DTU Compute
Department of Applied Mathematics and Computer Science

Scalability and Democratization of Real Time Strategy and Fighting Games: The Deterministic Lockstep Netcode Model with Client Prediction Combined with the Data Oriented Tech Stack.

Master Thesis

Michał Chrobot

Abstract

This master thesis investigates the process of creating deterministic games, highlighting their benefits, particularly for real-time strategy and fighting games. In addition to providing a basic netcode infrastructure, it focuses on detecting nondeterminism in games and methods for debugging it.

The aim is to develop a package that utilises a deterministic lockstep netcode model, a multiplayer synchronisation technique used for deterministic games, built into the Unity game engine on top of their Data-Oriented Technology Stack (DOTS). The package primarily focuses on detecting and debugging nondeterminism, using a sample Pong game implemented for this thesis as an example, and compares the created tools to existing solutions while exploring alternatives.

The research begins by highlighting key multiplayer concepts and examining existing solutions for determinism validation and debugging, particularly their limitations. It then details the implementation of a sample Pong game in DOTS, explaining how this framework works and the functionalities of the game. Next, a package providing a deterministic lockstep netcode model within Unity DOTS is created, demonstrating how the game can utilize it. This package can be used with different games made in DOTS, and the process of using it is explained in detail. Finally, the thesis discusses how determinism validation is performed and how debugging tools are implemented and used to identify sources of nondeterminism. It underlines the benefits of a data-oriented design (DOD) approach and the entity component system (ECS) pattern in identifying the causes of nondeterminism in simulations.

A key innovation of this thesis is the showcase of how to detect nondeterminism in DOTS and the development of per-system validation methods that enable more precise detection and ease debugging of nondeterministic behaviour in ECS-based frameworks. The results shall indicate that the per-system validation approach can reduce the scope of code to be tested, facilitating more precise identification of nondeterministic sources, although it is computationally expensive and more work would need to be done in order to use it with big RTS games.

Future work should focus on enhancing the netcode model with additional features for latency mitigation, as well as improving the validation tools performance for larger-scale games. The findings of this thesis contribute to the broader goal of democratising game development by providing smaller teams and independent developers with the means needed to create deterministic multiplayer games.

Contents

1 Introduction

The gaming industry that stands behind the creation of multiplayer online games is constantly pushing the boundaries of what's achievable with technology, aiming for scenarios where all participating players experience the same almost lag-free gameplay despite differing hardware, software, and/or network conditions. One way to achieve this is through deterministic gameplay, which allows for the implementation of effective lag mitigation techniques.

A simulation is considered deterministic if, given the same initial conditions and the same sequence of inputs, the simulation will always produce the same outputs, regardless of how many times the full simulation is run.

Deterministic simulation is employed in many different fields. One such field is simulating a real-world object in a virtual environment [1] (for example, the behaviour of a car on the road), and another is gaming.

Games that are deterministic come with many benefits. One of them is reducing the amount of data that needs to be sent over the internet by primarily transmitting player inputs rather than the entire game state. This in turn allows for the implementation of different proven techniques that are significantly reducing perceived lag [2] [3].

This is particularly beneficial in games with hundreds of units, as it significantly reduces the data transmission load, saves bandwidth, minimises the potential for delays, and means that every player sees the exact same simulation. Most larger multiplayer RTS games with a significant number of controllable units (like StarCraft 2 or Warcraft) are deterministic, and this is an industry standard for this type of game.

Creating fully deterministic code is harder than it sounds. Game engines (i.e., hobbyist and commercial software tooling licensed or used to develop video games) are rarely deterministic themselves. Additionally, there are differences in how operations are executed on various hardware or operating systems, like Windows or Mac. Those often come from variations in floating-point precision, processor architectures, compiler optimisations, and system-specific libraries. These factors can lead to subtle discrepancies in calculations on different platforms. On top of that, even if a game engine is deterministic, it's very easy to write non-deterministic code, especially as the vast majority of languages do not treat determinism as correctness and therefore do not offer tooling (via the compiler, for example) to prevent this kind of error. Moreover, the unpredictability of many operations in multithreaded contexts, which all modern games are expected to heavily utilise, further complicates the goal. A single bug in the code or any of the behaviours mentioned may introduce nondeterminism, leading to the same game implementation that runs on different machines to arrive at different game states for the same initial conditions and input sequence. This is why determinism validation is so critical if the game does, in fact, rely on determinism.

Because almost all game engines are not 100% deterministic and do not provide tools for determinism validation, game studios often need to rely on different solutions. The first is to create a deterministic game engine of their own, which requires a lot of time and resources. In addition, this game engine will probably be used only for the purpose of a single game or simulation made with it. The second is to use an existing deterministic engine solution, such as Photon Quantum [4]. While Photon Quantum is objectively a

good solution and provides general determinism validation tools, these tools could be improved to help developers find the cause of non-determinism more precisely.

Correct implementation of some of the aforementioned lag-mitigating solutions or determinism validation itself requires additional computational resources from the user's hardware. Some modern game engines support a programming approach called Data-Oriented Design (DOD) [5]. DOD focuses on structuring data in memory to optimise access patterns and improve cache efficiency [6]. This approach can reduce CPU load and speed up the computation process, making it well-suited for creating large-scale deterministic RTS games by enhancing players' hardware's computational capabilities. Additionally, the Entity Component System (ECS) pattern, often used in combination with DOD, separates code logic from data. This separation allows for more precise identification and narrowing down of nondeterministic behavior. Because systems typically represent small logical parts of the program, they can be validated individually, making it easier to pinpoint which system is causing nondeterminism in a simulation. In contrast, in object-oriented programs, code logic and data are often intertwined, making it more challenging to isolate the exact source of nondeterminism since entire objects and their interactions must be considered as a whole. This can complicate the debugging process and obscure the specific cause of nondeterministic behavior. Currently, Unity, the most widely used game engine [7] [8], offers the most advanced support for this technology through its Data-Oriented Tech Stack (DOTS).

If existing knowledge about determinism, its validation, and debugging were consolidated and more tools and general methodologies were provided, it would ease the creation of deterministic games. Additionally, offering a synchronisation method for online multiplayer games with these tools in the form of a package would further promote this way of game creation. Simplifying the creation of deterministic games will enable smaller teams with fewer resources to develop deterministic multiplayer games more easily, thereby democratising their usage.

Even if a game engine is not fully deterministic, game developers would benefit from the ability to test the potential game with built-in tools to detect non-deterministic behaviour. Developers could either create deterministic games using that engine or, if non-deterministic behaviour comes directly from the engine, find a solution for it or apply the same methodology with a different engine better suited to the game's needs. From another perspective, game engine developers could use such a tool to validate whether the functionalities of their game engines are indeed deterministic. Deterministic validation tools would facilitate the detection and resolution of non-deterministic behaviour, allowing developers to save time on debugging and testing their solutions.

There are many potential sources of non-deterministic behavior. Therefore, these tools will focus on identifying the parts of the engine or game that cause non-determinism and pinpointing the source as closely as possible, which should be human-readable, helping developers make the necessary changes in the code.

In conclusion this master thesis aims to assist game developers in creating fully deterministic gameplay. Specifically, it seeks to provide a package containing determinism validation and debugging tools with methodologies for Unity's Data-Oriented Technology Stack (DOTS) with the most common technique for network synchronisation used for deterministic games, known as the "Deterministic Lockstep Netcode Model." The application and usage of this package is demonstrated through the development and debugging of a showcase game. At the end, this thesis will provide a showcase example of how to debug nondeterminism cases using the implemented package, demonstrated through the case

of the implemented game.

For further clarification, this thesis initially aimed to focus on implementing the deterministic lockstep netcode model with lag mitigation techniques such as client prediction and rollback within DOTS. However, during the investigation, it became evident that determinism validation is an extensive and often underexplored topic. There are existing examples of deterministic lockstep implementation without any determinism validation tools. Proper implementation of these lag mitigation techniques would be ineffective without first ensuring the game's determinism through the development of validation tools.

Therefore, this thesis has shifted its focus towards implementing a basic deterministic lockstep netcode model in DOTS as the foundation for the game while investigating various techniques for detecting nondeterministic behaviour in online multiplayer games and methods for debugging it. The goal of democratisation is achieved by explaining the concepts that allow for deterministic simulation creation and providing tools that developers can use or build upon, enabling broader access to this technique. Scalability is highlighted by the use of a data-oriented approach in the form of DOTS, which supports faster computation processes than object-oriented programming and is recommended for large-scale games with many simultaneous actions.

The rest of the thesis is structured in the following order:
The **Concepts and Background** chapter provides a summary and explanation of common terms and concepts used throughout the thesis. These terms focus on the core concepts used in online multiplayer games and data-oriented design.
The **Goal of the Thesis** chapter discusses existing solutions, includes research questions, and outlines the primary objectives of this investigation.
The **Running Example** chapter presents a game sample in the form of a modified Pong game used to demonstrate the implemented package and serves as a reference for the further implementation process. This sample is used to illustrate various implementation steps later in the thesis.
The **Sample Pong game implementation** chapter demonstrates the implementation steps for the sample Pong game and provides deeper insights into how the Unity game engine works and core DOTS concepts.
The **Deterministic Lockstep Netcode Model** chapter goes deeper into the theory of the model, its implementation within the DOTS environment, and how the game is utilising it.
The **Determinism Validation and Debugging Tools** chapter finally provides an in-depth discussion on determinism, nondeterministic behaviours in games, their sources, and the tools developed for determinism validation and debugging. It also explains how these tools can be used to locate the source of nondeterminism in the code.
The **Evaluation** chapter presents the results and evaluates the effectiveness of the framework, determinism validation, and debugging tools. It identifies key findings, areas for improvement, and the current limitations of these tools.
The **Discussion** chapter interprets the findings, discussing their significance and pointing out what work could be done in the future in this area.
At the end, the **Conclusion** chapter answers the primary research question, summarising the thesis and concluding the investigation.

2 Concepts and background

Before going into the various aspects of the implementation process, it is crucial to grasp the basic terminology commonly used in the field of games, determinism and deterministic lockstep. This chapter will outline the terminology, concepts, and methodologies that will be extensively used in subsequent chapters, while not going into specific details but rather focusing on presenting the overall idea behind several concepts.

2.1 Netcode

When considering multiplayer games, the primary challenge is connecting multiple players on different devices and allowing them to play together. This requires a mechanism to keep all players synchronised, enabling them to see an accurate and consistent shared game state that is influenced by their inputs.

The game state includes various elements such as player actions, movement of units, scores, resources, and game events like shooting projectiles. These events can originate from player inputs or from the game itself, such as actions by non-player characters (NPCs). Synchronising these events and conditions across all devices is crucial. Without proper synchronisation, objects might appear in different locations for different players, leading to confusing or unfair gameplay.

Achieving this synchronisation across different devices, which are usually running at different speeds, while dealing with the physical limitations of networks like latency or limited bandwidth, all fall under the term “netcode.” By saying ”netcode model,” someone can refer to a specific implementation of all concepts related to handling interactions between remote clients.

2.2 Lockstep and deterministic lockstep

One netcode model involving the synchronisation of player actions is called the lockstep netcode model. Lockstep refers to a synchronisation method used in multiplayer online gaming where each client processes the same game ”simulation step” simultaneously. In its basic form, lockstep involves clients sending their data (such as position, actions, etc.) to a server. The server collects this information, waits for the data from all clients for the given step, processes it, and sends back to each client the combined data about all players’ actions for that simulation step. Each client waits to receive this information before advancing to the next simulation step, ensuring that all clients stay in sync. This method is called ”lockstep” because the game progresses for all clients at the same pace once all clients have sent their data for the current step.

Deterministic lockstep improves upon this by sending only player inputs rather than full game state updates, reducing data transmission size. This approach requires all clients to reproduce the same game state based on these inputs, enforcing the game to be fully deterministic. An example of this is shown in the next chapter in Figure 6.3.

Deterministic lockstep itself will not mitigate any latency. It will rather introduce it by forcing clients to wait, but by allowing them to send only inputs and process the simulation based on them, it creates a base for the implementation of lag-mitigating techniques. Deterministic lockstep also changes how the server operates. Instead of simulating the game, the server is solely responsible for gathering and sending back player inputs, which allows for greater scalability of games using it [9].

One technique, known as 'forced input latency', imposes a slight delay before the effects of user inputs are displayed. This delay is generally slight, and players get used to it during the game. This helps to smooth out common variations in connection speeds.

One such technique is 'forced input latency', which imposes a small, consistent delay before users see the effects of their inputs on the screen. This delay helps smooth out common variations in connection latency, making these delays less noticeable and more predictable for players.

Another one would be player prediction. Player prediction allows for the game to continue even without receiving a remote player's inputs for a short period of time using predicted inputs during temporary lag spikes. When the actual input is received, the game checks if the previous prediction was accurate and adjusts the game state accordingly. This approach is preferable to freezing the game during lag, as it usually provides a smoother and more continuous gameplay experience. There is much more to how those inputs are actually predicted, but the point is that this prediction and rollback mechanism are feasible due to the simplicity of the network data involved, which consists solely of player inputs.

All of those lag mitigating techniques combined and built on top of the 'Deterministic Lockstep Netcode Model' are commonly referred to as 'GGPO' (Good Game Paced Out). GGPO means both the concept and the first original implementation of Deterministic Lockstep [10] with those additional techniques to mitigate network latency.

2.3 Determinism and deterministic game

Determinism is not a concept that applies only to games, it applies broadly to various fields in computer science. However, it is easier to explain determinism in the context of games, where visual traces of nondeterminism are more apparent.

Nondeterminism can be defined as the characteristic of a process, system, or computation where the outcome is not entirely predictable from the initial conditions and inputs. This means that the same initial state and input can lead to different outcomes on different executions.

One very important thing is that "kind of deterministic" simulation code doesn't exist. A single bug, race condition, uninitialized variable, or anything else can break the entire determinism. Either a simulation is deterministic or it isn't. This is why validation is so critical if the game does, in fact, rely on determinism.

