requires three parameters, the pygame canvas window (to be drawn on),
the Game class object (

# the game to draw) and the player (according to which to draw)

def redrawWindow(win, game, p):

  win.fill((128,128,128))


  # there are not enough players to start the game

  if not(game.connected()):

    font = pygame.font.SysFont("comicsans", 80)

    text = font.render("Waiting for Player...", 1, (255,0,0), True)

    win.blit(text, (width/2 - text.get_width()/2, height/2 - text.get_height()/2))

  else:

    font = pygame.font.SysFont("comicsans", 60)

    text = font.render("Your Move", 1, (0, 255,255))

    win.blit(text, (80, 200))


    text = font.render("Opponents", 1, (0, 255, 255))

    win.blit(text, (380, 200))


    move1 = game.get_player_move(0)

    move2 = game.get_player_move(1)
```

```
# if both players played the move, display those moves
if game.bothWent():
    text1 = font.render(move1, 1, (0,0,0))
    text2 = font.render(move2, 1, (0, 0, 0))
else:
    if game.p1Went and p == 0: # if I played, show my move
        text1 = font.render(move1, 1, (0,0,0))
    elif game.p1Went: #if player 1 played but not me, then show him as locked
        text1 = font.render("Locked In", 1, (0, 0, 0))
    else: # none of the players played, then show waiting
        text1 = font.render("Waiting...", 1, (0, 0, 0))


    if game.p2Went and p == 1: # if player 2 played and am player 2, show my move
        text2 = font.render(move2, 1, (0,0,0))
    elif game.p2Went:
        text2 = font.render("Locked In", 1, (0, 0, 0))
    else:
        text2 = font.render("Waiting...", 1, (0, 0, 0))
# according to the above situations, decide the text to render on screen
if p == 1:
    win.blit(text2, (100, 350))
    win.blit(text1, (400, 350))
else:
    win.blit(text1, (100, 350))
    win.blit(text2, (400, 350))


for btn in btns:
    btn.draw(win)
    pygame.display.update()


btns = [Button("Rock", 50, 500, (0,0,0)), Button("Scissors", 250, 500, (255,0,0)), Button("Paper", 450, 
500, (0,255,0))]

# the game loop method, this method updates the state of the game for the player

def main():

    run = True

    clock = pygame.time.Clock()

    n = Network()  # establish the connection with the server

    player = int(n.getP())  # get the ID of the player for the game session decided and returned by the 
server to the client

    print("You are player", player)


    # game loop

    while run:

        clock.tick(60)

        try:

            game = n.send("get") # send the server a string to request the state of the game in Game 
class object

        except:

            run = False

            print("Couldn't get game")

            break


        # if both players played the move

        if game.bothWent():

            redrawWindow(win, game, player) # redraw the screen

            pygame.time.delay(500) # wait

            try:
```

```
        game = n.send("reset") # send the server signal to reset the round
    except:
        run = False
        print("Couldn't get game")
        break

    font = pygame.font.SysFont("comicsans", 90)
    # display the winning/losing screen to appropriate player
    if (game.winner() == 1 and player == 1) or (game.winner() == 0 and player == 0):
        text = font.render("You Won!", 1, (255,0,0))
    elif game.winner() == -1:
        text = font.render("Tie Game!", 1, (255,0,0))
    else:
        text = font.render("You Lost...", 1, (255, 0, 0))

    win.blit(text, (width/2 - text.get_width()/2, height/2 - text.get_height()/2))
    pygame.display.update()
    pygame.time.delay(2000)

for event in pygame.event.get(): # check if the user tried to quit the game
    if event.type == pygame.QUIT:
        run = False
        pygame.quit()

    # if the user clicks
    if event.type == pygame.MOUSEBUTTONDOWN:
        pos = pygame.mouse.get_pos()
        for btn in btns:
            if btn.click(pos) and game.connected():
                if player == 0:
                    if not game.p1Went:
                        n.send(btn.text) # send info to the server as string about which move button is
played
                else:
                    if not game.p2Went:
                        n.send(btn.text)

        redrawWindow(win, game, player) # redraw the screen



def menu_screen():
    run = True
    clock = pygame.time.Clock()

    while run:
        clock.tick(60)
        win.fill((128, 128, 128))
        font = pygame.font.SysFont("comicsans", 60)
        text = font.render("Click to Play!", 1, (255,0,0))
        win.blit(text, (100,200))
        pygame.display.update()

