IT UNIVERSITY OF COPENHAGEN

Modifying Multiplayer Networking in Godot: A Collaborative Exploration with Chasing Carrots

Fabio Mangiameli

Individual Project

M.Sc. Games Tech Track
IT University of Copenhagen
Denmark

15th December 2025

Abstract

This paper presents an individual project carried out in collaboration with the independent game studio Chasing Carrots, focusing on the modification of multiplayer networking in the Godot game engine. The work investigates whether a custom networking layer can provide greater flexibility than Godot's built in MultiplayerAPI while maintaining correctness and functionality. To achieve this, the project refactors existing network code to interact directly with the MultiplayerPeer and introduces a custom Communication Line System designed to better support the studio's modular workflow. In addition, the report explores the implementation of full peer to peer network meshes using both ENet and Epic Online Services, enabling direct client to client communication while preserving an authority based model. The new architecture was integrated into the studio's production codebase and evaluated through targeted test projects and internal play sessions. Results show that the custom system successfully replaces Godot's default multiplayer framework, supports multiple concurrent network meshes, and improves flexibility for features such as proximity voice chat. While the solution introduces additional complexity and requires further stability testing, it demonstrates that a tailored networking layer can better align with the needs of a professional game development workflow.

Contents

1 Introduction

This individual project was both an exploration into the work of an indie game studio and a deeper look into networking code in video games, including development with Godot. The two main focuses were becoming familiar with the general workflow and the tools used at Chasing Carrots, and developing an internal project that improves their flexibility in multiplayer development. A large part of this involved learning how the studio works with their custom Godot Engine build and the systems they have created to support their own games. The internal project centred around modifying the Godot networking layer to give Chasing Carrots more control during development. This project had two primary goals. First, the studio wanted to reduce reliance on the standard ***MultiplayerAPI*** [16] provided by Godot and instead work more directly with the underlying ***MultiplayerPeer*** [17]. Typically, the ***MultiplayerAPI*** wraps the peer and provides an easy way to send data between clients. While this system is simple to use, it also requires the ***MultiplayerSpawner*** [18] and ***MultiplayerSynchronizer*** [19] to function correctly. Chasing Carrots does not want these objects in their workflow, so the goal was to achieve a more flexible and direct interaction with the ***MultiplayerPeer*** without the extra systems that Godot expects by default. This raises the question: Can a custom networking layer provide greater flexibility than Godot's built-in ***MultiplayerAPI*** without sacrificing correctness and functionality?

The second focus was on implementing a network mesh for communication. The current setup uses a client-to-server peer-to-peer structure, where each client connects only to the server. Clients are not directly connected to each other. This means that if one client wants to send a packet to another, the server must act as a relay. In a mesh network, all clients are connected to each other via peer-to-peer links. There is still one client acting as the server because this is needed for game initialisation, spawning logic, and general ownership of the game state. However, the mesh makes certain features much more efficient. A good example is the proximity voice chat used in the new project at Chasing Carrots. Audio packets do not need to take an extra hop through the server and can instead be sent directly between clients, improving latency and reducing server load.

Overall, while this project provided a look behind the curtain at how a professional environment develops games, it was also a hands-on exploration of multiplayer systems, custom networking solutions, and the use of different network technologies to enable smooth and responsive gameplay.

2 Chasing Carrots

Chasing Carrots is an independent game studio based in Stuttgart, in the south of Germany. It was founded in 2011 and released its first game, Pressure [25], in March 2013. Pressure is a twin-stick shooter where you control a car driving through hordes of enemies, fighting your way to the end of each level. The next game, Cosmonautica [26], was released in 2015. It is a spaceship management game that combines humour with trading and crew simulation. After that, the studio developed a remake of Pressure called Pressure Overdrive [27], which came out in 2017. Pressure Overdrive was a complete overhaul featuring enhanced graphics, improved controls and much more content. It was also brought to the Nintendo Switch. After that, the studio focused on a new project called Good Company [29], released in 2022. This game marked a return to management gameplay. In Good Company, your task is to build a business, manage your employees and earn as much money as possible. Following Good Company, Chasing Carrots released their most successful game so far: Halls of Torment [28]. Inspired by Vampire Survivors [31], it is a bullet haven game that entered early access in 2023 and was fully released in 2024. It was also the studio's first title developed using Godot, which helped establish Chasing Carrots as a well-known developer in the Godot community.

While development on Halls of Torment is still ongoing, with a console release just a few months ago, the studio is now focusing on a new game which is also the reason for this individual project. Inspired by games like Lethal Company [33] and Phasmophobia [30], this upcoming title will offer a cooperative experience where up to four players can explore the harsh and frozen environment of Antarctica together. In the game, you play as part of a team of explorers travelling across the icy desert in a massive land cruiser. The cruiser serves as a hub where players can navigate, store items and warm up to avoid freezing to death. However, it needs constant maintenance and can break down if damaged, so players should drive carefully. The main goal of the expedition is to discover anomalies. These can be portals to other worlds, points of interest in the landscape or dangerous encounters. Valuable items and essential data can be found near these anomalies, which players must collect and extract safely. The game is planned for early access release in 2026 and is currently in the middle of development.

3 Related Work

Back in 2022, Chasing Carrots decided to swap their tech stack. While Good Company was developed with Unity, the studio decided to create Halls of Torment with Godot. Because the development was going quite well, they are also sticking with it for the next project. Therefore, the following section will describe the different technologies the studio interacts with during the development.

3.1 Godot Game Engine

Godot is an open-source game engine [9] that was first released as a stable version on December 15, 2014 [11]. Juan Linetsky and Ariel Manzur originally developed it with the goal of creating a fully open, community-driven engine that empowers developers to build games without restrictive licensing models or revenue sharing. In contrast to many commercial engines, Godot is and will remain completely free, both in price and in freedom, under the permissive MIT license [12]. Technically, Godot is written in modern C++ and uses a unique node- and scene-based architecture. Game content is structured as trees of ***Nodes***, each with a specific function, making projects highly modular, reusable, and easy to reason about. This design enables developers to compose complex behaviour out of simple building blocks, reducing both boilerplate and development time. To support rapid iteration, Godot provides its own high-level scripting language, GDScript. Tailored specifically to the engine, GDScript offers Python-like syntax, tight editor integration, and extremely fast iteration cycles since it does not require recompilation after changes. In addition to GDScript, the engine also supports C# and native code through C and C++. For teams like Chasing Carrots, Godot's extensibility is one of its most valuable characteristics.

There are four main options to modify the engine to personal preferences:

Plugins The first way of modification is plugins. Using GDScript or C#, developers can create tools, UI panels, importers, and automation features directly inside the editor. These add-ons can be shared through the Godot Asset Library or by sharing the source code and require no native compilation, making them easy to distribute and maintain. [14]

GDExtension system Another way to create plugins is to use the Godot GDExtension system [22]. GDExtensions are written in C++ and interface directly with the engine's core API. Developers can implement custom systems, bindings, or gameplay features and distribute them as platform-specific libraries. This allows teams to add highly performant native functionality without modifying the engine itself.

Modules Modules [2] sit one level deeper than extensions: they are compiled directly into the engine source but remain separated from core systems. This enables parallel development. Teams can maintain custom modules while still updating to new engine versions with relatively low merge overhead. Modules are ideal for systems that require tight engine integration.

Direct Engine Modification As a fully open-source project [9], Godot allows developers to modify any part of the engine codebase. While this grants complete control, it may lead to merge conflicts when updating to new versions, especially since Godot's architecture evolves significantly between major releases.

This project will place particular emphasis on the use of modules and GDExtensions. Since the studio already uses a modified engine version that employs a so-called ***CompositeNodeSystem*** implemented in modules. GDExtensions will be discussed in greater detail later, as an extension [13] was used for Epic Online Services (EOS) [7], which had to be modified to enable the use of multiple network meshes.

Godot also features built-in networking support suitable for real-time multiplayer games. Its multiplayer architecture revolves around the ***MultiplayerPeer*** class, an abstract interface that can be implemented in C++ or accessed through GDScript. The engine provides a default implementation using ENet called ***ENetMultiplayerPeer***, a reliable UDP-based networking library well-suited for fast-paced gameplay. Developers may also create custom peers to integrate external networking backends such as the ***EOSGMultiplayerPeer*** implemented to use EOS for example [13].