Deterministic games use a deterministic lockstep netcode model to exchange inputs and simulate the same next step of the game as other clients. To ensure and validate that these simulations are deterministic, the game state hash is computed and sent along with the input to the server. The server then validates if every player sent the exact same hash before sending back the combined inputs to the clients, allowing it to detect if nondeterminism occurred.

For simplicity, let's for now assume the perfect case in which it is possible to efficiently hash the entire game state, and clients send this calculation to the server for verification. Section 7 will later explain how games handle this process, why hashing the entire game state efficiently is not possible, and what is done instead. This section will also show how to hash the game state efficiently and reliably to ensure that nondeterminism is detectable only when it truly happens. For now, let's just keep in mind that by comparing the hash values from different clients, one can detect if their simulations are proceeding in a deterministic manner.

2.4 Players connection

To utilise the lockstep synchronisation method, the way how clients connect to each other and which of those methods is good for deterministic games and lockstep needs to be considered.

The way computers are connecting with each other is called ”network topology,” and it describes a specific arrangement of nodes (which can be visualised as clients) that interact. Figures 2.1, 2.2, and 2.3 showcase the most common network topologies.

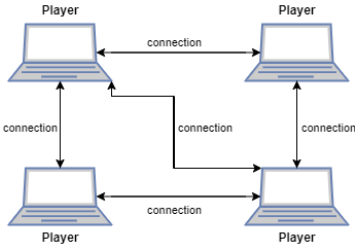

Figure 2.1: Example of P2P topology

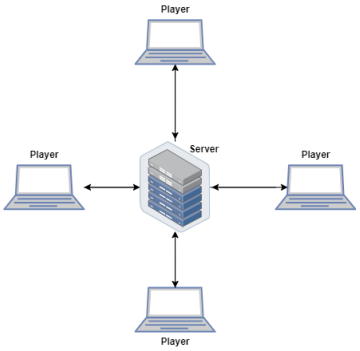

Figure 2.2: Example of client-server topology

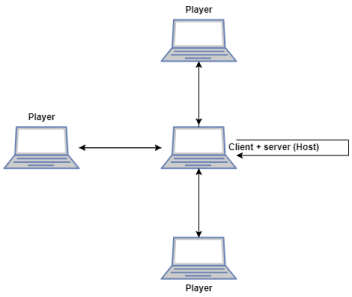

Figure 2.3: Example of client-server with authoritative client-host topology

In P2P netcode topology, each client connects to every other client in order to exchange game states and events, as shown in Figure 2.1. In that implementation, no single player can be defined as a game host, and instead, each client is in charge of managing their own game while accepting updates from other clients.

In a server-client topology, all clients connect to a separate game server instead of directly connecting to each other, as shown in Figure 2.2. This dedicated game server (DGS) runs the game and acts as the authoritative source for game state information. It receives updates from connected clients, integrates them into its game state, and then distributes the updated state to each client. Most major online games, such as Counter-Strike or League of Legends, operate under this model, with game studios running the servers [11].

While the deterministic lockstep model can use this network topology because it requires a server instance to gather and send back inputs, running a dedicated server is typically the domain of large, professional game companies. This thesis aims to showcase the implementation of deterministic lockstep, determinism validation, and debugging in simpler games, making it preferable to avoid the need for a dedicated server. Additionally, using a dedicated server presents maintainability issues, and it is more convenient for potential users of the package to test determinism on a local machine without having to set up a dedicated server first.

The preferred approach is to use one of the client’s machines as the server, as depicted in Figure 2.3. This method avoids the cost of a dedicated game server while still providing authority over the game state. The selected client becomes a client-host, simultaneously acting as the server for the game session and a client participating in it.

2.5 Desynchronization and Authority

When clients are connected, the term 'synchronization' refers to the process of changing or correcting their game state in order for it to be the same on every machine for a given point in the simulation time. Contrary to that, the term 'desynchronization' refers to the process in which, due to any reason, the game states on different machines start to differ from each other. The single instance of that behaviour is commonly referred to as 'desync'.

The terms "nondeterminism" and "desync" are often used interchangeably in deterministic simulations, but there is a difference between them. "Desync" refers to the state beginning to differ between simulations, while "nondeterminism" means that the state started to differ specifically because the simulation, given the same starting point and inputs, produced a different state. Therefore, every nondeterministic behaviour results in a desync, but not every desync is caused by nondeterminism, depending on the type of game.

As an example of a desync, one could consider a first-person shooter (FPS) game involving Player 1 and Player 2. In this scenario, let's imagine Player 1 sees Player 2 and shoots him with perfect timing according to his own screen. This will trigger an event that is sent simultaneously to Player2 (or the server) while also being processed on Player1's computer, confirming the hit and displaying Player2 as dead. However, due to signal delay, by the time the shot was processed on Player2's end, Player2 was no longer in the firing line, and, therefore, the shot missed. In this situation, Player1 perceives Player2 as eliminated while, according to Player2's, he remains alive, as in Figure 2.4. At this point, the game states of those clients are not syncronized anymore, and the that desyncronization occurred.

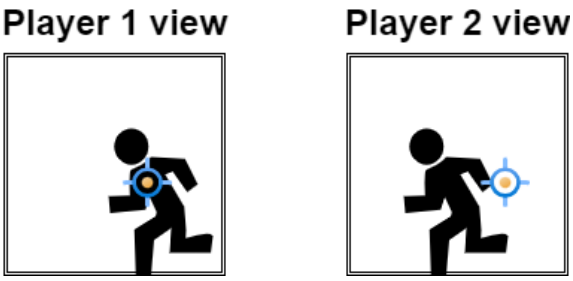

Figure 2.4: Player one sees player two as eliminated, but the state on player two's machine is different.

The only way for the game to continue would be to consider one of the players to have the "correct" version of events. This is called "authority." When a server or client is considered authoritative, the updates or events it sends to other clients are considered to be the correct sequence of events. Even if it conflicts with existing game states, the other client must update their game state to line up with the authoritative data. This is why more important game events, like scoring points or shooting another player, should not be visually executed until the game authority confirms that they really happened.

In the previous example of Figure 2.4, if Player1 is authoritative over Player2, then regardless of what Player2's machine calculates, the hit would be confirmed. Therefore, authority prevents desynchronization and allows both players to resynchronize. However, while authority may prevent desynchronization, it does not address issues caused by latency, which can manifest as seemingly perfect shots missing or apparently bad shots hitting.

Additionally, in deterministic games, the server is not responsible for processing the game state to provide authority over what really happened in the game. Instead, it collects and redistributes inputs, only being able to detect desynchronization without being able to determine which game version is correct. Since the server is not processing the simulation, it cannot ascertain which client processed the game incorrectly or what the current game state is.

2.6 Latency and Lag

Latency corresponds to the time it takes for a signal to go from point A (the source of the signal) to point B (the signal destination). When dealing with Netcode, one can never assume that a signal can get from one place to another instantaneously. A large part of netcode exists only to deal with this fact.

Network connection latency is primarily caused by several factors. One of them is the limit of the speed of light, which in itself wouldn't cause practically any delay, and it just means that data can't travel instantaneously from place to place. Light also travels slower in optic fibre glass or Ethernet cables. With that, various drawbacks of the way our technology works need to be taken into consideration. First of all, the way from point A to point B while using cables is never a straight line and can be unexpectedly prolonged by various factors. During this journey, the data needs to be read by network equipment at numerous points to determine its onward path, which may not always be the fastest one. It may also require multiple round trips if any part of this data is lost or corrupted. For example, if one of the routers (a piece of software that determines the next destination of the data) is overloaded with requests, it can simply drop our data. All of those factors can contribute to the to the additional time it takes for the data to arrive.

In terms of lockstep, each simulation step (also referred to as a tick) requires the client to wait for data to be received by the server along with all other clients' data. This process can take as much time as the delay experienced by the slowest client, as the simulation needs to gather all the data first. Then the data needs to be sent back, and only then can the game proceed. During this 'waiting time' the game seems to be frozen in place. If this delay is sufficiently long and thus noticeable, it is referred to as 'lag'. Again, it its pure implementation, detersministic lockstep netcode model does nothing to prevent the appearance of such lag.

From another aspect, every game has a pace of how many frames per second (FPS) it should simulate. The higher the value, the more smooth the game seems. Currently, all fast-paced games aim to enforce a rate of around 60 FPS, which is deemed responsive enough for the player. This translates to roughly 16.66667 milliseconds, during which the client should complete all calculations and send its data for the current game step to the server. If this process takes longer, it may also be a source of latency. Because of that, several more approaches were developed, which allow the simulation to run faster.

2.7 CPU power and Data Oriented Design (DOD)

Games, particularly real-time strategy (RTS) games, can consume a significant amount of processing power (referred to as CPU power). This is due to the high number of entities and interactions in the game world that need to be processed. That demand is critical because the traditional object-oriented programming approach tends to be inefficient, primarily due to its scattered memory access patterns. These patterns result in a substantial portion of the simulation step time being spent waiting for data to be accessed.

Data-oriented design is an umbrella term for a large collection of techniques and the condi-

tions under which those techniques should be used. In general, the goal is to analyse the access patterns of the different kinds of data in an application and select data structures that, for those access patterns, optimally leverage the underlying hardware. Hardware optimisations are often related (but not limited) to optimising usage of the CPU cache, limiting loading and storing to RAM (cache misses).

Object-oriented programming is about creating a model of the world in the programme and then solving problems by querying and modifying the model. Data-oriented design is about looking for what actually needs to happen in the real world and then structuring the programme around that. DOD is already a proven to work approach for resource intensive games as RTS or fighting games [12].

An example of DOD improvement could be in cache handling. In normal hardware, when a request for a certain piece of data is created, a whole chunk of data next to it is loaded into the cache. This is because cache is extremely fast to access compared to accessing the main memory [13]. In an object-oriented approach, the specific data a user wants to query, such as information about bullet objects, is scattered in memory. This scattering results in many memory requests to modify or read information about the same type of object, which does not utilise the cache efficiently.

In data-oriented design, objects that have the same type of data on them (are of the same archetype) are stored next to each other in memory. In the case of bullets, a data-oriented approach allows for requesting data from memory once and then utilising cache memory to the fullest. This approach focuses on updating or modifying one type of data at a time. When all bullet data is fetched, the data pulled into the cache is used efficiently, enabling the CPU to utilise its processing power more effectively instead of waiting for data to arrive from the main memory.

A good overview of Data-Oriented Design (DOD) and the differences between DOD and Object-Oriented Programming (OOP) is provided in the Unity tutorial [14], with the comparison illustrated in Figure 2.5 and visually represented in Figure 2.6.

In Object-Oriented Programming (OOP), once the player starts moving green balls around, the code iterates over an array of sphere objects to check the colour of each one and set the position of the green ones. Although the array is packed with contiguous data, it only contains references to Sphere objects, and the actual data can be scattered throughout memory, resulting in cache misses. In Data-Oriented Design (DOD), spheres would be decomposed into colour and position components and then packed into contiguous buffers which are easy to iterate on, resulting in fewer cache misses and much faster processing.

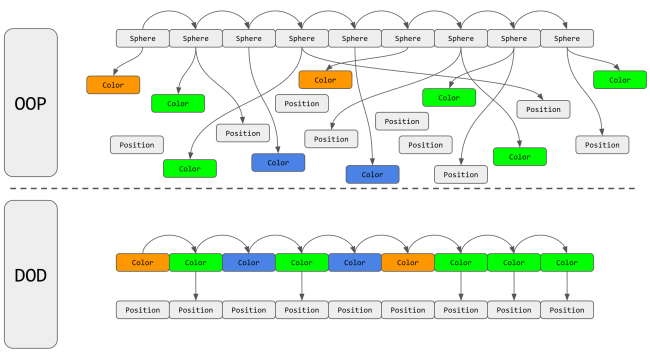

Figure 2.5: Comparison of data structure between OOP and DOD taken from Unity tutorial

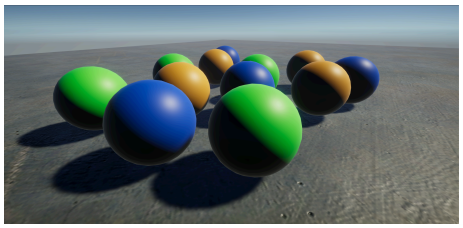

Figure 2.6: Visual representation of data used for comparison between DOD and OOP taken from Unity tutorial

In the context of games, reducing delays in data access can lead to significantly improved performance and smoother gameplay, especially when numerous data access actions and calculations need to be executed simultaneously. Consequently, DOD has been incorporated into several widely used game engines as an option. This shift represents a major change in how code and data are conceptualised and requires a different programming approach from developers.

2.8 ECS - Entity, Component, System

The Entity-Component-System (ECS) pattern is a software architectural pattern that is particularly well-suited to the principles of data-oriented design (DoD). It is important to note that while it is possible to write code that uses DoD principles without employing ECS, and vice versa, the ECS pattern naturally aligns with DoD optimisations.

An **entity** is a general-purpose object, typically represented by a unique identifier. Entities themselves do not contain any data or behaviour. **Components** are data containers that hold the informations about state of an entity. Each component represents a single aspect of the entity, such as its position, velocity, or health. Entity is thus used to group components together to represent certain element of the game (for example a player). **Systems** contain the logic that operates on entities with specific components. They change the state of the game by performing operations on the entities' components.

