server.py

```
import socket
import threading
import time

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
from cryptography.hazmat.backends import default_backend
import secrets # for randbits, Diffie Hellman

from sql_orm import Users_db

import struct
import math


# Player data structure
class Player:
    def __init__(self, username, id):
        self.username = username
        self.player_id = id
        self.dist2wall = -1
        self.position_x = -1
        self.position_y = -1
        self.rotation_x = -1
        self.rotation_y = -1

        self.health = 100
        self.dead = False
        self.score = 0

        self.last_message_time = None

        self.events = []

    def update_events(self, received_events: bytes):
        index = 0
        while index < len(received_events):
            event_size = received_events[index]
            event = received_events[index:index + event_size]
            if(event in self.events):
                self.events.remove(event)
            index += event_size

    def add_event(self, event: bytes):
        self.events.append(event)

# Define server address and port
TCP_PORT = 3000  # Separate port for TCP communication
UDP_PORT = 3001
players:list[Player] = []
struct_format = 'iffffii16s'
struct_size = struct.calcsize(struct_format)
player_binaries = b'\x00'
```

server.py

```
current_player_id = 0
key_bytes : dict = {}


# gets socket and string to send
def send(client, msg: str):
    send_bytes(client, msg.encode())

def send_bytes(client, msg: bytes):
    msg_length = len(msg) # get length int
    msg_length = msg_length.to_bytes(2, byteorder='little') # int to
    byte
    full_message = msg_length + msg # append msg length to msg
    client.send(full_message)

# returns msg bytes
def recvfrom(client) -> bytes:

    try:
        msg_length_bytes = client.recv(2)

        if len(msg_length_bytes) != 2:
            raise ConnectionError("Wrong message length received")

        msg_length = int.from_bytes(msg_length_bytes,
        byteorder='little')

        # Receive the actual message data
        message = client.recv(msg_length)

        if len(message) != msg_length:
            raise ConnectionError("Incomplete message received")

        return message

    except socket.error as e:
        raise ConnectionError(f"Socket error while receiving data:
        {e}") from e

def send_UDP(socket:socket.socket, address, msg: bytes):
    socket.sendto(msg, address)

# returns msg bytes
def recvUDP(udp_server : socket.socket) -> tuple[bytes, any]:

    try:
        msg_bytes, address = udp_server.recvfrom(256)


        # Receive the actual message data

        #if len(msg_bytes) != 33:
        #    raise ConnectionError("Incomplete message received (len:
```

```
        {0})", len(msg_bytes))

        return msg_bytes, address

    except socket.error as e:
        raise ConnectionError(f"Socket error while receiving data:
        {e}") from e


# # Function to broadcast game state to all players
# def broadcast_game_state(players):
#     # Prepare game state data (replace with your game state
representation)
#     game_state = [player.to_dict() for player in players]
#     data = pickle.dumps(game_state)

#     for player in players.values():
#         player.udp_socket.sendto(data, (SERVER_ADDRESS, address[1]))

def key_to_bytes(key):
    s = b''
    for i in range(16):
        s = int.to_bytes(key & 255, 1) + s
        key >>= 8

    return s

def pad_bytes(message_bytes: bytes, block_size):
    padding_size = block_size - (len(message_bytes) % block_size)
    padding = bytes([padding_size]) * padding_size
    return message_bytes + padding

def clean_bytes(dirty: bytes):
    return ' '.join([format(byte, '02X') for byte in dirty])

def unpad_bytes(padded_bytes: bytes):

    padding_size = padded_bytes[-1]  # Get the last byte, which
    represents the padding size
    if padding_size == 0 or padding_size > len(padded_bytes):
        raise ValueError(f"Invalid padding, looking at final char")

    # Verify that the padding bytes are all the same
    expected_padding = bytes([padding_size]) * padding_size
    if not padded_bytes.endswith(expected_padding):
        raise ValueError(f"Invalid padding, looking at the last
        {padding_size} bytes")

    # Remove the padding bytes
    unpadded_bytes = padded_bytes[:-padding_size]
    return unpadded_bytes

def encrypt_AES(plaintext: bytes, key):
```

```
    blocks = pad_bytes(plaintext, 16)
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=backend)
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(blocks) + encryptor.finalize()
    return ciphertext


def decrypt_AES(cipherbytes: bytes, key):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=backend)
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(cipherbytes) + decryptor.finalize()
    return unpad_bytes(plaintext)

def add_player(username):
    global players, player_binaries, current_player_id


    new_guy = Player(username, current_player_id)
    players.append(new_guy)


    player_binaries += int.to_bytes(current_player_id, 1) * struct_size
    current_player_id += 1

    player_binaries = int.to_bytes(len(players), 1) +
    player_binaries[1:]

    someone_joined(new_guy) # notify all players that this guy joined

    # tell all players about all players
    for player in players:
        give_usernames(player)


def remove_player(player_id):
    global player_binaries, players

    # Iterate through the players list
    for player in players:
        # Check if the current player's username matches the target
        username
        if player.player_id == player_id:
            # Remove the player from the list
            players.remove(player)
            break
    else:
        # If the player with the specified username is not found
        print(f"Player with id #{player_id} not found.")