Beyond its technical capabilities, Godot benefits from an active and rapidly growing community. The engine's open governance model encourages contributions from studios, hobbyists, and industry professionals alike. Extensive documentation, tutorials, and community plugins help support newcomers, while regular releases continue to introduce renderer improvements, performance gains, new tooling, and expanded platform support.

3.2 Multiplayer Technologies

Chasing Carrots utilizes two network technologies: ENet and Epic Online services [4] [7]. The ***ENetMultiplayerPeer*** is primarily used for debugging, since setting up a connection is faster than with EOS. With EOS, a connection must first be initialised on the EOS servers, and testing with multiple instances requires an additional EOS authoring tool [23]. EOS will be the main technology palyers will interact with when the game is shipped, because it handles online functionality such as matchmaking, lobbies, and cross-platform support. In comparison, ENet is more lightweight and allows quick local testing with several instances of the game running on the same machine. Another advantage is that ENet can always be used on a local network to enable multiplayer gameplay without relying on any external service. EOS depends heavily on Epic Games, and Chasing Carrots cannot assume that these online services will always be available or fit every development scenario. Because of this, time is invested in supporting both technologies.

3.2.1 ENet

In the backbone of the default Godot multiplayer framework lies the ENet C++ library [4]. It is an open source networking library built on top of the User Datagram Protocol (UDP) [24]. ENet was initially developed for the game Cube [32] as its UDP network layer. The main idea behind ENet was to create a network layer that combines beneficial properties of both Transmission Control Protocol (TCP) [3] and UDP. To achieve this, a uniform protocol was designed that sits on top of UDP and adds features such as optional reliability, packet ordering, channels, and automatic packet fragmentation. Godot integrates ENet by exposing it through the ***ENetMultiplayerPeer***. This makes it easy to create connections, send packets, and manage peers without interacting with the ENet library directly.

3.2.2 Epic Online Services

As the name suggests, Epic Online Services [7] is a collection of online services that provide networking features to developers. Epic Games developed EOS and offers it free of charge to all users. The services include features such as matchmaking, voice chat, user authentication, friends and presence systems, and anti-cheat integration. EOS is designed to support cross-play across platforms, enabling players on PC and consoles to play together seamlessly. Epic Games provides an SDK that allows developers to integrate EOS into their own games. To integrate EOS into Godot, Chasing Carrots uses an open source extension called Epic Online Services Godot (EOSG)[13]. This extension communicates with the official EOS API and exposes its functionality through GDScript. It also offers a custom EOS peer called ***EOSGMultiplayerPeer*** that can be used the same way as the ***ENetMultiplayerPeer***. This makes switching between the two technologies straightforward and allows the team to test both solutions without changing the rest of the networking logic.

4 Customizing Godot for Studio Workflow

Chasing Carrots has been working with the Godot Game Engine since the development on Halls of Torment began in 2022. The team originally picked Godot out of curiosity and built prototypes with it to see how it felt. Since those early experiments went well, the studio shifted more and more of its workflow toward Godot and away from Unity. Cost played an important role, too. Unity requires a paid plan for studios, while Godot is completely free to use and fully open source. When Unity made headlines in September 2023 because of its new pricing policy [8], it confirmed the studio's decision to move on. By that time, Halls of Torment had already proven itself in early access, and it became clear that the team had chosen the right tool for the job. One of Godot's most significant advantages, as with many open-source projects, is how easy it is to modify. Chasing Carrots maintains its own fork [1] of the official Godot repository [9]. The goal is to stay as close as possible to the latest engine version while still allowing integration of custom changes. When necessary, the team cherry-picks experimental updates from the main Godot repository, especially when a feature is needed sooner rather than later. On top of that, they add their own features whenever the engine is missing something essential.

The following section will take a closer look at how the studio works, with a particular focus on how the Godot engine fits into their development process. The following section focuses specifically on the modifications the studio has made to the Godot engine to support a more modular development style. It does not deal with the studio's broader working environment or the additional tools used in day-to-day production. Version control systems, project management software, and internal communication are therefore not part of the discussion. All the following modifications were implemented with the Godot engine module system. Consequently, it was written directly in Godot's source code and requires a custom build to use the changes in the editor.

4.1 Composite Node System

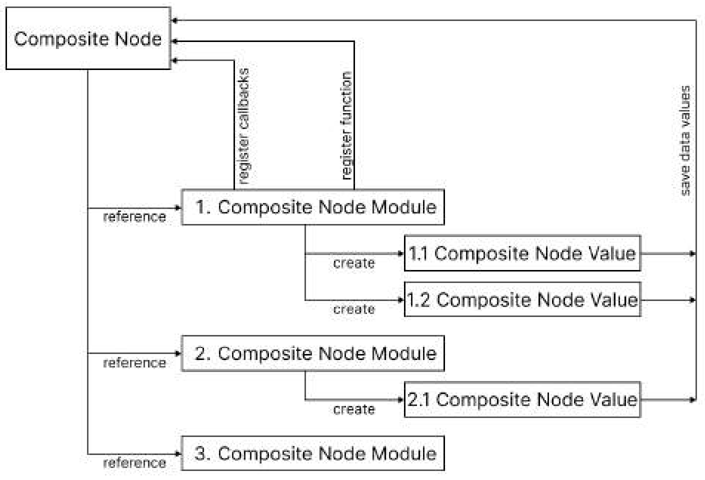


Figure 1: Example of the Composite Node System

As mentioned earlier, Godot organises its game objects in a tree structure, where every object inherits from the base class ***Node***. To support more modular development, Chasing Carrots introduced a new type called ***CompositeNode***. These ***Nodes*** can store data, create functions, or register callbacks, making them highly flexible building blocks for game logic. Figure 1 illustrates a typical use case for the ***CompositeNode*** system. The three main classes involved are ***CompositeNode***, ***CompositeNodeModule***, and ***CompositeNodeValue***. It

```gdscript
extends CompositeNodeModule

var data : CompositeNodeValue
var synchronized_data : CompositeNodeValue

var inital_value : int = 0

func _ready_composite_node() -> void:
  data = create_non_synchronized_value(&"Data", inital_value)
  synchronized_data = create_synchronized_value(
    & "SynchronizedData",
    inital_value,
    CompositeNode.OnChange,
    CompositeNode.U32
  )

  register_function(&"PrintData", PrintData)
  register_callback(&"DataPrinted", data_was_printed)
  register_data_updated_callback(&"SynchronizedData", on_
    synchronized_data_updated)

func _ready_authority() -> void:
  # Do something when the authority is set
  pass

func PrintData() -> void:
  print(data.value)
  _composite_node.CallCallback(&"DataPrinted", [])

func data_was_printed(data: int) -> void:
  # Do something with the data after it was printed
  pass

func on_synchronized_data_updated(new_data: int) -> void:
  print(new_data)

```

Listing 1: Example implementation of a Composite Node Module in GDScript

is important to note that using the last two classes is optional. They are provided primarily as convenient helper classes. For example, a ***CompositeNodeModule*** automatically stores a reference to its parent ***CompositeNode*** and provides functions that can be overridden to run logic when the ***Node*** or its authority is initialised. A ***CompositeNode*** can have multiple modules as children, and it can also contain other ***CompositeNodes***. In the figure, the example ***CompositeNode*** has three modules as children. The first module registers callbacks and functions on the ***CompositeNode***, and it creates two ***CompositeNodeValues*** to store its data. The other two modules, however, do not register functions or callbacks, and they do not need ***CompositeNodeValues***. But they would be able to trigger the callback, call the function or access the data registered on the ***CompositeNode*** by ***CompositeNodeModule 1***. This demonstrates the flexibility of the system. Modules can be lightweight or fully featured depending on the needs of the project. A simple example of a ***CompositeNodeModule*** implementation in GDScript is shown in Listing 1. This illustrates the basic interactions with the ***CompositeNode*** system. ***CompositeNodeValues*** are created and stored on the module as variables within the **_ready_composite_node()** function. This function is called once the parent ***CompositeNode*** has finished its own initialisation. In the example, one value is synchronised, while the other is not. A synchronised value automatically propagates changes to all clients, which is particularly useful for multiplayer setups. In this case, an unsigned 32-bit integer is sent to every client whenever the value changes. Functions and callbacks are also registered in **_ready_composite_node()**. The function **PrintData()** is registered on the ***CompositeNode*** and can be called by any object that holds a reference to it. Chasing Carrots deliberately uses PascalCase for registered functions to distinguish them from normal functions written in snake_case. A function call looks like this: **_composite_node.CallFunction(&"PrintData", [ ])**. Callbacks work similarly. They trigger a provided function when called. In the example, calling **_composite_node.CallCallback(&"DataPrinted", [ ])** triggers **data_was_printed(data)**. Multiple ***Nodes*** can register for the same callback to attach different logic. It is also possible to register a callback for a specific data value that is triggered whenever the defined value changes. With **register_data_updated_callback()**, that kind of callback can be registered on the ***CompositeNode*** and is called whenever the data changes. This mechanism allows logic to automatically run on clients when a value is updated by the authority, streamlining multiplayer synchronisation. It is also worth noting that all these function calls use StringName instead of regular strings. StringName is a specialised Godot data type that treats two StringNames with the same content as the same object. This enables extremely fast, efficient comparisons, which is crucial when dealing with frequent callbacks and function calls [21].