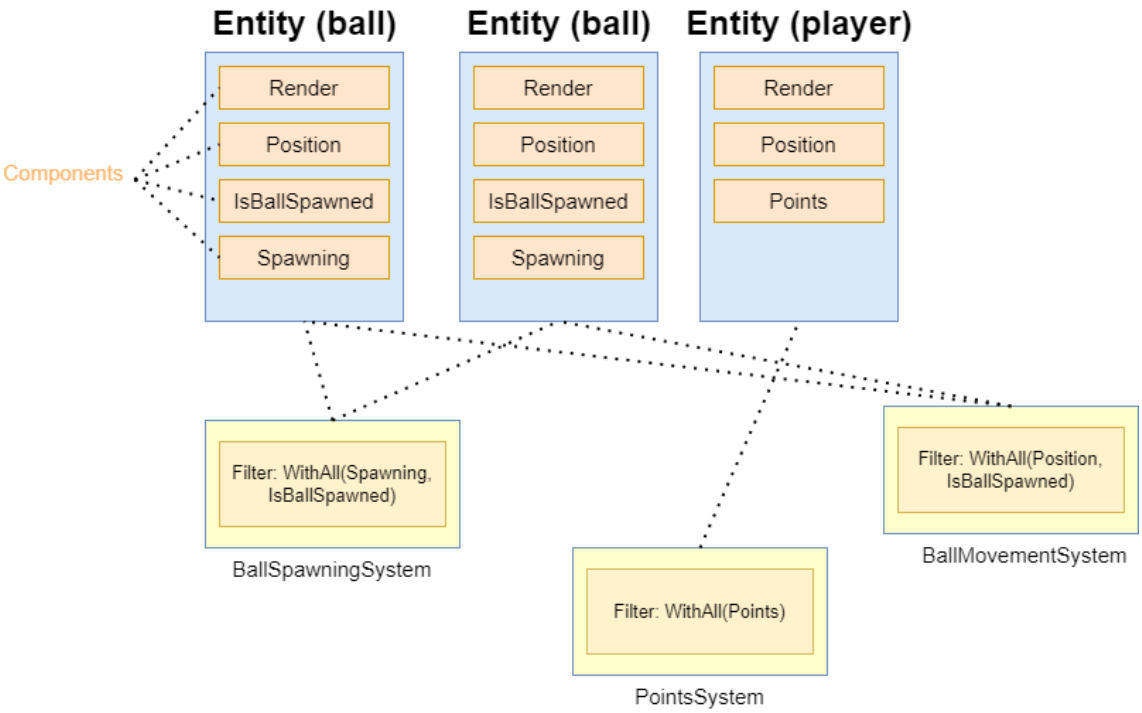

Figure 2.7: Example of ECS design

The components that constitute an entity are grouped into archetypes, meaning entities with the same types of components. These entities' components are stored next to each other in memory, allowing ECS to make better use of the CPU cache, reducing cache misses, and improving performance. For example, when changing the colors of balls, iterating through the same archetype maximizes cache memory usage by reducing the time spent fetching data from memory, leading to faster and more efficient processing.

Entities within the same archetype share the same set of components, even if their values differ. This contiguous memory arrangement enables systems to operate on entities of a single archetype at a time, improving performance through optimized cache usage.

The ECS pattern lends itself well to DOD due to its separation of data and logic. It cleanly separates data from behaviour, making it easier to manage and reason about each system and for what behaviour it's responsible.

In practice, many ECS frameworks adopt a storage design that allows applications to leverage the optimisations enabled by DOD. For example, Unity's DOTS framework uses an ECS pattern that iterates over contiguous arrays of components, thereby maximising the benefits of data locality.

This is why ECS and DOD are often incorrectly used interchangeably. However, they are almost always used together, especially in the context of games and game engines.

An example of ECS design is shown in Figure 2.7 where components are grouped together by entities, and systems query specific components to modify them.

2.9 Game engines and Unity

In order to implement or use either DOD or any tools and frameworks for the game development process, the game developers can choose between two options. Either they can create all of the necessary tools themselves from scratch to, in essence, build software that would facilitate the creation of the game (called a game engine), or they can already use one of the existing game engines on the market. Those game engines represent years of development and hundreds of different functionalities aiming to help developers create games faster and easier.

Game engines like, for example, Unity, Unreal Engine, or Godot are usually built with a modular architecture, where each module provides functionalities of different types for the game, like, for example, netcode, physics, or AI. Game engines help developers save time and resources by providing a foundation that handles the core elements of a game, allowing developers to focus more on the unique functionalities.

Unity, in this context, is one of the better-known game engines in the industry [7]. According to several sources, like Wikipedia [15], apps made with Unity game engine account for 50 percent of all mobile games. It is renowned for its flexibility, ease of use, and extensive feature set, which includes a strongly developed Data-Oriented Design (DOD) approach.

2.10 Unity Data Oriented Technology Stack (DOTS)

The DOD approach in the Unity game engine is represented in the form of Data Oriented Technology Stack (DOTS), which is a combination of technologies and packages that delivers a data-oriented design approach to building games. DOTS incorporates several essential packages, including, among others, ECS and the job system.

DOTS, which was for the first time announced in 2018 and production ready for release on May 25, 2023, represents a shift towards a data-oriented design in Unity, leveraging technology to ensure high-speed execution and efficient use of memory and resources. This in turn allows for the creation of much bigger, more complex, and performance-intensive games. Examples of those include games like 'diplomacy is not an option' which features hundreds of units [16] or Unity's own sample 'Megacity Metro' which allows for concurrent online play for up to 128 players online [17].

3 Goal of the thesis

This chapter aims to provide an overview of existing work in the field of deterministic games, determinism validation and debugging, highlighting the areas this thesis aims to address and contribute to. It defines the specific goals to be achieved, the questions to be answered, and the hypotheses to be tested.

3.1 Research Focus

The primary focus of this thesis, beyond investigating how to implement a deterministic lockstep netcode model in DOTS as a package, which serves as a base for the creation of online multiplayer deterministic games and has been implemented by various developers [18], is to explore existing approaches in the area of determinism validation and debugging, and investigate how to build such tools in DOTS, incorporating them into the package.

The package is developed in Unity DOTS, researching how validation and debugging can be performed in Data-Oriented Design (DOD) and Entity Component System (ECS) based software. It examines the benefits of working with ECS in the validation process and explores how to build these tools in software that is not itself 100% deterministic.

3.2 Problem Statement

Creating a deterministic online multiplayer game which allows for implementation of proven lag-mitigating techniques [3] is challenging and currently not many solutions exist which help with its creation. Especially in terms of determinism validation and debugging which is essential for ensuring that all players experience the same simulated game state, despite differences in hardware, software, and network conditions. If the game states were to frequently diverge in production, the resulting player dissatisfaction would be enormous, likely harming the game, business, and associated brands. Players have a right to have a product work as advertised, and therefore to return faulty games.

Determinism is essential because if clients will desync, this breaks the entire gameplay experience of the user. It's not possible to recover after a desync because game in this case is not synchronising state. It's the core point of the entire architecture, so game don't have full determinism, its not playable

The topic of determinism validation and debugging is often overlooked, with some existing solutions providing necessary netcode infrastructure without providing tooling to detect desyncs [19] and making sure that the simulation needs to be acceptably deterministic (i.e. encountering an acceptably low frequency of desync bugs in testing, and in the wild). For such infrastructure to function effectively, it is essential to ensure that the game simulation is deterministic. This is the main focus of this thesis. Currently, developers have limited solutions and would benefit from an alternative that helps to debug issues related to nondeterminism, as well as a foundational framework for deterministic lockstep to build upon.

3.3 Objective

The primary objective of this thesis is to develop a package for Unity DOTS that includes determinism validation and debugging tools along with an implementation of a deterministic lockstep netcode model. These tools are designed to aid in the creation of deter-

ministic multiplayer online games based on a deterministic lockstep model, providing a foundation that can be later improved upon. The package will offer essential debugging tools to identify and resolve nondeterminism issues, easing the implementation and validation of deterministic networking features in games. On top of that this methodology will help with verifying determinism in functionalities of even a game engine that is not fully deterministic which in turn will help the engine developers to locate the issues and game developers to see if an engine is suitable for their determinism needs.

The usage of this package will be demonstrated with a sample game and can be downloaded from the GitHub page [20].

3.4 Hypothesis

The central hypothesis of this research is that, in addition to implementing a deterministic lockstep netcode model in DOTS, effective determinism validation and debugging are possible within a not fully deterministic DOD/ECS framework like DOTS. The developed determinism validation tools will demonstrate that using ECS can result in more accurate detection of the sources of nondeterminism by pinpointing, in addition to the usual game frame and field, the exact system causing the nondeterministic behavior.

The hypothesis will be evaluated by assessing the accuracy and effectiveness of the package's determinism validation and debugging tools in detecting and resolving nondeterminism issues in code that aims to be deterministic.

3.5 Research Questions

To validate the hypothesis, the research will focus on several key questions:

1. How can the Deterministic Lockstep Netcode Model be effectively incorporated within Unity's DOTS?
2. How nondeterministic behaviour can be efficiently detected in DOTS?
3. How can the source of nondeterministic behaviour in DOTS-based games be identified?
4. What are the most common sources of non-deterministic behaviour?

3.6 Research Goals and Methods

The goals and methodologies implemented within this master thesis and outlined below aim to address the research questions previously identified, guiding the exploration into the integration of determinism validation tools and the deterministic lockstep netcode model within Unity's DOTS package which may be used by Unity projects to build deterministic multiplayer online games.

- **Goal 1:** Design and implement integration of the Deterministic Lockstep Netcode Model within Unity's DOTS framework.
- **Goal 2:** Create a sample game that will showcase the usage of the lockstep model and be the basis on which determinism validation will be tested.
- **Goal 3:** Create and describe a methodology for detecting nondeterministic behaviour within the DOTS environment, and explain how developers can use it to detect sources of nondeterminism and debug them.

- **Goal 4:** Offer insights, recommendations, and best practices for developers aiming to implement deterministic games within modern game development frameworks, particularly focusing on Unity's DOTS.

To accomplish the outlined research goals, the following methodologies will be employed:

- **Review of existing solutions:** Conduct a review of existing literature and software related to the deterministic lockstep netcode model, Unity's DOTS, and determinism validation and debugging.
- **Case Studies:** Investigate existing examples of determinism validation tool integration within game engines. The aim is be to identify common patterns, challenges encountered, and best practices.
- **Prototyping:** Develop a prototype game that integrates the package containing determinism validation and debugging tools and the deterministic lockstep netcode model within Unity's DOTS framework.
- **Evaluation:** Evaluate the effectiveness of determinism validation and how to locate the source of nondeterminism.
- **Documentation:** Create a package README document, as well as an overall description of the investigation process, critical findings, and insights obtained throughout the study.

By utilising these research methods, this study aims to accomplish its intended goals, provide valuable insights and recommendations, and make the creation of multiplayer online games more accessible for developers.

3.7 Prior work and existing knowledge

Numerous studies highlight the impact of different game development techniques on player reception and gameplay. Evidence suggests that games employing deterministic lockstep with built-in lag mitigation techniques see substantial improvements in player experience and immersion [21]. This is primarily due to better handling of network latency, which is a major challenge for developers of multiplayer online games [22]. However, implementing forced input latency, player prediction, or other lag mitigation techniques on top of deterministic lockstep requires deterministic code. To use this netcode model, a set of determinism validation tools is necessary to properly identify and address any nondeterministic behaviour in the game.

The concepts of deterministic lockstep and GGPO are well-established [10], [19], with many open-source implementations available in GitHub repositories claiming to provide deterministic functionalities. However, most of these repositories are outdated, not designed for DOTS, or lack support for validating determinism [23], [18], [24]. None of these repositories offer a clear and practical method for verifying determinism in games. This can be mostly due to the many factors that can introduce non-deterministic behaviour and the fact that it's not so complicated to tell that a desync has occurred, but it's difficult to do it efficiently and point to where in the code nondeterminism originates. In addition to those repositories, there are several written sources about deterministic prototyping in DOTS [25], deterministic networking in general [26] [27], and lockstep concepts in general [28] which can be utlised as a learning resource.

When it comes to creating a deterministic game and validating it with existing public game engines, most are not 100% deterministic and do not provide built-in tools for determinism

validation and debugging. Factors contributing to the fact that this is not supported out of the box include the non-deterministic nature of many engine functionalities and the significant development effort required to change this. Additionally, the complexity and scale needed to support a wide range of features and genres, the limited demand for deterministic behavior, and the availability of third-party tools like Photon Quantum [4] also play a role. Although detailed information about the exact state of determinism in DOTS is scarce, features like deterministic float mode [29] and the deterministic nature of ECS as stated on the Unity webpage [30] suggest potential for future support for determinism in Unity and by this also other game engines.

Developers aiming to implement GGPO-style deterministic gameplay currently often resort to creating custom deterministic game engines tailored to their specific game requirements, such as the Snowplay engine used for the ”Stormgate” game [31]. This approach, also seen in games like Warcraft 3 and StarCraft 2, yields the best results but requires significant resources and time, making it feasible primarily for large studios planning to create massive games that will offset the costs.

An example of such a private deterministic game engine is the one used by Riot for League of Legends.

3.7.1 Riot approach

Since this is a private engine, its code and implementation details are not publicly accessible, making it necessary to rely solely on available public information for insights. Riot's blog about League of Legends provides an excellent example of how they support determinism in their League of Legends game [32] and how they implemented determinism validation tools [33]. While the specifics of what happens from a player's perspective when a nondeterminism is detected remain unclear due to limited public information, the blog offers a valuable overview of the determinism validation implementation and debugging of such cases.

Riot, first, emphasises the importance of ensuring that their game behaves deterministically to support their netcode model, which differs from lockstep in the sense that their servers are also simulating the game state in order to control the correct state of the game and prevent cheating. Recognising that making 100% of their game engine and server deterministic was not feasible, they focused on the validation of the gameplay-essential state.

A common technique for evaluating game determinism is to hash the game state and compare the results between all connected players after some period of game update frames. To support later debugging, Riot chose to log the game state hashed changes during the game in readable form as a set of hierarchical key-value pairs in a file. This approach allows them to compare the outputs of these logs and use them to see what exact variable caused the desync.

The drawback of this method, as they are stating, is that log file sizes can be huge, and recording to and reading from these logs has severe performance implications. The logs are captured by them at the end of every frame, though the sampling rate can be decreased or logging can be limited to specific segments of frames to save space. Additionally, they log the state of game variables only when their values change, which helps manage file sizes.

