“Limb-For-Limb: This Time It’s Global!” Networking Technical Specification

This document details in-depth how the networking and database systems for my individual extension of the game “Limb-For-Limb” are set up.

# Introduction

The game uses a client-server model with a combined TCP/UDP protocol. The server is hosted by an Oracle Cloud Computing Web Instance running on Ubuntu, and it houses the 24/7 server code and database.

The low-level networking code is handled by the SFML Networking API and the database is handled on a .db file managed through the SQLITE3 C-Library

# Structure Overview

## Connection

The following diagram are some node maps for the high-level networking infrastructure.

A diagram of a network manager

Description automatically generated

A diagram of a server

Description automatically generated

When a client (AKA a Network Manager) wants to connect to the server, they send a TCP connection request to the server. On the server’s side, a new TCP socket is made each frame until a connection request is received, at which point the server allocates that socket to the client, calculates the network manager index for the client (its position in the map), adds a new entry to the connnectedNetworkManagers map, and then sends the network manager index back to the client for them to store it. The client’s UDP socket is bound to a free port, this port is then sent through a packet to the server and stored in the server’s connectedUdpPorts map. The client is now connected to the server.

## Disconnection

For a server which runs 24/7, it’s important that the disconnection process for clients is extremely robust. This means there are two different ways that a client can be disconnected from the server.

1. Disconnection request – This is the standard way that a client should be disconnected from the server. If the user presses the “Go Offline” button, or quits the program, or anything similar which requires specific input from the user that the program can pick up on, the network manager will send a disconnect request to the server through a TCP connection. When the server receives such a request, they will remove all knowledge of the client. This request can be sent by itself through a function public to the NetworkManager class but is also contained within the class destructor.
2. Unexpected disconnection – This is a backup plan for if something unexpected occurs causing the client to no longer have communication with the server (E.g.: If the user loses internet connection or the power goes out or the program crashes). In such cases where the program didn’t have time to send a proper disconnection request, the network manager will still be disconnected from the server because the server periodically sends packets through a TCP connection to the client to check that they’re still there. If this packet fails to send, the client is no longer reachable and can be disconnected.

## Packets

“Packet” is the name given to a stream of data sent over a network. In my program, all data sent is contained in packets. The key point is that it’s vitally important that both sides of the network agree on the structure of packets or else device A won’t know how to decode device B’s messages.

Devices can only communicate through these packets, so it isn’t possible to send a message saying how your message is encoded, because then how will the recipient know how *that* message is encoded!

The solution to this is to have a fixed structure for packets that both the client and the server have been hard programmed to follow. In my implementation, that structure looks like this:



Now as much as I’ve just yapped on about how important it is for this structure to be rigid and hard-coded, it’s actually not rigid nor hard-coded!

In fact, the above diagram is just a general structure, and the packet code always comes first when being sent client to server, but depending on what the packet code is, the structure of the following data can change.

Wait. What’s a packet code?

### Packet Codes

A packet code is a short descriptor for the *kind* of data that is contained within a packet. **This** is the thing that the client and server agrees on. Here is a list of all the packet codes:

* UDPConnect
* RemoveNetworkManager
* KeyChange
* PositionChange
* Grounded
* Flip
* Health
* ItemBoxSpawn
* Nums
* UsernameRegister
* UsernameAvailabilityStatus
* UUID
* Login
* LoginStatus
* Ranking
* Logout
* UsernameInvite
* UserExists
* UserOnline
* UserFree
* InvitedUserNetworkManagerIndex
* MatchInvitation
* MatchAcceptanceClientToServer
* MatchAcceptanceServerToClient
* PlayerNum
* MatchSceneLoaded
* MatchStart
* MatchWin
* MatchLeave
* MatchEnd

The important takeaway is that packets might be paired with different data, but if a packet has a certain packet code, it will **always** have the same type of data, and so will always be handled by the client and server in the same way.

