TypeMelanyHome

Multiplayer Network
Game Programming Report

Team Members

Huang Wei Jhin
Kew Yu Jun
Glenn Lee
Lor Xaun Yun Michelle
Lee Hsien Wei, Joachim
Xie Zhi Xiong

Table of Contents

1. Game Idea

1.1 Game Theme

Our game, Type Melany Home, is a dual player space-themed typing and shooting hybrid game. Players can enter their name and choose from 3 colors, blue, red and yellow. Names and colors have to be unique. Players will have to navigate their ship and shoot asteroids with lasers by pressing the corresponding key with the asteroids containing a letter and their color. There will be a timer counting down, upon reaching zero. The player with the higher score will win.

1.2 Game How to play

The controls of the game are very simple, only requiring keyboard inputs A to Z and arrow keys. Keys A to Z are used to shoot the laser. Arrow keys are used to move the space ship. Players can also toggle the Interpolation by pressing (Ctrl + I), Reconciliation by pressing (Ctrl + O) and Prediction by pressing (Ctrl + P).

1.3 Game Mechanism

Connected players have a color that they choose and the asteroid letters will inherit the player's colors. Players will need to shoot down the asteroid containing the letter of their chosen color. In order to do so, the player needs to press the corresponding key (A-Z) to shoot a laser of the letter shown on the asteroid.

If the laser hits an asteroid of the same color and letter, the asteroid will be destroyed and the player will earn 5 points from that. Each player will do the same and at the end of the timer, the player with the higher score will win. Players are not allowed to overlap and be at the same position as each other.

1.4 Game Levels

The game has infinite levels, with each level increasing in difficulty. As the level progresses, it will become harder to score as the asteroids will get faster. Furthermore, the number of lanes that the asteroid will fall from increases as the level progresses. Level 1 still has 3 lanes, level 2 will have 4 lanes, and so on with each level increasing by 1 lane.

Figure 1.2 How to Play

Figure 1.4 Game View of Level 1

2. System design and implementation

2.1 Game Engine

The game uses a game engine from a Software Engineering Project 3 called VI engine from team MemoryLeak. There are three members from team Memory leak, namely Wei Jhin, Yu Jun and Joachim.

The game engine had a vast amount of functionalities such as a Level Editor, Animations, Physics and collision, Audio, MovementAI, text and graphics rendering. Hence more focus could be made on the networking side of the game as making the game was relatively easy.

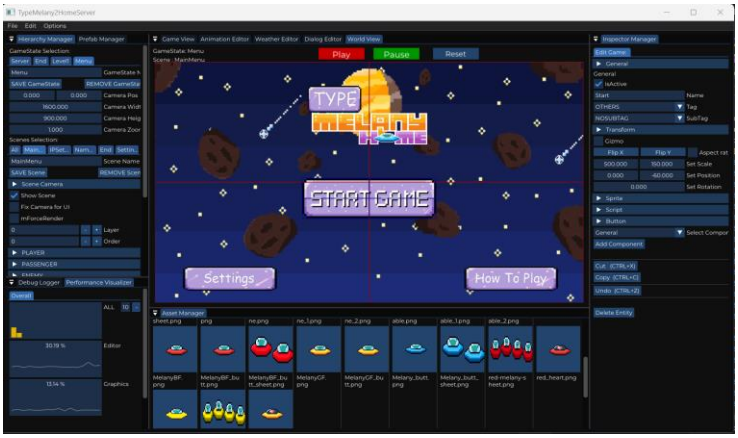


Figure 2.1 Editor view of VI engine

2.2 System Design

The game comprises 2 projects, MelanyServer and MelanyClient. The other 2 projects, TOOLS and VI, are for the level editor and game engine.

MelanyServer is the server application that the clients connect to. This runs the main game logic and handles the clients.

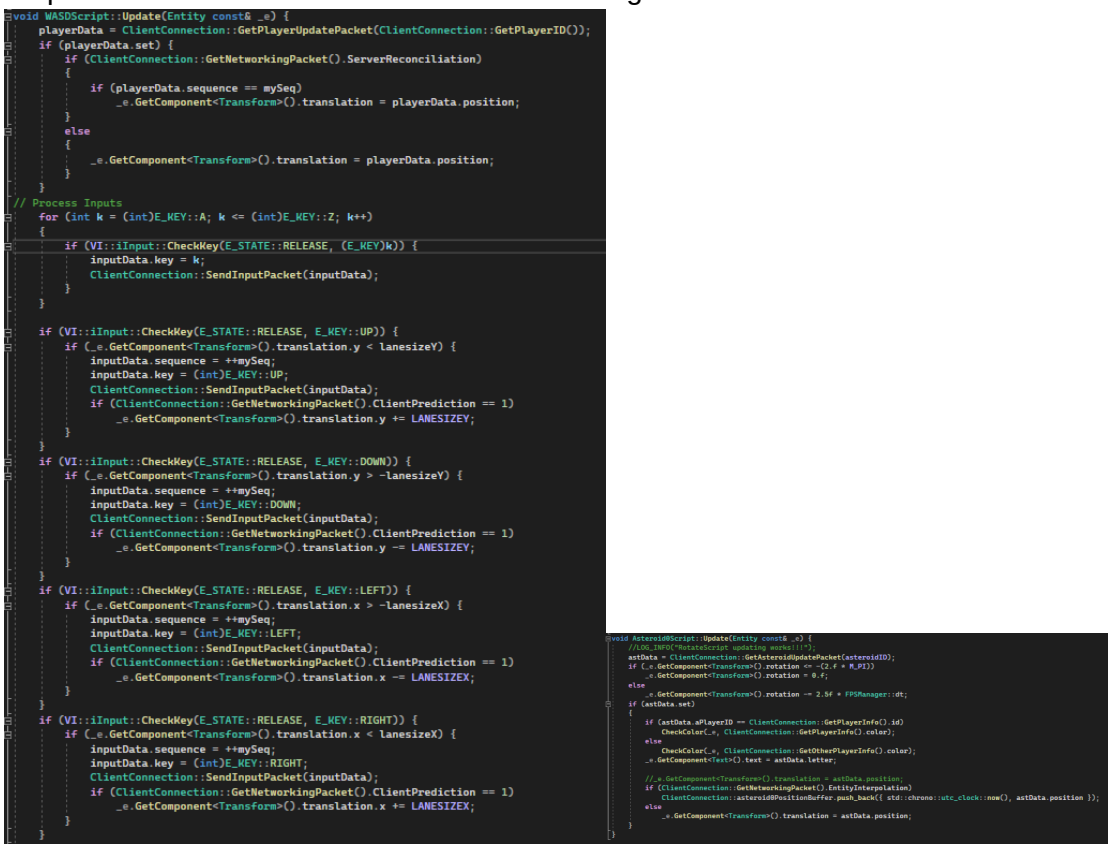
MelanyClient is the client application that the clients will use to connect to the server to play the game. The client application only handles rendering of the screen, gets inputs, animations and audio. Packets are sent from the client to the server and vice versa for communications and syncing of data between clients.

2.3 Game Implementation

The game largely relies on scripts attached to entities to run. Each NPC game object on the client side has a script to update its position, color and text. Additionally, the main player has a script to get inputs to send to the server. Client prediction and Server Reconciliation is also included in the script.

The other NPC game objects also have Entity interpolation included. Each object takes care of its own animation. On the server side, there are similar game objects. However they have additional colliders for the player and do not have animation.

The game logic is done on the server scripts. Data such as inputs are sent from the client and processed. Collision between players, asteroid shooting collisions are validated and the updated position is sent back to the client for rendering.



```
void WASDScript::Update(Entity const& _e) {
    playerData = ClientConnection::GetPlayerUpdatePacket(ClientConnection::GetPlayerID());
    if (playerData.set) {
        if (ClientConnection::GetNetworkingPacket().ServerReconciliation)
        {
            if (playerData.sequence == mySeq)
                _e.GetComponent<Transform>().translation = playerData.position;
        }
        else
        {
            _e.GetComponent<Transform>().translation = playerData.position;
        }
    }
// Process Inputs
    for (int k = (int)E_KEY::A; k <= (int)E_KEY::Z; k++)
    {
        if (VI::iInput::CheckKey(E_STATE::RELEASE, (E_KEY)k)) {
            inputData.key = k;
            ClientConnection::SendInputPacket(inputData);
        }
    }

    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::UP)) {
        if (_e.GetComponent<Transform>().translation.y < lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::UP;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y += LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::DOWN)) {
        if (_e.GetComponent<Transform>().translation.y > -lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::DOWN;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y -= LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::LEFT)) {
        if (_e.GetComponent<Transform>().translation.x > -lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::LEFT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x -= LANESIZEX;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::RIGHT)) {
        if (_e.GetComponent<Transform>().translation.x < lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::RIGHT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x += LANESIZEX;
        }
    }
}
```

```
void Asteroid0Script::Update(Entity const& _e) {
    //LOG_INFO("RotateScript updating asteal!!");
    astData = ClientConnection::GetAsteroid0UpdatePacket(asteroidID);
    if (_e.GetComponent<Transform>().rotation <= -(2.f * M_PI))
        _e.GetComponent<Transform>().rotation = 0.f;
    else
        _e.GetComponent<Transform>().rotation -= 2.5f * FPSManager::dt;
    if (astData.set)
    {
        if (astData.aPlayerID == ClientConnection::GetPlayerInfo().id)
            CheckColor(_e, ClientConnection::GetPlayerInfo().color);
        else
            CheckColor(_e, ClientConnection::GetOtherPlayerInfo().color);
        _e.GetComponent<Text>().text = astData.letter;

        //_e.GetComponent<Transform>().translation = astData.position;
        if (ClientConnection::GetNetworkingPacket().EntityInterpolation)
            ClientConnection::asteroid0PositionBuffer.push_back({ std::chrono::utc_clock::now(), astData.position });
        else
            _e.GetComponent<Transform>().translation = astData.position;
    }
}
```

Figure 2.3 Example of a script for Player Movement and Asteroid Receive

3. Network Game Architecture and Mechanisms

3.1 Overview

The game comprises 2 projects, MelanyServer and MelanyClient. There is a ServerConnection class in MelanyServer and a ClientConnection class in MelanyClient handling all the networking chicanery.