Riot uses two levels of detail for game state logging. **Basic logging** is fast enough for real-time recording, determines if the game state is sufficiently similar by considering only the most important subset of the game state (for example, players, moving objects, etc.),

and results in file sizes of tens of megabytes. This option won't guarantee that the game is 100% deterministic, but it will guarantee that it is as deterministic as necessary for fair gameplay. The other option is **Detailed logging**, which captures more data, is too slow for real-time recording but is detailed enough to allow to find the first exact variable that diverged, and results in file sizes of tens of gigabytes.

To process these logs, Riot developed a script that compares the game logs and returns a sufficient ”diff-log,” highlighting the details of the first known desync. As discussed in their blog, only the first divergence really matters, since the game state will quickly diverge into chaos after a single discrepancy.

In addition to saving game state recordings from each game, Riot also maintains server network recordings. These recordings contain all the data exchanged between the server and clients, allowing them to replay the game on Riot's machines to replicate and analyse the same nondeterministic behaviour.

Their workflow involves the following steps:

1. Record a server network recording and basic game state log on one of the Riot environments for a real-world game.
2. On a separate PC, play back the server network recording and create a new basic game state log.
3. Compare the two basic game-state logs to see if the issue can be replicated.
4. If the basic game state logs are different, re-run the playback twice using detailed logging, comparing the detailed logs to each other.
5. Investigate the desync, starting with the variables that diverged.

Additionally, they built a test automation process that runs on dedicated hardware configured to closely resemble the actual servers used for League's game instances. This automated process operates on a large set of games, replaying beta-phase games where users expect occasional errors. This setup helps catch divergence-causing bugs before they go live or reach e-sports environments.

This approach presents really good and advanced software with many hints and tips for implementation.

3.7.2 Photon Quantum approach

In contrast to pre-made game engines, there exists one publicly accessible deterministic game engine known as ”Photon Quantum”. Photon Quantum brands itself as the only 100% deterministic multiplayer game engine on the market [4] that can be used by other developers. It offers more practical approach for smaller studios or individual developers then implementing deterministic engine from scratch. Photon Quantum integrates with Unity, managing code execution while Unity handles presentation, input gathering, and platform support. It incorporates its own variant of the ECS, job system, GGPO implementation, and general multiplayer-focused functionalities. Many games requiring determinism use this solution, including the popular ”Stumble Guys” game.

It's important to note that this investigation knowledge is based on Quantum 2, as Quantum 3 is only available to paid users. Therefore, while the new version may contain improvements and new functionalities related to determinism validation, focus of the thesis will be on the features available in Quantum 2.

Photon Quantum's approach involves performing game state hashes once per frame and identifying the frame where nondeterminism occurred. If nondeterminism is detected, the game is stopped permanently, and logs of the first frame where nondeterminism was detected are shown on the screen. Similar to Riot's approach, these logs provide a readable game state, helping developers pinpoint the exact frame and component/variable where the nondeterminism originated, narrowing down the search for the source of nondeterministic behavior.

For local validation, developers can control the hashing interval value, which determines how often game state hashes are sent to the server for verification. Quantum recommends using this feature during development but setting it to zero for release, meaning no verification. This is because, currently, when a divergence is detected, the log is displayed and the game stops without the possibility of further play which would drastically disturb the game.

While this approach could lead to the possibility of divergence in the released game, Quantum's engine is deterministic, and the assumption is probably that any nondeterminism results from user code and will be eliminated during testing. This is not a perfect solution but is probably considered sufficient given other priorities.

Additionally, there are various log options that specify the level of detail in the logs, similar to Riot's approach, in case the default settings are insufficient. The basic log shows which component caused the desync, but it is possible to generate more advanced logs that include component values, allowing developers to see precise variable values instead of just hashes or component names. These options provide more detailed information to pinpoint the differences, as illustrated in Figures 3.1 and 3.2.

When a divergence is detected, the logs show which field caused the desync, the frame it occurred on, and data related to all object values and initial files that may affect the simulation. The debugging options also allow for manual hashes checks, enabling custom implementations.

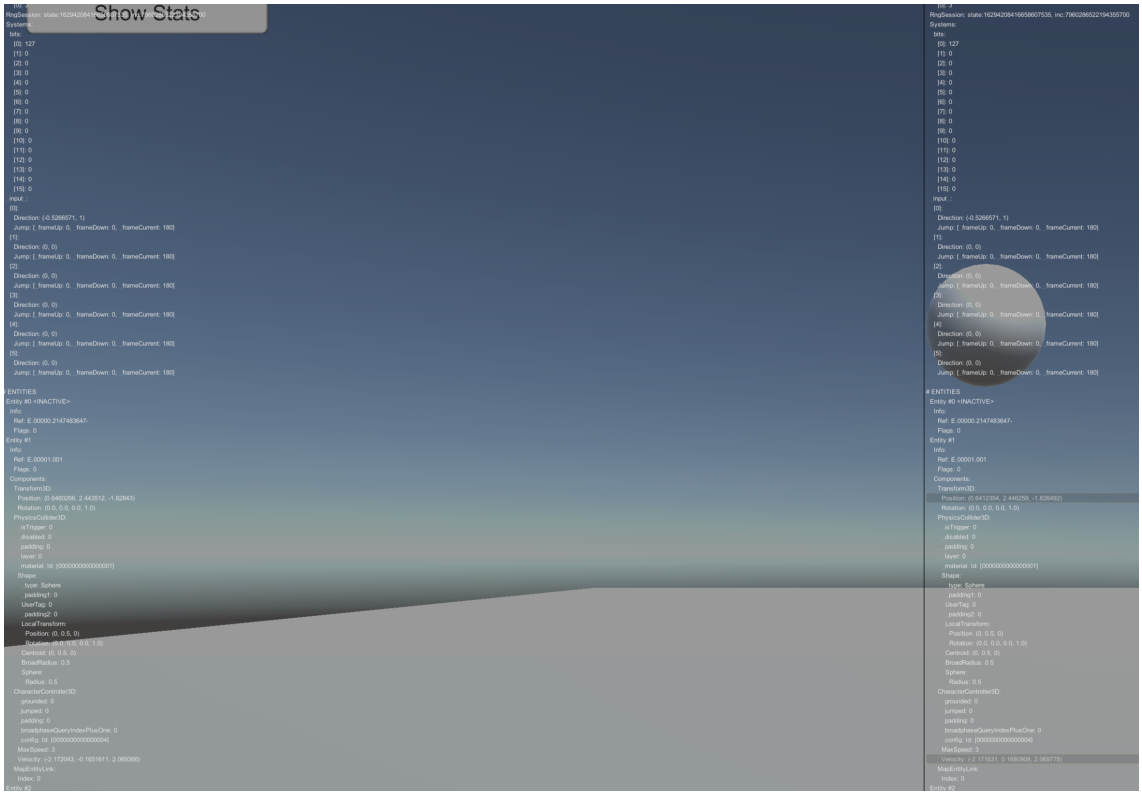

Figure 3.1: Photon Quantum log shown when divergence was detected with basic log options

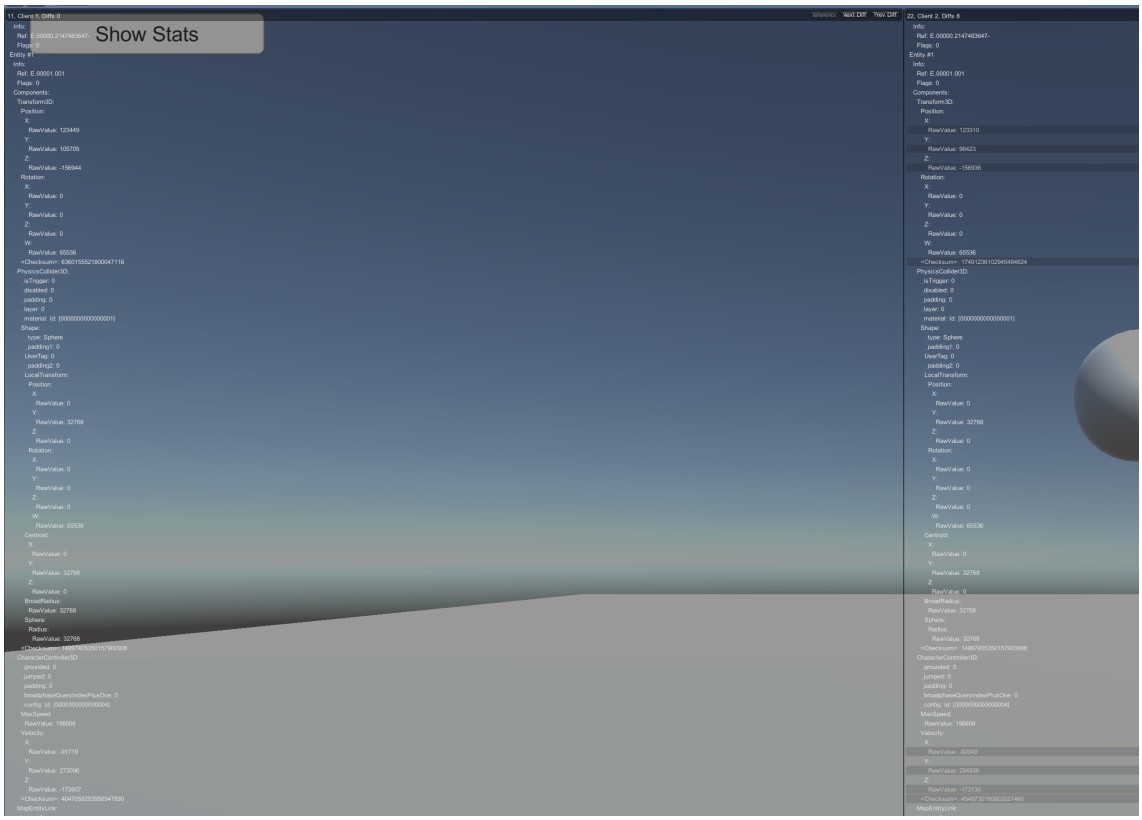

Figure 3.2: Photon Quantum log shown when divergence was detected with all log options

Since most game engines used for creating deterministic games likely offer some form of determinism validation tool, it is difficult to claim that the list provided is explaining all existing and best determinism validation and debugging tools. There are probably other, even more advanced, solutions. However, based on publicly available examples, one can only really consider the ones mentioned.

3.8 Proposed approach

First of all, Unity and DOTS were chosen as the base framework for the thesis because of its potential to provide greater computational speed, which is beneficial for runtime determinism validation and potential future extension of the framework with lag mitigation techniques like prediction and rollback, as these require additional computational resources. The use of the DOD paradigm enhances performance from this perspective [5]. Moreover, creating a publicly available package for the most popular game engine has significant value. It can serve as a foundation for implementing more complex systems and act as a learning resource for developers. On top of that ECS frameworks like DOTS are showing potential for implementing more precise determinism debugging tools. Due to these factors, Unity DOTS was chosen as the base framework for implementing the proposed solution.

The validation process is similar to Quantum in that the game stops when a desync is detected, and a log detailing the first frame that diverged will be created. However, unlike Quantum, this thesis investigates whether validation during runtime can be fast enough to be continuously active in the game (same as Riot).

Additionally, a replay functionality is created, similar to Riot's approach, allowing for replaying the game with different validation options. The replay logs the entire game state, and the computational expense of this approach will be evaluated. Showcase example on how to debug nondeterminism cases is also provided in form of debugging the sample Pong game.

While proposing a better solution is challenging, one potential improvement involves leveraging the ECS pattern. ECS separates system logic from data, leading to a clear division of code into distinct parts. This separation allows for per-system validation, which can pinpoint the exact system causing divergence, thereby limiting the code base that developers usually need to investigate. It's unclear why Photon Quantum is not utilizing this option since they implemented and use their version of ECS. It may be part of Quantum 3, was deemed too resource-intensive for games, or their current priorities are different. Because systems are usually responsible for very specific operations (e.g., only the movement of the balls), this method could significantly reduce the scope of a code that needs to be checked, making the debugging process more efficient and precise.

This thesis also proposes building determinism validation tools in an environment that is not 100% deterministic. This presents an interesting investigation into the potential problems that may arise and how they could be overcome.

The general steps to implement this solution involve first creating a sample game in Unity DOTS. Next, a package is developed to facilitate the use of the deterministic lockstep netcode model as a base for deterministic games, which the sample game utilizes. The goal of this package is to enable various game implementations to leverage it. Finally, determinism validation and debugging tools are integrated into the package.

More concepts are explained in detail in the following sections, but it is important to understand the general aim of this thesis. The created framework serves as an example of

implementation that can be expanded upon. Further expansion ideas are discussed in the Discussion section.

4 Running example

This chapter introduces a game that will serve as the showcase example for all code implementations throughout the remainder of the thesis. The primary goal is to guide the reader by providing visualizations, demonstrating the implementation steps, and ultimately detecting and debugging nondeterministic behavior in the game and what steps the developer would need to take in order to find the source of it and fix it. Additionally, it aims to illustrate the integration process with the developed package.

4.1 Sample Pong game description

The game itself is relatively simple and involves two players competing in a modified version of the classic 'Pong' game. Each player controls a paddle that can move vertically along their side of the screen, with the objective of hitting a ball back and forth across the field. The goal is to prevent the ball from passing their paddle while attempting to get it past their opponent's paddle, scoring points in the process. The game continues until a predetermined score is reached.

This version differs in that, besides counting points and determining a winner, it features not one ball but a thousand, which are spawned over the span of the game. This design choice is intended to demonstrate the handling of a large amount of data in a deterministic lockstep/DOTS-based game and to make nondeterminism detection more comparable to real-life scenarios in terms of entities number. Additionally, this game features only two players because the approach is easier to explain and would work similarly with more players.

Due to the limited time frame of several months for this investigation, it was not feasible to implement a larger game that would present real-life nondeterminism examples. Instead, this game focuses on simpler cases, demonstrating the methodology and potential of these validation tools in real-life scenarios.

Because of that, it won't provide realistic estimates for numbers, such as the time it takes to validate determinism in each frame. To provide a comparison, Unity's own sample "Megacity Metro" is used as a reference point [17]. Megacity Metro is a vertical slice (a proof of concept) and it is large enough and accessible to provide a realistic estimate in terms of amount of entities and systems existing in a production game. Megacity Metro will be used for comparison of potential validation time and file sizes for bigger games in Section 7.

