Software Development Project

FHWS

Computer Science

2021/2022

Henrique Araújo – k54804

# Abstract

This project is developed within the course of “Software Development Project” offered by FHWS.

It consists on a multiplayer game developed in the Unity game engine. To that end, it was developed two sub-projects separated in two different Unity solutions. Namely a sub-project for the server side of the video game, and another sub-project for the client-side.

The way the game is going to operate over the network goes as follows:

1. On the client project, there is a representation of the world that is simulated also on the server. In this sense, both projects are rendering the same world.
2. The client sends player inputs to the server, including movement inputs, shooting, jumping and other mechanics.
3. The server then takes those inputs, calculates a new game state and then sends that game state to the client so that they are both in sync.

# Networking

In this section, we’ll talk about how the connection between server and client is implemented. In doing so we will explain some of the .NET socket’s code as well as in-house networking code. Throughout this section you will encounter notice that the project is divided into two projects – server and client.

Diagram

Description automatically generatedNote that, in this section, we will not be discussing any game logic. Only how these two counterparts are communicating with each other.

Figure 2 - Server UML diagram

A picture containing diagram

Description automatically generated

Figure - Client diagram

## Diagram Description automatically generatedNetwork Flow Overview

Figure - Flow diagram of network overview

At a first instance, the server is setup to start listening to TCP and UDP requests coming from a client. To accomplish this, we use the “System.Net” namespace to incorporate our TcpListeners’s and UdpListeners’s “BeginAcceptTcpClient” and “BeginReceive” methods accordingly. In these methods we handle the connection logic between server and client.

Next, on the server side, we define, a dictionary that maps the incoming client packet’s ID to the method they’re going to trigger on the server. This happens both on the client and the server. Also on both of these, there is a class named “Packet” that holds information about the packets that the server sends, as well as the packets clients send. Moreover, in that same class, we have a packet initializer, reader, writer and helper functions regarding packets that we use to send both TCP and UDP packets.

On both server and client instances, there is a client class that, within it holds a TCP and UDP sub-class. These custom sub-classes are used to send TCP and UDP packets respectively, as well as establish the first connection. It is worth mentioning that this class also contains a reference to a Player class, so that every connected client is mapped to an actual player within the game.

On the client side, a request of connection can be made through the Client’s class “ConnectToServer” method. Much like the server side, this class holds the TCP and UDP classes to send data from the client to the server. In this sense, it contains a Dictionary to map the incoming server’s packet id to the correct method that that packet should trigger. This ensures the synchronization between client and server.

## TCP Connection

### Server

The first thing the server does when it starts, is start listening for incoming TCP connections. In this sense, if you open up the server Unity Project and take a look in the hierarchy, you can see an empty game object named “NetworkManager”. Upon closer inspection of that object, you can see that there are two C# scripts attached to it – *NetworkManager.cs* and *ThreadManager.cs*. For now, only the first one will interest us.

A screenshot of a computer

Description automatically generated with medium confidence

If you open up the *NetworkManager.cs* script, you will notice that on the Start method, – the first method to get called when Unity starts the game; Constructor - we first set the target frame rate to 30 frames per second. This limits how many times per second our game state is updated. In other words, the server’s tick rate.

You will also notice that we call a static method from the *Server.cs* script named Start passing the 50 as the maximum player count that our Server will handle and 26950 dictating the Port of the Server. This number was decided having in mind the well-known Port numbers.

Text

Description automatically generated

If we drill down the code and take a closer look at *Server.cs* file, we can see that this class provides a few attributes that will be used on the Start method or the methods that follow it:

* MaxPlayers
* Port
* Clients: dictionary of Integer, Client to store the connected client’s information.
* packetHandlers: dictionary to map the client’s packet to a handler
* TcpListener
* UdpListener

On the *Server.cs* Start method, we start by initializing the variables passed as parameters and then we call the InitializeServerData method.

Text

Description automatically generated

On the InitializeServerData method, we start by populating the “clients” dictionary. Note that, even though we have no connected clients, this is reserve slots for them according to the MaxPlayers attribute.

After that, we initialize the packetHandlers dictionary. This dictionary will map the incoming packets from different clients to an action on the server. Notice how we take an ID coming from the packet (through an enumerator called ClientPacket) and map it to a method in the *ServerHandle.cs* class. Please note that this dictionary is not crucial to the connection.

Text

Description automatically generated

Back on the Start method, we now initialize the TCP listener passing as parameter IP.Any, which means that we don’t have to assign our server’s IP statically, so we just need to know the current IP address of the machine in which the server is currently running. Notice that this may change if there is a dedicated machine for the server.

We then call the listener’s Start method and BeginAcceptTcpClient method passing in the TCPConnectCallback method as the parameter and null as the transfer object.

Text

Description automatically generated

On the TCPConnectCallback method, we first get the TCP Client returned by the TCP listerner’s EndAcceptTcpClient method. Once a player connects, we want to make sure that we continue listening to connections, so we recursively call the TCPConnectCallback method again.