Server and Client connections are done within the respective projects under ClientConnection.h/.cpp & ServerConnection.h/.cpp.

The game is run on the server, with the client only handling inputs and displaying the objects on the screen. The server will store the information of the game world and update the client when the object moves. The score and timer is also stored on the server and updated to the client. The server will send information to the client about the spawning of an asteroid and its movement.

The client will send its inputs and/or updates to the server for validation. Such data includes player movement, player shooting, bullet travel etc. The server will then validate the received data with its own data before sending the response back to the client. If successful, the action is permitted (e.g. destruction of the asteroid, increasing score; movement). Otherwise, the server will send an appropriate response to the client.

The way the server and client communicate using different packets with each other can be seen from the flowchart in **Annex B**. The server can differentiate each client by using their unique socket number.

```
struct CLIENT_INFO
{
    struct sockaddr_in clientAddr {};
    SOCKET clientSocket{};
    bool setup = false;
    bool ready = false;
    int nLength =0;
    PLAYER_INFO player_info{};
    NetworkingPacket networkdata{};
    std::queue<InputPacket> playerInput{};
};

class ServerConnection
{
public:
    static int Init();
    static BOOL WINAPI HandleClients_Thread(LPVOID lpData);
    static int ServerListen_Thread();
    static int ServerExit();
    //game
    static int InGame();
    static int NumClients();
    static int ValidateClient(std::string name, ColorType color, int id);
    static int Getlevel() { return level; }
    static void SetLevel(int lvl) { level = lvl; }
    //data getters
    static CLIENT_INFO* GetClientInfo(int player);
    static PlayerInfoData* GetClientData(int player);
    static std::string GetClientName(int player);
    static ColorType GetClientColor(int player);
    static std::queue<InputPacket>* GetInput(int player);
    //data setters
    static void SetInputPacket(InputPacket);
    static void SetSendDataTimer(short timer);
    static void SetSendData(int player, PlayerInfoData playerData);
    static void SetSendData(int player, AsteroidData asteroidData);
    static void SetSendData(int player, LaserData laserData);
    static void SendUpdateData();
    static void SendEndGameData(int winnerID);

private:
    static bool InitWinSock2_0();
    static void BroadcastMessageToAll(Packet packet);
    static void SendMessageToClient(SOCKET user, Packet packet);
    static std::vector<CLIENT_INFO*> client_infos;
    static std::string szServerIPAddr;
    static int nServerPort;
    static SOCKET hServerSocket;
    static Packet data;
    static Packet sendData;
    static UpdatePacket updatepacket;
    static int level;
```

```
class ClientConnection
{
public:
    static int Init();
    static int SendInitPacket(InitPacket initPacket);
    static int SendInputPacket(InputPacket inputPacket);
    static int SendNetworkingPacket(NetworkingPacket networkingPacket);
    static BOOL WINAPI ClientInputThread();
    static int ClientExit();
    //init
    static InitResponsePacket GetInitResponsePacket();
    static StartPacket GetStartPacket();
    static void ResetStartPacket(int start);
    //update
    static UpdatePacket GetUpdatePacket(int obj, int id);
    static PlayerInfoData GetPlayerUpdatePacket(int id);
    static AsteroidData GetAsteroidUpdatePacket(int id);
    static LaserData GetLaserUpdatePacket(int id);
    static short GetUpdatePacketTimer();
    //Networking
    static NetworkingPacket GetNetworkingPacket();
    static void SetClientPrediction(bool toggle);
    static void SetServerReconciliation(bool toggle);
    static void SetEntityInterpolation(bool toggle);
    //endgame
    static EndGamePacket GetEndGamePacket();
    static void ResetAllPackets();
    static ExitPacket GetExitPacket();
    //player
    static int GetPlayerID();
    static PLAYER_INFO GetPlayerInfo();
    static PLAYER_INFO GetOtherPlayerInfo();
    static bool SetServerIPAddress(std::string serverIPAddr);
    static int GetLevel() { return level; }
private:
    static int ClientSend(Packet data);
    static bool InitWinSock2_0();
    static SOCKET hClientSocket;
    static std::string szServerIPAddr;//= "127.0.0.1";
    static int nServerPort;//= 5050;
    static char szBuffer[1024];//= "";
    static PLAYER_INFO player_info;
    static PLAYER_INFO otherPlayer_info;
    //Packet data;
    static InitPacket initPacket;
    static NetworkingPacket networkingPacket;
    static InitResponsePacket initResponsePacket;
    static StartPacket startPacket;
    static UpdatePacket updatePacket;
    static EndGamePacket endGamePacket;
    static ExitPacket exitPacket;
    static int level;
```

Figure 2.2 ServerConnection and ClientConnection Class Declarations

3.2 Networking Packets

When sending data across the network, the data will be packaged into a char pointer. The first 4 bytes will denote the type of data sent, for example during connection setup, Init and InitResponse types are used, Input and Update type during the game, etc. The remaining bytes will be used for the data that needs to be sent.

This way the recipient knows how to extract the data from the char pointer into the correct struct based on the type. All packets have a member set. Set is set to 1 when sending the data. After getting the data set is set to 0 indicating that the data has been processed. As the set is checked when getting the data, data which is not set is not processed as it is already processed. This saves computation time by not processing the same data if it is not updated. The data sent is also encrypted with Affine Cipher encryption and decryption on the receiver side. This allows secure transfer of data packets.

Packets:

1) InitPacket:

- set - denoting of the packet is updated but the data not yet processed
- state - current state of the connection setup
- color - set the color enum to the player
- name - set the name of the player

2) InitResponsePacket

- set - denoting of the packet is updated but the data not yet processed
- isName - check if name is already taken by other player
- isColor - check if color is already taken by other player
- id - client ID

3) StartPacket

- set - denoting of the packet is updated but the data not yet processed
- startGame - toggle for game start
- player_info - array of client's info
- numPlayer - number of clients

4) NetworkingPacket

- set - denoting of the packet is updated but the data not yet processed
- ClientPrediction - prediction toggle
- ServerReconciliation - reconciliation toggle
- EntityInterpolation - interpolation toggle

5) InputPacket

- set - denoting of the packet is updated but the data not yet processed
- sequence - sequence number in order of packets for server reconciliation
- key - input keypress

6) UpdatePacket

set	- denoting of the packet is updated but the data not yet processed
timeleft	- timer
asteroid	- array of asteroid's data
player	- array of client's data
laser	- array of laser's data

7) EndGame

set	- denoting of the packet is updated but the data not yet processed
ended	- end game toggle
winnerID	- winning client ID

8) ExitPacket

set	- denoting of the packet is updated but the data not yet processed

Datas:

1) PLAYER_INFO

id	- client's ID
name	- client's Name
color	- client's Color

2) PlayerInfoData

set	- denoting of the packet is updated but the data not yet processed
aPlayerID	- client's ID
sequence	- sequence number in order of packets for server reconciliation
position	- client's position
score	- client's Score

3) AsteroidData

set	- denoting of the packet is updated but the data not yet processed
aPlayerID	- client's ID
letter	- prompt input alphabet
position	- asteroid's position
score	- client's Score

4) LaserData

set	- denoting of the packet is updated but the data not yet processed
aPlayerID	- client's ID
alpha	- laser's alpha
position	- laser's position
score	- client's Score3

3.3 Server Connection

This class is implemented in the server application to handle all the network connections.

The first function Init() sets up the server's Transmission Control Protocol (TCP). We use TCP because it is a reliable data transfer protocol as it reduces the loss of packets. The function binds the server IP address and server port and creates a server thread ServerListen_Thread() to listen for clients to connect.

Next the ServerListen_Thread() actively listens and waits for clients to connect. When a client is connected, it stores the IPaddress, socket number and assigns an ID (0 or 1). This is stored in a client_infos vector. A new thread for that client, HandleClients_Thread(), is then created for that client.

Next, the HandleClients_Thread() sends a InitResponse packet containing the client's ID to the client. This is used to subscript broadcast data sent in the future. It then goes into a loop to actively listen for clients' messages. Clients start at the not setup state.

When it receives a message, it checks the type of the message.

1) InitPacket
   If Init's state is 0, it validates the client's name and color to ensure that it is not already taken. And then send a InitResponse packet back indicating if the name and color is valid. If so, the client will be in a setup state.
   It will also send a Start packet broadcast to all clients connected containing its player info and other players info. Clients can use its ID to subscript the player info array to get its name or other players name or color. Start packet's .start set to 0 as not all players are ready.
   If the Init's state is 1, the server will change the client state to ready and send a Start packet broadcast to all clients connected containing its all connected player info as well as Start packet's .start set to 1 if all connected player states are ready.

2) InputPacket
   When it receives an input packet from a client, it pushes the input packet to the client's input queue. This ensures that all inputs are processed in order and not overridden.

3) NetworkingPacket
   This updates the clients networking selection on the server side for each client. Selections include Client prediction, Server Reconciliation and Entity interpolation toggles.