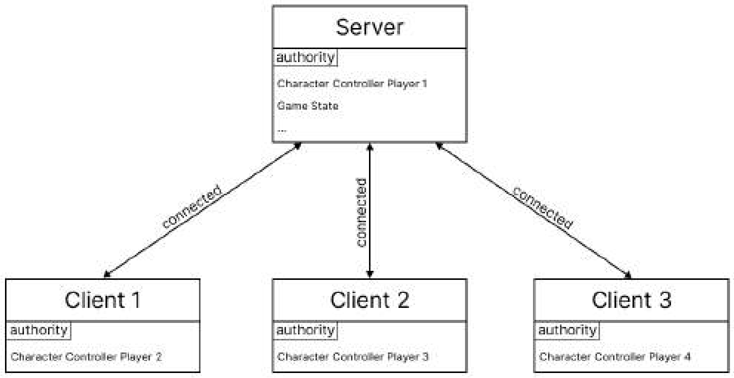

Server
authority
Character Controller Player 1
Game State
...
connected
connected
connected
Client 1
authority
Character Controller Player 2
Client 2
authority
Character Controller Player 3
Client 3
authority
Character Controller Player 4

Figure 2: Client to Server visualisation

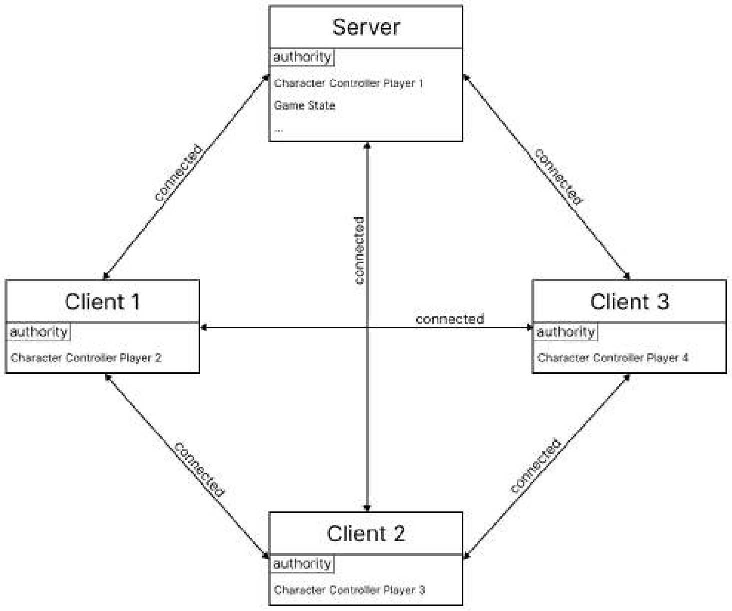


Figure 3: Network Mesh visualization

4.2 Current Multiplayer Architecture

Before this individual project began, the game already had a networking structure in place. The current setup follows a classic client-to-server model. Each client is connected to one specific client that takes the role of the server, as shown in Figure 2. The server is responsible for distributing all relevant data to the clients that need it. Another essential part of the networking model is the concept of authority. In the Godot ***MultiplayerAPI***, every ***Node*** in the

SceneTree has exactly one client assigned as its authority. The authority is usually the client that processes game logic or calculates new values for that node. A typical example is the character controller. Each client has authority over its own character controller, which means player input is processed locally. The results are then sent to the server, which forwards them to the other clients.

Chasing Carrots wants to keep the authority system, but they also need direct client-to-client communication. For this reason, the project aims to introduce a network mesh. As shown in Figure 3, the idea of a server and authorities remains the same, but all clients are also connected directly to one another. This allows a client to send requests directly to another client without routing them through the server. Direct communication reduces latency and can improve responsiveness, especially in situations where many small updates need to be exchanged.

4.3 Communication Line System

Inspired by the ***MultiplayerAPI*** used by Godot [16], Chasing Carrots developed its own system for transferring data between clients. The aim was to create a lightweight system which directly communicates with the ***MultiplayerPeer***. Before this project, the ***CommunicationLineSystem*** used the Godot ***MultiplayerAPI*** itself. This is why one of the goals for this project was to remove the functionality that relied on the ***MultiplayerAPI*** and directly access the ***MultiplayerPeer***. In this chapter, I will present the already modified version of the ***CommunicationLineSystem*** (CLS) with the modifications I applied, as this is the target architecture for this system.

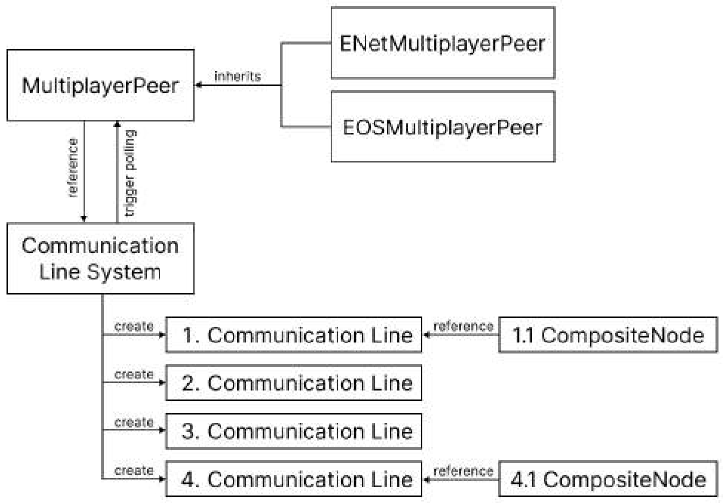


Figure 4: Communication Line System Architecture

Figure 4 shows, how the ***CommunicationLineSystem*** is structured. The diagram illustrates how the CLS interacts with the ***MultiplayerPeer*** and how it integrates with ***CompositeNodes***. The framework allows multiple ***CommunicationLineSystems*** to exist within a single application. Each CLS manages its own set of ***CommunicationLines***. In the game code, ***CommunicationLines*** can be created wherever needed and can be used immediately. Remote functions can be defined on these lines and will be executed on other clients when triggered. This approach is inspired by Godot's Remote Procedure Call (RPC) system [16]. ***CommunicationLines*** also support check bits that can filter remote calls. For example, a client can look up which peer carries the authority bit mask and send the remote call only to that peer. This makes it easy to route messages to specific clients with specific roles, such as the server. Every ***CompositeNode*** automatically holds a ***CommunicationLine*** as a member variable. This is necessary for registering synchronised data values, as described in Section 4.1, and for invoking functions on the authority. However, ***CommunicationLines*** are not restricted to ***Nodes***. They can be created anywhere in the codebase, even outside the ***SceneTree***. This makes them more flexible than Godot's built-in ***MultiplayerAPI***, which always requires a ***Node*** with access to the API instance. ***CommunicationLines*** can also be used in RefCounted objects [20], which opens the door for networking code that does not depend on scene structure.