Text

Description automatically generated

After that we iterate through the client’s dictionary and determine if we have any available slots left. If we do, we assign the first available slot to the new connection through the *Client.cs’s TCP* Connect method.

Text

Description automatically generated

On the *Client.cs* class, there is a subclass named *TCP.* This class is responsible for sending and receiving TCP data from the Client. On the Connect method, we first initialize the TCP related data of the newly connected client and we start reading the stream of data provided by the tcp listener passed by parameter as seen on the previous image. After that, we send a welcome packet to the client to trigger other connectivity related algorithms.

Text

Description automatically generated

For now we will focus on the ReceiveCallback method since it is the method that handles the incoming data. In a first instance we call the stream’s EndRead method which waits for the pending asynchronous read to complete and returns the length of the bytes that we are going to read.

If that length is 0 or less (should never be less), we disconnect the player because the client is no longer connected to the server.

Next call the Packet’s (receivedData) Reset method passing in the Boolean method HandleData to check if all data has been handled and, if it is, reset the Packet (again, receivedData).

After that, we recursively call the ReceiveCallback method through the stream’s BeginRead method until all handle has been dealt with.

We wrap this logic in a try-catch block since there are a lot of point in which our code can throw an exception. In which case, we disconnect the player.

Text

Description automatically generated

At this point the server has the ability to listen to new TCP connections but has no ability to send any data to it’s connected clients. Please note that for the sake of simplicity, a lot of already implemented methods were omitted in this explanation. This section only focuses on how the server is set up to listen to TCP connections. Eventually all of the remaining code will be referenced.

### Client

On the client side, we are now ready to establish a connection with the server. To analyze this, please turn your attention to the “Connect” button on the menu. You will see that it triggers the *UIManager.cs* ConnectToServer method which, as the name suggests, connects the client to the server.



Graphical user interface, text, application

Description automatically generated

Taking a closer look at the method, we can see that, besides doing some game logic, it calls the *Client.cs*’s ConnectToServer method.

Text

Description automatically generated

On the *Client.cs*, we have a singleton situation. Furthermore, we set an integer attribute named dataBufferSize to 4096 (bytes) which corresponds to 4MB. We also have an ip, port and id attributes, as well as a reference to it’s TCP Class

Text

Description automatically generated

Text

Description automatically generated

On that TCP class, we have a method, similarly with the Server’s Start method, we have a call to the InitializeClientData method and a call to the subclass TCP’s Connect method. The InitializeClientData method is not crucial to the initial connection between Server and Client as it maps the Packet’s ID to a method or action.

Graphical user interface, text

Description automatically generated

On the *Client*.cs TCP class, we have a TcpClient reference named socket, a NetworkStream for reading the stream of data a Packet attribute and a receivedBuffer byte array.

Text

Description automatically generated

On the TCP class Connect method, we use the parent’s class (Client) attributes to create a new TcpClient and assign it to the socket attribute. After that we refine the TCP’s receivedBuffer byte array to be length of the one defined on the parent class. Then we can call the TcpClient’s (socket) BeginConnect method, passing the singleton variable instance’s ip, port, the ConnectCallback method as a callback method and the TcpClient we just created.

Text

Description automatically generated

On the ConnectCallback method, we first call the socket’s EndConnect passing in the IAsyncResult as the parameter, then we check if the socket is connected or not in which case we return the method. Then we assign the stream attribute the the socket’s stream attribute and similarly to the Server, we call the stream’s BeginRead method passing in the corresponding attributes and the ReceiveCallback method.

Text

Description automatically generated

For now we will focus on the ReceiveCallback method since it is the method that handles the incoming data. In a first instance we call the stream’s EndRead method which waits for the pending asynchronous read to complete and returns the length of the bytes that we are going to read.

If that length is 0 or less (should never be less), we disconnect the player because the client is no longer connected to the server.

Next call the Packet’s (receivedData) Reset method passing in the Boolean method HandleData to check if all data has been handled and, if it is, reset the Packet (again, receivedData).

After that, we recursively call the ReceiveCallback method through the stream’s BeginRead method until all handle has been dealt with.

We wrap this logic in a try-catch block since there are a lot of point in which our code can throw an exception. In which case, we disconnect the player.

Text

Description automatically generated

At this point, the server can recognize the connection from the client. But it is worth noting that there is not yet an implementation for the sending of traffic between the hosts.

## Packets

Before we continue with the documentation on how the sending of packets between hosts works, we need to look at what are packets and the project’s *Packet.cs* class.

Since you can not send, for example, an integer or string through a wire, you need to, instead, send a byte array that represents a certain data structure and its value. To accomplish this, we use the in-house implementation of the *Packet.cs* class.

The premise is that a packet has a unique ID known to both client and server. So, in that sense, we have 2 enumerator types on the *Packet.cs* class. One to keep track of the available packets that a client can send/server can receive and one to keep track of the packets that the client can receive/server can send.

So, in that sense, if the client sends a packet with the ID = 1, the server knows that the client wants to jump. So we map that ID to an action and every time the client sends a packet with the ID = 1, the corresponding client jumps on the server. It is worth noting that, besides that ID, a host can also send other information including player ID, positions, rotations, usernames etc. It is also worth noting that this mapping is an architectural decision of the developers of the .NET and .NET.Sockets namespaces and they use a dictionary of <int, PacketHandler> that will be discussed on the TCP transport layer section of this document.