4) ExitPacket
   Denote that the connection has ended.

The server connections has helper functions to be used in the scripts such as:

BroadcastMessageToAll()
: To send data packets to all clients..kn.

SendMessageToClient()
: To send data packets to an individual client.

InGame()
: Checks if all the players are in ready state.

NumClients()
: Get the number of clients connected.

ValidateClient()
: Check if the player name and color is already taken.

Getlevel()
: Get the current level.

Setlevel()
: Set the current level.

GetClientInfo()
: Get the details of the client such as input queue, network, ip address and socket.

GetClientData()
: Get the player data of the client such as name, color and ID.

GetClientName()
: Get the client name.

GetClientColor()
: Get the client color.

GetInput()
: Get the input queue of the client.

SetInputPacket()
: Set the input packet for first player testing purpose on the server only

SetSendDataTimer()
: Sets the timer of the updatePacket to be sent by SendUpdateData() every frame

SetSendData(PlayerData)
: Sets the PlayerData of the updatePacket to be sent by SendUpdateData() every frame

SetSendData(AsteroidData)
: Sets the AsteroidData of the update packet to be sent by SendUpdateData() every frame

SetSendData(LaserData)
: Sets the LaserData of the updatePacket to be sent by SendUpdateData() every frame

SendUpdateData()
: To send update data packets every game loop when in game.

SendEndGameData()
: To send endGame data packets containing the winnerID when the game ends

```
int ServerConnection::ServerListen_Thread()
{
    // Start the infinite loop
    while (true)
    {
        SOCKET hClientSocket;
        struct sockaddr_in clientAddr;
        int nSize = sizeof(clientAddr);
        hClientSocket = accept(hServerSocket, (struct sockaddr*)&clientAddr, &nSize);
        if (hClientSocket == INVALID_SOCKET)
        {
            std::cout << "accept( ) failed" << std::endl;
        }
        else
        {
            int ID = (int)client_infos.size();
            CLIENT_INFO* CI = new CLIENT_INFO;
            client_infos.emplace_back(CI);
            client_infos[ID]->clientAddr = clientAddr;
            client_infos[ID]->clientSocket = hClientSocket;
            client_infos[ID]->player_info.id = ID;
            HANDLE hClientThread;
            DWORD dwThreadId;

            std::cout << "Client connected from " << inet_ntoa(client_infos[ID]->clientAddr.sin_addr) << std::endl;
            std::cout << "Client socket :  "<< hClientSocket << std::endl;
            // Start the client thread
            hClientThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)(HandleClients_Thread), (LPVOID)(client_infos[ID]), 0, &dwThreadId);
            if (hClientThread == NULL)
            {
                std::cout << "Unable to create client thread" << std::endl;
            }
            else
            {
                CloseHandle(hClientThread);
            }
        }
    }
    return 0;
}
```

Figure 3.3.1 Server thread to listen for new client and store client info

```
BOOL WINAPI ServerConnection::HandleClients_Thread(LPVOID lpData)
{
    //send client id as initial hand shake packet
    CLIENT_INFO* pClientInfo = (CLIENT_INFO*)lpData;
    Packet setup;
    setup.type = TYPE::INITRESPONSE;
    setup.initResponsePacket.set = 1;
    setup.initResponsePacket.id = pClientInfo->player_info.id;
    setup.initResponsePacket.isName = 0;
    setup.initResponsePacket.isColor = 0;
    std::cout << "\n=====================================\n";
    std::cout << "Sending first setup data type: " << (int)setup.type << " to player" << " : " << pClientInfo->player_info.id << "\n";
    PrintingDebug::PrintPacket(setup);
    SendMessageToClient(pClientInfo->clientSocket, setup);

    while (1)
    {

        char szBuffer[MAX_BUFFER];
        int nLength;
        nLength = recv(pClientInfo->clientSocket, szBuffer, sizeof(szBuffer), 0);
        if (nLength > 0)
        {
            char* tempBuff = szBuffer;
            Packet recvData = tempBuff;

            std::string NclientIPAddr = std::string(inet_ntoa(pClientInfo->clientAddr.sin_addr));
            std::cout << "\n=====================================\n";
            std::cout << "Received data type: " << (int)recvData.type << " from player" << " : " << pClientInfo->player_info.id << "setup" << (int)pClientInfo->setup << std::endl;
            PrintingDebug::PrintPacket(recvData);
            if (recvData.type == TYPE::INIT)
            {
                //send to only client InitResponsePacket
                Packet reply;
                reply.type = TYPE::INITRESPONSE;
                reply.initResponsePacket.set = 1;
                reply.initResponsePacket.id = pClientInfo->player_info.id;
                if (!pClientInfo->setup || !pClientInfo->ready)//check if set up
                {
                    if (recvData.initPacket.state == 0)
                    {
                        int status = ValidateClient(recvData.initPacket.name, recvData.initPacket.color, pClientInfo->player_info.id);
                        if (status == 0) //ok
                        {
                            reply.initResponsePacket.isName = 1;
                            reply.initResponsePacket.isColor = 1;
                            strncpy_s(pClientInfo->player_info.name, recvData.initPacket.name, MAXNAME);
                            pClientInfo->player_info.color = recvData.initPacket.color;
                            pClientInfo->setup = true;
                        }
                        else if (status == 1)//wrong name
                        {
                            reply.initResponsePacket.isName = 0;
                            reply.initResponsePacket.isColor = 0;
                        }
                        else if (status == 2)//wrong col
                        {
                            reply.initResponsePacket.isName = 1;
                            reply.initResponsePacket.isColor = 0;
                        }
                        std::cout << "\n=====================================\n";
                        std::cout << "Sending first init response data type: " << (int)reply.type << " to player" << " : " << pClientInfo->player_info.id << "\n";
                        PrintingDebug::PrintPacket(reply);
                        SendMessageToClient(pClientInfo->clientSocket, reply);
                    }
                    else if (recvData.initPacket.state == 1)
                    {
                        if(pClientInfo->setup)
                            pClientInfo->ready = true;
                    }
                }
                if (recvData.initPacket.state == 2)
                {
                    pClientInfo->ready = false;
                }
                data.type = TYPE::START;
                data.startPacket.set = 1;
                data.startPacket.startGame = InGame()* level;
                data.startPacket.numPlayer = client_infos.size();
                for (int s = 0; s < client_infos.size(); s++)
                {
                    if (s > MAXPLAYER) break;
                    data.startPacket.player_info[s].id = client_infos[s]->player_info.id;
                    strncpy_s(data.startPacket.player_info[s].name, client_infos[s]->player_info.name, MAXNAME);
                    data.startPacket.player_info[s].color = client_infos[s]->player_info.color;
                }
                std::cout << "\n=====================================\n";
                std::cout << "Sending START broadcast data \n";
                PrintingDebug::PrintPacket(data);
                BroadcastMessageToAll(data);
            }
            if (recvData.type == TYPE::INPUT)
            {
                if(InGame())
                {
                    pClientInfo->playerInput.push(recvData.inputPacket);
                    std::cout << "\nPushing input  data into ID:  " << pClientInfo->player_info.id << "\n";
                }
            }
            if (recvData.type == TYPE::NETWORKING)
            {
                pClientInfo->networkdata = recvData.networkingPacket;
                std::cout << "\nSetting network data into ID:  " << pClientInfo->player_info.id << "\n";
            }
        }
        else
        {
            std::cout << "Player" << pClientInfo->player_info.id << ", " << inet_ntoa(pClientInfo->clientAddr.sin_addr) << " Left...\n";
            closesocket(pClientInfo->clientSocket);
            return TRUE;
        }
```

Figure 3.3.1 Server thread to handle Client receive messages

3.4 Client Connection

This class is implemented in the client application to handle all the network connections. When the client wants to connect to the server, it calls the Init() function to establish a connection. The init() function will create a new socket and a new client thread ClientInputThread() once connected to the server.

The ClientInputThread() handles messages received from the server. It checks the type of message and puts the packet in the respective data members e.g. InitResponse, Start and Update.

1) InitResponsePacket
   Updates its own ID and puts the data in a private member initResponsePacket which the scripts can access.

2) StartPacket
   Puts the data in a private member startPacket which the scripts can access. If startGame is not set, it updates player name and color as well as the other player's name, color and ID. If startGame is set, it updates the current level.

3) UpdatePacket
   Puts the data in a private member updatePacket which the scripts can access to update the game objects data.

4) EndGamePacket
   Puts the data in a private member endGamePacket which the scripts can access to know that the level has ended.

5) ExitPacket
   Puts the data in a private member exitPacket to determine if the connection has ended.

The client connections has helper functions to be used in the scripts such as:

SendInitPacket()

Creates a packet with type Init to place the Init packet. Calls ClientSend() to send the packet to the server.

SendInputPacket():

Creates a packet with type Input to place the Input packet. Calls ClientSend() to send the packet to the server.

SendNetworkingPacket():

Creates a packet with type networking to place the Networking packet. Calls ClientSend() to send the packet to the server.

GetInitResponsePacket():

Getter to the Init Response packet data.

GetStartPacket():

Getter to the Start packet data.

ResetStartPacket():

Set the startGame to true or false

GetUpdatePacket();

Getter to the update packet which contains player, asteroid and laser updates.

GetUpdatePacketTimer():

Getter to the update packet time left variable.

GetPlayerUpdatePacket();

Getter to the update packet for the Player.

GetAsteroidUpdatePacket();

Getter to the update packet for the Asteroid.

GetLaserUpdatePacket();

Getter to the update packet for the Laser.