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                run = False
            if event.type == pygame.MOUSEBUTTONDOWN:
                run = False
    main()


while True:
    menu_screen()
```

- **network.py**

```
import socket
import pickle


# Network Class
# an object of Network class is instantiated on the side of the client
# this object immediately attempts to connect to the server
# on successful connection, it is assigned an int (0 or 1) symbolizing whether the client will
# act as player 1 or player 1


class Network:
    def __init__(self):  # constructor
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # become a socket client
        self.server = "192.168.48.129"
        self.port = 5677
        self.addr = (self.server, self.port)
        self.p = self.connect()  # attempt to connect to the server and let the server assign whether am
player 1 (
        # p=0) or player 2 (p=1)


    # getter function which returns if the client is player 1 for player 2 according to the server
    def getP(self):
        return self.p


    # function which tried to connect to the server and receives an int from the server regarding
player id
    def connect(self):
        try:
            self.client.connect(self.addr)
            return self.client.recv(2048).decode()
        except socket.error as e:
            print(e)


    # function to send some data to the server
    def send(self, data):
        try:
            self.client.send(str.encode(data))  # send the encoded data as string
            return pickle.loads(self.client.recv(2048 * 2))  # accept a pickled object back from the server
and
            # unpickle it
        except socket.error as e:
            print(e)
```

- **game.py**

```
# Game class
# an object of class Game is an instance of a game session running
# with two players connected to it
# it contains all the information about the Game session between those two connected players
# (and the rounds they play, during that game session)
# terminology - two players (player 1 and player 2) connect to the server
# the server starts a Game session between them (object of this class)
# during that session multiple rounds of the rock, paper and scissors are played
class Game:
    def __init__(self, id):  # constructor
        self.p1Went = False  # has player 1 selected his move for the round?
        self.p2Went = False  # has player 2 selected his move for the round?
        self.ready = False  # is this game session ready to be started?
        self.id = id  # unique ID of this game session
        # (note: the sever will manage multiple game sessions and each session will be able to
        # accommodate multiple rounds of RPS
        self.moves = [None, None]  # moves decided by player 1 and player 2 for the latest round
        self.wins = [0, 0]  # win count of player 1 and player 2 throughout the session
        self.ties = 0  # number of ties throughout the session

    # this function takes the ID of the player (player 1 (ID:0) or player 2 (ID:1)) and returns what move
that player
    # has decided
    def get_player_move(self, p):
        """
        :param p: [0,1]
        :return: Move
        """
        return self.moves[p]

    # this function takes the ID of the player (0 or 1) along with the move decided, it updates that
move and marks
    # the player as move played: true
    def play(self, player, move):
        self.moves[player] = move
        if player == 0:
```

```
        self.p1Went = True
    else:
        self.p2Went = True

# return: is the game session ready to be started?
def connected(self):
    return self.ready

# return: has both players played their own moves or not?
def bothWent(self):
    return self.p1Went and self.p2Went

# game logic
    # 1. Get the moves of the players
    # 2. Decide the winner (or tie) based on the rules of the game.
    # 3. returns the verdict (0 = Player 1 won, 1 = Player 2 won, -1 = tie)

def winner(self):

    p1 = self.moves[0].upper()[0]
    p2 = self.moves[1].upper()[0]

    winner = -1
    if p1 == "R" and p2 == "S":
        winner = 0
    elif p1 == "S" and p2 == "R":
        winner = 1
    elif p1 == "P" and p2 == "R":
        winner = 0
    elif p1 == "R" and p2 == "P":
        winner = 1
    elif p1 == "S" and p2 == "P":
        winner = 0
    elif p1 == "P" and p2 == "S":
        winner = 1

