MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DOOM 64 ONLINE

LAN based multiplayer game.

Mini Project Report

CSE3162 COMPUTER NETWORKS LAB

Student: Rudra Patel

Registration No: 210905324

Course: CSE3162

10-Nov-2023

Report

Abstract

This project, Doom64 Online, revisits the classic Doom 64 game, introducing a LAN-based multiplayer aspect. Central to the project is the development of an efficient server-client architecture to ensure smooth data exchange among multiple players. The game offers a first-person shooter experience with continuous gameplay until server shutdown. Key features include robust multi-threaded server support, seamless join-and-play mechanics, and client-side rendering for fluid graphics. Player interactions, like movement and shooting, are managed through server-client communications using JSON for data transfer. Resources for the graphical user interface were adapted from an existing repository, focusing mainly on the computer networking components of the game. This project serves as a model for implementing multiplayer functionality in retro-styled games, leveraging modern network programming techniques.

Keywords: Multiplayer, LAN, FPS, Doom 64, Server-Client Architecture, Networking, Game Development, Python, Pygame, JSON

Contents

1. Introduction

This report delves into the analysis of a complex and intriguing LAN-based multiplayer adaptation of the classic game Doom 64, named "Doom64 Online." The project has been selected for its intricate blend of retro gaming elements and advanced networking concepts. This fusion presents an opportunity to explore and understand the challenges and intricacies involved in modernizing a classic game for contemporary networked multiplayer environments.

1.1 Features of Doom64 Online

In "Doom64 Online," we have adapted and implemented several core features from the original Doom 64, tailored to create an engaging LAN-based multiplayer experience. The key features implemented in our adaptation are:

- **Multiplayer Interaction:** Players can see and interact with other players within the game environment, enhancing the multiplayer experience and fostering a sense of community and competition.
- **Shooting Mechanics:** A fundamental aspect of the game is the ability to engage in combat by shooting at other players, adding a competitive and interactive dimension to the gameplay.
- **Player Movement:** Players have the ability to navigate freely around the game environment, enabling exploration and strategic positioning during gameplay.
- **3D Visualized Environment:** The game offers a fully immersive 3D environment, meticulously designed to replicate the atmospheric and intricate level design of the original Doom 64.

These features are central to "Doom64 Online," offering players a modernized version of the classic game with enhanced networked multiplayer capabilities.

1.2 Reason for Selection

The decision to analyze this project is driven by the following:

- **Technical Insights:** The use of Python and Pygame [4] offers a unique opportunity to apply networking principles in a game development context, providing insights into real-time data transfer, concurrency handling, and network efficiency in multiplayer games.
- **Educational Value:** It serves as an educational tool for understanding the application of socket programming, UDP communication, and game loop management in a Python-based environment.
- **Personal Interest:** The choice is also influenced by a personal fascination with the technical aspects of game development and the unique challenges posed by networked multiplayer game programming.

- **Multiplayer Game Development:** The project offers an opportunity to learn how AAA multiplayer games are implemented at a lower level by building it from scratch, instead of merely following pre-established frameworks or methodologies. This approach allows for a deeper understanding of the underlying mechanics and challenges involved in high-end game development.

1.3 Analysis Focus and Hypotheses

The analysis revolves around several key areas:

- **Networking Implementation:** How does the project handle real-time communication and data synchronization between the server and multiple clients?
- **Concurrency Management:** How are multiple concurrent player interactions managed without compromising game performance?
- **Gameplay Integrity:** How does the networked environment affect gameplay, and what measures are in place to maintain game integrity and responsiveness?

The hypothesis is that the Doom64 Online project effectively utilizes Python's networking capabilities to create a seamless multiplayer experience, overcoming the challenges of latency and data consistency.