GetNetworkingPacket()

Getter to the networking packet.

SetClientPrediction()

Toggle the Client Prediction Variable.

SetServerReconciliation()

Toggle the Server Reconciliation Variable.

SetEntityInterpolation()

Toggle the Entity Interpolation Variable.

GetEndGamePacket()

Getter to the endgame packet.

GetExitPacket()

Getter to the exit packet.

ResetAllPackets()

Resets all the private packet data in the client connection class.

GetPlayerID()

Gets the player's ID.

GetPlayerInfo()

Gets the players info such as name and color.

GetOtherPlayerInfo()

Gets the other players info such as name and color.

SetServerIPAddress()

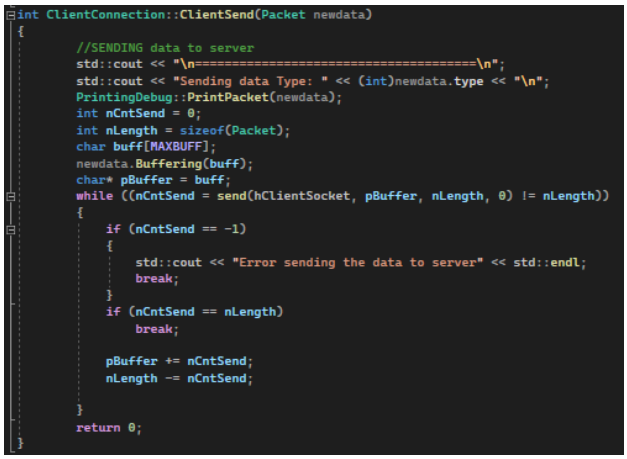
Sets the IP address to connect to.

GetLevel()

Get the current Level.

```
BOOL WINAPI ClientConnection::ClientInputThread()
{
    while (1)
    {
        int nLength;
        nLength = recv(hClientSocket, szBuffer, sizeof(szBuffer), 0);
        if (nLength > 0)
        {
            Packet data = szBuffer;
            if (data.type == TYPE::INITRESPONSE)
            {
                if (!startPacket.startGame)
                {
                    initResponsePacket = data.initResponsePacket;
                    if(player_info.id ==-1)
                        player_info.id = initResponsePacket.id;
                }
            }
            else if (data.type == TYPE::START)
            {
                startPacket = data.startPacket;
                if (!startPacket.startGame)
                {
                    startPacket.set = 1;
                    for (int p = 0; p < startPacket.numPlayer; p++)
                    {
                        if (p == player_info.id)
                        {
                            strncpy_s(player_info.name, startPacket.player_info[p].name, MAXNAME);
                            player_info.color = startPacket.player_info[player_info.id].color;
                        }
                        else //if (otherPlayer_info.id == -1)
                        {
                            otherPlayer_info.id = startPacket.player_info[p].id;
                            strncpy_s(otherPlayer_info.name, startPacket.player_info[p].name, MAXNAME);
                            otherPlayer_info.color = startPacket.player_info[p].color;
                        }
                    }
                }
                else
                    level = startPacket.startGame;
            }
            else if (data.type == TYPE::UPDATE)
            {
                updatePacket = data.updatePacket;
            }
            else if (data.type == TYPE::ENDGAME)
            {
                endGamePacket = data.endGamePacket;
            }
            else if (data.type == TYPE::EXIT)
            {
                exitPacket = data.exitPacket;
            }
        }
```

Figure 3.4.1 ClientInputThread to handle received message from the server



```
int ClientConnection::ClientSend(Packet newdata)
{
        //SENDING data to server
        std::cout << "\n=====================================\n";
        std::cout << "Sending data Type: " << (int)newdata.type << "\n";
        PrintingDebug::PrintPacket(newdata);
        int nCntSend = 0;
        int nLength = sizeof(Packet);
        char buff[MAXBUFF];
        newdata.Buffering(buff);
        char* pBuffer = buff;
        while ((nCntSend = send(hClientSocket, pBuffer, nLength, 0) != nLength))
        {
            if (nCntSend == -1)
            {
                std::cout << "Error sending the data to server" << std::endl;
                break;
            }
            if (nCntSend == nLength)
                break;

            pBuffer += nCntSend;
            nLength -= nCntSend;

        }
        return 0;
}
```

Figure 3.4.2 Client send function to send packet to server

3.5 Client Prediction

If turned on, the clients prematurely move the position of the player first and wait for a validation from the server regarding the current position of the player.

This is implemented in our movement script of the player when a movement input key is detected from the user. When not enabled, the input is just sent to the server. When enabled, the player's position will be updated as well.

```
// Process Inputs
    for (int k = (int)E_KEY::A; k <= (int)E_KEY::Z; k++) {
        if (VI::iInput::CheckKey(E_STATE::RELEASE, (E_KEY)k)) {
            inputData.key = k;
            ClientConnection::SendInputPacket(inputData);
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::UP)) {
        if (_e.GetComponent<Transform>().translation.y < lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::UP;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y += LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::DOWN)) {
        if (_e.GetComponent<Transform>().translation.y > -lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::DOWN;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y -= LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::LEFT)) {
        if (_e.GetComponent<Transform>().translation.x > -lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::LEFT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x -= LANESIZEX;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::RIGHT)) {
        if (_e.GetComponent<Transform>().translation.x < lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::RIGHT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x += LANESIZEX;
        }
    }
```

Figure 3.5 - Client-Side Prediction

3.6 Server Reconciliation

When the client enters a movement input key, it increments an internally tracked sequence number before sending the input data to the server. When responding back to the client, the server will copy the same sequence number to the packet before sending it back to the client.

In the following update loop, the client will process the received packet and check if the sequence number of the received packet is the same as the internally tracked sequence number. If so, it will update the position of the entity. This is implemented in the movement script of the player.

```
void WASDScript::Update(Entity const& _e) {
// Process server events
    playerData = ClientConnection::GetPlayerUpdatePacket(ClientConnection::GetPlayerID());
    if (playerData.set) {
        //std::cout << "SERVER RECON: " << ClientConnection::GetNetworkingPacket().ServerReconciliation << std::endl;
        if (ClientConnection::GetNetworkingPacket().ServerReconciliation)
        {
            //std::cout << "SEQUENCE: " << playerData.sequence << " " << mySeq << std::endl;
            if (playerData.sequence == mySeq)
                _e.GetComponent<Transform>().translation = playerData.position;
        }
        else
        {
            _e.GetComponent<Transform>().translation = playerData.position;
        }
    }

// Process Inputs
```

Figure 3.6 - Server Reconciliation

3.7 Entity Interpolation

This is implemented for the falling asteroids and other players to animate smoothly. Currently, this is done within the receive movement script of the other player and each asteroid receiving scripts.

When enabled, within the update function of the entity script, the update positions of the entity are added into a container that stores the timestamp it was processed and the position to update to. In the late update function of the entity script, the program will erase any old positions that have already been processed based on the computed time frame to render.

Once done, it will check if there is more than 1 position available for it to interpolate. If so, it will interpolate between the two positions.

```
void ReceiveWASDScript::Update(Entity const& _e) {
    //LOG_INFO("RotateScript updating works!!!");

    playerData = ClientConnection::GetPlayerUpdatePacket( ClientConnection::GetOtherPlayerInfo().id);
    if (playerData.set) {
        if (ClientConnection::GetNetworkingPacket().EntityInterpolation)
            ClientConnection::otherPlayerPositionBuffer.push_back({ std::chrono::utc_clock::now(), playerData.position });
        else
            _e.GetComponent<Transform>().translation = playerData.position;
    }
    if (VT::iInput::CheckKey(E_STATE::RELEASE, E_KEY::LEFT_CONTROL))
```

Figure 3.7.1 - Entity Interpolation Part 1

```
void ReceiveWASDScript::LateUpdate(Entity const& _e) {
    (void)_e;

    // Interpolate other entities
    if (ClientConnection::GetNetworkingPacket().EntityInterpolation) {
        std::chrono::time_point<std::chrono::utc_clock, std::chrono::duration<double, std::chrono::utc_clock::period>> now{ std::chrono::utc_clock::now() };
        std::chrono::time_point<std::chrono::utc_clock, std::chrono::duration<double, std::chrono::utc_clock::period>> timeStamp{ now - std::chrono::duration<double, std::ratio<1,1>>(1.0/60.0) };

        while (ClientConnection::otherPlayerPositionBuffer.size() >= 2 && ClientConnection::otherPlayerPositionBuffer[1].first <= timeStamp)
            ClientConnection::otherPlayerPositionBuffer.erase(ClientConnection::otherPlayerPositionBuffer.begin());

        // Interpolate between the two positions
        if (ClientConnection::otherPlayerPositionBuffer.size() >= 2 && ClientConnection::otherPlayerPositionBuffer[0].first <= timeStamp && timeStamp <= ClientConnection::otherPlayerPositionBuffer[1].first) {
            Math::Vec2 x0{ ClientConnection::otherPlayerPositionBuffer[0].second },
                       x1{ ClientConnection::otherPlayerPositionBuffer[1].second };
            auto t0{ ClientConnection::otherPlayerPositionBuffer[0].first},
                 t1{ ClientConnection::otherPlayerPositionBuffer[1].first };

            _e.GetComponent<Transform>().translation = x0 + (x1 - x0) * ((timeStamp - t0) / (t1 - t0));
        }
    }
}
```