The *Packet.cs* class has 3 attributes:

* Buffer: is a List of bytes
* ReadableBuffer: array of bytes
* readPos: int

Text

Description automatically generated

There are also three constructors. One to create an empty packet, another one to create a packet with an ID and another one for received bytes on the stream.

Text

Description automatically generated

Text

Description automatically generated

### Packet functions

To help the manipulation of bytes, we created some functions with informational comments

#### SetBytes

Initializes the Packet’s data using a given byte array.

Text

Description automatically generated

#### WriteLength

Inserts the byte length of the packet at the start of the list.

Text

Description automatically generated

#### InsertInt

Inserts the ID of the packet. Not to be confused with the Insert(int value) method which inserts an integer that is not associated with the ID of the packet.

Text

Description automatically generated

#### ToArray

Returns the buffer’s content as an array.

Text

Description automatically generated

#### Length

Returns the length of the buffer

Text

Description automatically generated

#### UnreadLength

Returns the Length of the buffer minus the current reading position of the cursor.

Text

Description automatically generated

#### Reset

Resets the buffer according to Boolean. If false, we just unread the last integer (4 is the number of bytes for an integer)

Text

Description automatically generated

### Write data

Writing data to the packet is straightforward. You just need to add a sequence of bytes to the byte List. These are automatically converted from a primitive data structure (int, bool etc) to byte format, as the following example demonstrates.

Text

Description automatically generated

### Reading data

Reading data is a little bit more complex. Let’s take as an example the ReadInt method. If the size of the buffer is greater than the read position (cursor), that means there are no integers to read in the byte array and we throw an exception. Otherwise we take the next set of bytes from the list and convert them to an integer. If the local variable \_moveReadPos is true, we increase the read position by 4 (because an integer is represented by 4 bytes). The read position tells us which bytes in the packet we have converted and which ones are next. At the end, we return the value.

Text

Description automatically generated

### Sending a packet

To send a packet from the server to the client we use the *ServerSend.cs* class. In this class you can define a static method and, within your static method you can create and send TCP and UDP packets to all users, a single user, or every user except one, as the figure demonstrates.

Make sure that, on the client’s *Client.cs* class, there is a proper packet handler for the packet id you’re sending on the *ClientHandle.cs* class. The opposite happens if you want to send a packet from the client to the server.

Text

Description automatically generated

### Receiving a packet

At the top of the *Packet.cs* class, we have two enumerators starting with 1 each. As the names suggests, the ServerPackets are sent from the server to the client and the ClientPackets are sent from the client to the server.

Text

Description automatically generated

At the Server’s *Server.cs* class, on the InitializeServerData, we can see how each packet name is mapped to a handler. So, when a client sends, for example, a welcomeReceived packet (id = 1), the *ServerHandle.cs* WelcomeReceived method gets called. This happens under the hood on the .NET and .NET.Sockets namespace.

As you might guess, the same thing happens on the client side under the *Client.cs* InitializeClientData method

Text

Description automatically generated

Figure 4 - Client's IntializeClientData method

Text

Description automatically generated

Figure 5 - Server's InitializeServerData method

When receiving a packet, **make sure you are reading from the packet with the exact order you wrote to the packet.**

Let’s take the WelcomeReceived packet as an example. As you can see, on the client side, when we are sending the packet, we first write the ID of the client and only then it’s username. Note that since the *Packet.cs* class uses IDisposable, using the “using” keyword will automatically dispose of the object.

Text

Description automatically generated

On the server side, when we are receiving the packet (*ServerHandle.cs*), we are first reading the ID and only then we read the username.

Text

Description automatically generated

**Not complying with this might cause unpredictable errors as well as server crashing.**

## TCP transport layer

The TCP/IP protocol suite is the most widely used protocol suite on the world wide web. Its premise is to guarantee that the packet reaches its destination and there is no packet loss. However, this approach will cause an increase in server latency since every packet needs to be checked.

### Transmitting data from the server and receiving in the client

To send a TCP packet, first we created a SendData method on the *Client.cs* class’s TCP class. In that method, we check if the socket (TcpClient) is not null, and after that we call the stream’s BeginWrite. In this method we pass the packet in array format, offset of 0, the packet’s length as the length and null as the object state. We wrap this logic in a try-catch block to avoid any runtime errors that might crash our server.

A screenshot of a computer

Description automatically generated with medium confidence

Next, on the *ServerSend.cs* class, we add a SendTCPData method that takes in the client’s ID and the packet itself. Within this method we call the packet’s WriteLength method and use the correct client’s ID to send a packet to a specific client.

Text

Description automatically generated

Following the same logic as the previous method, we also have a method that send a TCP packet to all connected clients as well as a method to send a TCP packet to all connected clients except one.

Text

Description automatically generated

Text

Description automatically generated

After this, we can use these methods to send a TCP packet to the client. The following is how the Welcome packet is being sent on the *ServerSend.cs*.