1.4 Research questions/hypotheses

The following research questions and hypotheses are formulated to guide the analysis of the Doom64 Online project and will be pivotal in user evaluation:

1. **Concurrency and Multiplayer Interaction:** How are concurrent actions from multiple players managed and synchronized by the server? Does the game maintain consistency in the shared game world across all clients?
2. **Scalability of the Game Architecture:** Can the network architecture of Doom64 Online efficiently scale to accommodate an increasing number of players without degrading performance or gameplay experience?
3. **Gameplay Mechanics and Network Dependence:** To what extent do the game's mechanics rely on network performance? Are there fallback mechanisms to handle network instability?
4. **Implementation Challenges:** What are the primary challenges faced during the implementation of networked multiplayer features in Doom64 Online, and how were they addressed?
5. **Comparison with AAA Game Standards:** How does the networking implementation in Doom64 Online compare with standard practices in AAA multiplayer games? Are there any noticeable differences in approaches to network management, data handling, and player synchronization?

2. Background

The challenge of efficient data sharing in multiplayer gaming environments is a well-explored domain, with numerous solutions proposed over the years. Fundamentally, this challenge revolves around the rapid and reliable transmission of game-related data such as player positions, health status, and various state variables. Two primary approaches have emerged as the most prevalent in addressing this problem: peer-to-peer (P2P) and client-server architectures.

In the peer-to-peer model, as detailed in existing literature [2], each player directly connects with other players, facilitating a decentralized exchange of data. While this approach enables direct data sharing among participants, it lacks a centralized validation mechanism, potentially leading to issues such as cheating and data inconsistency.

Conversely, the client-server model [3] centralizes data exchange, where all players connect to a dedicated server. This server acts as an authoritative source, validating and relaying data among players. This method enhances security and data integrity, as the server verifies all shared information before dissemination. However, its major drawback is the dependency on a single point of failure; server downtime directly impacts all connected players, rendering the game non-functional.

This project aims to delve into the intricacies of implementing a robust client-server communication system for a specific multiplayer game. We explore both UDP and TCP protocols to determine their suitability in different gaming scenarios, comparing their efficiency in handling real-time data exchange. Additionally, we evaluate various data packet formats, ranging from binary (like Pickle) to text-based (such as JSON), to ascertain the most effective method for data serialization and transmission.

To address concurrency, a critical aspect of multiplayer gaming, we investigate several programming constructs and system calls. These include traditional methods like os.fork, as well as more modern approaches such as the multithreading and multiprocessing libraries in Python. Each method is tested for its ability to handle simultaneous requests and data processing, ensuring that the game maintains consistent performance and responsiveness under varying load conditions.

3. Data

The dataset for this project primarily revolves around the core assets and gameplay mechanics obtained from the open-source repository for a basic implementation of a DOOM-style game, accessible at [5]. This foundational codebase serves as the starting point for the implementation of our networked multiplayer adaptation, "Doom64 Online."

3.1 Data Preparation

The adaptation of the base game into a multiplayer environment necessitated the development of a robust application layer from scratch. A critical aspect of this development was the design and implementation of custom communication message classes. These classes are instrumental in managing the data flow between the game clients and the server, ensuring efficient and reliable data exchange.

For data serialization and transmission, we opted for JSON, owing to its widespread use and ease of integration with Python. The message classes were designed to encapsulate various types of game-related information, such as player positions, game state updates, and login requests/responses. The following code snippet provides an overview of these classes:

Code snippet for custom message classes

```
from settings import *
import json



class PacketClass:
    type = None
    msg = None
    def   init  (self,type=None,msg=None):
        self.type=type
        self.msg=msg

class LoginResMsg:
        x=0
        y=0
        id=None
        def   init  (self,x,y,id):
                self.x=x
                self.y=y
                self.id=id
        def getJson(self):
                return json.dumps(self.  dict  )

# this is an example implementation for one of the message
   class used for login
# (Note: For complete code, refer to the project repository
   )
```

These classes are a cornerstone in the architecture of Doom64 Online, facilitating the translation of game actions into network messages and vice versa. The JSON conversion capability integrated within these classes ensures a standardized format for data exchange, contributing to the overall consistency and reliability of the multiplayer experience.