        return

    # get index of this players buffer
```

```
    buffer_index = 1
    while True:
        if buffer_index >= len(player_binaries):
            print("Something fucked when looking for this player in the
            binaries", buffer_index)
            return None

        if player_binaries[buffer_index] == player_id:
            break

        buffer_index += struct_size

    # remove this players buffer from the binaries
    player_binaries = player_binaries[:buffer_index] +
    player_binaries[buffer_index+struct_size:]

    player_binaries = int.to_bytes(len(players), 1) +
    player_binaries[1:]

    event = int.to_bytes(5, 1) # someone left
    event += int.to_bytes(player_id, 1) # user id
    send_event(event)

    print(f"Player with id #{player_id} removed successfully.")


def handle_client(client_socket, address, users_db:Users_db, lock:
threading.Lock):
    global players, key_bytes, current_player_id



    # Diffie Hellman
    (p, g) = 170141183460469231731687303715884105757,
    340282366920938463463374607431768211507
    secret = secrets.randbits(128)


    # X2
    x2 = pow(g, secret, p)
    send(client_socket, 'X' + str(x2) + 'X')

    # X1
    x1 = recvfrom(client_socket).decode()
    if(not (x1[0] == 'X' and x1[-1] == 'X')):
        print("Incorrect X1 received. disconnecting client")
        send(client_socket, "ERROR")


    x1 = int(x1[1:-1])

    # Encryption
```

server.py

```
# Key
key = pow(x1, secret, p)

current_key_bytes = key_to_bytes(key)

#print("key_bytes: " + str(key_bytes))

cipherbytes = recvfrom(client_socket)
# print("cipherbytes: " + ciphertext.hex())


message = decrypt_AES(cipherbytes, current_key_bytes)

#print("got this: " + str(message))

parts = message.decode().split('~')
print(parts)

# users_db.insert_new_user(parts[1], parts[2])
# users_db.remove_user(parts[1])

lock.acquire()

response = "ERROR~Message Unrecognized"
if(parts[0] == "LOGIN"):
    if(users_db.user_exists(parts[1], parts[2])):
        response = "SUCCESS~" + str(current_player_id) + "~"
        key_bytes[current_player_id] = current_key_bytes

        add_player(parts[1])

    else:
        response = "FAIL~Username Or Password Incorrect~"
elif(parts[0] == "SIGNUP"):
    if(users_db.username_exists(parts[1])):
        response = "FAIL~Username Already Exists~"
    elif(users_db.insert_new_user(parts[1], parts[2])):
        response = "SUCCESS~"
    else:
        response = "FAIL~Can't Add User For Some Reason~"



lock.release()
# print("sending this: " + ' '.join([format(byte, '02X') for byte
in cipherbytes]))
print(f"{response=}")
send_bytes(client_socket, encrypt_AES(response.encode(),
current_key_bytes))



client_socket.close()
```

server.py

```
def dot_product(x1, y1, x2, y2):
    return x1*x2 + y1*y2

def send_event(event: bytes):
    event = int.to_bytes(len(event)+1 , 1) + event # size
    for player in players:
        player.add_event(event)

# give the new player the usernames of the rest
def give_usernames(new_player : Player):
    for player in players:
        if player == new_player: continue

        print(f"telling {new_player.player_id} about
        {player.player_id}'s username")

        event = int.to_bytes(4, 1) # username giving
        event += int.to_bytes(player.player_id, 1) # user id
        event += int.to_bytes(player.score, 1) #score
        event += player.username.encode() + b'\0' # username
        event = int.to_bytes(len(event)+1 , 1) + event # size

        print("sending ", clean_bytes(event))

        new_player.add_event(event)

# events: size byte, type byte, data bytes
def someone_joined(player: Player):
    event = int.to_bytes(3, 1) # new user event
    event += int.to_bytes(player.player_id, 1) # user id
    event += player.username.encode() + b'\0' # username

    send_event(event)


def someone_died(killer: Player, victim: Player):
    killer.score += 1

    event = int.to_bytes(2, 1) # killing
    event += int.to_bytes(killer.player_id, 1) # killer
    event += int.to_bytes(victim.player_id, 1) # killee
    send_event(event)

def someone_got_shot(shooter: Player, shootee : Player):
    if shootee.dead:
        return # can't shoot a dead person

    print(f"player {shooter.player_id} shot player
    {shootee.player_id}")