### Network Listeners

A network listener is a client-side template class which can be specialised to any other class. Here’s how it works:

* A class needs to be affected by the network (e.g. an Online Player needs to have its position changed based on data from the network).
* A user-defined NetworkListener<OnlinePlayer> specialisation is made which holds a reference to an OnlinePlayer.
* In the NetworkListener<OnlinePlayer> specialisation, an InterpretData() method is written to take in a packet as an argument and extract its data to apply it to the OnlinePlayer reference (e.g.: it might receive a position change packet and use the data to update the OnlinePlayer reference’s position).
* The OnlinePlayer class can now hold a pointer to a NetworkListener<OnlinePlayer>.
* In the OnlinePlayer constructor, the NetworkManager::GenerateNetworkListener() method can be called, passing in a reference to itself.
* The network manager creates a new NetworkListener<OnlinePlayer> object and stores it in a vector of all the network listeners.
* NetworkManager::GenerateNetworkListener() returns a pointer to this network listener for the OnlinePlayer object to store.

Now here’s what happens when the network manager receives a packet from the server:

* The network manager receives a packet from the server.
* It extracts the network listener index from the packet if there is one.
* It indexes into the vector of network listeners to get the associated network listener.
* It calls the InterpretData() method of the network listener, passing in the packet (in the structure PacketCode+Data).
* The network listener extracts the PacketCode and uses it to figure out what kind of operation to apply to its parent reference (e.g.: the OnlinePlayer).
* It then finally extracts the data to apply the exact operation to its parent reference (e.g.: setting the position of the OnlinePlayer using the x and y value from the packet).

### Reserved Network Listeners

The above implementation works, but it runs into the same issue we had with packets, which is that of structure.

In other words, if client A wants to send a packet to client B to update something (to stay on trend, let’s say the player’s position), how is client A supposed to know what the appropriate network listener index is? If network listeners are just added into the vector when NetworkManager::GenerateNetworkListener() is called, it would seem client A would need some kind of knowledge of client B’s object-instantiation-order. Surely not, right? There has to be a better way.

As mentioned above, we’ve already ran into a similar issue before when we were trying to decide on the structure packets, and the solution was to hard code both sides to understand these “Packet Codes”.

My implementation does a similar thing with the network listeners, creating a public enum in the NetworkManager and Server classes known as “ReservedEntityIndexTable” to hold a collection of entities that should be given a reserved spot in the vector. These reserved entities are:

* PLAYER\_1
* PLAYER\_2
* REGISTRATION\_SCREEN
* LOGIN\_SCREEN
* SEND\_INVITE\_SCREEN
* NETWORK\_SCENE
* MATCH\_INVITATION\_INTERRUPT

When the NetworkManager is first constructed, it stores one nullptr for every reserved entity there is. Then, when NetworkManager::GenerateNetworkListener() is called, an optional ReservedEntityIndexTable parameter can be provided and the network manager will create the network listener at the reserved position.

Now, if clients A and B are in a match, and client A is player 1, then if client A needs to send position data, they can send it alongside the PLAYER\_1 reserved entity index and client B will know to send the packet to player 1.

## Combined TCP/UDP Protocol

This project was my first time ever looking into networking and before I started, I didn’t even know what TCP, UDP or a protocol in general was.

After minimal research, I hastily decided that I would use UDP for my program since I thought it was the standard for real time games due to the low latency that it’s famous for. Unfortunately, after implementing my entire networking infrastructure based on the UDP protocol, I ran into immense packet loss problems which it is also famous for.

So I then spent a great deal of time converting my program code to work with the TCP protocol, and it was fantastic! Thanks to TCP, I had no more packet loss issues! I was so happy that I almost didn’t realise just how bad the latency had gotten.

It seemed impossible. If I use UDP, I get great speeds but horrible packet loss, and if I use TCP, I get no packet loss but horrible speeds.

Obviously, the solution I eventually landed on was to use a combined TCP/UDP protocol, this was ideal because it let me decide for myself whether I want data to be sent over a TCP connection or a UDP connection.

In the end, I only ended up needing the UDP protocol for position data, since this is sent every frame that the player is moving. In this instance, reducing latency is of the utmost priority, and if a few packets get lost here and there, it doesn’t matter because another one will most likely be sent the next frame, correcting the previous one that was missed. TCP is used for everything else since it’s very important that things like attacks (or logging in!) definitely arrive, more important than the speed at which those things arrive.