4.2 Functionalities in the game

The Pong sample game consists of a menu where the player can choose to either host or connect to a game, with the option to play locally on one computer, as shown in Figure 4.1. If a player wants to connect to a game, the player only needs to enter the host machine's address and the port on which the game is running, as all other game settings will come directly from the host machine. When hosting a game, the player can specify the port on which the game will run and the simulation speed (ticks per second).

In this case, some validation parameters can be chosen in the game menu. This feature would not typically be available in a released game, where such settings would be configured by the developer. It was added to simplify the validation process and eliminate the

need to rebuild the game each time a validation option is changed. Additionally, there is a checkbox that allows the player to replay the game from a file by reading inputs saved in it. The process of how this is done is explained in Section 7. This feature, like the validation options, is intended for deterministic debugging and would not normally be part of a normal game but is used here for simplicity.

When hosting or connecting to a game, the player will first see a lobby screen as in Figure 4.2, indicating that other players can still join the game and that the game has not started yet. Joining will not be possible after the game starts, as this would require additional logic to connect the player to an already-running deterministic simulation.

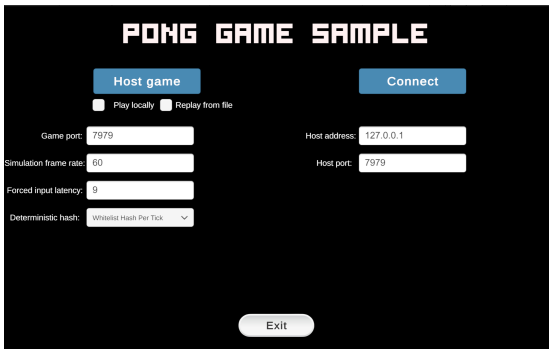

Figure 4.1: ”In game” menu

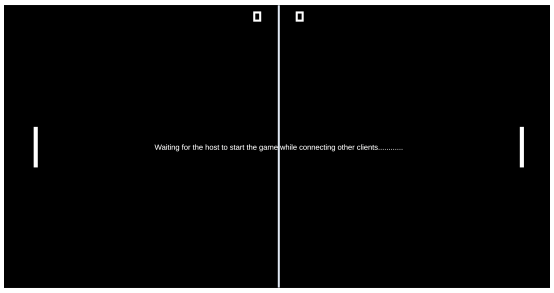

Figure 4.2: Lobby screen which signifies that other players can join the game

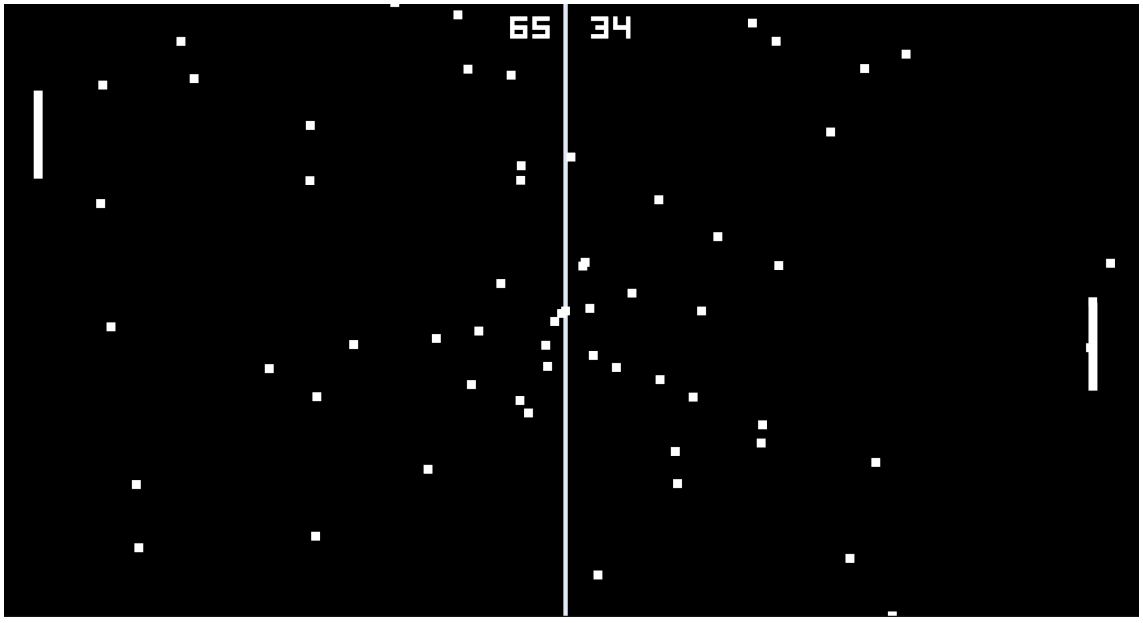

Figure 4.3: Visualization of the sample Pong game

Several gameplay modes were implemented. The first mode is initiated by pressing ”host” and starting the game without anyone joining, allowing the game to be played in single-player mode. This allows for validation if the game behaves as expected (i.e., if balls are visually bouncing correctly between the walls and player’s paddle). The second mode, activated by pressing ”Play locally,” enables two players to play on the same machine (one player uses the arrow keys and the other uses the ”W” and ”S” keys). This mode allows for local game simulation where each paddle represents a separate client, which aids

in debugging same-platform determinism since the validation will be performed between them. Finally, if a remote player joins to control the second paddle, the game is normally played in online multiplayer mode. The general game looks like in Figure 4.3

4.3 Implementation overview

Regarding the deterministic lockstep netcode model and simulation, developers creating a game will have control over its basic settings. These settings will be implemented in Section 6. For the initial step, which is the game implementation, one will only consider a local game without any netcode functionality to keep things simpler. Later settings will include options like frame rate, the number of allowed connections, and other tuning parameters that may vary depending on the type of game. Right now, local game settings only include parameters related to the game itself, like, for example, the number of balls to spawn and their speed.

The diagram in Figure 4.4 illustrates the necessary parts of the sample Pong game in Unity DOTS, which are implemented in Section 5. The details of this diagram will be discussed and explained in the following sections, expanding on them during the implementation of the deterministic lockstep as well as the determinism validation and debugging tools. It provides an overview of what is being implemented.

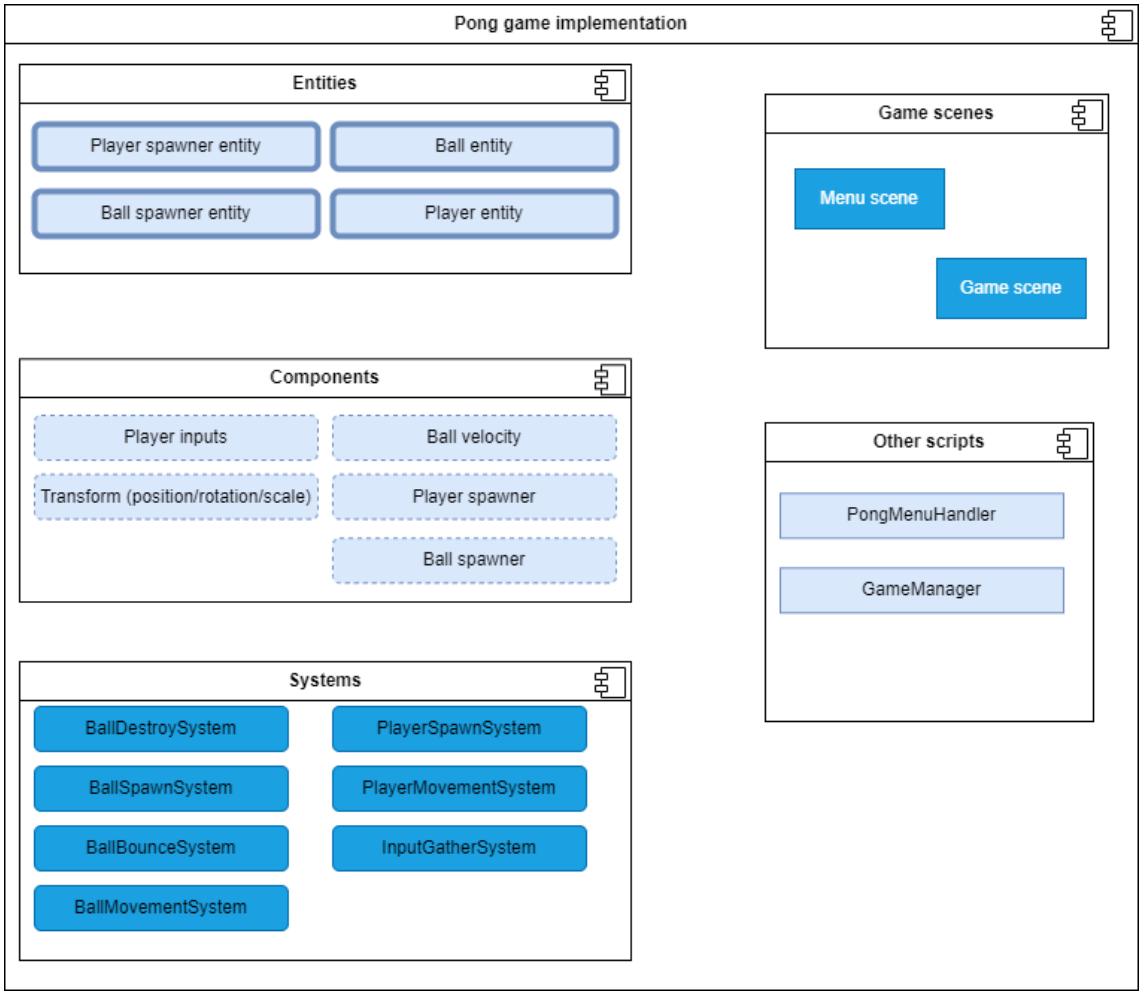

Figure 4.4: Illustration of Pong game elements which are implemented in the next section

5 Sample Pong game implementation

This section provides insights into the implementation of the sample Pong game. First, an overview of the most important Unity and DOTS technical concepts will be provided to understand how it works. Then, this implementation will focus on the local game setup without using the lockstep netcode model or any network connection so the basic game implementation can be understood. The networked version, utilizing the lockstep netcode model, will be implemented and showed in the next section and this will focus on pure game implementation.

5.1 Technical concepts of Unity and DOTS

Unity is one of several game engines, where each offers unique benefits for game development [34]. However, Unity stands out with its advanced implementation of Data-Oriented Design principles and computational speed through DOTS. Practically all DOTS-based games are known for handling numerous moving units efficiently and delivering better performance.

5.1.1 Unity base concepts

Let's begin by exploring the concepts generally used by the Unity game engine that are not associated with DOTS. Unity's traditional method of implementing games has been object-oriented, and not everything can yet be solved by a data-oriented approach. This is why DOTS is optional and can be mixed with object-oriented functionalities within a single game. Additionally, since DOTS represents a different way of thinking about data and code, it couldn't simply replace object-oriented coding but had to be added as an optional feature. By starting with an exploration of object-oriented functionalities, the basic concepts that will also be used in the sample Pong game can be understood.

Game objects and prefabs

In Unity, GameObjects and Prefabs are fundamental building blocks used to create and manage game elements. As the name suggests, a game object is the most basic object in games made with Unity. It acts as a container for various components that define its properties and behaviour. These components can include physical attributes like transform, which defines the game object's position, rotation, and scale, as well as scripts, colliders, renderers, and more. GameObjects are versatile and can represent anything from characters and props to cameras and lights, making them essential for constructing the game world. GameObject has essentially the same function as an entity, which is to group components and script into a singular object.

Prefabs, in turn, are saved instances of GameObjects. The usual approach is to create a GameObject in the editor, modify it, and then drag and drop it into one of the folders, which automatically converts it into a Prefab. This Prefab can then be referenced by scripts, such as a spawning script, to instantiate the premade GameObject. For example, a ball Prefab with all its components and scripts already assigned can be spawned as a fully functional object within the game.

Those concepts will be used in the sample Pong game in order to create saved instances of objects such as ball or player.

Scenes in Unity

In Unity, a scene serves as a container for the environments, objects, and settings that comprise separate game levels. It allows for loading and unloading groups of objects or

entities without needing to destroy all entities in the world and create new ones, which would constitute a new level.

This approach is particularly useful for creating distinct sections of a game, such as a menu scene and a game scene in Pong, as these are completely different in terms of content and functionality.

DeltaTime

DeltaTime is a crucial parameter in game development. It represents the time it took to process the previous frame. DetlaTime is calculated as the difference between the current frame's timestamp and the previous frame's timestamp. This value, referred to as "delta time" or "elapsed time," is used in most per-frame calculations.

Because different machines have varying computational power, the simulation of a full game frame might take 16 ms on one machine and 30 ms on another. These values are then saved as DeltaTime and can be used for various purposes in the next frame.

For example, if developer wants a player object to move 1 metre per second, he cannot simply use its velocity and move it by this value every frame because it would result in the player object moving faster on a more powerful machine, introducing nondeterminism. Instead, he must multiply the velocity by DeltaTime. This ensures that both machines will experience the same movement, maintaining consistent gameplay across different hardware since, after 1 second, the player will have moved the same distance on both machines.

Another application is ensuring that all clients process the game simulation at the same speed (i.e., 60 FPS). The method for enforcing this is explained in detail in Section **??**.

The concept of delta time is used both in DOTS and in traditional game development within Unity.

Managed and unmanaged variables

Another concept used both by standard Unity and DOTS is understanding the difference between managed and unmanaged variables. Managed variables are those whose memory management is automatically handled by the garbage collector. This simplifies development, as developers do not need to manually allocate and deallocate memory because garbage collector automatically detects which parts of memory can be set free (deallocated) to be used by other parts of the simulation. However, the convenience of managed variables comes with a performance cost due to the overhead introduced by the garbage collector.

In contrast, unmanaged variables require developers to manually manage memory allocation and deallocation. This manual management can lead to significant performance improvements and more determinism because memory operations are more predictable and efficient without the intervention of the garbage collector. Unmanaged memory management is crucial for performance-sensitive applications, such as games, where consistent and high-speed performance is required.