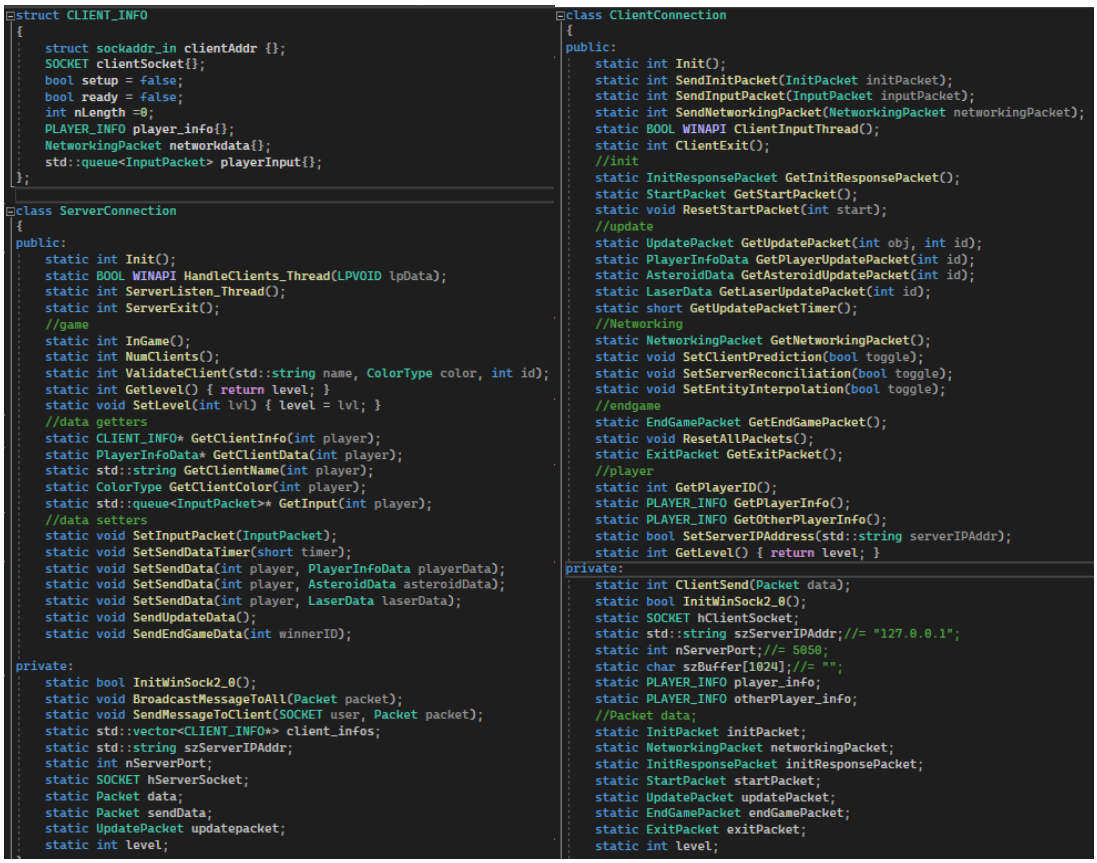
Figure 1.7.2 - Entity Interpolation Part 2

Annex A - Code Snippets

```
void WASDScript::Update(Entity const& _e) {
    playerData = ClientConnection::GetPlayerUpdatePacket(ClientConnection::GetPlayerID());
    if (playerData.set) {
        if (ClientConnection::GetNetworkingPacket().ServerReconciliation)
        {
            if (playerData.sequence == mySeq)
                _e.GetComponent<Transform>().translation = playerData.position;
        }
        else
        {
            _e.GetComponent<Transform>().translation = playerData.position;
        }
    }
// Process Inputs
    for (int k = (int)E_KEY::A; k <= (int)E_KEY::Z; k++)
    {
        if (VI::iInput::CheckKey(E_STATE::RELEASE, (E_KEY)k)) {
            inputData.key = k;
            ClientConnection::SendInputPacket(inputData);
        }
    }

    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::UP)) {
        if (_e.GetComponent<Transform>().translation.y < lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::UP;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y += LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::DOWN)) {
        if (_e.GetComponent<Transform>().translation.y > -lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::DOWN;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y -= LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::LEFT)) {
        if (_e.GetComponent<Transform>().translation.x > -lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::LEFT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x -= LANESIZEX;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::RIGHT)) {
        if (_e.GetComponent<Transform>().translation.x < lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::RIGHT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x += LANESIZEX;
        }
    }
}
```

```
void Asteroid0Script::Update(Entity const& _e) {
    //LOG_INFO("RotateScript updating works!!!");
    astData = ClientConnection::GetAsteroidUpdatePacket(asteroidID);
    if (_e.GetComponent<Transform>().rotation <= -(2.f * M_PI))
        _e.GetComponent<Transform>().rotation = 0.f;
    else
        _e.GetComponent<Transform>().rotation -= 2.5f * FPSManager::dt;
    if (astData.set)
    {
        if (astData.aPlayerID == ClientConnection::GetPlayerInfo().id)
            CheckColor(_e, ClientConnection::GetPlayerInfo().color);
        else
            CheckColor(_e, ClientConnection::GetOtherPlayerInfo().color);
        _e.GetComponent<Text>().text = astData.letter;

        //_e.GetComponent<Transform>().translation = astData.position;
        if (ClientConnection::GetNetworkingPacket().EntityInterpolation)
            ClientConnection::asteroid0PositionBuffer.push_back({ std::chrono::utc_clock::now(), astData.position });
        else
            _e.GetComponent<Transform>().translation = astData.position;
    }
}
```

Figure 2.3 Example of a script for Player Movement and Asteroid Receive



```
struct CLIENT_INFO
{
    struct sockaddr_in clientAddr {};
    SOCKET clientSocket[];
    bool setup = false;
    bool ready = false;
    int nLength =0;
    PLAYER_INFO player_info{};
    NetworkingPacket networkdata{};
    std::queue<InputPacket> playerInput{};
};

class ServerConnection
{
public:
    static int Init();
    static BOOL WINAPI HandleClients_Thread(LPVOID lpData);
    static int ServerListen_Thread();
    static int ServerExit();
    //game
    static int InGame();
    static int NumClients();
    static int ValidateClient(std::string name, ColorType color, int id);
    static int Getlevel() { return level; }
    static void SetLevel(int lvl) { level = lvl; }
    //data getters
    static CLIENT_INFO* GetClientInfo(int player);
    static PlayerInfoData* GetClientData(int player);
    static std::string GetClientName(int player);
    static ColorType GetClientColor(int player);
    static std::queue<InputPacket>* GetInput(int player);
    //data setters
    static void SetInputPacket(InputPacket);
    static void SetSendDataTimer(short timer);
    static void SetSendData(int player, PlayerInfoData playerData);
    static void SetSendData(int player, AsteroidData asteroidData);
    static void SetSendData(int player, LaserData laserData);
    static void SendUpdateData();
    static void SendEndGameData(int winnerID);

private:
    static bool InitWinSock2_0();
    static void BroadcastMessageToAll(Packet packet);
    static void SendMessageToClient(SOCKET user, Packet packet);
    static std::vector<CLIENT_INFO*> client_infos;
    static std::string szServerIPAddr;
    static int nServerPort;
    static SOCKET hServerSocket;
    static Packet data;
    static Packet sendData;
    static UpdatePacket updatepacket;
    static int level;
```

```
class ClientConnection
{
public:
    static int Init();
    static int SendInitPacket(InitPacket initPacket);
    static int SendInputPacket(InputPacket inputPacket);
    static int SendNetworkingPacket(NetworkingPacket networkingPacket);
    static BOOL WINAPI ClientInputThread();
    static int ClientExit();
    //init
    static InitResponsePacket GetInitResponsePacket();
    static StartPacket GetStartPacket();
    static void ResetStartPacket(int start);
    //update
    static UpdatePacket GetUpdatePacket(int obj, int id);
    static PlayerInfoData GetPlayerUpdatePacket(int id);
    static AsteroidData GetAsteroidUpdatePacket(int id);
    static LaserData GetLaserUpdatePacket(int id);
    static short GetUpdatePacketTimer();
    //Networking
    static NetworkingPacket GetNetworkingPacket();
    static void SetClientPrediction(bool toggle);
    static void SetServerReconciliation(bool toggle);
    static void SetEntityInterpolation(bool toggle);
    //endgame
    static EndGamePacket GetEndGamePacket();
    static void ResetAllPackets();
    static ExitPacket GetExitPacket();
    //player
    static int GetPlayerID();
    static PLAYER_INFO GetPlayerInfo();
    static PLAYER_INFO GetOtherPlayerInfo();
    static bool SetServerIPAddress(std::string serverIPAddr);
    static int GetLevel() { return level; }
private:
    static int ClientSend(Packet data);
    static bool InitWinSock2_0();
    static SOCKET hClientSocket;
    static std::string szServerIPAddr;//= "127.0.0.1";
    static int nServerPort;//= 5050;
    static char szBuffer[1024];//= "";
    static PLAYER_INFO player_info;
    static PLAYER_INFO otherPlayer_info;
    //Packet data;
    static InitPacket initPacket;
    static NetworkingPacket networkingPacket;
    static InitResponsePacket initResponsePacket;
    static StartPacket startPacket;
    static UpdatePacket updatePacket;
    static EndGamePacket endGamePacket;
    static ExitPacket exitPacket;
    static int level;
```