    # event details what happened.
```

```
    event = int.to_bytes(1, 1) # shooting
    event += int.to_bytes(shooter.player_id, 1) # shooter
    event += int.to_bytes(shootee.player_id, 1) # shootee
    send_event(event)

    # decrease health
    shootee.health -= 40
    if shootee.health <= 0:
        someone_died(shooter, shootee)


def player_distance(one:Player, two:Player):
    return math.sqrt((one.position_x - two.position_x)**2 +
    (one.position_y - two.position_y)**2)

def is_pointing_at(pointer: Player, pointee: Player):

    dir_x, dir_y = math.cos(pointer.rotation_x),
    math.sin(pointer.rotation_x)
    a = dot_product(dir_x, dir_y, dir_x, dir_y)

    qc_x, qc_y =  pointee.position_x - pointer.position_x,
    pointee.position_y - pointer.position_y
    b = -2 * dot_product(dir_x, dir_y, qc_x, qc_y)

    r = 0.4
    c = dot_product(qc_x, qc_y, qc_x, qc_y) - r*r

    discy = b*b -4*a*c
    return discy > 0


def handle_gun_shot(player: Player):
    for i in range(len(players)):
        if players[i] == player: continue

        dist2other = player_distance(player, players[i])

        if dist2other < player.dist2wall and is_pointing_at(player,
        players[i]):
            someone_got_shot(player, players[i])


def update_player(player_info: bytes):
    global players, player_binaries

    if(len(player_info) != struct_size):
        print("Invalid Player Info Received")
        return None

    player_id, dist2wall, pos_x, pos_y, rot_x, rot_y, flags, score,
    username_bytes = struct.unpack(struct_format, player_info)
```

```
    for player in players:
        if player.player_id == player_id:
            break
    else:
        print("Got message from nonexistant player, id=", player_id)
        return None

    player.position_x = pos_x
    player.position_y = pos_y
    player.rotation_x = rot_x
    player.rotation_y = rot_y
    player.dist2wall = dist2wall


    has_quit = flags & 8
    if(has_quit):
        remove_player(player_id)
        return None

    gun_shot = flags & 4
    if(gun_shot):
        handle_gun_shot(player)

    dead_flag = flags & 32
    if(not dead_flag and player.dead): # player came back to life
        player.health = 100

    player.dead = dead_flag

    # get index of this players buffer
    buffer_index = 1
    while True:
        if buffer_index >= len(player_binaries):
            print("Something fucked when looking for this player in the
            binaries", buffer_index)
            return None

        if player_binaries[buffer_index] == player_id:
            break

        buffer_index += struct_size

    # update this players buffer in the binaries
    player_binaries = player_binaries[:buffer_index] + player_info +
    player_binaries[buffer_index+struct_size:]

    player.last_message_time = time.time()

    return player

def handle_game():
    global key_bytes, player_binaries
```

server.py

```
# UDP
udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_socket.bind(('0.0.0.0', UDP_PORT))



while(True):
    msg, address = recvUDP(udp_socket)


    player_id, encrypted = msg[0], msg[1:]

    correct_key_bytes = key_bytes[player_id]

    try:
        decrypted = decrypt_AES(encrypted, correct_key_bytes)
        player_info = decrypted[:struct_size]
        received_events = decrypted[struct_size:]

    except Exception as e:
        print(f"invalid message from {address}, error: {e}")
        print(f"encrypted:", clean_bytes(encrypted))
        print()
        continue



    player = update_player(player_info)

    if(player == None):
        continue

    # print(f"Sending Player #{player_id} These Binaries:",
    player_binaries[0])
    # for i in range(len(players)):
    #     print(f"player {i}.",
    clean_bytes(player_binaries[1+struct_size*i:1+struct_size*i+struct_size])
    # print()


    #delete the events that the player already received
    player.update_events(received_events)

    response = player_binaries + b''.join(player.events)
    #player.events = b'' # reset events.


    udp_socket.sendto(encrypt_AES(response , correct_key_bytes),
    address)


    #print(end - start)
    #print("boutta send UDP: " + ' '.join([format(byte, '02X') for
```

```
        byte in others]))

def remove_idles():
    while(True):
        now = time.time()
        for player in players:
            if(player.last_message_time and now -
            player.last_message_time > 2):
                print(f"player {player.player_id} is idle.
                disconnected.")
                remove_player(player.player_id)
        time.sleep(2)


# Main server function
def main():
    global players


    # TCP
    SERVER_ADDRESS = ('0.0.0.0', TCP_PORT)
    tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_socket.bind(SERVER_ADDRESS)
    tcp_socket.listen(5)


    users_db = Users_db()
    lock = threading.Lock()

    threading.Thread(target=handle_game).start()

    threading.Thread(target=remove_idles).start()
    # Threading for concurrent client handling

    print("Server listening on", SERVER_ADDRESS)

    while True:
        client_socket, address = tcp_socket.accept()
        threading.Thread(target=handle_client, args=(client_socket,
        address, users_db, lock)).start()

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