    return winner
# reset both player move status, i.e prepare them for next round
def resetWent(self):
    self.p1Went = False
    self.p2Went = False
```

- **button.py**

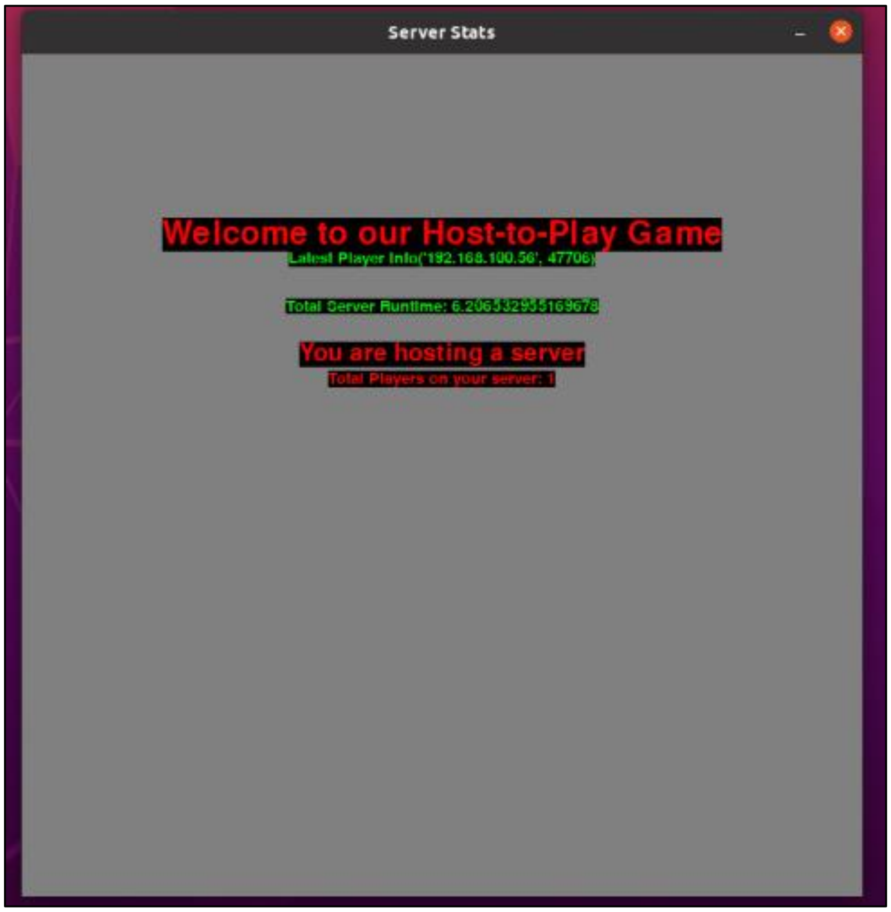
```
import pygame
class Button:
    def __init__(self, text, x, y, color):
        self.text = text
        self.x = x
        self.y = y
        self.color = color
        self.width = 150
        self.height = 100

    def draw(self, win):
        pygame.draw.rect(win, self.color, (self.x, self.y, self.width, self.height))
        font = pygame.font.SysFont("comicsans", 40)
        text = font.render(self.text, 1, (255,255,255))
        win.blit(text, (self.x + round(self.width/2) - round(text.get_width()/2), self.y +
round(self.height/2) - round(text.get_height()/2)))