3.2 Significance of Data Preparation

The design and implementation of these message classes were pivotal in achieving several project objectives:

- **Efficient Data Handling:** By using JSON for data serialization, the project benefits from a lightweight and easily parsable data format, crucial for real-time gaming scenarios.
- **Scalability:** The modular nature of these classes allows for easy expansion and modification, essential for accommodating future enhancements or changes in game mechanics.
- **Network Transparency:** Custom message classes provide a clear and structured way to handle various types of network communication, enhancing the maintainability and readability of the code.

In conclusion, the careful preparation and design of the data handling mechanisms in Doom64 Online play a critical role in ensuring the seamless operation and scalability of the game's multiplayer functionality.

4. Approach

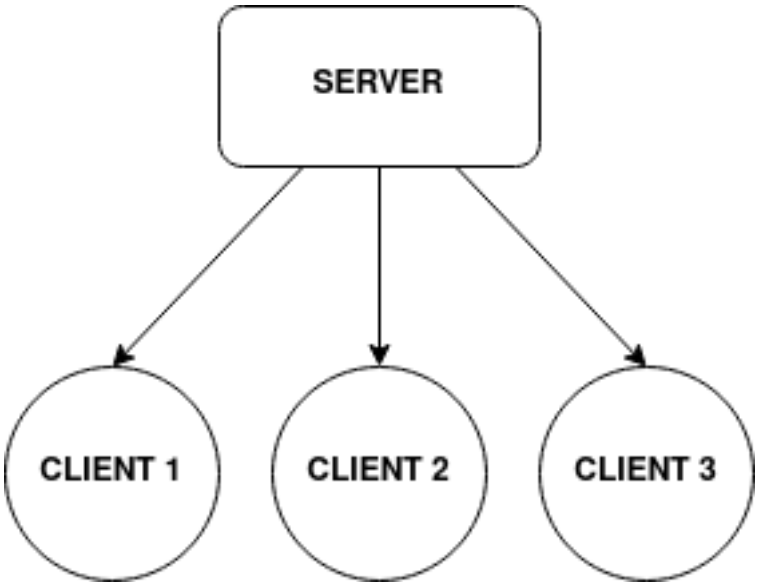
This section presents the approach adopted in the development of "Doom64 Online".

4.1 L0: Design

The project utilizes a server-client architecture for its communication system. The key features of this implementation include:

- **Concurrent Server**: The server is equipped to handle multiple clients concurrently, aiming to provide low latency for each player.
- **Game State Management**: Server-side management ensures synchronization of game events across all connected clients, maintaining gameplay consistency.
- **Client Communication System**: Custom message classes are implemented for effective encoding and decoding of messages sent between the server and clients, tailored to the type of message transmitted over the network.
- **Server Finder**: To avoid the impracticality of manually entering the server's IP address on each client, a server finder feature is implemented for automatic detection of the server on the local network.

L0 DIAGRAM:



4.2 L1: Detailed Overview

The primary communication components are consistent throughout all implementations of this project.

4.2.1 Server

The server comprises four fundamental components:

- **Shared Memory Object**: Named GAME STATE, this component holds all player data, a list of game events, and other vital information to synchronize game events across players.
- **Main Listener Class**: This communication API establishes a connection listener, assigns unique ports to clients, and initiates a new thread for each connection, managing all client communications.
- **Client Communication Thread**: A dedicated thread for each client manages all communication with that client using a custom message class for encoding and decoding messages.
- **MultiCliCon Class**: This communication API facilitates interaction between the server thread and clients. It manages message processing, login requests, and responses.

4.2.2 Client

The client includes three base components:

- **Main Connection Class**: This communication API connects to the server and handles all server communications, using a custom message class for message processing.

- **Main: Connection**: A method in the Main Connection Class for establishing a connection with the server.
- **Main: Login**: Notifies the server of client availability, receiving the player's spawn location and ID.
- **Main: Player Update**: This method sends player updates to the server, including position, yaw angle, health status, shooting actions, and target player ID. The server's response includes data for all players, determining subsequent game events.