5 Network Code Refactor

One of the first takeaways from this project was getting a look at a professional game development environment and seeing how an independent studio works with Godot on a daily basis. The second major part of the learning process was developing my own project inside the network code. This section explains how I moved away from the built-in Godot ***MultiplayerAPI*** toward a system where ***MultiplayerPeer*** ownership is handled entirely through the ***CommunicationLineSystem***. Shifting this responsibility away from the engine and into custom logic was an essential step because it enabled much greater control over how authority, routing, and synchronisation are managed. Once that part was working reliably, the next challenge was to implement complete network meshes using both ENet and EOS. Through this project, I learned a lot about multiplayer synchronisation and different network architectures, and gained deeper insight into ENet and Epic Online Services.

5.1 Networking implementation in Godot

Godot's multiplayer architecture is shown in Figure 5. At the centre of this system is the ***MultiplayerAPI***, which can be implemented or replaced as needed. The default implementation is ***SceneMultiplayer***, which inherits from the base

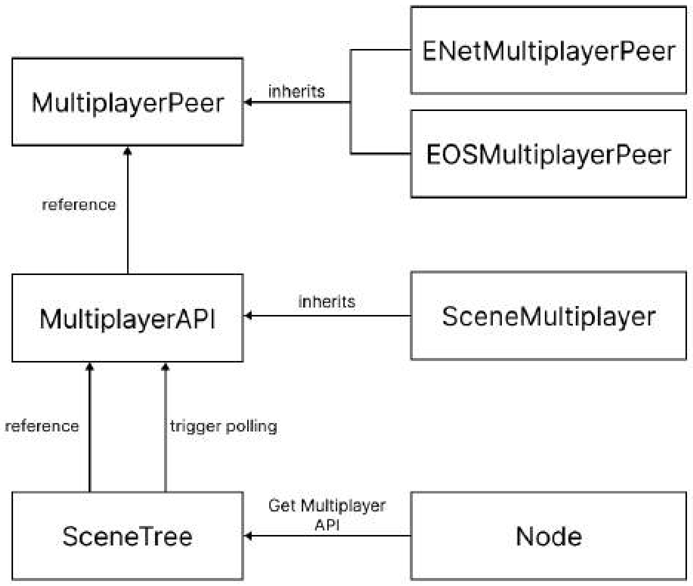


Figure 5: Godot networking architecture

MultiplayerAPI and provides Godot's built-in RPC system. Since developers are free to implement their own ***MultiplayerAPI***, the framework is quite flexible.

In our case, we decided not to build our own ***MultiplayerAPI***, because we did not want to use Godot's RPC layer at all. Instead, we created a separate system, which has already been touched upon in section 4.3 and will be discussed in more detail in the next chapter (section 5.2), that works alongside Godot's architecture without relying on it. Below the API layer sits the ***MultiplayerPeer*** class. This is the part that handles the actual transport layer, and it can be inherited from to integrate different networking technologies. Chasing Carrots uses two implementations: the default ***ENetMultiplayerPeer***, which Godot provides [6], and the ***EOSGMultiplayerPeer***, which is part of the open source extension Epic Online Services Godot (EOSG) [13]. Thanks to Godot's system design, swapping peers is straightforward and does not require changes to the game logic. For development, the ***ENetMultiplayerPeer*** is primarily used because it is lightweight and easy to test with. Once the game is released, the EOS peer will be the primary option for players, since it supports features such as platform services, matchmaking and cross-platform connectivity.

Accessing the multiplayer system inside the game code is simple. Every ***Node*** in Godot has access to the ***SceneTree***, and the ***SceneTree*** holds a reference to the currently active ***MultiplayerAPI***. This means that any ***Node*** can call networking functionality without needing global managers or additional infrastructure. It also allows developers to write and test networking logic directly in GDScript in the editor, keeping iteration times short and making it easy to experiment with new ideas. Godot integrates its networking features in the same spirit as the rest of the engine: simple to set up and quick to use. As a developer, you can get something running over the network within minutes when using the provided example code and documentation from Godot [10]. Writing synchronised gameplay logic directly in GDScript feels natural and convenient. However, the strength of this simplicity also becomes a limitation.

The built-in system does not scale well for more complex projects, as writing RPC calls in every script can get hard to maintain. It also relies heavily on Godot-specific concepts such as the ***MultiplayerSpawner*** and the ***MultiplayerSynchronizer***. These tools work well for small games, but they can feel restrictive as a project grows or when the architecture requires greater control over how data flows through the network. For Chasing Carrots, this made it clear that a custom networking layer would be a better long-term solution. Building their own framework allowed them to shape the networking model around the game rather than shaping the game around Godot's networking model. It also opened the door to cleaner modularity and custom authority rules. In short, the decision to move away from the built-in system was not because Godot's networking is bad, but because it is designed first for accessibility. For a project of this scale and complexity, Chasing Carrots needed something more specialised and flexible.

5.2 Transitioning to the Communication Line System

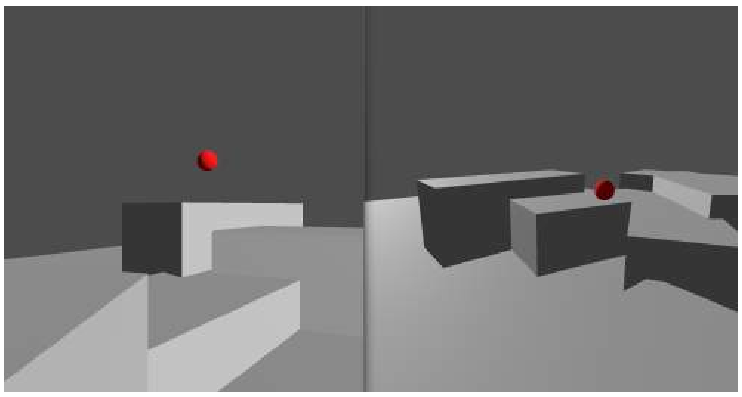


Figure 6: Test scene with red spheres as controllable objects

This section covers the first primary task of my project at Chasing Carrots. The goal was to move from the networking architecture shown in figure 5 to the new ***CommunicationLineSystem*** architecture shown in figure 4. My starting point was a customised version of Godot 4.5 that already included several engine modifications and a working networking setup, as described in section 4.2. The custom engine changes are explained in section 4. With this as a foundation, the first step was to build a small test project that mimicked the networking layer used in the current prototype of Chasing Carrots' upcoming title. The test project included a lobby system, transitions between different game states, and a set of debugging tools. To verify that data was still transferred correctly after swapping out the underlying ***MultiplayerPeer***, I created a dedicated testing scene in Godot (figure 6). The scene included a straightforward player controller, so I could immediately see whether the player's input was synchronised correctly. Each client can control one of the red spheres in the scene with six degrees of freedom. The screenshot shows two connected clients, but the setup supported up to four simultaneous players.

```
func _process(_delta: float) -> void:
  if not _composite_node.IsAuthority():
    return
  var gametime : float = GameTime.GameTime
  if gametime >= _next_update_gametime:
    while gametime >= _next_update_gametime:
      _next_update_gametime += 5.0

    _U8_value.value = randi_range(0, 255)
    _U16_value.value = randi_range(0, 65535)
    _U32_value.value = randi()
    _S8_value.value = randi_range(-128, 127)
    _S16_value.value = randi_range(-32768, 32767)
    _S32_value.value = randi_range(-2147483648, 2147483647)
    _HalfFloat_value.value = randf()
    _Float_value.value = randf()
    _Double_value.value = randf()
    _Vector2Type_value.value = Vector2(randf(), randf())
    _Vector3Type_value.value = Vector3(randf(), randf(), randf())

```

Listing 2: Script function, which periodically sends data to clients

The scene also contained a script that periodically sent values of every supported data type across the network. This allowed me to check whether any of them failed to synchronise. Listing 2 shows how this was implemented. The script extends a ***CompositeNodeModule***, which gives it access to its ***CompositeN-ode*** and, through that, the ***CommunicationLine***. Each assigned variable is a synchronised ***CompositeNodeValue***. The early return ensures that only the ***CompositeNodes*** authority updates the values. Once a value changes, the system triggers a remote update on all connected clients so they mirror the data.