Text

Description automatically generated

And we call this packet on the *Client.css* TCP class, on the connect method.

Text

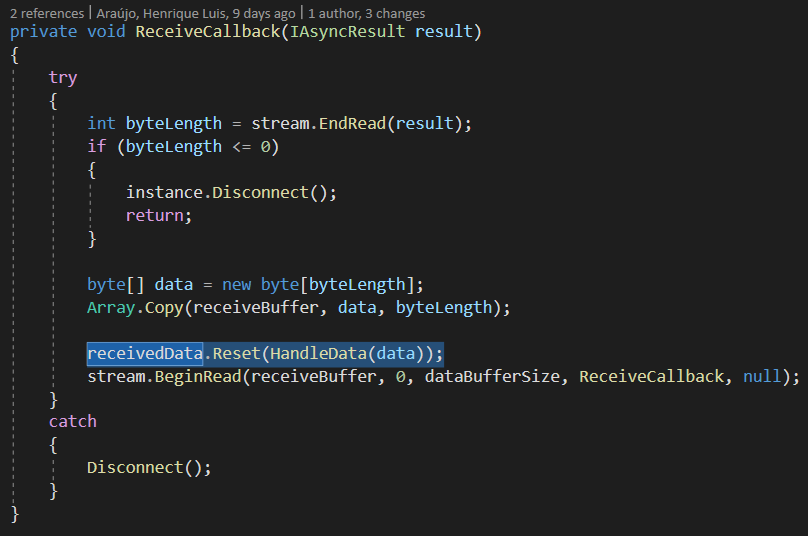
Description automatically generated

At this point, the packet is being sent to the client.

On the client side, we need to update the packetHandler’s dictionary on the *Client.css* and link the packet to a new method in *ClientHandle.css.* This was previously discussed on Packets/Receiving a packet topic.

Right before we call the BeginRead method from the stream attribute, we call the receivedData (*Packet.css*) Reset method. This will take a Boolean return from the HandleData method and determine if we reset the packet or not.

Since TCP is stream based, it sends a continuous stream of information. It will ensure all packets we send are delivered and in the correct order. While the chunks of data being sent are guaranteed to arrive, they aren’t guaranteed to be delivered in one piece. In this sense, when we send a packet it will be added to a larger list of bytes, and once enough bytes accumulate they’re sent in one bigger delivery. TCP leaves it up to the developer to handle cases where a packet is split between two deliveries which is why we don’t always want to reset the received bytes. There could still be a piece of a packet in it that the rest of it hasn’t arrived.



On the HandleData method, we check if the receivedData has more than 4 unread bytes. If it does, it means that we have the start of one of our packets because a integer consists of 4 byes and the start of our packet always begins with an integer representing the length of the packet.

Next we check if that integer is less than 1. If it is, we reset the data.

Text

Description automatically generated

Next, and still no the same method, we do a while loop in which we check if the packet length is greater than 0 and less than the unread length of the received data. If this is true, it means we still have another complete packet that we need to handle.

Inside a different thread from what the Unity framework is running, we then create a new packet using the bytes that were sent over the network. With this new packet, grab the correct packet handler (discussed previously) using our packet ID and invoke it passing it the packet instance. This is how the ID get’s mapped to the actual byte array.

Text

Description automatically generated

After that we check for the same condition as the beginning of the method to check if the packet has been processed in its entirety.

Text

Description automatically generated

Outside the loop, we check if the packet length is less or equal to 1, in which case we return true. If it is not, that means that what was processed was part of the packet and not its entirety and we should continue to handle it to its completion.

Text

Description automatically generated with medium confidence

At this stage, the server can send TCP packets to the server but the client can not respond.

### Transmitting data from the client and receiving it on the server

Like the server, in the client’s *Client.cs* class, we have a TCP sub-class with a method named SendData. This method is responsible for sending TCP data over to the server. We start by checking if the socket (TcpClient) is null and, if it’s not, we call the stream’s BeginWrite method, passing the packet as an array of bytes, an offset of 0, the packet length null for a callback and null for a object state.

We wrap this logic in a try-catch block since the method can produce an exception that could potentially crash the server.

Text

Description automatically generated

Much like to the server, we have a *ClientSend*.*cs* class where the sending logic is written on the client side. Inside that class, you will find a SendTCPData method that sends TCP packets to the server.

Graphical user interface, text

Description automatically generated

At this stage the client can now send TCP packets to the server. To illustrate this, let’s take a look at the WelcomeReceived packet. Much like the server, we use the using keyword to dispose of the packet after we have sent it.

Text

Description automatically generated

On the *ClientHandle.cs* Welcome method, we call the WelcomeReceived method over on the *ClientSend.cs.* This is because we want to inform the server that the transmission was successful as well as some other topics that will be discussed further on.

Text

Description automatically generated

Back on the server side, we need to be able to handle the incoming TCP packets coming from the client. This happens within the *ServerHandle.cs* class WelcomeReceived method which will be the method that we call once we detect the WelcomeReceived packet coming from the client.