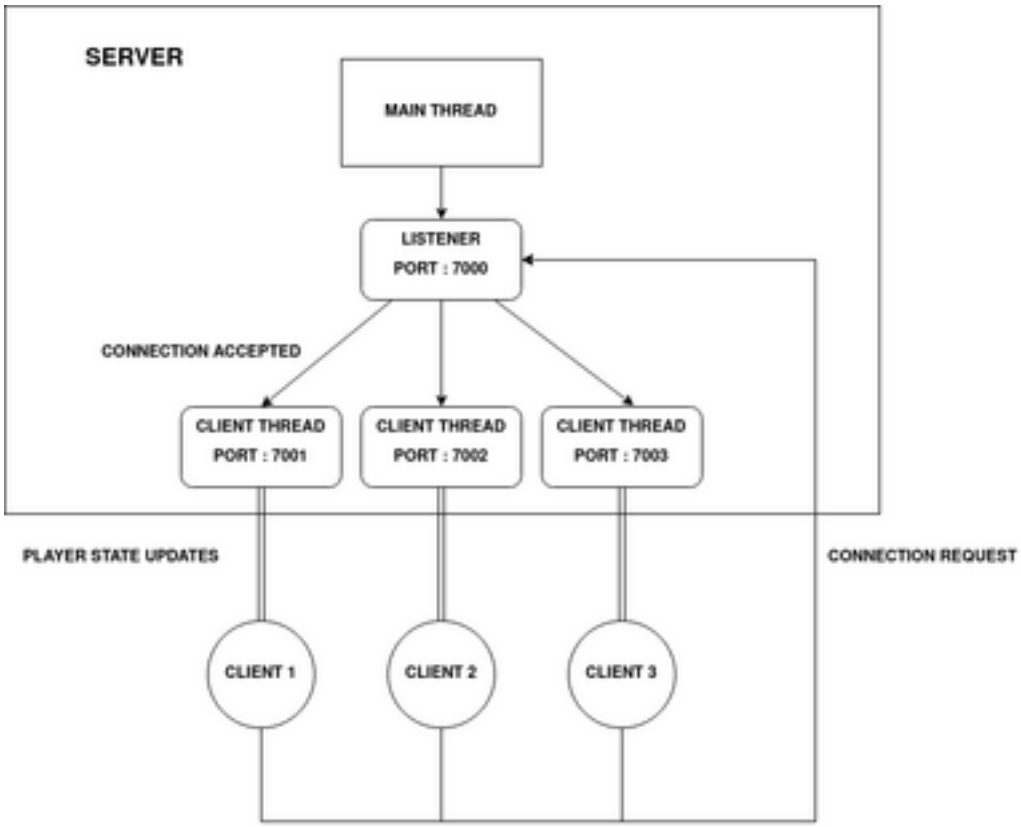
4.2.3 Communication Flow

The communication flow remains largely constant throughout the project's implementations, as follows:

- **0: SERVER**: Initiate the server on any PC within the LAN.
- **0: CLIENT**: Start the client on any LAN-connected PC and send a connection request to the server.
- **1: SERVER**: Accept the connection, register the client, and assign a unique communication port.
- **1: CLIENT**: Receive the assigned port and send a login request.
- **2: SERVER**: Process the login request, sending the client its spawn location and ID.
- **2: CLIENT**: Receive spawn details and begin the game at the assigned location.
- **3: CLIENT**: Regularly send player updates to the server.
- **3: SERVER**: Update the game state based on received player data and distribute this updated state to all clients.
- **4 : CLIENT**: Receive the updated game state and render the game accordingly.
- **Repeat steps 3 and 4 until the game ends.**

Further details for the connection between '0: SERVER' and '0: CLIENT' steps will be explored in the server finder implementation discussion.

L1 DIAGRAM:



4.3 TCP Server Implementation

The initial implementation utilized the TCP protocol, chosen for its ease of implementation and built-in features for handling new client connections via port assignments.