Figure 7 shows the debugging window that lists all synchronised values for each client. This tool made it easy to spot mismatched data or missed updates. Before starting the actual refactor, I made sure the old system was fully understood, fully functional, and behaving consistently. Only then did I begin

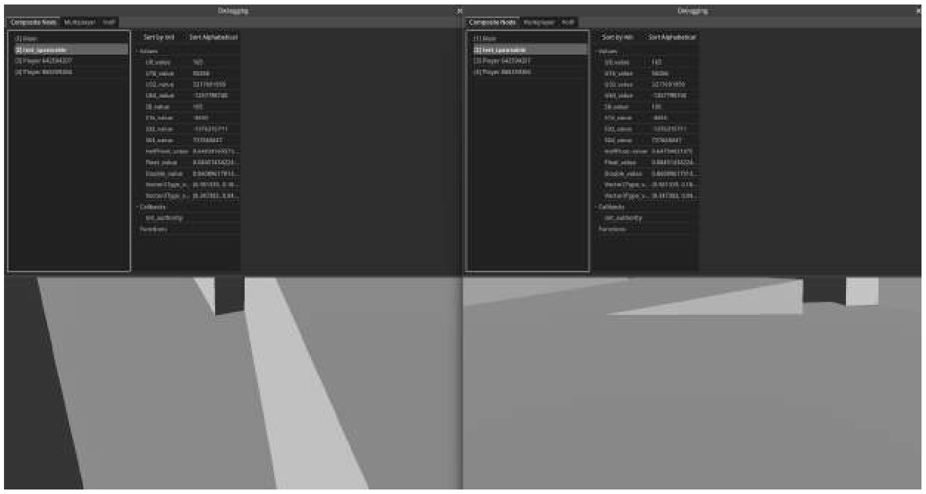


Figure 7: Debugging screen to see data synchronisation

modifying the networking layer itself.

The first changes had to be made inside the custom modules in the Godot source code. These parts are written in C++. The ***CommunicationLineSystem*** currently relies on the ***MultiplayerAPI*** to access the ***MultiplayerPeer*** and stores a reference to the API. To remove this dependency, I added initialisation functionality so the ***CommunicationLineSystem*** can store a direct reference to the ***MultiplayerPeer***. Normally, the ***MultiplayerAPI*** triggers the polling of network data, so the ***CommunicationLineSystem*** needed its own function to poll data. I reused the polling logic from Godot's ***SceneMultiplayer*** and removed the code that was not relevant. Because Chasing Carrots plans to use a multiplayer mesh topology, the relay server features were not needed and could be removed. I also removed parts of the code that were only required for Godot's built-in RPC system and the ***MultiplayerSynchronizer***, since we no longer use those features.

By default, the authority concept in Godot assigns a unique peer ID to a ***Node***. This value is synchronised across all clients, so each can compare the ***Nodes*** authority ID with its own. Our new framework keeps the idea of authority but adjusts how it is handled. The ***CommunicationLineSystem*** uses a bitmask to filter requests and direct them to the correct peers. Because this bitmask was already a core part of the system, we decided to use it to mark authority. I defined the first bit of the mask as the authority bit and updated the authority initialisation code to set this bit instead of assigning a peer ID.

In Godot, every script that extends the ***Node*** class can access the ***Multiplayer-API***. This is used to check authority status or to retrieve the peer ID. With our changes, the built-in helper functions, such as **is_multiplayer_authority()**, are no longer valid. To replace them, I exposed new backend functions from C++ to GDScript. These functions are accessible through the ***CommunicationLine*** or the ***CompositeNode***. The new authority check is called with **communication_line.is_authority()**. This triggers a check for the authority bit in the bitmask and returns a boolean. I also implemented helper functions to retrieve the unique ID of the peer with the authority bit set and a check if a peer is the server.

Once the refactor was complete, I moved back to the test project. I cleaned up the project and replaced every point where the old ***MultiplayerAPI*** was used. This included authority and server checks, removing all RPC calls and replacing them with remote calls over the ***CommunicationLine***, and updating the initialisation process so that the correct ***MultiplayerPeer*** instance is passed to the ***CommunicationLineSystem***. I removed all references to ***MultiplayerSpawner*** and ***MultiplayerSynchronizer*** and replaced them with logic built directly on top of the new system. After everything was updated and the project was running again, I verified that data synchronisation still worked. The periodically sent values from listing 2 were correctly synchronised, and all red spheres in the test scene moved consistently for every client when receiving player input.

6 Network Mesh Implementation

The implementation of the network meshes (figure 3) has several advantages compared to the traditional client-server structure shown in figure 2. The most important benefit is that clients can send packets directly to each other. This reduces latency because data no longer needs to be routed through a server first. Lower latency is especially useful for features like voice chat and the synchronisation of player inputs, where fast and responsive communication matters a lot. A downside of this approach is the increased complexity of setting up the mesh. ENet and EOS both support peer-to-peer communication, but they handle discovery and connection management in different ways. Because of this, each backend needs its own implementation.

There is also a security aspect that should be mentioned. When clients can send packets directly to other clients without a central authority validating the data, the risk of cheating increases. A peer-to-peer system makes it easier for a malicious user to manipulate the game state or send incorrect information. In competitive or large-scale online games, studios usually invest a lot of effort into server-side validation and anti-cheat systems to prevent this. In our case, the upcoming title is a cooperative game that is mainly played with friends or small groups. Because of this, the studio decided that adding complex anti-cheat measures would not be worth the additional development time and maintenance.

6.1 ENet Network Mesh

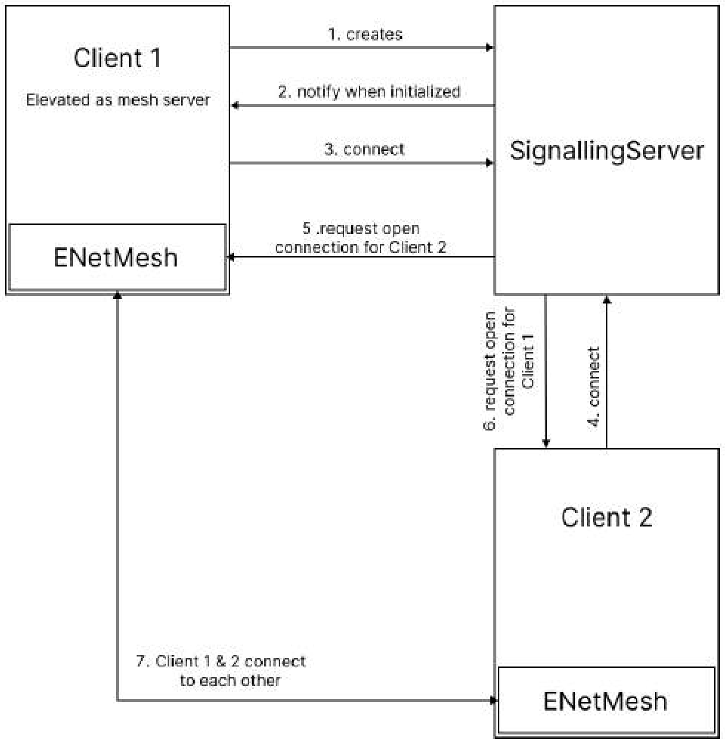

Client 1
Elevated as mesh server
ENetMesh
1. creates
2. notify when initialized
3. connect
5 .request open connection for Client 2
SignallingServer
6. request open connection for Client 1
4. connect
Client 2
ENetMesh
7. Client 1 & 2 connect to each other

Figure 8: ENet Mesh representation with numbered initialisation sequence

Documentation for implementing an ENet network mesh is very limited. The only valuable source I found was a blog post about networking changes introduced in Godot 4.0 [15]. The post included a simple example but relied on static IP addresses hard-coded into the project. This makes the example unsuitable for our use case, so several adjustments were required. The blog post also suggests creating a signalling server. This server is responsible for distributing the information that clients need in order to connect to one another. Since ENet itself does not provide any discovery features, the signalling server becomes the place where clients register, exchange their connection details and request mesh links. To establish a peer-to-peer connection, each client has to create its own ***ENetConnection*** instance [5]. The process works as follows:

1. Client 1 creates an ENet host.
2. Client 1 attempts to connect to an ENet host created on Client 2.
3. Client 2 creates its own host and binds it to the incoming connection request from Client 1.