In that method, we start by reading the data of the packet **in the same order that it was written.** This data will be used to spawn the player and print some good logging statements that we will take a look later on.

Text

Description automatically generated

On the *Server.cs,* much like the *Client.cs* on the client side, we need a dictionary of integers, PacketHandlers. Since the server is handling multiple clients at the same time, the PacketHandler itself will also have an ID.

We initialize that dictionary on the InitializeServerData method.

Text

Description automatically generated

Next, we need to handle the data we receive. Since the HandleData method on the client side works also for the server side, we just copy it and don’t need to change anything. We just need to also add the receivedData’s (Packet) Reset method, and pass the HandleData method as the parameter, much to the likes of the client side.

Text

Description automatically generated

At this stage, the client and both send and receive TCP packets to and from the server and the server can send packets to one, or more clients whilst receiving packets from multiple clients.

## UDP transport layer

The UDP/IP protocol suite serves the same purpose as its counterpart, the TCP/IP protocol suite. The main difference between these two is that whilst the TCP guarantees the delivery of packets in their correct order at the cost of latency, the UDP just sends it to the client disregarding if it gets there or not. This is mostly appropriate if we need to send packets every tick of the server.

### Client

The client connects via UDP to the server, once the first TCP packet has been received (Welcome packet, to be discussed further on). When this packet is received, on the client’s *ClientHandle.cs* Welcome method, the *Client.cs* subclass UDP’s method Connect is called, passing the already established TCP local port.



On the Connect method, we start by initiating the socket (UdpClient) with the local port passed in previously as an IPEndPoint. Note that this local port is a port in which the client is communicating and not the port to which we’re trying to send data to.

After that, we call the socket’s Connect method to start a UDP connection with the server and the BeginReceive method to start receiving data from the server which will be talked more in detail further on.

We also send an empty packet with the Client ID to the server so that our attempt to connect via UDP gets recognized and handled. This packet’s purpose is only to initiate a connection with the server. The SendData method is responsible for adding the client ID to the packet since it is vital for proper UDP usage.

Text

Description automatically generated

On the SendData method, we start by inserting the client ID on the packet. We do this because we’ll be using this value on the server to determine who sent it. Because of the way UDP works, we can’t give each client their own UDP client instance on the server without running into issues with closing of ports. We can validate the ID by checking if the endpoints match on the server.

After that, we check if the socket isn’t null and start sending our packet. The UdpClient’s BeginSend method is similar to the TcpClient’s.

Text

Description automatically generated

On the ReceiveCallback method, we start by assigning a byte array to the value returned by the socket’s EndReceive method, passing the IASyncResult as a parameter.

Immediately after, we call the socket’s BeginReceive method again, passing it the same parameters as before.

After that, we check if the byte array length is less than 4 (again, because 4 if the size of an integer) and, in that case, we disconnect the Client because the connection has terminated.

After that we call the HandleData method to retrieve the information from the byte array. We wrap all of this logic in a try-catch block and, should an exception be throw, we disconnect the UDP.

*Text

Description automatically generated*

On the handle data method, we start by creating a new packet with the bytes we received. Next, we read the length of the packet and store the packet’s information except its size on the “\_data” variable. In this way, the next integer we read from the packet will be a packet ID.

After that, we use the *ThreadManager.cs* main thread to read the packet ID and invoke the corresponding method using the packetHandlers.

Text

Description automatically generated

Now, on the *ClientSend.cs* file, we can add a SendUDPData method for sending UDP packets. Just like its TCP counterpart, we first write the length of the packet and then call the *Client.cs* instance’s UDP class to send the data.

Text

Description automatically generated

### Server

Text

Description automatically generatedSimilarly to its client counterpart, on the *Client.cs* UDP class, you can find a Connect method that assigns and endpoint to the connection.

Text

Description automatically generatedAfter that, we have a SendData method so we can send UDP packets to a certain client. Inside it, we simply call the *Server.cs* SendUDPData method, passing our assigned endpoint and a packet.

Much to the liking of the client, the UDP’s HandleData’s first two lines of code aims to remove the length of the packet from the byte array so that the next integer to be read will be the packet ID.

Text

Description automatically generatedWe use the *ThreadManager*.cs main thread to then read out the packet ID and invoke the appropriate method.

On the *Server.cs*, we have a UdpClient object named udpListener to keep the consistency with the TCP implementation.

Text

Description automatically generated with medium confidence

On the *Server.cs* Start method, initialize this new variable with the server’s port number and, after that, we call the udpListener’s (UdpClient) BeginReceive method, passing the UDPReceiveCallback method as a parameter.

Text

Description automatically generated

On the UDPReceiveCallback method, we start by creating an IPEndPoint with no specific IP address or port number. After that, we assign a byte array to the udpListener’s EndReceive method passing as second parameter a reference to the IPEndPoint we just created. This will not only get the bytes coming from the connection but will also populate the IPEndPoint with the information from the machine that the connection came from. Next we call the BeginReceive method again in case there are more connections that need handling.

Text

Description automatically generated

After that we check if the byte array is less than 4 bytes longs, in which case we return out of the method because there is no integer with the packet’s byte size to be read.