    def click(self, pos):
        x1 = pos[0]
        y1 = pos[1]
        if self.x <= x1 <= self.x + self.width and self.y <= y1 <= self.y + self.height:
            return True
        else:
            return False
```

4.5 Sample Runs

The followings figures have shown a group of screenshots for the proposed application and the interfaces for both client and server perspectives.

1.**The main screen of the system**

Figure 7.The Main Screen of The System



Welcome to our Host-to-Play Game

Latest Player Info('192.168.100.56', 47706)

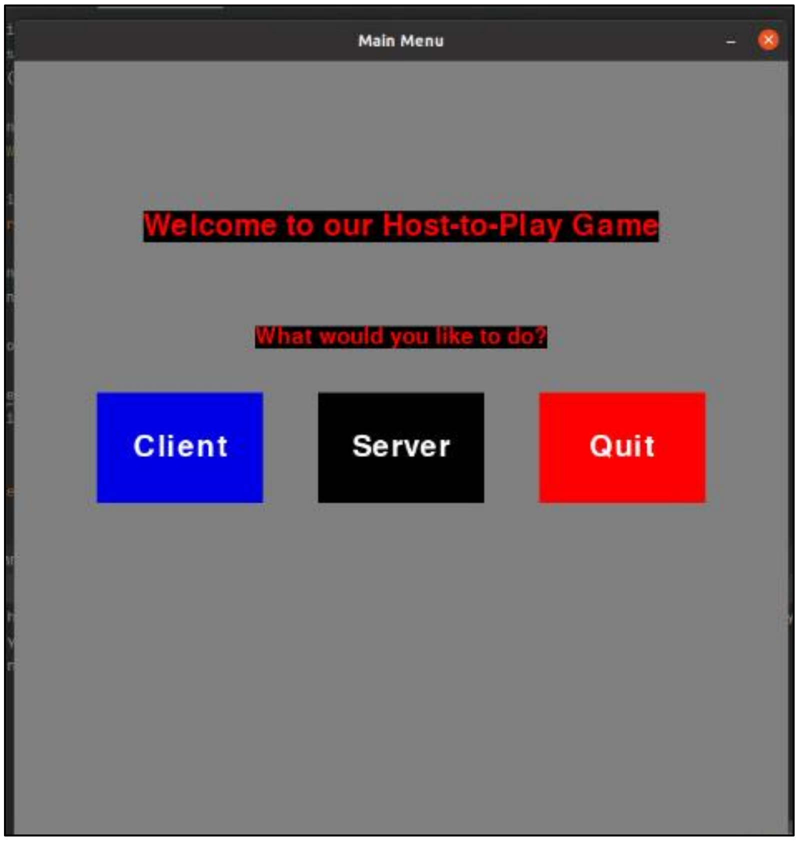
Total Server Runtime: 6.206532955169678

You are hosting a server

Total Players on your server: 1

2. **The menu screen of the system where the users have to choose to be either client or server and may close the system.**

Figure 8. Menu Screen of the System



3. **The main screen of the client interactive window where the users can start playing the game.**

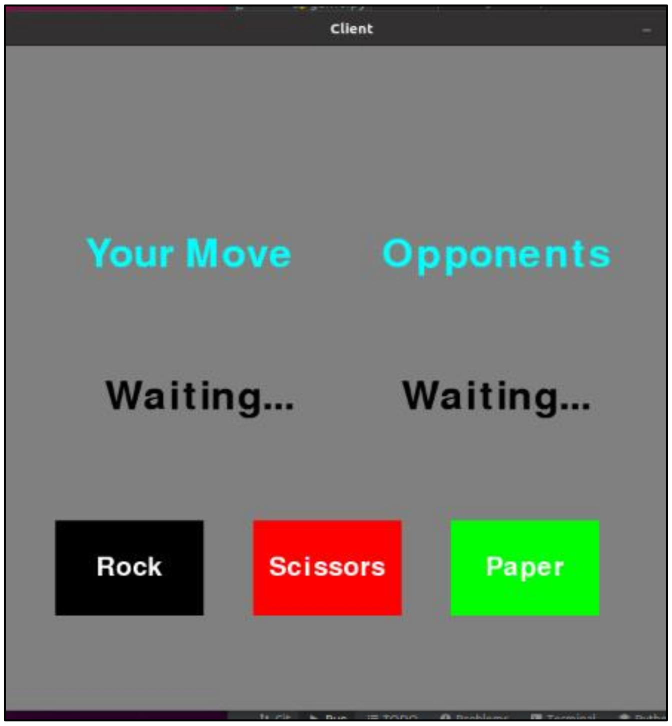
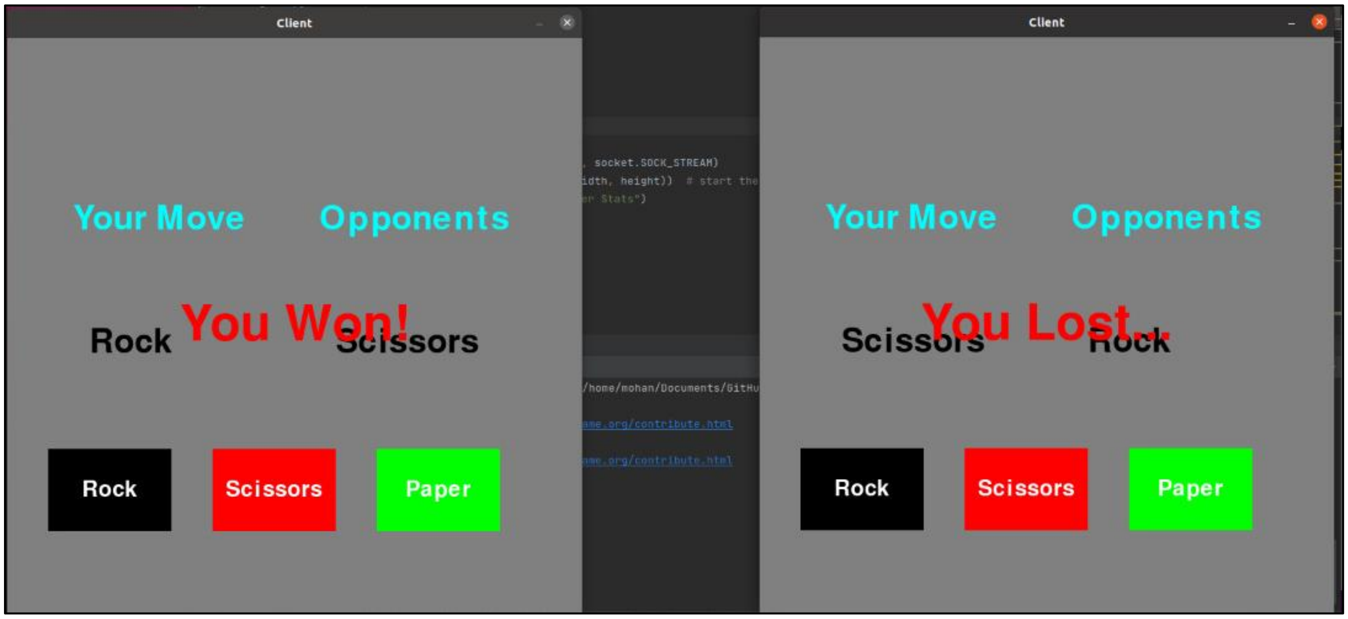


Figure 9. Main Screen as Client

4. **The below screen shows the end of the game when the game is over after a pair of clients has played the game. It shows also who is the winner and who is the loser.**