4.3.1 TCP Fallbacks and Challenges

While TCP was employed in the initial server setup, it proved suboptimal for real-time multiplayer gaming [1]. The protocol introduced noticeable latency and wait times for clients, attributable to TCP's inherent design for reliable data transmission through packet acknowledgment. Although this ensures data integrity, crucial in many applications, it hampers the speed and responsiveness needed in real-time gaming.

Nevertheless, certain project features were conducive to TCP implementation. Consequently, a hybrid approach was adopted, incorporating UDP for faster communication while integrating a simplified version of TCP's reliability system for specific functionalities.

4.4 UDP Implementation

4.4.1 UDPServer Class

The UDPServer class is built on the UDP protocol, prioritizing reduced latency over reliability to enhance real-time communication efficiency.

UDP Server Functionality In this implementation, the server retains some TCP functionalities: it waits for client connection requests, assigns unique ports to each client for dedicated communication, and creates new threads for concurrent communication.

Client-Server Communication Communication between the client and server is managed through custom message classes. These classes encode messages into JSON strings for network transmission. The code snippets for these custom classes are as follows:

```
class PacketClass:
    type = None
    msg = None

    def __init__(self,type=None,msg=None):


class LoginResMsg:
    type = None
    msg = None

    def __init__(self,x,y,id):

    def getJson(self):
        return json.dumps(self.__dict__)



class JsonPacket:
    type = None
    msg = None
    def __init__(self,type=None,msg=None):
        if type :
            self.type=type
            if self.type!=LOGIN_REQ and self.type !=
               UDP_CON_REQ:
                self.msg=json.loads(msg)
            else:
                self.msg = msg

    def udpConnReq(self):
        self.type = UDP_CON_REQ
        self.msg = None
        return self.getJson()

    def udpConnRes(self,socket):
        self.type = UDP_CON_RES
```

```
        self.msg = str(socket)
        return self.getJson()

    def loginReq(self,username):
        self.type=LOGIN_REQ
        self.msg=username
        return self.getJson()

    def loginRes(self,spawn):
        self.type=LOGIN_RES
        self.msg=spawn.getJson()
        return self.getJson()

    def playerUpdate(self,player):
        self.type=PLAYER_UPDATE
        self.msg=player.getJson()
        return self.getJson()

    def gameStateUpdate(self,game_state):
        self.type=GAME_STATE
        self.msg=game_state.getJson()
        return self.getJson()

    def getJson(self):
        return json.dumps(self.__dict__)




class DataPlayer:
    position=[0,0]
    yaw=0
    shoot=False
    health = None
    shotWho=None
    def __init__(self,position=[0,0],yaw=0,shoot=False,
       health=None,shotWho=None):
        self.position=position
        self.yaw=yaw
        self.shoot=shoot
        self.health=health
        self.shotWho=shotWho
    def update(self,position=[0,0],yaw=0,shoot=False):
        self.position=position
        self.yaw=yaw
```

```
        self.shoot=shoot
    def show(self):
        print(self.__dict__)
    def __str__(self):
        return str(self.__dict__)
    def getJson(self):
        # print(self.__dict__)
        return json.dumps(self.__dict__)
```

4.5 Game State Management

The game state is dynamically updated based on data received from each client. In this architecture, the server primarily functions as a communication medium, with the game logic being executed client-side. To illustrate this concept, consider an example involving two players, Player A and Player B. When Player A shoots Player B, the client-side code of Player A recognizes this action and updates the server. Subsequently, the server informs Player B's client to adjust its health accordingly.

4.5.1 Game State Class Functions

The Game State Class encompasses several key functions:

- It stores and continuously updates player data.
- It provides methods for serializing the game state into JSON format, facilitating efficient network transmission.
- It implements specific game logic, such as ray casting algorithms, to determine player visibility and other in-game interactions.