A picture containing text, device, meter, gauge

Description automatically generated

After that, we create a new packet with the bytes received in the “\_data” variable and start by reading the client ID. We do a quick check to see if it’s 0 which it should never be (starts at 1).

Immediately after, we check if the client that sent the packet has an active UDP connection. If it doesn’t, we connect it using the *Client.cs* UDP class Connect method and passing in the endpoint that was populated by the udpListener’s EndReceive method. Note that if this is the case, we know that the packet sent was the initial connection packet that opens up the client’s port.

Text

Description automatically generated

If a connection is present, we then check the endpoints of the client in our server’s dictionary and compare it to the one that we got over the network. If these match, we can be sure that there is no impersonation (ID and endpoints match), and we call the UDP’s HandleData packet.

Text

Description automatically generated

Lastly, on the *ServerSend.cs* class, we add a SendUDPData, SendUDPDataToAll and SendUDPDataToAllExeptOne methods, much to the liking of its TCP counterpart.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

At this point, we can just call the *ServerSend.cs*  methods regarding UDP and passing a normal packet to send a UDP packet to the server or, coming from the client, we can use the *ClientSend.cs* UDP methods to send a normal packet to the server.

# Game Logic

Game development includes a lot of “smoke and mirrors” to seem as though something is what it appears to be. In this section of this document, we aim to explain some of these tricks both in the code and on the Unity’s editor. Besides that, we will also explain all of the more general game logic including the trigger methods on both the client and server side (*ServerHandle.cs* and *ClientHandle*.*cs*).

There is not going to be a separation explaining the server side and then the client side because these are too connected to have their separate explanations. Even so, the documentation aims to be easy to follow along.

## Spawning a player

When the client instance is first launched, the user is presented with a text input box and a connect to server button. That button is linked to a method in the *UIManager.cs* class named ConnecToServer. In this sense, when a user presses the button, that method will run.

A computer screen capture

Description automatically generated with low confidence

A black rectangular object with white text

Description automatically generated with low confidence

Inside that method, we first set the UI elements of the menu to inactive as to not clutter the play screen and we call the  *Client.cs* instance ConnectToServer method.

Text

Description automatically generated

Inside that method, as already explained on the Networking section of this document, we connect via TCP.

Text

Description automatically generated

On the server side, the server acknowledges this connection on a TCPConnectCallback method and calls its own *Client.cs* Connect method that will, among other, send a Welcome packet to the client that has just connect. This Welcome packet is a very important connection packet that will initialize the client’s data on the server.

Text

Description automatically generated



Back on the client side, the client will receive this packet on the *ClientHandle.cs* and amongst initialing a UDP connection, will send a WelcomeReceived packet to the server carrying a username and the ID of the already IP connected client.

Text

Description automatically generated

Back on the server side’s *ServerHandle.cs* class, we receive that WelcomeReceived packet that carries the previously mentioned player information. We can access the client’s ID through the parameter and through the packet’s content since we wrote that two times. We check to see if they match before continuing. If that validation passes, we then call the *Client.cs* method SendIntoGame, passing the username as a parameter.

Text

Description automatically generated

In that method, we call the *NetworkManager.cs* InstantiatePlayer method that will instantiate the player game object that is attached to the *Client.cs* class at a specified location. In this way, a player game object belongs to one *Client.cs* and one *Client.cs* belongs to a player game object.

Text

Description automatically generated

Text

Description automatically generated

Immediately after that, we call that newly instantiated player game object’s (*PlayerManager.cs*) Initialize method which, in its own right, it’s the class contructor.



At this point, we have a physical player on the server’s game state but not on any of the client’s game states.

To synchronize the two sides, we send the information about this new player to all connected players and then we send information about all connected players to the newly joined player by sending them, in that logic, a SpawnPlayer packet that will make it so a player game object will be instantiated on the client side, and thus synchronizing both sides.

Text

Description automatically generated

This SpawnPlayer packet contains information about the transform properties of the object, as well as it’s corresponding to *PlayerManager.cs* attributes, which include its ID and username. We write to the packet according to the image below.

Text

Description automatically generated

On the client side, we start by reading the packet’s information in the same order that we wrote them and, immediately after we can spawn a player. Whether that player will be a local player or a remote player is decided on the *GameManager.cs* method SpawnPlayer

Text

Description automatically generated

On the *GameManager.cs* SpawnPlayer method, if the id coming in from the packet is different from the ID that we hold, we spawn a remote player. Otherwise we spawn a local player.

After that we initialize the player’s attributes (*PlayerManager.cs*) and we add it to our Dictionary of players.