## Online Matches

### From The User’s Perspective

If a user wants to send a match invitation to one of their friends, the process is detailed as follows:

A diagram of a program

Description automatically generated

### From The Server’s Perspective

From the server’s perspective, it receives a packet with the “UsernameInvite” packet code which contains the invited username, the checks are handled as follows:

* User exists – This is checking if the entered username is present in the database’s AccountInfo table.
* User online – When a user logs in, their NMI and username is added to a map (called onlineUsers), so the user-online check is just checking if the username is present in this map.
* User free – When a user enters an online match with another user, those two users’ NMIs are added to a map called matchedUsers, so the server will check if the invited user is in this map, if they are, then they’re currently in an online match and therefore they are “not free”.
* At this point, all the checks have been passed, so the invitation is sent to the invited user in a packet containing the inviting user’s username, ranking, and NMI. The network listener associated with this packet is a reserved one: MATCH\_INVITATION\_INTERRUPT – a network listener of this type can be attached to a scene class to allow it to receive invitations.
* The invited user receives the match invitation packet and can choose whether to accept or deny it, this is then sent back to the server who sends it on to the inviting user.
* If the invitation is accepted, both clients start loading the network scene.
* When a client finishes loading the network scene, they send a packet with the packet code “MatchSceneLoaded” to the server.
* Once the server has received a MatchSceneLoaded packet from both clients, the match can commence, the server adds both NMIs to the matchedUsers map and sends a MatchStart packet to both clients.

The point of these MatchSceneLoaded packets is to avoid the scenario where one client has a much more powerful computer than the other, meaning their scene loads faster and they can start fighting sooner.

### Ending Matches

A match can be ended in one of two different ways, these are:

1. A player winning – much like a local match, the typical way for a player to win is by getting their opponent’s health to 0. At which point, the winning player sends a MatchWin packet to the server and the rankings are updated accordingly.
2. A player leaving through disconnection or the pause menu – In this instance, the other player will win and the rankings are updated accordingly.

### Communication During Matches

While two players are in a match, they communicate using each other’s NMIs, their opponent’s NMI is sent to them by the server during the invitation process.

This implementation makes things really easy for the server which uses user’s NMIs as a bridge between everything (username, ranking, match status, online status, etc.).

### The OnlinePlayer Class (And a Note on Confusing Terminology)

Once players join the match, they are given their player number by the server (i.e.: Player 1 or Player 2). The match scene on both client’s devices will have two OnlinePlayer objects (one for the user and one for the opponent).

The OnlinePlayer class has a bool flag attribute to state whether the OnlinePlayer will be controlled by the user or the network (the opponent). I am mentioning this because the flag (“isLocal”) leads to some rather confusing terminology, that being, a “Local OnlinePlayer” and an “Online OnlinePlayer”.

So to be clear: on a single client’s device, they have a scene with two OnlinePlayer objects, one of them will be controlled by the client and will send its data (position, attacks, health, etc.) to the opponent, this OnlinePlayer is referred to as the “Local OnlinePlayer”. The other OnlinePlayer is controlled by a network listener and will get its position data from the opponent who is controlling it virtually, this OnlinePlayer is referred to as the “Online OnlinePlayer”. Apologies for the confusing terminology.

# Post-Mortem

The networking portion of this project taught me a lot about the basics of networking and I feel like I’ve learnt a lot. My implementation here is very basic as far as modern games go but it still ended up working really well I think.

With that being said however, here are some things I would’ve loved to look into but just didn’t have the time to:

* Anti Cheat – This is by far one of the biggest reasons for having a client-server model and I didn’t get the chance to really delve into it.
* Prediction Algorithms – This wasn’t an issue of time; real time network prediction is just notoriously a really confusing part of networking for games. I imagine I’ll learn more about this in our 4th year networking module.

It’s also quite buggy in some places but there are just so many possible edge cases to handle so I wasn’t able to get them all. I focused more on the server’s robustness than the client’s, since if the client’s game crashes, it’s annoying, but it’s not nearly as big of a problem as the server crashing.