The code implementation for these functions is provided below:

```
class Game_State:
    running=False
    players={}
    # score=0
    map_obj=None

    def __init__(self,running=False,players={},map_obj=None
       ):
        self.running=running
        self.players=players
        self.map_obj=map_obj

    def get_players_json(self,id):
        temp={}
        for i , player in self.players.items():
            temp[str(i)]=str(player.__dict__)
```

```
        temp[str(-1)]=str(id)
        return json.dumps(temp,indent=1)

    def show(self,id =None):
        print("Game State: ###########\n")
        if id:
            for i, player in self.players.items():
                if i != id:
                    print(f"Player {i}: {player}")
        else:
            for i, player in self.players.items():

                print(f"Player {i}: {player}")
        print("############\n\n")
        # for i in self.players:
        #       i.show()

    def getJson(self):
        return json.dumps(self.players)
```

4.6 Pseudocode for Server Flow

1. UDPMultiCliCon Class:
- Create a UDP socket and bind it to a specified address and port.
- Include methods to:
  - Receive data from clients.
  - Accept login requests and allocate spawn positions.
  - Update the game state and send it to clients.

2. UDPServerCon Class:
- Create a UDP socket for the main server connection.
- Include methods to:
  - Accept incoming messages.
  - Send connection information to clients (like assigned ports).

3. UDPServer Class:
- Initialize game state and a dictionary to manage client handlers.
- Assign new ports for clients to maintain separate communication channels.
- In the `start` method:
  - Continuously listen for new connections.
  - For each new connection, assign a port and start a new thread for handling t
  - Send the client their unique communication port.
- In the `handle_client` method:
  - Manage communication with individual clients.
  - Update the game state based on client actions.
  - Handle client disconnections and remove them from the game state.

4. Main Server Execution:
   - Define the server host and port.
   - Create an instance of the UDPServer class.
   - Call the `start` method on the server instance to begin listening for and hand

4.7 Client-Side Implementation

The client-side architecture of the project is centered around a primary class, referred to as the Main Connection Class, which is responsible for managing communication with the server. This class utilizes the previously mentioned message classes to format data for transmission. Key functionalities include:

- Establishing and maintaining a stable connection with the server.
- Encoding and decoding data using custom message classes.
- Handling real-time data transmission to ensure synchronized gameplay.

Detailed code for the Main Connection Class and its associated functionalities will be provided in the final section of this document.

```
class UDPCon:
    s=None
    sendMsg=None
    revMsg=None
    cliAddr = None
    port = None
    host=HOST
    def __init__(self,host=None,port=None):
        self.s = socket.socket(socket.AF_INET, socket.
           SOCK_DGRAM)
        self.s.settimeout(0.5)
        if host:
            self.host=host
    def accMsg(self):
        self.revMsg , self.cliAddr = self.s.recvfrom(
           RECIEVE_BUFFER_SIZE)
        self.revMsg = json.loads(self.revMsg)
        print("DATA␣REC␣"+self.recvMsg["type"],self.
           recvMsg["msg"])
        self.recvMsg=JsonPacket(self.recvMsg["type"],
           self.recvMsg["msg"])
        if self.recvMsg.type != UDP_CON_REQ:
            return None,None
        return self.recvMsg,self.cliAddr

    def sendConnMsg(self):
        self.sendMsg=JsonPacket()
        self.sendMsg=self.sendMsg.udpConnReq()
```

```
        sent =False
        while not sent:
            try:
                self.s.sendto(self.sendMsg.encode("utf
                    -8"), (self.host,PORT))
                self.revMsg = self.s.recvfrom(
                    RECIEVE_BUFFER_SIZE)[0]
                sent = True
            except Exception as e:

                print(e)
                sent = False
                continue
        self.revMsg = json.loads(self.revMsg)
        print("DATA␣REC␣"+self.revMsg["type"],self.
            revMsg["msg"])
        return int(self.revMsg["msg"])
        # return 7001

    def login(self,username):

        print(self.host,self.port)

        self.sendMsg=JsonPacket()
        self.sendMsg=self.sendMsg.loginReq(username)
        sent = False
        while not sent:
            try:
        # sleep(0.5)
                self.s.sendto(self.sendMsg.encode("utf
                    -8"), (self.host,self.port))
            # except timeout of socket


                self.revMsg = self.s.recvfrom(
                    RECIEVE_BUFFER_SIZE)[0]
                sent = True
            except Exception as e:
                print(e)
                sent = False


        print(self.revMsg)
        self.revMsg = json.loads(self.revMsg)
        self.revMsg=JsonPacket(self.revMsg["type"],self
            .revMsg["msg"])
        # print([int(self.revMsg.msg['x']),int(self.
```

```
            revMsg.msg['y']),int(self.revMsg.msg['id'])
            ])
        return [int(self.revMsg.msg["x"]),int(self.
            revMsg.msg["y"]),int(self.revMsg.msg["id"])]

    def playerUpdate(self,player):
        self.sendMsg=JsonPacket()
        self.sendMsg=self.sendMsg.playerUpdate(player)

        sent = False
        while not sent:
            try:

                self.s.sendto(self.sendMsg.encode("utf
                    -8"),(self.host,self.port))
                self.revMsg = self.s.recvfrom(
                    RECIEVE_BUFFER_SIZE)[0]
                sent = True
            except Exception as e:
                print(e)
                sent = False
                continue



        print(self.revMsg)
        self.revMsg = json.loads(self.revMsg)
        self.revMsg = JsonPacket(self.revMsg["type"],
            self.revMsg["msg"])
        return self.revMsg
```