Text

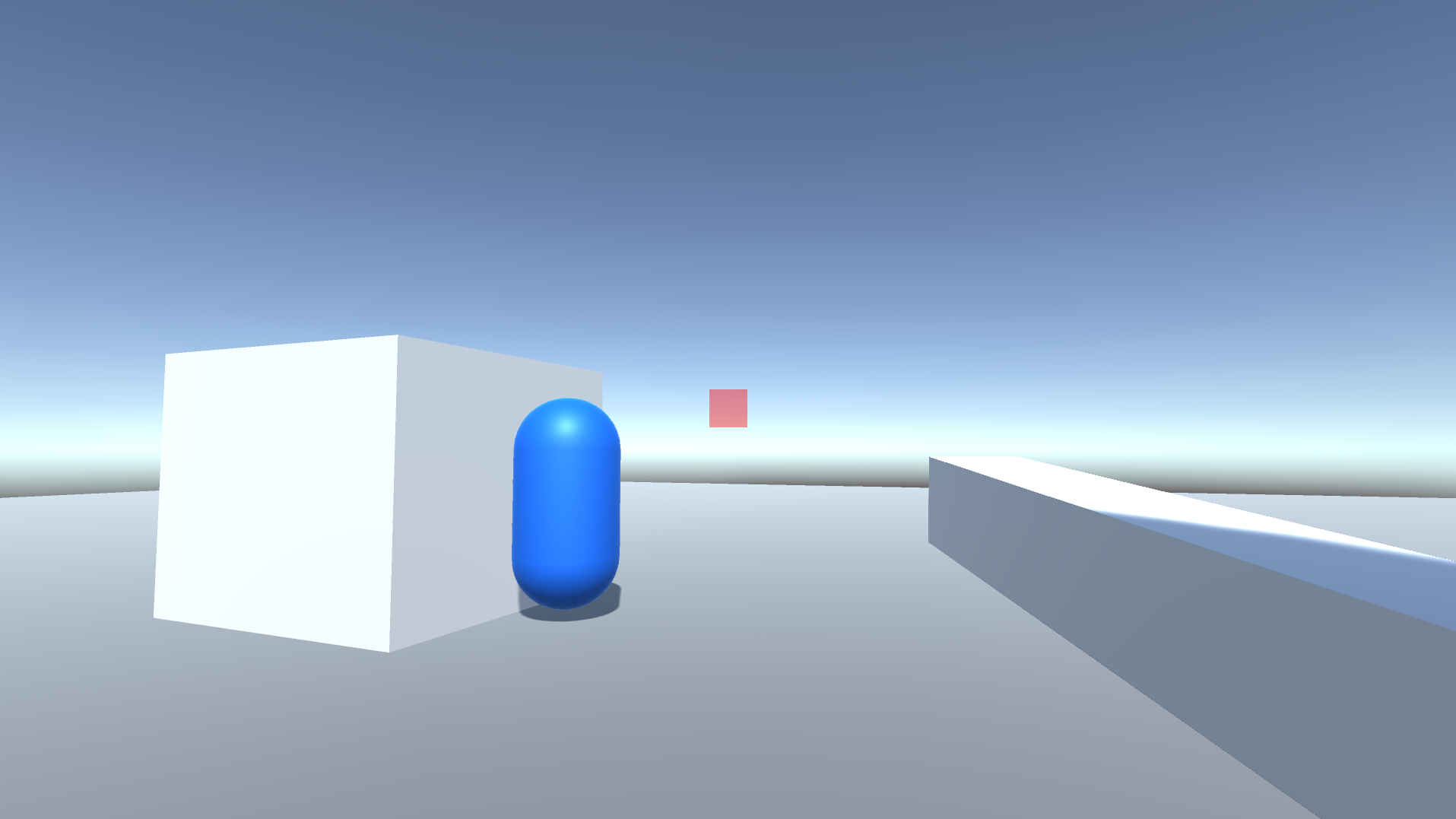
Description automatically generated

### Remote Player vs Local Player

A remote player (playerPrefab on the client) is a game object on a client’s game state that represents another client over the network. A remote player will always be updated by the server and never will it be controllable by the client that is seeing it.

A local player (localPlayerPrefab) is a game object that is controlled by the client and Is constantly sending updates on the inputs you are giving it to the server. It is the game object you are supposed to control.

In this next figure, you can see a local player holding the weapon in a first person view and a remote player represented by a blue cylinder.



And on the Unity editor

A picture containing logo

Description automatically generated

## Player Movement (rotation and position)

As previously stated, this project consists of an authoritative server. This means that the player position is determined by the server, through the user’s inputs, and sent back to the client. In this sense, we can start by taking a look at the client’s *PlayerController.cs* script which is going to be in charge of sending the user’s inputs to the client.

This is done by sending an array of Booleans, each representing a pressed key. We check which keys the user has pressed and then the array to the server. This is done every tick of the server (FixedUpdate method).

Text

Description automatically generated

In the *ClientSend.cs* PlayerMovement method, we first write that array to the packet keeping in mind that we have to read it the same way that we wrote it on the server side. In that same packet, we also write the Quaternion which represents the player’s rotation. As you can see, the movement is authoritative but the rotation of the player isn’t. This is because doing so would cause major screen jitter and fixing this issue would involve client side prediction which goes beyond the scope of this project.

We send this packet via UDP because it will be checked every tick of the server, so it doesn’t really matter if we lose a packet but it matters how long the packet takes to reach the server.

