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.

# Network Flow Overview

Diagram

Description automatically generated

Figure 1 - 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.

# Server/Client connection

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. This explanation will be divided in two – server side and client side.

## 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 Cient. 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 2 - Client's IntializeClientData method

Text

Description automatically generated

Figure 3 - 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.

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.