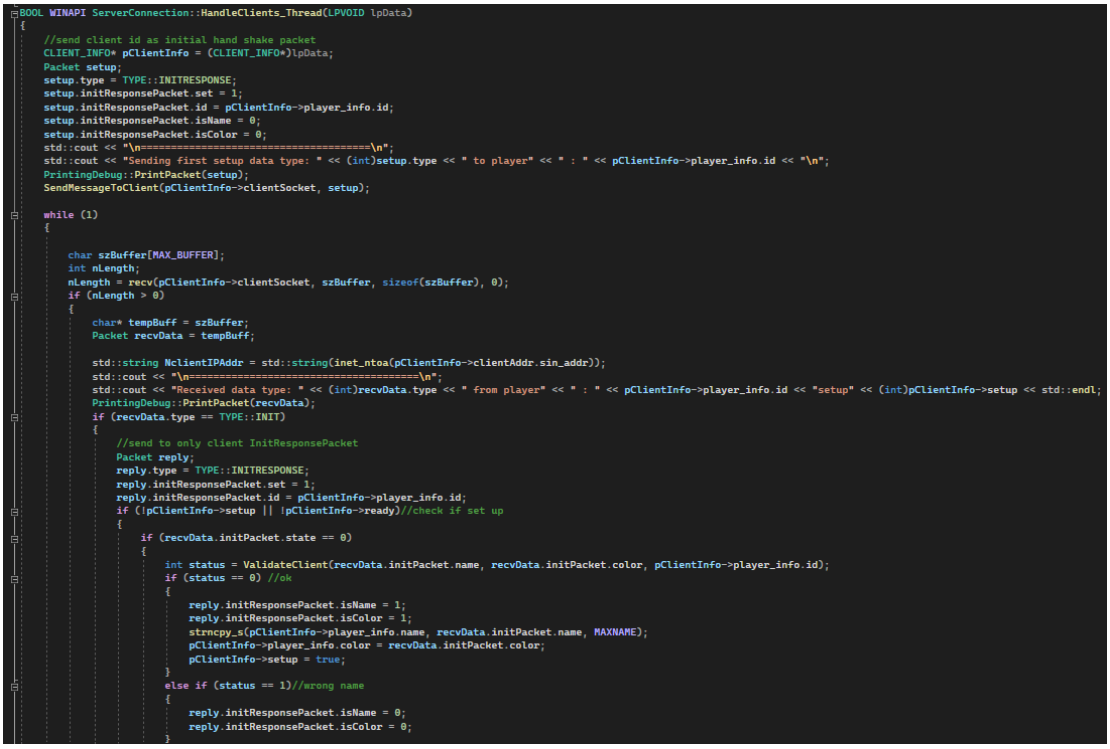
Figure 2.2 ServerConnection and ClientConnection Class Declarations

```cpp
#define KEY1 5
#define KEY2 9
#define MAXBUFFER 1024
class Cipher
{
public:
#define cipherMax 256
    static void encrypt(char* buffer)
    {
        for (int i = 0; i < MAXBUFFER; i++)
        {
            buffer[i] = ee(buffer[i], KEY1, KEY2);
        }
    }
    static void decrypt(char* buffer)
    {
        for (int i = 0; i < MAXBUFFER; i++)
        {
            buffer[i] = dd(buffer[i], KEY1, KEY2);
        }
    }



    static char ee(char c, int key1, int key2)
    {
        if (key1 % 2 == 0) //key need to be odd
            key1++;
        key1 %= cipherMax;
        key2 %= cipherMax;
        int r = ((int)c * key1) + key2;
        r = r % cipherMax;
        return (char)r;
    }

    static char dd(char c, int key1, int key2)
    {
        if (key1 % 2 == 0)
            key1++;
        key1 %= cipherMax;
        key2 %= cipherMax;
        int key1_inv = 0;
        for (int i = 0; i < cipherMax; i++)
        {//finding key1 inverse
            if (((key1 * i) % cipherMax) == 1)
            {
                key1_inv = i;
                break;
            }
        }
        int r = ((int)c + cipherMax - key2) * key1_inv;

        r = r % cipherMax;
        return (char)r;
    }
};
```

Figure 3.2 Affine cipher encryption and decryption

```cpp
int ServerConnection::ServerListen_Thread()
{
    // Start the infinite loop
    while (true)
    {
        SOCKET hClientSocket;
        struct sockaddr_in clientAddr;
        int nSize = sizeof(clientAddr);
        hClientSocket = accept(hServerSocket, (struct sockaddr*)&clientAddr, &nSize);
        if (hClientSocket == INVALID_SOCKET)
        {
            std::cout << "accept( ) failed" << std::endl;
        }
        else
        {
            int ID = (int)client_infos.size();
            CLIENT_INFO* CI = new CLIENT_INFO;
            client_infos.emplace_back(CI);
            client_infos[ID]->clientAddr = clientAddr;
            client_infos[ID]->clientSocket = hClientSocket;
            client_infos[ID]->player_info.id = ID;
            HANDLE hClientThread;
            DWORD dwThreadId;

            std::cout << "Client connected from " << inet_ntoa(client_infos[ID]->clientAddr.sin_addr) << std::endl;
            std::cout << "Client socket :  "<< hClientSocket << std::endl;
            // Start the client thread
            hClientThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)(HandleClients_Thread), (LPVOID)(client_infos[ID]), 0, &dwThreadId);
            if (hClientThread == NULL)
            {
                std::cout << "Unable to create client thread" << std::endl;
            }
            else
            {
                CloseHandle(hClientThread);
            }
        }
    }
    return 0;
}
```

Figure 3.3.1 Server thread to listen for new client and store client info

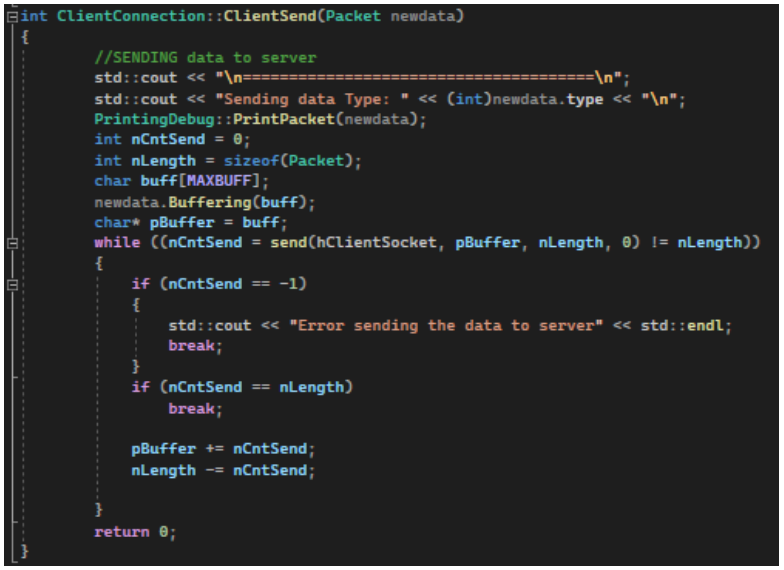


```
BOOL WINAPI ServerConnection::HandleClients_Thread(LPVOID lpData)
{
    //send client id as initial hand shake packet
    CLIENT_INFO* pClientInfo = (CLIENT_INFO*)lpData;
    Packet setup;
    setup.type = TYPE::INITRESPONSE;
    setup.initResponsePacket.set = 1;
    setup.initResponsePacket.id = pClientInfo->player_info.id;
    setup.initResponsePacket.isName = 0;
    setup.initResponsePacket.isColor = 0;
    std::cout << "\n=====================================\n";
    std::cout << "Sending first setup data type: " << (int)setup.type << " to player" << " : " << pClientInfo->player_info.id << "\n";
    PrintingDebug::PrintPacket(setup);
    SendMessageToClient(pClientInfo->clientSocket, setup);

    while (1)
    {

        char szBuffer[MAX_BUFFER];
        int nLength;
        nLength = recv(pClientInfo->clientSocket, szBuffer, sizeof(szBuffer), 0);
        if (nLength > 0)
        {
            char* tempBuff = szBuffer;
            Packet recvData = tempBuff;

            std::string NclientIPAddr = std::string(inet_ntoa(pClientInfo->clientAddr.sin_addr));
            std::cout << "\n=====================================\n";
            std::cout << "Received data type: " << (int)recvData.type << " from player" << " : " << pClientInfo->player_info.id << "setup" << (int)pClientInfo->setup << std::endl;
            PrintingDebug::PrintPacket(recvData);
            if (recvData.type == TYPE::INIT)
            {
                //send to only client InitResponsePacket
                Packet reply;
                reply.type = TYPE::INITRESPONSE;
                reply.initResponsePacket.set = 1;
                reply.initResponsePacket.id = pClientInfo->player_info.id;
                if (!pClientInfo->setup || !pClientInfo->ready)//check if set up
                {
                    if (recvData.initPacket.state == 0)
                    {
                        int status = ValidateClient(recvData.initPacket.name, recvData.initPacket.color, pClientInfo->player_info.id);
                        if (status == 0) //ok
                        {
                            reply.initResponsePacket.isName = 1;
                            reply.initResponsePacket.isColor = 1;
                            strncpy_s(pClientInfo->player_info.name, recvData.initPacket.name, MAXNAME);
                            pClientInfo->player_info.color = recvData.initPacket.color;
                            pClientInfo->setup = true;
                        }
                        else if (status == 1)//wrong name
                        {
                            reply.initResponsePacket.isName = 0;
                            reply.initResponsePacket.isColor = 0;
                        }
                        else if (status == 2)//wrong col
                        {
                            reply.initResponsePacket.isName = 1;
                            reply.initResponsePacket.isColor = 0;
                        }
                        std::cout << "\n=====================================\n";
                        std::cout << "Sending first init response data type: " << (int)reply.type << " to player" << " : " << pClientInfo->player_info.id << "\n";
                        PrintingDebug::PrintPacket(reply);
                        SendMessageToClient(pClientInfo->clientSocket, reply);
                    }
                    else if (recvData.initPacket.state == 1)
                    {
                        if(pClientInfo->setup)
                            pClientInfo->ready = true;
                    }
                }
                if (recvData.initPacket.state == 2)
                {
                    pClientInfo->ready = false;
                }
                data.type = TYPE::START;
                data.startPacket.set = 1;
                data.startPacket.startGame = InGame()* level;
                data.startPacket.numPlayer = client_infos.size();
                for (int s = 0; s < client_infos.size(); s++)
                {
                    if (s > MAXPLAYER) break;
                    data.startPacket.player_info[s].id = client_infos[s]->player_info.id;
                    strncpy_s(data.startPacket.player_info[s].name, client_infos[s]->player_info.name, MAXNAME);
                    data.startPacket.player_info[s].color = client_infos[s]->player_info.color;
                }
                std::cout << "\n=====================================\n";
                std::cout << "Sending START broadcast data \n";
                PrintingDebug::PrintPacket(data);
                BroadcastMessageToAll(data);
            }
            if (recvData.type == TYPE::INPUT)
            {
                if(InGame())
                {
                    pClientInfo->playerInput.push(recvData.inputPacket);
                    std::cout << "\nPushing input  data into ID:  " << pClientInfo->player_info.id << "\n";
                }
            }
            if (recvData.type == TYPE::NETWORKING)
            {
                pClientInfo->networkdata = recvData.networkingPacket;
                std::cout << "\nSetting network data into ID:  " << pClientInfo->player_info.id << "\n";
            }
        }
        else
        {
            std::cout << "Player" << pClientInfo->player_info.id << ", " << inet_ntoa(pClientInfo->clientAddr.sin_addr) << " Left...\n";
            closesocket(pClientInfo->clientSocket);
            return TRUE;
        }
```