Once both sides have a host object and both hosts acknowledge the connection, ENet establishes a peer-to-peer link between the two machines. A key requirement of this system is that each connection must use a dedicated port. The IP addresses of both clients and an open port for the connection must be known to the signalling server so it can communicate them to the peers. During development, I tested whether a single port could be shared among multiple ***ENetConnections*** within the same mesh. Debugging showed that this does not work because ENet reports an error when a second host tries to bind to a port that is already in use. This means that each connection in the mesh needs its own port on both clients. At this stage, the peer-to-peer link exists only on the ENet level. Godot's ***MultiplayerPeer*** is not aware of it yet. To solve this, I had to keep track of the ENet hosts that were created for each connection. I implemented this using a simple host buffer, which is shown in listing 3.

```
func _process(_delta: float) -> void:
  if multiplayer_peer:
    for host_id in host_buffer:
      var host : ENetConnection = host_buffer[host_id]
      var event = host.service()
      if event[0] == host.EVENT_CONNECT:
        # Add host peer
        multiplayer_peer.add_mesh_peer(int(host_id), host)

        host_buffer.erase(host_id)

```

Listing 3: ENetMesh host management

Each host in this buffer represents one active ENet link. For Godot to recognise the connection, each client needs to call **add_mesh_peer(peer_id, host)** on its own ***MultiplayerPeer***. This function takes the ENet host representing the connection and the peer ID assigned to the remote client. Because of this, the signalling server must provide the following information to each client:

- the IP address of the target client,
- the port that should be opened for the connection,
- the target peer's unique ID to use inside the multiplayer system.

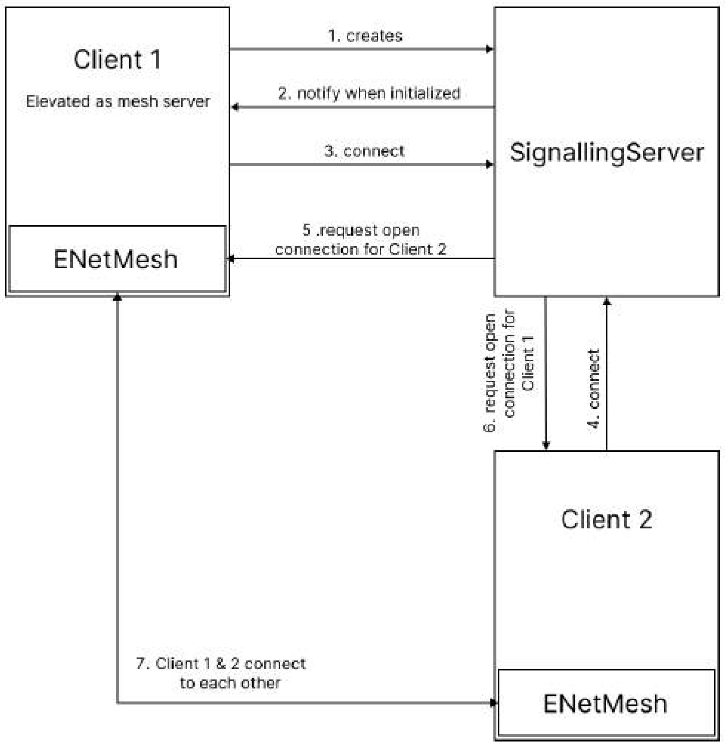
Once this data is available, the clients can create their ENet hosts, establish a direct connection, and finally register each other via the **add_mesh_peer(peer_id, host)** function call. This completes one link in the ENet network mesh.

Figure 8 shows the order in which an ENet mesh connection is established. I implemented the signalling server by reusing the existing ENet client-to-server structure that the studio already uses in the current networking architecture. When a player creates a lobby, their client also starts the signalling server. Once the signalling server has finished its initialisation, the client connects to it. Because this is the first client to join, it is promoted to become the mesh server. The mesh server has an important role, because it is responsible for handling global game state, coordinating transitions such as starting the game and managing any spawning logic. Even though the final mesh is peer-to-peer, the project still needs one server to coordinate these high-level actions. The signalling server generates a unique ID for every client that connects. It also stores the client's IP address so it can share this information with other peers later. When a second client joins, it does not connect directly to Client 1. Instead, it connects to the signalling server using the IP address of Client 1 and the defined port for the signalling server. Once Client 2 is connected, the signalling server sends both clients the information needed to create their ENet hosts. This includes the IP address of the other client and a port that should be opened for the connection. The signalling server keeps track of all used ports to prevent duplicates, because each ENet mesh link requires a dedicated port. After both clients create their hosts and listen on their assigned ports, they establish the actual ENet peer-to-peer connection. If a third client joins later, the process repeats. The signalling server sends the new client the data for all already connected clients. It also sends the existing clients the information for the new one. With this, every client receives the correct IP address, port and unique peer ID for each connection. All clients then create the missing ENet hosts and establish the required mesh connections until every peer is linked with every other peer. This process continues for all players who join the lobby, resulting in a fully connected ENet mesh topology.

To verify that the mesh setup worked correctly, I used my test scene to check whether all data was still synchronised and whether the behaviour matched the previous client-server structure. Since the server relay functionality was removed, the only way clients can communicate is by sending data directly to each other. Because of this, I wrote a small piece of test code that sends a message from one client directly to another. This can only succeed if the mesh was created correctly on both sides. During the test, I was able to trigger a remote call between two clients. This confirmed that the peer-to-peer links were active and that my implementation of the network mesh was functioning as intended (Appendix: A).

6.2 EOS Network Mesh

Once the ENet network mesh was working as intended, I moved on to implementing the same system for Epic Online Services (EOS). The setup for an EOS network mesh is shown in figure 9. EOS handles the process of establishing connections between peers on its own, so there is no need for a separate signalling server. Before a mesh connection can be set up, both clients must log in to EOS and be fully initialised. EOS supports several login methods. For internal testing, Chasing Carrots uses an anonymous login based on a device ID generated automatically by EOS. This works well when testing with colleagues

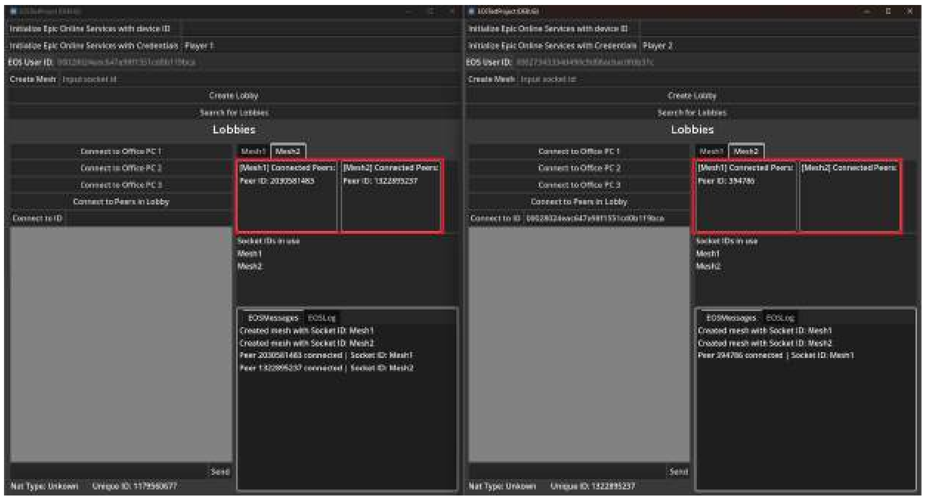


Client 1
Elevated as mesh server
ENetMesh
1. creates
2. notify when initialized
3. connect
5 .request open connection for Client 2
SignallingServer
6. request open connection for Client 1
4. connect
Client 2
ENetMesh
7. Client 1 & 2 connect to each other

Figure 9: EOS Mesh representation with numbered initialisation sequence

in the studio. When I needed to test alone on my machine, I had to use the authorisation tool provided by Epic Games [23]. The tool allows logging in with whitelisted Epic Accounts in the studio's EOS backend for development. Without this tool, I would not have been able to simulate two separate clients on one machine.