In Unity, both managed and unmanaged variables are used, but in DOTS, the recommendation is to use unmanaged variables to speed up code execution. There are specific types of systems, which will be explained later, that use only unmanaged code or allow a mix of unmanaged and managed code.

An example of utilising unmanaged variables in Unity's DOTS is the NativeList, which is an unmanaged version of a standard list. Developers are responsible for managing its memory, which involves explicitly allocating and freeing memory when it is no longer

needed. This responsibility gives developers better control over memory usage and helps avoid the performance overhead associated with garbage collection.

Using NativeList and other native collections prefixed with ”Native” when used with DOTS can lead to performance gains. These structures are optimised for use with the Job System and Burst Compiler in Unity’s DOTS, enabling more efficient parallel processing and deterministic behaviour.

5.1.2 DOTS related concepts

Now let’s understand the basic concepts which will be used from DOTS side in the Pong implementation.

ECS in DOTS

Entities in DOTS are represented by a unique identifier, as shown in Figure 5.1. Entities themselves do not contain any data or behaviour; instead, their purpose is to allow to group different components together. An entity can conceptually represent anything in the game environment, such as a player or a ball. However, it is important to remember that entities only group together components that constitute such objects (e.g., a player).

In DOTS, entity creation, destruction, associating components with entities, and other operations are performed using the EntityManager. The EntityManager handles the necessary memory restructuring based on the intended action. An example usage is shown in Figure 5.1.

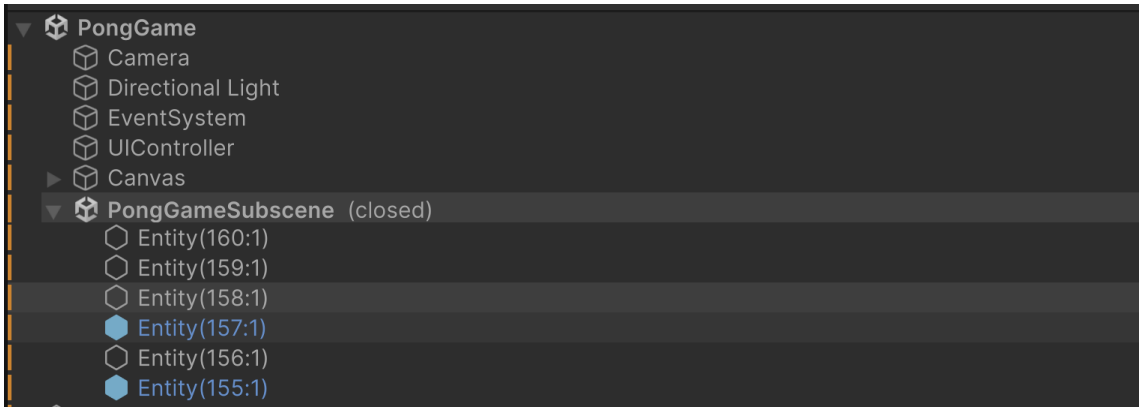

Figure 5.1: An example of entities in Unity

A **Component** in DOTS is a data container. Each component represents a single aspect of an entity, such as a transform component (containing position, rotation, and scale), ball velocity, or input values, but contains no logic related to them. Components do not possess any behaviour on their own, they are purely data. For instance, a ”Velocity” component stores the speed and direction of the ball entity.

In summary, a component is a container of data, and an entity is a container of components. Figure 5.2 illustrates the view of components associated with an entity in the Unity editor. This view allows to inspect various data contained within the components.

A component can either have associated data or be created without any data, in which case it is treated as a ”tag component.” Tag components are used to mark entities for specific purposes. For example, an entity can be marked as spawned by adding the PlayerSpawned tag component.

In Unity, an example of component implementation and its association with an entity is shown in the code listing 5.1. First one can see how component can be implemented and then how to add it to a newly created entity by using EntityManager.

It is worth to mention that there are two ways to connect a component to an entity. One way is via the EntityManager, as demonstrated in the same code listing 5.1. The other way is to manually add components to an entity in the editor, which is useful for pre-making objects that exist in the game, eliminating the need to perform this operation in the code.

As an example in Figure 5.2 an example of components added to an entity through the editor is shown, which can then be accessed, modified, added, or removed via code.

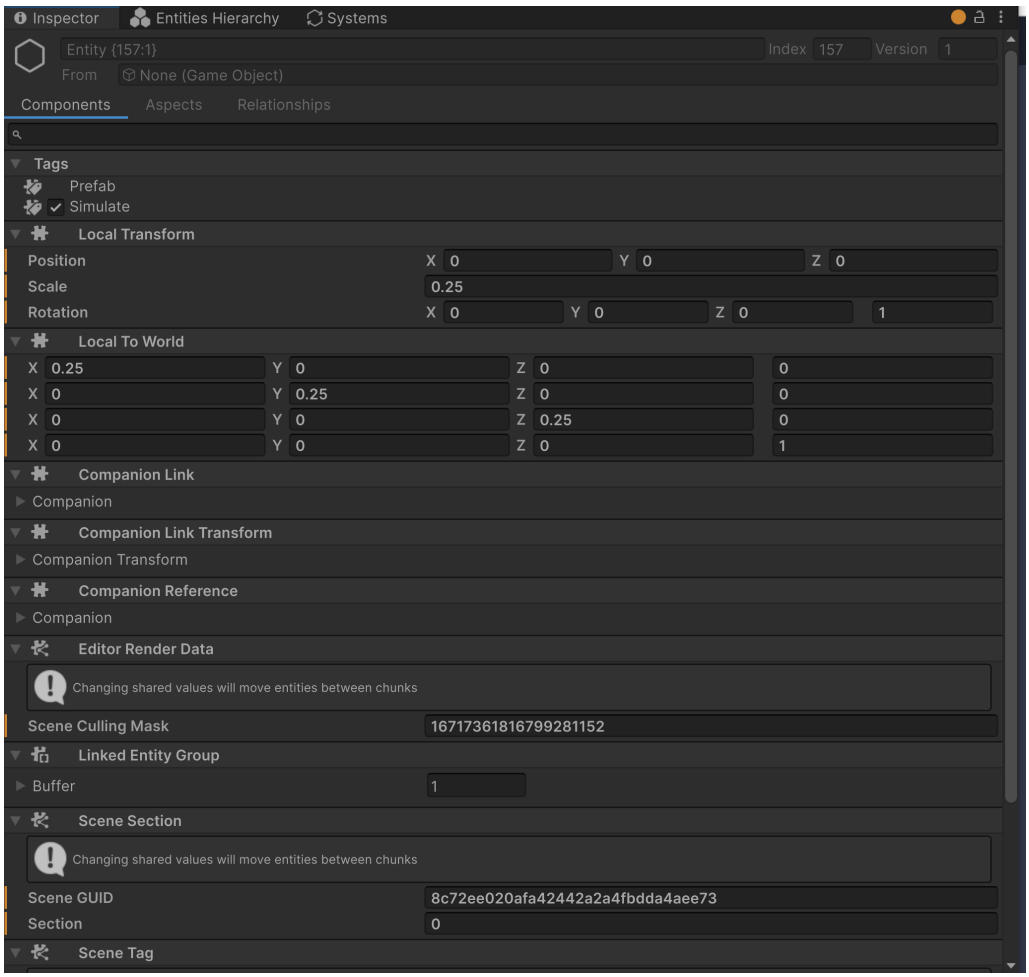

Figure 5.2: An example of components in Unity

```
public struct ExampleComponent : IComponentData
{
    public Entity exampleEntity;
    public int Value;
}

public partial class ExampleSystem : SystemBase
{
    var testEntity;

    protected override void OnUpdate()
    {
```

```
        testEntity = EntityManager.CreateEntity();
        var newComponent = new ExampleComponent(){
            exampleEntity = testEntity,
            Value = 5
        }

        EntityManager.AddComponentData(testEntity, newComponent);
    }
}
```

Listing 5.1: Example component holding data in Unity DOTS and operations on it with EntityManager

A **System** in DOTS and in the context of ECS architecture is a piece of logic or functionality that operates on a set of entities containing specific components. Systems are responsible for processing and updating the state of these components, often based on specific criteria or events. Systems focus on a single aspect of the game logic, such as spawning balls in the Pong game, moving players, etc., making the code modular and easier to maintain. They interact with the data stored in components, reading, modifying, and updating components to drive the behaviour and state of entities. A system typically iterates over all entities that possess the necessary components to perform its operations, for example, a BallMovementSystem processes all entities with a ”Velocity” and Transform components. Systems operate independently of each other and do not maintain direct references to entities, instead, they operate on component data, promoting decoupling and flexibility in the architecture. Systems are usually executed in a specific order within the game loop, ensuring that the game logic is processed correctly, which is crucial for maintaining proper game behaviour. By using systems in ECS, the architecture promotes a clean separation between data (components) and behaviour (systems), which is particularly beneficial when debugging nondeterministic issues.

As an example, let’s consider the following DOTS systems from the Pong game (just their limited example scope) listed in Code Listing: 5.2

This example highlights several features of DOTS systems. Firstly, it demonstrates the ability to control the order of system execution using attributes. By using attributes such as ”[UpdateInGroup]” and ”[UpdateAfter]”, one can specify the relative positioning of systems within system groups, which act as containers for grouping systems together. In this case, both systems will run in the DeterministicSimulationSystemGroup, with the BallSpawnSystem being executed before the BallMovementSystem.

Secondly, ”SystemAPI,” which provides a set of methods and properties for interacting with the ECS framework, allows for querying for specific components. BallMovementSystem modifies the position value based on the Velocity component value. The foreach loop iterates over entities that have both Transform and Velocity components, which in this game will exist together only on ball entities, and updates their values.

Furthermore, the use of RefRW (reference ReadWrite) and RefRO (reference ReadOnly) helps indicate whether a component is read-only or read-write, improving access security, especially when executing parallel jobs.

By adding, removing, or changing components, one can influence which entities will be processed by a given system. For example, if the Velocity component is removed or disabled, the Transform component on the same entity will not be updated by the BallMovementSystem.

This dynamic component-based structure is a key feature of DOTS, enabling flexible and efficient entity processing and can be visualised as in Figure 5.3.

It's also worth noting that DOTS can be mixed with standard object-oriented classes, which are often used for tasks like connecting buttons and executing commands. Not everything in Unity can be implemented using DOTS (e.g., User Interface (UI) elements and scene management), so these approaches can be combined to use the benefits of both paradigms.

```
[UpdateInGroup(typeof(DeterministicSimulationSystemGroup))]
[UpdateBefore(typeof(BallSpawnSystem))]
public partial struct BallMovementSystem : ISystem
{
    public void OnUpdate(ref SystemState state)
    {
        float deltaTime = SystemAPI.Time.DeltaTime;

        foreach (var (transform, velocity) in SystemAPI
            .Query<RefRW<LocalTransform>, RefRO<Velocity>>())
        {
            transform.ValueRW.position += velocity.ValueRO * deltaTime;
        }
    }
}

[UpdateInGroup(typeof(DeterministicSimulationSystemGroup))]
[UpdateAfter(typeof(MovementSystem))]
public partial struct BallSpawnSystem : ISystem
{
    public void OnUpdate(ref SystemState state)
    {
        var ball = EntityManager.Instantiate(prefab);
        EntityManager.SetComponentData(ball, new LocalTransform
        {
            Position = new float3(0, 0, 13),
            Scale = 0.2f,
            Rotation = quaternion.identity
        });
    }
}
```

Listing 5.2: Example of system in DOTS

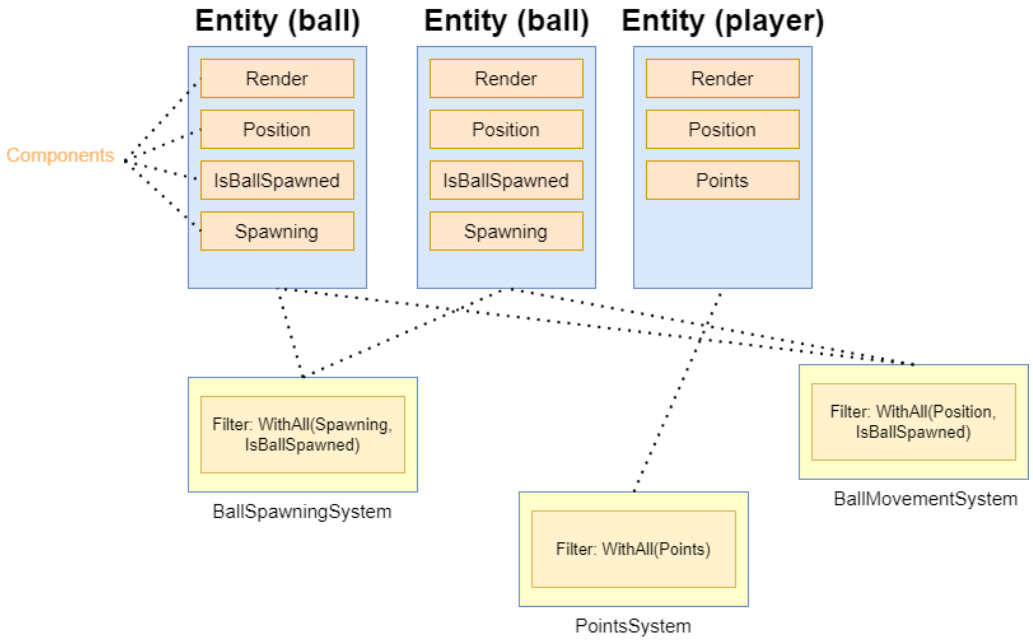

Figure 5.3: ECS flow in DOTS

Worlds

A world in Unity is a context or scope within which entities, components, and systems exist and operate. Each world contains its own set of entities and systems, allowing for isolated or parallel processing of different parts of a game or application. A world contains its own unique EntityManager, which developers use to create, destroy, and manage entities.