Figure 3.3.1 Server thread to handle Client receive messages

```
BOOL WINAPI ClientConnection::ClientInputThread()
{
    while (1)
    {
        int nLength;
        nLength = recv(hClientSocket, szBuffer, sizeof(szBuffer), 0);
        if (nLength > 0)
        {
            Packet data = szBuffer;
            if (data.type == TYPE::INITRESPONSE)
            {
                if (!startPacket.startGame)
                {
                    initResponsePacket = data.initResponsePacket;
                    if(player_info.id ==-1)
                        player_info.id = initResponsePacket.id;
                }
            }
            else if (data.type == TYPE::START)
            {
                startPacket = data.startPacket;
                if (!startPacket.startGame)
                {
                    startPacket.set = 1;
                    for (int p = 0; p < startPacket.numPlayer; p++)
                    {
                        if (p == player_info.id)
                        {
                            strncpy_s(player_info.name, startPacket.player_info[p].name, MAXNAME);
                            player_info.color = startPacket.player_info[player_info.id].color;
                        }
                        else //if (otherPlayer_info.id == -1)
                        {
                            otherPlayer_info.id = startPacket.player_info[p].id;
                            strncpy_s(otherPlayer_info.name, startPacket.player_info[p].name, MAXNAME);
                            otherPlayer_info.color = startPacket.player_info[p].color;
                        }
                    }
                }
                else
                    level = startPacket.startGame;
            }
            else if (data.type == TYPE::UPDATE)
            {
                updatePacket = data.updatePacket;
            }
            else if (data.type == TYPE::ENDGAME)
            {
                endGamePacket = data.endGamePacket;
            }
            else if (data.type == TYPE::EXIT)
            {
                exitPacket = data.exitPacket;
            }
        }
```

Figure 3.4.1 ClientInputThread to handle received message from the server



```
int ClientConnection::ClientSend(Packet newdata)
{
        //SENDING data to server
        std::cout << "\n===================================\n";
        std::cout << "Sending data Type: " << (int)newdata.type << "\n";
        PrintingDebug::PrintPacket(newdata);
        int nCntSend = 0;
        int nLength = sizeof(Packet);
        char buff[MAXBUFF];
        newdata.Buffering(buff);
        char* pBuffer = buff;
        while ((nCntSend = send(hClientSocket, pBuffer, nLength, 0) != nLength))
        {
            if (nCntSend == -1)
            {
                std::cout << "Error sending the data to server" << std::endl;
                break;
            }
            if (nCntSend == nLength)
                break;

            pBuffer += nCntSend;
            nLength -= nCntSend;

        }
        return 0;
}
```

Figure 3.4.2 Client send function to send packet to server

```
// Process Inputs
    for (int k = (int)E_KEY::A; k <= (int)E_KEY::Z; k++) {
        if (VI::iInput::CheckKey(E_STATE::RELEASE, (E_KEY)k)) {
            inputData.key = k;
            ClientConnection::SendInputPacket(inputData);
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::UP)) {
        if (_e.GetComponent<Transform>().translation.y < lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::UP;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y += LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::DOWN)) {
        if (_e.GetComponent<Transform>().translation.y > -lanesizeY) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::DOWN;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.y -= LANESIZEY;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::LEFT)) {
        if (_e.GetComponent<Transform>().translation.x > -lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::LEFT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x -= LANESIZEX;
        }
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::RIGHT)) {
        if (_e.GetComponent<Transform>().translation.x < lanesizeX) {
            inputData.sequence = ++mySeq;
            inputData.key = (int)E_KEY::RIGHT;
            ClientConnection::SendInputPacket(inputData);
            if (ClientConnection::GetNetworkingPacket().ClientPrediction == 1)
                _e.GetComponent<Transform>().translation.x += LANESIZEX;
        }
    }
```

Figure 3.5 - Client-Side Prediction

```
void WASDScript::Update(Entity const& _e) {
// Process server events
    playerData = ClientConnection::GetPlayerUpdatePacket(ClientConnection::GetPlayerID());
    if (playerData.set) {
        //std::cout << "SERVER RECON: " << ClientConnection::GetNetworkingPacket().ServerReconciliation << std::endl;
        if (ClientConnection::GetNetworkingPacket().ServerReconciliation)
        {
            //std::cout << "SEQUENCE: " << playerData.sequence << " " << mySeq << std::endl;
            if (playerData.sequence == mySeq)
                _e.GetComponent<Transform>().translation = playerData.position;
        }
        else
        {
            _e.GetComponent<Transform>().translation = playerData.position;
        }
    }

// Process Inputs
```

Figure 3.6 - Server Reconciliation

```
void ReceiveWASDScript::Update(Entity const& _e) {
    //LOG_INFO("RotateScript updating works!!!");

    playerData = ClientConnection::GetPlayerUpdatePacket( ClientConnection::GetOtherPlayerInfo().id);
    if (playerData.set) {
        if (ClientConnection::GetNetworkingPacket().EntityInterpolation)
            ClientConnection::otherPlayerPositionBuffer.push_back({ std::chrono::utc_clock::now(), playerData.position });
        else
            _e.GetComponent<Transform>().translation = playerData.position;
    }
    if (VI::iInput::CheckKey(E_STATE::RELEASE, E_KEY::LEFT_CONTROL))
```

Figure 3.7.1 - Entity Interpolation Part 1

```
void ReceiveWASDScript::LateUpdate(Entity const& _e) {
    (void)_e;

    // Interpolate other entities
    if (ClientConnection::GetNetworkingPacket().EntityInterpolation) {
        std::chrono::time_point<std::chrono::utc_clock, std::chrono::duration<double, std::chrono::utc_clock::period>> now{ std::chrono::utc_clock::now() };
        std::chrono::time_point<std::chrono::utc_clock, std::chrono::duration<double, std::chrono::utc_clock::period>> timeStamp{ now - std::chrono::duration<double, std::ratio<1,1>>(1.0/60.0) };

        while (ClientConnection::otherPlayerPositionBuffer.size() >= 2 && ClientConnection::otherPlayerPositionBuffer[1].first <= timeStamp)
            ClientConnection::otherPlayerPositionBuffer.erase(ClientConnection::otherPlayerPositionBuffer.begin());

        // Interpolate between the two positions
        if (ClientConnection::otherPlayerPositionBuffer.size() >= 2 && ClientConnection::otherPlayerPositionBuffer[0].first <= timeStamp && timeStamp <= ClientConnection::otherPlayerPositionBuffer[1].first) {
            Math::Vec2 x0{ ClientConnection::otherPlayerPositionBuffer[0].second },
                       x1{ ClientConnection::otherPlayerPositionBuffer[1].second };
            auto t0{ ClientConnection::otherPlayerPositionBuffer[0].first},
                 t1{ ClientConnection::otherPlayerPositionBuffer[1].first };

            _e.GetComponent<Transform>().translation = x0 + (x1 - x0) * ((timeStamp - t0) / (t1 - t0));
        }
    }
}
```