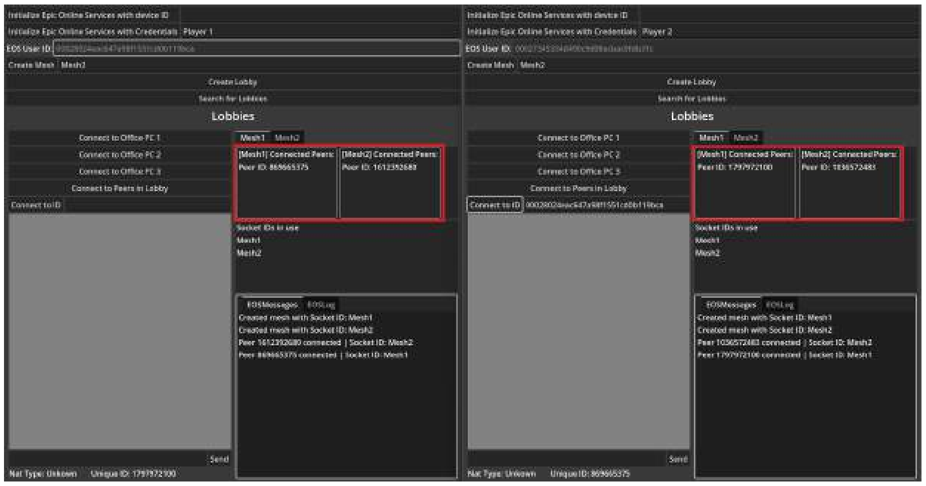
Once both clients are logged in and initialised, the mesh can be created. As before, Client 1 starts a lobby, which automatically designates them as the server in the mesh. EOS assigns a user ID to every logged-in client, which is used to establish the connection. With the user ID of Client 1, Client 2 can begin the connection process by calling **add_mesh_peer(remote_user_id)**. The user ID replaces the Peer ID and ***ENetConnection***, which were needed for the ENet Mesh in the section before. EOS distributes the request sent by Client 2 to Client 1. Client 1 then opens a connection for Client 2. The EOS Godot extension [13] was implemented in a way, that once a connection was opened in mesh mode, it would also instruct Client 1 to send its own connection request to Client 2. After both sides complete these steps, the mesh link between the clients is established, and they can start communicating (Appendix: B).



```
Player 1
[Mesh1] Connected Peers:   [Mesh2] Connected Peers:
Peer ID: 2030581483        Peer ID: 1322895237
Socket IDs in use
Mesh1
Mesh2
Created mesh with Socket ID: Mesh1
Created mesh with Socket ID: Mesh2
Peer 2030581483 connected | Socket ID: Mesh1
Peer 1322895237 connected | Socket ID: Mesh2
Nat Type: Unkown     Unique ID: 1179560677
```

```
Player 2
Connect to ID 00028024eac647a98f1551cd0b119bca
[Mesh1] Connected Peers:   [Mesh2] Connected Peers:
Peer ID: 394786
Socket IDs in use
Mesh1
Mesh2
Created mesh with Socket ID: Mesh1
Created mesh with Socket ID: Mesh2
Peer 394786 connected | Socket ID: Mesh1
Nat Type: Unkown     Unique ID: 1322895237
```

Figure 10: EOS testing interface, Mesh 2 failed to establish a connection



```
Player 1
Create Mesh  Mesh2
[Mesh1] Connected Peers:   [Mesh2] Connected Peers:
Peer ID: 869665375         Peer ID: 1612352680
Socket IDs in use
Mesh1
Mesh2
Created mesh with Socket ID: Mesh1
Created mesh with Socket ID: Mesh2
Peer 1612392680 connected | Socket ID: Mesh2
Peer 869665375 connected | Socket ID: Mesh1
Nat Type: Unkown     Unique ID: 1797972100
```

```
Player 2
Create Mesh  Mesh2
Connect to ID 00028024eac647a98f1551cd0b119bca
[Mesh1] Connected Peers:   [Mesh2] Connected Peers:
Peer ID: 1797972100        Peer ID: 1836572483
Socket IDs in use
Mesh1
Mesh2
Created mesh with Socket ID: Mesh1
Created mesh with Socket ID: Mesh2
Peer 1036572483 connected | Socket ID: Mesh2
Peer 1797972100 connected | Socket ID: Mesh1
Nat Type: Unkown     Unique ID: 869665375
```

Figure 11: EOS testing interface, Mesh 2 succeeded in establishing a connection

After I finished implementing the EOS network mesh, I verified it using the same test environment that I had already used for the ENet network mesh. Once I confirmed that everything behaved correctly, I asked the programming lead to review the changes. After getting approval, I began integrating the mesh into the main project on a separate branch. This step required a lot of work. It was not enough to add the new mesh logic. The entire codebase had to be updated to work with the new ***CommunicationLineSystem***. This meant removing all references to the multiplayer API and replacing every RPC call with a call through the new ***CommunicationLine***. Only after these changes were completed was I able to test the mesh inside the real project. At first, everything seemed to work. However, it quickly became clear that voice chat did not work with the EOS mesh. A look into the debug tools showed that no peers were connected to the voice chat system on the client. The project uses a proximity voice chat feature, and this system uses its own network package handling. Because of this, it must run its own mesh. This was already the case before my work. One peer-to-peer connection is used for gameplay data, and another one is used for the voice chat. For the game, this means at least two meshes will run in parallel. Even after extensive debugging, it was challenging to identify the cause of the problem. I was then instructed to isolate the issue in a smaller example project, which is shown in figure 10. This example only provided a simple UI that allowed me to create a mesh step by step. With one mesh, everything worked without any issues. The interesting behaviour appeared only when creating a second mesh. Figure 10 shows the peers connected in each mesh, highlighted in red. Client 1, which also acts as the server, correctly registered Client 2's peer in the second mesh. However, Client 2 never showed Client 1 as a peer. This meant the connection was not registered on the ***MultiplayerPeer*** of Client 2, and therefore, the second mesh was incomplete (Appendix: C).

I systematically debugged the code base to identify where the problem occurred. After confirming that the issue was not caused by the GDScript code used in my testing environment, I investigated the EOS Godot Extension itself [13]. While stepping through the extension, I found that Client 1 was correctly sending a connection request to Client 2, but the packet never arrived on the other side. The most likely explanation is that the EOS SDK, on which the extension is built, processes the packet internally and never hands it over to the extension. As a result, the ***MultiplayerPeer*** on Client 2 never receives the request and cannot register the connection, preventing communication between the peers. Since I could not determine the exact root cause, I implemented a workaround by modifying the extension in C++. Client 1 now sends several connection requests to Client 2 until it receives confirmation that the connection has been established or it runs out of reattempts. As shown in figure 11, the workaround functions as intended and the second mesh successfully registers the peer connection (Appendix: D). After implementing the workaround, I compiled the EOS Godot Extension to generate a new DLL and imported it into the main project. With these changes in place, the voice chat system started working again, and all peers were correctly registered in the debugging tools. This confirmed that the workaround was effective and that both meshes, including the one used for voice over IP, were fully operational.

7 Evaluation

The evaluation of this project focuses on functional correctness and practical performance implications of replacing Godot's built-in ***MultiplayerAPI*** with the ***CommunicationLineSystem*** and introducing full peer-to-peer network meshes using ENet and Epic Online Services (EOS). Since the work was carried out within an active production environment, the evaluation emphasises validation through testing and integration rather than synthetic benchmarks.

Functional correctness was validated using a dedicated test project that mirrored the networking setup of the studio's current prototype. The test scene allowed up to four clients to connect simultaneously, each controlling an individual player object with six degrees of freedom. Player movement, authority handling, and state synchronisation behaved consistently before and after the refactor. In addition, a ***CompositeNodeModule*** was used to periodically transmit values of all supported synchronised data types across the network. A debugging interface displayed the received values for each client, and no desynchronisation or missed updates were observed during testing. This confirms that the ***CommunicationLineSystem*** successfully replaced Godot's RPC-based synchronisation for the tested scenarios.

Authority handling was preserved through the introduction of a bitmask-based system integrated into the ***CommunicationLineSystem***. State updates and input processing were restricted to the designated authority peer, while remote calls were routed only to peers matching the relevant bitmask. Newly exposed helper functions allowed GDScript code to perform authority checks without relying on Godot's built-in multiplayer helpers. In practice, this approach proved sufficient and did not introduce ambiguity or inconsistent behaviour during testing.