The difference between a scene and a world is that while Unity scenes are about visually arranging and managing game elements, DOTS worlds are about optimising the underlying data and processing for better performance.

By default, Unity creates a ”Default World,” where most gameplay logic occurs and to which every system and entity is assigned. Developers have the option to create additional worlds, which is useful for isolating systems, components, and entities for specific situations or logical groupings. For example, in online multiplayer games, it is common to separate client and server logic. DOTS provides world flags such as ClientWorld or ServerWorld, allowing developers to create distinct worlds for server and client operations. This enables grouping specific systems to run exclusively on the client, server, or both.

To assign and run a system in a specific world, an attribute can be used, as shown in the modified BallSpawnSystem in Listing 5.3.

This division ensures that only server-related systems run in the server world, while client-related systems run in the client world. The server world typically does not need to perform any visual updates and therefore usually only contains systems that manage incoming data from clients.

Without such a division, systems intended only for the server (e.g., a system that gathers and validates data in a lockstep model) would also run on the client machine. This could either influence the gameplay negatively or, at best, add unnecessary overhead to the client’s processing. The client machine would then need to detect that this system shouldn’t execute because it is not a server, adding complexity and inefficiency.

```
1
2  public class PongMenuHandler : MonoBehaviour
3  {
4      private static World CreateClientWorld(string name)
5      {
6          var world = new World(name, WorldFlags.GameClient);
7      }
8  }
9
10 [WorldSystemFilter(WorldSystemFilterFlags.ClientSimulation)]
11 public partial struct BallSpawnSystem : ISystem
12 {
13
14 }
```

Listing 5.3: Flag for assigning system to a world

5.2 Pong game implementation

Now, knowing the basic concepts of DOTS, let's focus on insights of the sample Pong game implementation.

5.2.1 Scenes and Prefabs

First, game scenes need to be implemented to serve as the main game environment. The game consists of two scenes. One of them is the menu scene, which consists of buttons shown in Figure 5.4, whose functionality was described in the previous Section 4. Second is a game scene, which contains the basic game elements like the middle line, points text, and player spawner as well as ball spawner entities shown in Figure 5.5. Ball spawner and player spawner entities only include the transform component and are not visually represented.

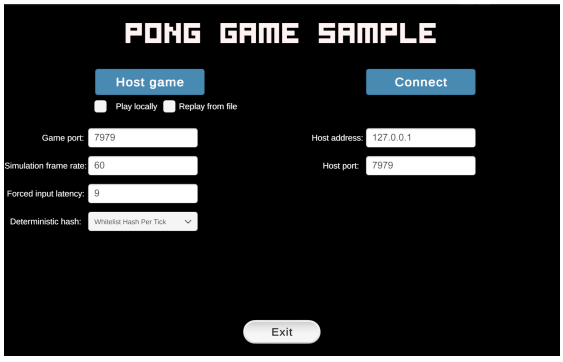

Figure 5.4: In game menu

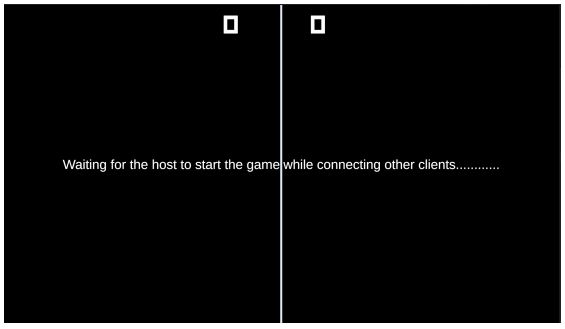

Figure 5.5: Pong game scene

Next, all the prefabs that will be used were created, which in this case consist of a ball Prefab (shown as an example in Figure 5.6) and a player prefab.

In Figure 5.6, one can see the structure of a ball Prefab and its components. It is important to note that these are not yet ECS components, they will be created when this object is instantiated by DOTS. Currently, the Prefab consists of a transform component, a sprite renderer for visualising the object, and an authoring script. Authoring scripts are used during entity creation to add components. For the ball Prefab mentioned, the authoring script, shown in Listing 5.4, adds a velocity component while the entity is created. This ensures that every time the ball is spawned, it already has the velocity component assigned.

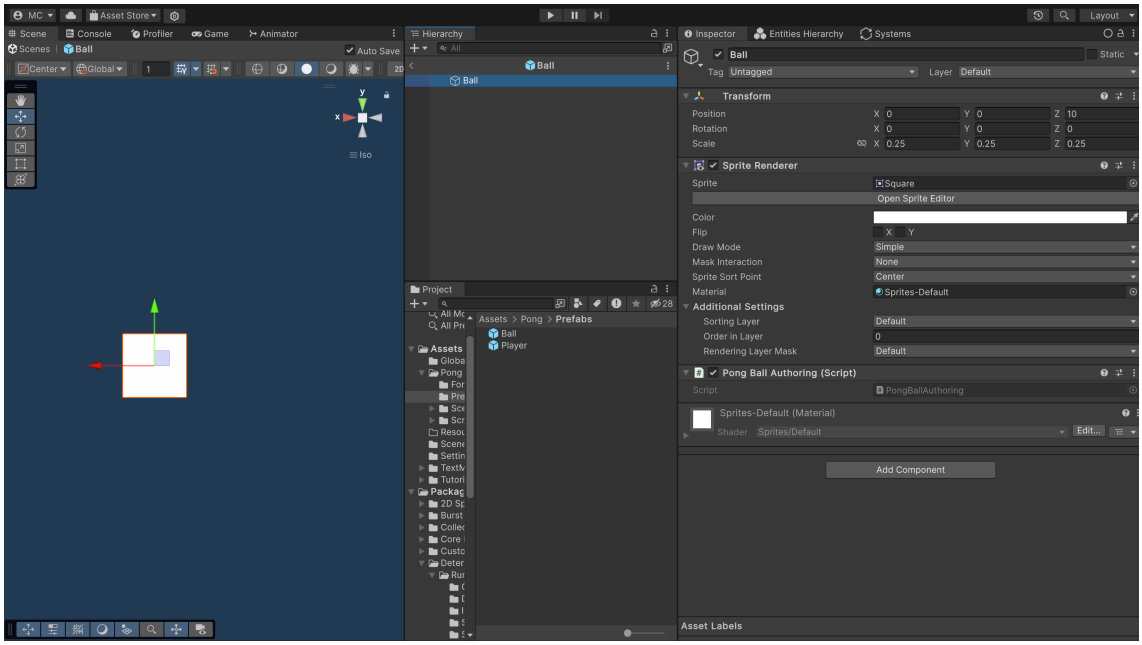

Figure 5.6: Ball Prefab instance

```
1
2  public struct Velocity : IComponentData
3  {
4      public float3 value;
5  }
6
7  public class PongBallAuthoring : MonoBehaviour
8  {
9      class Baker : Baker<PongBallAuthoring>
10     {
11         public override void Bake(PongBallAuthoring authoring)
12         {
13             var entity = GetEntity(TransformUsageFlags.Dynamic);
14
15             var velocity = default(Velocity);
16                 AddComponent(entity, velocity);
17         }
18     }
19 }
```

Listing 5.4: Pong Ball Authoring script

5.2.2 Entities and components

Let's examine all the components and entities that constitute the game.

Transform component

Transform is a component that every entity possesses from the start. It specifies the position, rotation, and scale of the object in the game world, serving as its coordinates. This component is automatically assigned to any entity upon its creation and serves as the base data for an object in Unity.

Velocity component

The velocity component is implemented in the game and is used to indicate the ball's speed and direction in the form of a float3 value. This component, as mentioned earlier, is assigned to an entity when the Prefab is created as an entity by the spawner. The value

of this component is set randomly on entity creation to randomise the speed and direction of the ball with constraints of maximum and minimum speed and angle.

PlayerInput component
PlayerInput is a singleton component, meaning only one instance of this component exists in the game. It holds the current player input, which, in the case of the Pong game, is represented by a VerticalInput float value indicating the direction and magnitude of the player's movement. In particular, PlayerInput does not store raw input (such as key presses like ”w”), but rather the transformed input that signifies the exact action. Later in this chapter, it is shown how the InputGatherSystem interprets key presses and updates the PlayerInput struct accordingly.

PongBallSpawner component
PongBallSpawner component contains ball entity object to spawn and its baked automatically into BallSpawner Entity.

PongPlayerSpawner component
Similarly to PongBallSpawner component, PongPlayerSpawner component contains player entity object to spawn and its baked automatically into player spawner entity.

Player component
The Player component is a tag component used to mark entities that represent players. This is important for querying purposes, as otherwise, the player entities would only have a Transform component assigned, which is insufficient for creating player-specific queries.

Ball Spawner Entity
A ball spawner entity is created in the scene and positioned in the middle of the game field. It has an authoring component to which the ball Prefab is attached, which is later used by the BallSpawnSystem to reference and spawn it. The ball will be spawned by the BallSpawnSystem later, which will use the ball spawner entity position and spawn the ball Prefab there.

Figure 5.7 shows the view from the editor perspective, where one can see how this setup looks like in the editor.

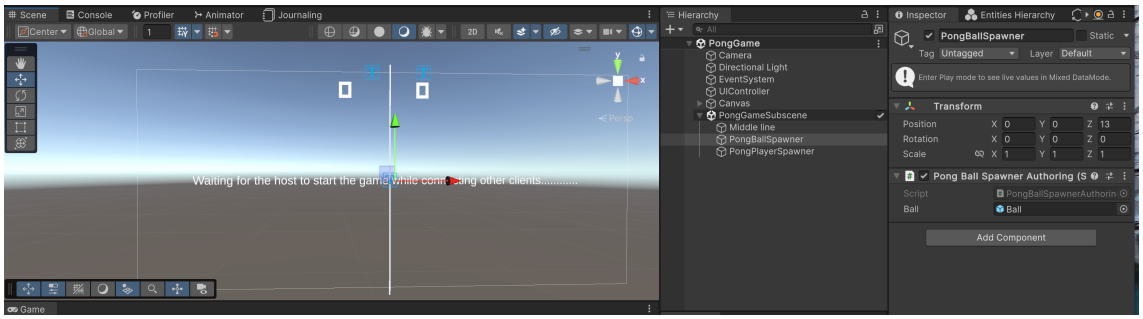

Figure 5.7: PongBallSpawner setup in the game scene

Player Spawner Entity
Similarly to the BallSpawner entity, the PlayerSpawner entity is also created in the subscene and has an authoring component with PongPlayerSpawner component where a player Prefab is attached. It is used by the PlayerSpawnSystem to spawn players at the start of the game while referencing the player prefab.

Player entity
Two **player entities** are created at the beginning of the game via the PlayerSpawnSystem, transforming the created player Prefab into an entity. They are created at predefined positions on each side of the game scene.

Ball entity
Ball entites are created once per simulation tick by the BallSpawnSystem. They are spawned in the position of a ball spawner entity, transforming into the ball Prefab into the ball Prefab assigned to it. Then the velocity component value is set, and the entity is considered spawned.

5.2.3 Systems

Systems are filter specific entity archetypes, which are entities containing the same types of components, such as ball entities that share common component types. Since these entities are stored contiguously in memory, they can be fetched and iterated on together by the system, improving the efficiency of operations.

The components mentioned earlier are then fetched and modified by these systems.

PlayerSpawnSystem
The PlayerSpawnSystem is responsible for filtering the PongPlayerSpawner component, of which only one instance exists in the scene, and spawning two players. It queries the PlayerSpawner component, retrieves the associated entity, and spawns two players at the beginning of the game. After this initial spawning, the system is disabled, as there is no need to spawn additional players.

InputGatherSystem
The InputGatherSystem runs before the PlayerMovementSystem in each frame, detecting and interpreting player key presses to populate the PlayerInput component. It queries the PlayerInput component, a singleton component with only one instance in the game, and updates its value. In the sample Pong game, the ”w” and ”s” inputs are interpreted as values of 1 or -1 for the verticalInput in this component, indicating whether the player should move up by 1 distance unit, move down, or stay in place.

PlayerMovementSystem
The PlayerMovementSystem queries for player and transforms components and updates their positions based on the PlayerInput component, which is retrieved through another query. Since the base game is not yet multiplayer, both players move up or down by the value from PlayerInput, multiplied by deltaTime to ensure consistent movement.

BallSpawnSystem
The BallSpawnSystem is responsible for filtering the PongBallSpawner component, of which only one instance exists in the scene, and spawning one ball each time it runs until 1000 balls have been spawned. It queries the BallSpawner component, retrieves the associated entity, and spawns one ball per frame at the spawner’s position. It then randomly sets the velocity component of the spawned entity, which will later be used by the BallMovementSystem.

BallMovementSystem
The BallMovementSystem moves the balls every frame according to the values in their Velocity component. The system filters for transform and velocity components (which exist together only on ball entities). This system is more complicated since, because of the number of balls to move, it utilises parallel jobs to make the process more efficient.

Code listing 5.5 shows the implementation.

This system is executed after the BallBounceSystem and begins by creating a query to gather all entities with the LocalTransform and Velocity components. It then stores specific components in temporary arrays, which are used in a parallel job to update the ball positions. The main thread delegates the actual movement calculations to the BallMovementJob, which runs in parallel for each ball.

The job utilises EntityCommandBuffer, which is used to save commands from parallel jobs, and then one can use it to execute them after the job finishes, which ensures that all updates are performed safely in parallel. The job calculates the new positions based on the velocity and deltaTime and writes the new position back to the LocalTransform component of each ball entity using the EntityCommandBuffer.ParallelWriter. After completing the job, the command buffer is played back to apply the changes, and all temporary arrays are disposed of to free up memory.