Text

Description automatically generated

Over on the server side, we receive that packet in the *ServerHandle.cs* PlayerMovement method and we start by reading the data in the correct order. After that, we take the client’s ID (*\_fromClient*) and access that client’s *Player.cs* object. Inside that, we call the SetInput method that will set the attributes of inputs and rotation, making the server’s player respond to the client’s input.

Text

Description automatically generated

We do that by, each tick of the server, mapping the first 4 elements of the Boolean array (AWDS keys) to a vector2, and then using a CharacterController to make the player move upon that vector, in the Move method.

Text

Description automatically generated

In the Move method, we start by mapping the Vector2 passed in as a parameter to a final Vector3 and multiplying that by the moveSpeed variable so that we can have some control over the player’s movement speed. After that, we check if the 5th element of the input array is true (space bar) and make the player’s y velocity change accordingly.

Finally, we call the CharacterController’s Move method and send that player’s position and rotation to all connected clients, through the *ServerSend.cs* PlayerPosition and PlayerRotation methods.

Text

Description automatically generated

In the PlayerPosition method, we send the computed position to all connected clients, including the client that sent the input array. We, again, do this through the UDP listener because this method is getting called every frame and it won’t matter if a packet gets lost.

Text

Description automatically generated

In the PlayerRotation method, we send the player’s rotation to everyone except the client that sent the Quaternion value to the server. This is because, in this instance, the player’s rotation is not server authoritative, hence, we’re just informing the other players of the rotation that a certain player has.

Text

Description automatically generated

## Player shooting, health and respawn

On the client side’s *PlayerController.cs* script, we check every frame of the client if he has pressed the Mouse0 button. We do this in the Unity’s MonoBehaviour Update method because it is imperative that a shot is registered. When we detect such input, we then call the *ClientSend.cs* PlayerShoot method, passing in the camera’s transform which will be our shooting origin.

Graphical user interface, text

Description automatically generated

On the *ClientSend.cs* PlayerShoot method, we initialize a packet passing in the “playerShoot” ID and write the vector passed in through the previous method. We send this packet through TCP because we can not afford to lose it, as losing it could possibly be game breaking.

Text

Description automatically generated

On the server side’s *ServerHandle.cs* script, we receive the packet in the PlayerShoot method and start by reading the direction in which the player was facing when he pressed mouse0 on the client side. After that, we access that client’s *Player.cs* script and call the Shoot method, passing in that same vector.

It is worth noting that we could simply use the server’s render of the player’s rotation for this but since the rotation is not server authoritative, it is better for the client to send in which direction he actually was facing instead of relying on a possible outdated state of game that the server holds.

Text

Description automatically generated

On the *Player.cs* Shoot method, we cast a ray and, if that ray hits a game object with the tag “Player”, he then accesses that game object’s *Player.cs* script and calls the TakeDamage method, passing in the amount of damage that the weapon he is currently holding deals.

Text

Description automatically generated

On the TakeDamage method, we first check if the health is already below zero, in which case it doesn’t make sense to continue with the logic.

Text

Description automatically generated

After that, we subtract the damage from the player’s current health and check if it is now below or equal to zero. If it is, we start by disabling its ability to move and set a new position for it (respawn point). We have to send this new position back to all clients, as we do for every movement the player makes on the server side.

Text

Description automatically generated

After that, we start a simple Coroutine Respawn that is going to reset the player’s properties and make it so it is “alive” again and send a packet informing the other players of this.

Text

Description automatically generated

After the Coroutine, we call the *ServerSend.cs* method PlayerHealth which is going to update that player’s health by sending it to all connected clients. We do this by creating a packet with the playerHealth ID and writting in it the player ID and its health and sending It through TCP so it doesn’t get lost.

Text

Description automatically generated

### Player health

The *ServerSend.cs* PlayerHealth method is used to synchronize all player’s health throughout all connected players. In this since, we first create a packet with the ID of playerHealth and write on it the player’s id and health values. We send this data over TCP to all connected clients.

Text

Description automatically generated

On the client side’s *ClientHandle.cs* PlayerHealth method, we start by reading the values in their correct order and, with those values, access the right player instance residing on the *GameManager.cs* player list and we set it’s health to the one the server provided.

Text

Description automatically generated

On the *PlayerManager.cs* SetHealth class, we set the health of the player to the one given by the server and, if that health is equal or below zero, we call the Die method.

Graphical user interface, text, application, chat or text message

Description automatically generated

The Die method, simply turns the mesh renderer of the player off, whilst the server is repositioning the player to it’s respawn position

Text

Description automatically generated

### Player respawned

On the *ServerSend.cs* PlayerRespawned method, we simply create a new packet with the playerRespawned ID and write the player’s ID to it. We then send it to all connected players via TCP.

Text

Description automatically generated

Over on the client side’s *ClientHandle.cs* PlayerRespawned method, we take action upon that packet and start by reading the ID of the player. We use this ID to access the *GameManager.cs* players list and call the connect player’s Respawn method.

Text

Description automatically generated

On the Respawn method, we simply turn the mesh back to visible and set the player’s health back to its max health.

Text

Description automatically generated