The network mesh implementations were verified for both ENet and EOS. For ENet, a signalling server distributed IP addresses, ports, and peer ID's, allowing each client to establish direct peer-to-peer connections with all others. Direct remote calls between non-server clients succeeded, confirming that communication no longer depended on server relaying. For EOS, mesh connections were established using EOS without the need for a signalling server. A limitation was discovered when running multiple meshes in parallel, which prevented peer registration in secondary meshes. This issue was resolved through a workaround implemented in the EOS Godot extension, after which both gameplay and voice chat meshes functioned correctly.

While no quantitative latency or bandwidth measurements were conducted, the architectural implications are clear. In the previous client–server model, client-to-client communication required an additional hop through the server. The mesh-based approach reduces this to a single hop, lowering latency and reducing server load. Formal performance benchmarking under adverse network conditions is left for future work.

Integrating the new system into the main project required replacing all remaining uses of Godot's ***MultiplayerAPI***, including RPC calls, authority checks and cases where the ***MultiplayerSpawner*** or ***MultiplayerSynchronizer*** was used. Although this migration required significant effort, the resulting codebase is more modular and less dependent on Godot's scene structure. The ability to use ***CommunicationLines*** outside the ***SceneTree*** improves code reuse and architectural flexibility.

Overall, the evaluation shows that the project achieved its primary goals. The ***CommunicationLineSystem*** fully replaced Godot's built-in multiplayer framework for the tested use cases, enabled direct peer-to-peer communication through network meshes, and integrated successfully with both ENet and EOS into the main project. While the solution introduces additional complexity, it provides the flexibility required for the studio's cooperative multiplayer game.

Ultimately, the game was tested during a studio play session attended by the whole team. The team successfully connected and completed a full playthrough. While in previous playtests the multiplayer connection seemed unstable and led to disconnects of single clients, the refactored version increased stability and ran better than in previous tests. Unfortunately, the stability is still not good enough. In later playtests, the team experienced further connection issues. The cause of these problems doesn't necessarily need to be the newly introduced changes. However, they can be, so further debugging and stress testing are required in order to create a multiplayer experience as stable as possible.

8 Conclusion

This individual project, carried out in collaboration with Chasing Carrots, has been an immensely valuable and insightful learning experience. I am proud to present the results in this report and to have contributed meaningfully to the studio's ongoing development efforts. For me, this project was an exceptional opportunity to dive deeper into the world of network code for multiplayer games. I explored various network topologies, including classic client-server architectures and full-mesh networks, and implemented both approaches myself. Working closely with the Godot Engine's source code as well as with internal studio code significantly strengthened my ability to read, understand, and work within large-scale codebases written by others.

Throughout the project, I also learned to handle frustration and remain persistent—especially during long debugging sessions that sometimes lasted days before revealing the actual underlying issue. A particularly memorable part of the work involved the Godot EOS Extension. It showed me how open-source projects, while powerful, can sometimes be prone to unexpected bugs. The workaround I implemented ultimately solved the issue, though it was more of a temporary, "hacky" fix than the clean solution I would aim for in the long term. If time permits, I intend to create a detailed problem description and proper documentation of the fix, and share them with the extension's main author to collaborate on a more robust solution.

While the new game at Chasing Carrots is still far from finished, I am genuinely pleased to have contributed to its development. Future steps will include stress-testing the new networking framework to evaluate how it handles large amounts of synchronised data and how it behaves under heavy latency or packet-loss conditions. Overall, I am very grateful for the opportunity to collaborate with Chasing Carrots and gain deeper insight into the world of indie game development. I participated meaningfully in the project, and I am proud that my work can be helpful to the studio moving forward.

Bibliography

[1] *Chasing Carrots Godot Fork on GitHub*. [Online]. Available: https://github.com/ChasingCarrots/godot.

[2] *Custom modules in C++ - Godot Documentation*. [Online]. Available: https://docs.godotengine.org/en/stable/engine_details/architecture/custom_modules_in_cpp.html.

[3] W. Eddy, "Transmission Control Protocol (TCP)," Internet Engineering Task Force, Request for Comments RFC 9293, Aug. 2022, Num Pages: 98. DOI: 10.17487/RFC9293. [Online]. Available: https://datatracker.ietf.org/doc/rfc9293.

[4] *ENet: Reliable UDP networking library*. [Online]. Available: http://enet.bespin.org/.

[5] *ENetConnection*. [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_enetconnection.html.

[6] *ENetMultiplayerPeer - Godot Documentation*. [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_enetmultiplayerpeer.html.

[7] *Epic Online Services (EOS) Overview - Epic Online Services Developer*. [Online]. Available: https://dev.epicgames.com/docs/epic-online-services/eos-overview.

[8] M. Farokhmanesh, "Unity May Never Win Back the Developers It Lost in Its Fee Debacle," *Wired*, Sep. 2023, Section: tags, ISSN: 1059-1028. [Online]. Available: https://www.wired.com/story/unity-walks-back-policies-lost-trust/.

[9] *Godot Engine - Free and open source 2D and 3D game engine*. [Online]. Available: https://godotengine.org/.

[10] *High-level multiplayer - Godot Documentation*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html.

[11] L. Juan, *Godot Engine reaches 1.0, first stable release*, Dec. 2014. [Online]. Available: https://godotengine.org/article/godot-engine-reaches-1-0/.

[12] *License Godot Engine*, en. [Online]. Available: https://godotengine.org/license/index.html.

[13] D. Lourenco, *EOSG - Epic Online Services for Godot*, Sep. 2025. [Online]. Available: https://github.com/3ddelano/epic-online-services-godot.

[14] *Making plugins - Godot Documentation*. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/plugins/editor/making_plugins.html.

[15] *Multiplayer in Godot 4.0: ENet wrappers, WebRTC.* [Online]. Available: https://godotengine.org/article/multiplayer-changes-godot-4-0-report-3/.

[16] *MultiplayerAPI - Godot Documentation.* [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_multiplayerapi.html.

[17] *MultiplayerPeer - Godot Documentation.* [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_multiplayerpeer.html.

[18] *MultiplayerSpawner - Godot Documentation.* [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_multiplayerspawner.html.

[19] *MultiplayerSynchronizer - Godot Documentation.* [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_multiplayersynchronizer.html.

[20] *RefCounted - Godot Documentation.* [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_refcounted.html.

[21] *StringName - Godot Documentation.* [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_stringname.html.

[22] *The GDExtension system - Godot Documentation.* [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/index.html.

[23] *Tool: Developer Authentication - Epic Online Services Developer.* [Online]. Available: https://dev.epicgames.com/docs/epic-account-services/developer-authentication-tool.

[24] "User Datagram Protocol," Internet Engineering Task Force, Request for Comments RFC 768, Aug. 1980, Num Pages: 3. DOI: 10.17487/RFC0768. [Online]. Available: https://datatracker.ietf.org/doc/rfc768.

Ludography

[25] Chasing Carrots, *Pressure*, Mar. 2013.

[26] Chasing Carrots, *Cosmonautica*, Jul. 2015.

[27] Chasing Carrots, *Pressure Overdrive*, Jul. 2017.

[28] Chasing Carrots and Erabit, *Halls of Torment*, Sep. 2024.

[29] Chasing Carrots and The Irregular Corporation, *Good Company*, Mar. 2020.

[30] Kinetic Games, *Phasmophobia*, Sep. 2020.

[31] poncle, *Vampire Survivors*, Oct. 2022.

[32] Wouter van Oortmerssen, *Cube*, Sep. 2001.

[33] Zeekerss, *Lethal Company*, Oct. 2023.

A Appendix: ENet Showcase

Video material of the created test project, showcasing a functional ENet network mesh with the refactored network code: "AppendixA_ENetShowcase4.mp4"

B Appendix: EOS Showcase

Video material of the created test project, showcasing a functional EOS network mesh with the refactored network code: "AppendixB_EOSShowcase4.mp4"

C Appendix: EOS Failed Connection

Video material of the EOS test project showing the second network mesh will fail to establish a connection: "AppendixC_EOSConnectionFailed.mp4"

D Appendix: EOS Successful Connection

Video material of the EOS test project showing the second network mesh successfully established a connection with the implemented workaround in the EOSG Extension: "AppendixD_EOSConnectionSuccessful.mp4"