The client is initialized and connected to the server by sending a formatted connection message. The initial connection setup is as follows:

```
connector = UDPCon(host=host)
porttemp = connector.sendConnMsg()
```

Once connected, the player logs into the game to receive their initial spawn location and ID, as detailed in the UDPCon class:

```
spawn_location = connector.login(username)
```

Subsequently, during each game cycle, the playerUpdate method in the connector is invoked, and player updates are sent to the server:

```
obj = DataPlayer([self.player.pos], self.player.angle, self
    .player.shot, self.player.health, self.player.shotWho)
game_state_res = self.connector.playerUpdate(obj)
```

The player update includes the following data:

- New position
- New angle
- Shooting status (whether the player has shot or not)
- Health of the player
- ID of the player who has been shot, if applicable

This process ensures real-time synchronization of player actions and states between the client and the server.

4.8 Server Finder

The Server Finder is an integral part of the system, as discussed in Section 4.2.3. This tool bridges the gap between starting the server and establishing a connection with it. The Server Finder operates on the local network using the UDP protocol to discover the server. It broadcasts a message on a predefined port, which is consistent across all clients.

A parallel UDP communication system runs alongside the main server on the designated server PC. This system listens for broadcast messages on the specified port. Upon receiving a broadcast, the script responds with the main server's IP address and port number, thus facilitating the client's connection to the server.

The process involves these steps:

- Broadcasting a UDP message across the local network.
- Listening for these broadcasts on the server-side via a dedicated UDP script.
- Responding to the client with the server's IP address and port number.

The Server Finder utilizes network scanning tools, such as the 'nmap' command, to identify active interfaces on the local network, enabling the effective location of the server. This approach ensures a seamless and automated process for clients to connect to the server within the local network.

```
def find_all_ips():
    hostname = socket.gethostname()
    IPAddr = socket.gethostbyname(hostname)


    print("Your␣Computer␣Name␣is:" + hostname)
    print("Your␣Computer␣IP␣Address␣is:" + IPAddr)
    secs=IPAddr.split(".")
    netId= ""
    for i in range(len(secs)-1):
        netId+= secs[i]+"."
    netId+="0"
    print(netId)
```

```
os.system("nmap␣-n␣-sn␣"+netId+"/24␣-oG␣-␣|␣awk␣'/Up$/{
   print␣$2}'␣>␣ips.txt")
```

Continuing with the Server Finder process, the system utilizes the previously identified active IP addresses on the local network. These addresses are used to broadcast a server-finding message. The sequence of actions is as follows:

- The system broadcasts a Server Finder message to the stored IP addresses on the local network.
- The server, upon receiving this message, is programmed to respond with a specific acknowledgment.
- If the response matches the expected format, the IP address and port of the server are then captured and stored by the client.