Figure 1.7.2 - Entity Interpolation Part 2

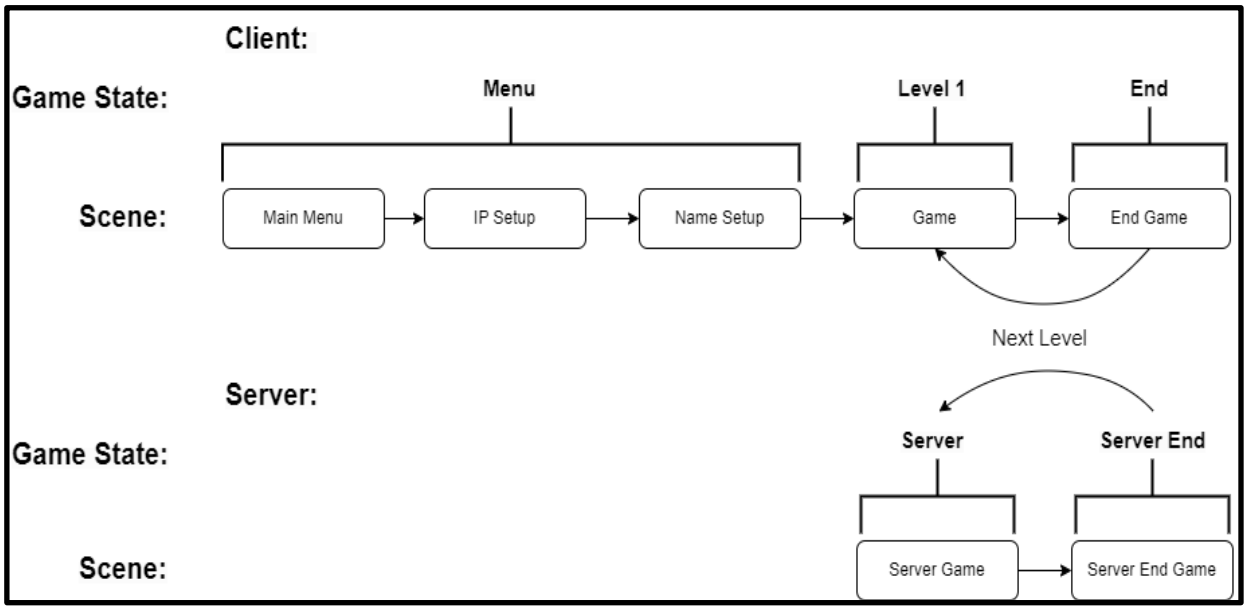


Figure 1.1 Game flow

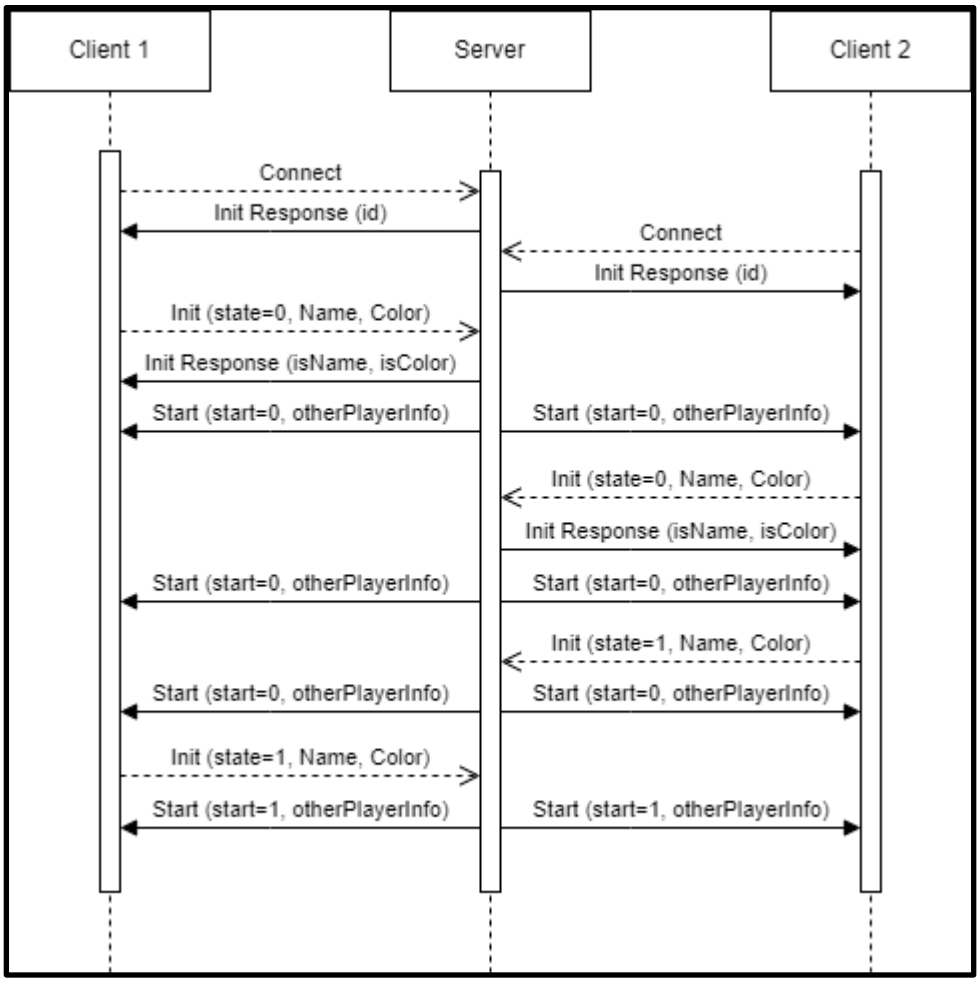


Figure 1.2 Packets sent during Client-Server connection setup and client validation

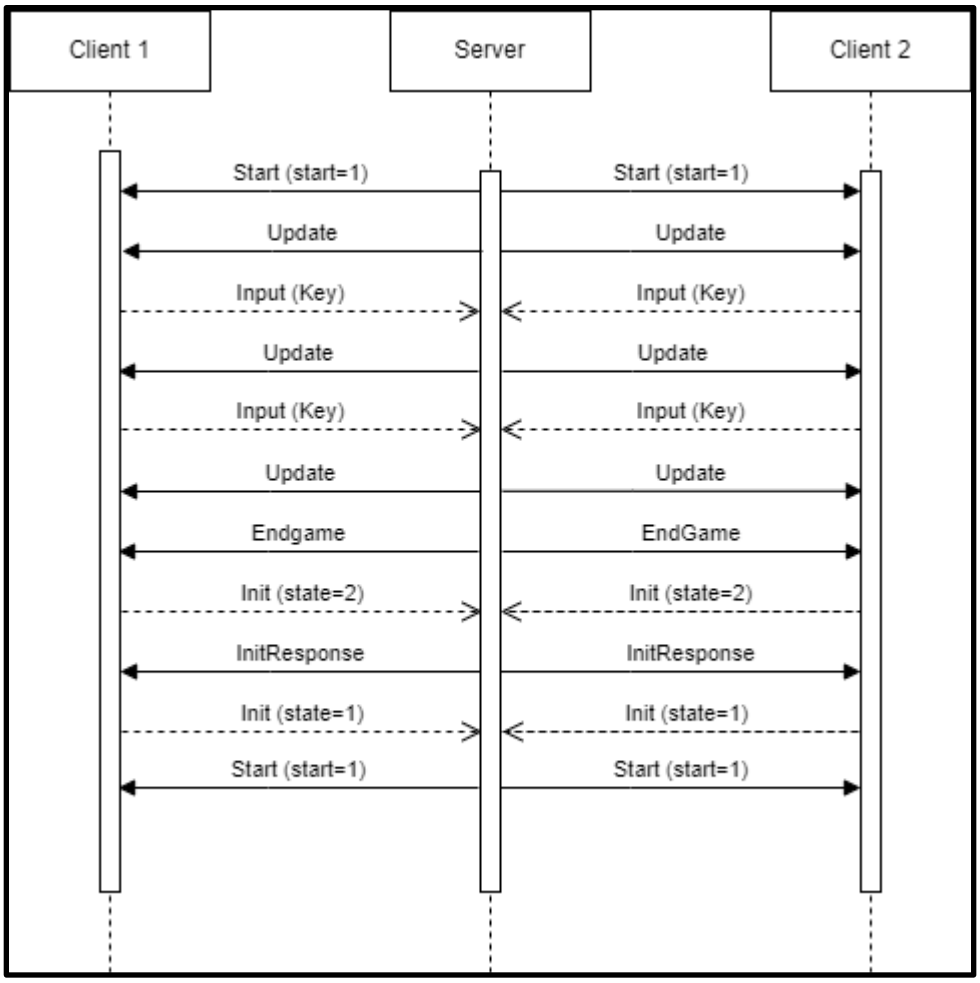


Figure 1.3 Packets sent during game and restarting of level

Annex C - Work Distribution

Huang Wei Jhin

- Team organization
- Task allocation
- Brainstorming & implementation of the packet data sent
- Server Connection class
- Client Connection class
- Server reconciliation
- Base engine setup
- Game idea brainstorm
- Report flowchart
- Debugging

Kew Yu Jun

- Game scene setup
- Brainstorming & implementation of the packet data sent
- Gameplay programming on server
- Entity sending/receiving information on server
- Game idea brainstorm
- Asteroid collision and spawning
- Laser collision and spawning
- Player collision
- Debugging

Glenn Lee

- Brainstorming & implementation of the packet data sent
- Server Connection class
- Client Connection class
- Game idea brainstorm
- Report writing
- Report flowchart
- Debugging

Lor Xaun Yun Michelle

- Game scene setup
- Gameplay programming on client
- Client side prediction
- Game idea brainstorm
- Menus
- Validate Client names and color
- Entity sending/receiving information
- Video script writing
- Debugging

Lee Hsien Wei, Joachim

- Brainstorming & implementation of the packet data sent
- Entity interpolation
- Base engine setup
- Engine debugging
- Game idea brainstorm
- Report writing & formatting
- Video script writing
- A Lot of Debugging

Xie Zhi Xiong

- Game scene setup
- Gameplay programming on client
- Game idea brainstorm
- Game settings
- Entity sending/receiving information
- Debugging