```
, socket.SOCK_STREAM)
idth, height))  # start the
er Stats")

/home/mohan/Documents/GitHu

ame.org/contribute.html

ame.org/contribute.html
```

Figure 10. Game Over

5 Conclusion

Distributed systems are everywhere in our life. They facilitate our life in many various ways. They open the doors in front of us to communicate and share different resources and services with each other using different tools and facilities that exist everywhere to be easy to use and reach. In this project, we have developed a modern distributed application which is called ‘A Client-Server based Multiplayer Rock, Paper and Scissor Game’ using various concepts and components of the distributed systems like using the different components, software, networks and others. This game application is based on a client-server interaction model that used sockets to communicate and pass messages among each other. Therefore, the main goal of this system is to allow a pair of players to play against each other anywhere and at any time they want. It is a simple game that depends on understandings the basic rules of the game and everyone in the group plays a gesture, thus within a small period of time the game is over and shows the winner and the loser. We also extended the functionality of the game client by providing an incentive based option for the game client to stop the game and act as a server for other players instead. This let us demonstrate that for large distributed systems such as Massive Multiplayer Online Games (MMOG), the responsibility of hosting servers can itself be distributed.

6 References

1. Coulouris G. , Dollimore J., Kindberg T., and Blair G. DISTRIBUTED SYSTEMS Concepts and Design. Pearson Education, Inc. 2012.
2. Corey Clark and Myque Ouellette. 2017. Video games as a distributed computing resource. In Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG '17). Association for Computing Machinery, New York, NY, USA, Article 28, 1–7. DOI:https://doi.org/10.1145/3102071.3102099
3. Jennings, N. (2018). *Socket Programming in Python - Guide*. Retrieved from Real Python: https://realpython.com/python-sockets/
4. IBM Corporation. (2015). *How sockets work*. Retrieved from IBM: https://www.ibm.com/docs/en/i/7.3?topic=programming-how-sockets-work
5. Kalita, L., 2014. Socket programming. *International Journal of Computer Science and Information Technologies*, *5*(3), pp.4802-4807.
6. geeksforgeeks.org. (2021, Aug 31). *Socket Programming in Python*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/socket-programming-python/
7. Diot C. and Gautier L., "A distributed architecture for multiplayer interactive applications on the Internet," in IEEE Network, vol. 13, no. 4, pp. 6-15, July-Aug. 1999, doi: 10.1109/65.777437.
8. Coursera.org. (2021, Sep 22). *What Is Python Used For? A Beginner's Guide*. Retrieved from Coursera: https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python