This mechanism ensures that clients within the network can automatically detect and connect to the server without manual input of the server's IP address and port. The process not only streamlines the connection setup but also adds a layer of dynamic interaction in network environments where server details might change or multiple servers might be present.

```
def find_server():
    file1 = open('ips.txt', 'r')
    Lines = file1.readlines()

    count = 0

    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(FINDER_TIMEOUT)
    line =None
    # Strips the newline character
    for line in Lines:
        try:
        # print("-"+line[:-1]+"-")
            s.sendto("server?".encode("utf-8"), (line[:-1],
               SERVER_FINDER_PORT))
            revMsg , cliAddr = s.recvfrom(
               RECIEVE_BUFFER_SIZE)
            if revMsg:
                print("request␣->␣"+line+ "\nresponse␣->␣"
                   +revMsg.decode())
                break
            # os.system("ping "+line+" "+str(
               SERVER_FINDER_PORT))
            count += 1
        except Exception as e:
            print(e)
```

```
            continue
    if line:
        return line[:-1]
    else:
        return HOST
```

5. Results

The final implementation of the multiplayer first-person shooter game showcases a robust network architecture, capable of handling real-time communications with minimal latency. The game state synchronization across clients was successfully achieved, allowing for a seamless multiplayer experience. Key findings include the efficacy of the hybrid TCP/UDP implementation and the dynamic server discovery mechanism, both of which significantly enhanced the game's performance and usability.

6. Discussion

The results demonstrate a successful balance between reliable data transmission and low-latency communication, crucial for real-time multiplayer gaming. The implementation of a concurrent server with game state management validated the project's approach to handling multiple client interactions simultaneously. The effectiveness of the Server Finder tool in simplifying network connectivity also stands out as a significant achievement.

6.1 Limitations and Challenges

Given more time, further optimization of the client-server data exchange could be explored to enhance game performance under various network conditions. One challenge faced was balancing the trade-off between the reliability of TCP and the speed of UDP, particularly in maintaining game state consistency across clients.

7. Conclusion

This project has yielded valuable insights into the design and implementation of network systems for multiplayer games. The key achievements include the development of an efficient game state synchronization mechanism and a user-friendly server discovery process. However, further investigations into optimizing network traffic and exploring alternative data transmission methods could provide additional improvements.

8. Reflections on own work

- The scope of the project was refined to focus primarily on network implementation, driven by the challenges in real-time multiplayer gaming.
- Extensive research into network protocols and multiplayer game architectures guided the project's direction.
- Implementing, testing, and validating the results were informed by a combination of academic research and practical examples in the gaming industry.
- Sources like technical forums, networking textbooks, and multiplayer gaming case studies were instrumental in overcoming technical challenges.
- Starting over, more emphasis would be placed on early-stage testing of network protocols to better understand their impact on gameplay.
- In retrospect, exploring more advanced server discovery mechanisms could have further enhanced the project.

Further work on this project could involve the integration of WebSockets to enable internet-based gameplay, expanding the game's reach beyond local networks. This would involve tackling additional challenges related to broader network variability and security considerations.

Bibliography

[1] Kuan-Ta Chen, Chun-Ying Huang, Polly Huang, and Chin-Laung Lei. An empirical evaluation of tcp performance in online games. page 5, 06 2006. doi: 10.1145/1178823.1178830.

[2] Abdulmotaleb El Saddik and A. Dufour. Peer-to-peer suitability for collaborative multiplayer games. pages 101– 107, 11 2003. ISBN 0-7695-2036-7. doi: 10.1109/DISRTA.2003.1243003.

[3] Abdul Khan, Ivica Arsov, Marius Preda, Sophie Chabridon, and Antoine Beugnard. Adaptable client-server architecture for mobile multiplayer games. page 11, 03 2010. doi: 10.4108/ICST.SIMUTOOLS2010.8704.

[4] Pete Shinners. Pygame. http://pygame.org/, 2011.

[5] StanislavPetrovV. Doom-style-game. https://github.com/StanislavPetrovV/DOOM-style-Game/tree/main, 2020.