This approach ensures efficient and smooth movement of a potentially large number of balls by leveraging Unity's job system for parallel processing.

```
1
2  [UpdateAfter(typeof(BallBounceSystem))]
3  public partial struct BallMovementSystem : ISystem
4  {
5      private EntityQuery ballsQuery;
6      private NativeArray<LocalTransform> ballTransform;
7      private NativeArray<Velocity> ballVelocities;
8      private NativeArray<Entity> ballEntities;
9
10     public void OnUpdate(ref SystemState state)
11     {
12         var deltaTime = SystemAPI.Time.DeltaTime;
13
14         ballsQuery = SystemAPI.QueryBuilder().WithAll<LocalTransform, Velocity
               >().Build();
15         ballTransform = ballsQuery.ToComponentDataArray<LocalTransform>(
               Allocator.TempJob);
16         ballVelocities = ballsQuery.ToComponentDataArray<Velocity>(Allocator.
               TempJob);
17         ballEntities = ballsQuery.ToEntityArray(Allocator.TempJob);
18
19         var ecb = new EntityCommandBuffer(Allocator.TempJob);
20
21         var ballMovementJob = new BallMovementJob
22         {
23             ECB = ecb.AsParallelWriter(),
24             ballVelocities = ballVelocities,
25             localTransform = ballTransform,
26             entities = ballEntities,
27             deltaTime = deltaTime
28         };
29
30         JobHandle ballMovementHandle = ballMovementJob.Schedule(ballTransform.
               Length,1);
31         ballMovementHandle.Complete();
32         ecb.Playback(state.EntityManager);
33
34         ecb.Dispose();
35         ballTransform.Dispose();
36         ballVelocities.Dispose();
37         ballEntities.Dispose();
38     }
39 }
40
41 public struct BallMovementJob : IJobParallelFor
42 {
43     public EntityCommandBuffer.ParallelWriter ECB;
44
45     public NativeArray<Entity> entities;
46     public NativeArray<LocalTransform> localTransform;
```

```
47      public NativeArray<Velocity> ballVelocities;
48
49      public float deltaTime;
50
51      public void Execute(int index)
52      {
53          LocalTransform transform = localTransform[index];
54          Velocity velocity = ballVelocities[index];
55          Entity entity = entities[index];
56
57          var newPosition = transform.Position + deltaTime * velocity.value;
58
59          var newTransform = new LocalTransform
60          {
61              Position = newPosition,
62              Rotation = transform.Rotation,
63              Scale = transform.Scale
64          };
65
66          ECB.SetComponent(index , entity, newTransform);
67      }
68  }
```

Listing 5.5: BallMovement system implementation in the game

BallBounceSystem
The BallBounceSystem has two queries: one for entities with velocity and transform components, and another for entities with transform and player components. The system checks if the ball is colliding with a wall or a player, using the player query to obtain their current positions. It then updates the ball's velocity component to reflect the bounce accordingly.

BallDestroySystem
The BallDestroySystem queries for the transform and velocity components and checks if a ball has crossed the boundary line at any part of the field. If so, the entity is destroyed, and points are added accordingly via the UIManager class.

5.2.4 Other scripts

There are few scripts that aren't systems but rather classic object-oriented scripts. Those are used to handle behaviours that are currently not possible to do with DOTS, like handling button presses, which need to be assigned to a function, or managing UI components on the scene.

GameManager class
Functions from the GameManager are called by the BallDestroySystem when a ball is destroyed. The GameManager is responsible for managing points during the game, determining if the game has ended (with a total of 1000 points scored), and identifying which player won. If the game is finished, the GameManager stops the execution of other systems, displays a message indicating which player won, waits 5 seconds, and then exits the game.

PongMenuHandler class
The PongMenuHandler class manages the behaviour of various buttons in the menu scene, such as starting the game or creating server and client worlds when the "Host Game" button is pressed. The implementation details for creating server and client worlds are covered in the next section when the netcode model is added.

All of these systems, components, entities, and classes work together in a local simulation. In this implementation, the game does not function in multiplayer mode but allows for local

player movement. The next step is to implement the netcode model to make the game multiplayer.

6 Deterministic Lockstep Netcode Model

A well-established netcode model for deterministic games is lockstep. This model is implemented as a package that can be added to the game and utilised by developers. While the implemented Pong game showcases the usage of this package, it can also be used to create more complex multiplayer online games.

This chapter begins by outlining the theoretical concepts behind lockstep, progressing to its implementation, and how it should be used by the game. In the end, it results in a basic deterministic lockstep-based multiplayer online game to which determinism validation and debugging tools are added.

Figure 6.1 shows the new elements implemented in this chapter.

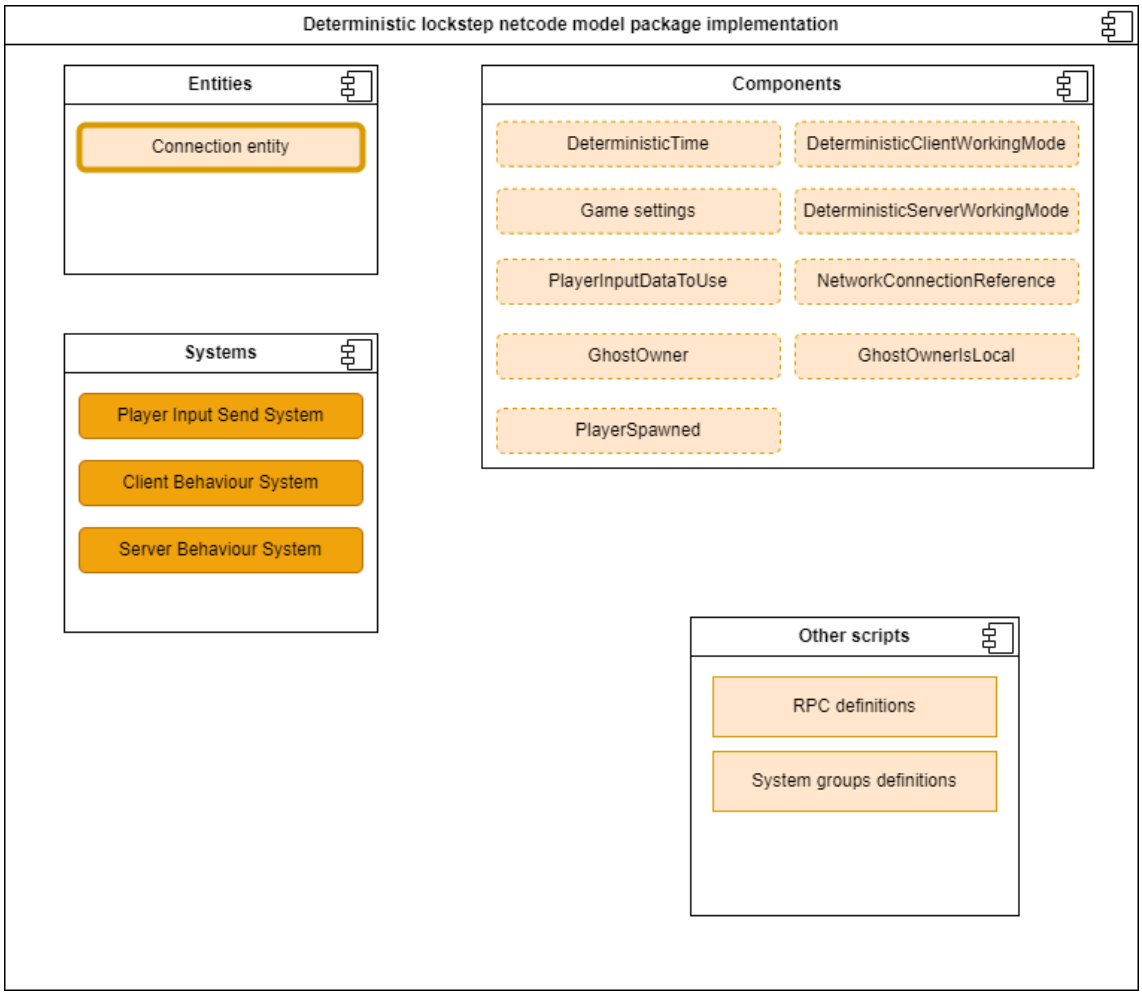

Figure 6.1: Illustration of deterministic lockstep netcode model elements which are implemented in the this section

6.1 Deterministic Lockstep theory

Deterministic Lockstep consists of several interrelated concepts that work together to ensure a synchronized and consistent game state across all clients. Lets start with the

simplified and not yet optimized concept of lockstep and then expand the understanding of it by adding next parts which constitite the final version. The final version will be used to support deterministic games in the package.

6.1.1 Naive, non-deterministic lockstep

In general, lockstep revolves around ensuring that each game client processes the same game step (also called a 'tick') at the same time. Consequently, no client sees a different or older version of the game state compared to the others. Each client advances the game state locked in step with each other, hence the name "lockstep." The visual explanation in Figure 6.2 illustrates this approach. It is called lockstep because all clients wait for inputs from all players (sent together from the server) to arrive before proceeding to the next step.

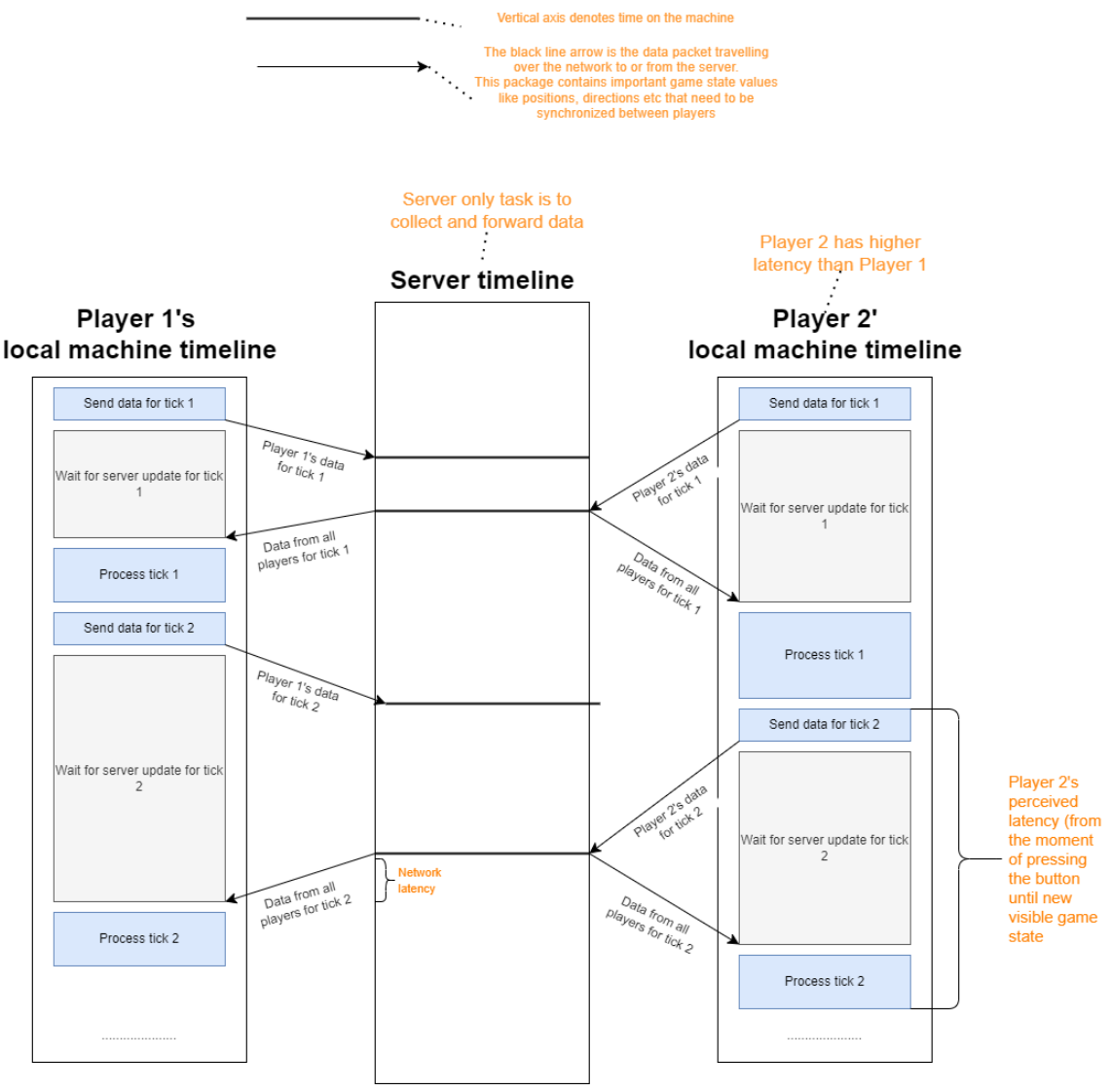

Figure 6.2: An example of lockstep running on two game clients connected to a server. The clients send their data to a server and then wait for all the inputs that are needed to be applied in the game before processing a game tick. The client waits as long as necessary to receive that data.

The primary function of Lockstep, from the server's perspective, is to wait for the game

state data of the player at the respective client (such as positions of objects, movements, etc.) for a given tick. Once all clients' state data for that tick is received, the server combines this data into one packet and sends it back to all clients.

However, having to wait for the data transmission comes with the limitation that all clients need to fully complete their game tick and their transmission before the next tick can happen. In this simplified lockstep example, the problem is that any network latency at all causes the game to pause.

Basic lockstep tends to work acceptably on a local network within the same house or neighbourhood where latencies are low. However, on the internet, higher latencies can cause lockstep games to run slowly or experience frequent lag. With the naive lockstep model, it's not just frequent lag (latency), it's frequent pauses and resumes, resulting in stuttering. Extensive stuttering can make the game appear to move in slow motion. For example, a connection between Copenhagen and Toronto has a round-trip latency of about 100 ms [35]. This would require a lockstep game to run at 10 game steps per second or slower, which is adequate for turn-based games but insufficient for faster-paced games.

In order to address the lockstep imperfections, there are several improvements that make the method usable for online games that need faster tick rates.

6.1.2 Replacing state with inputs – Deterministic lockstep

Deterministic lockstep differs in that it sends players' control inputs (e.g., keyboard key presses) exclusively rather than the resulting game state (e.g., the character controller's current position and velocity). In the case of the sample Pong game, these inputs are already processed into precise commands, specifically the vertical input value indicating where the player should move. The other clients then process these inputs by injecting them into the next tick of their own local game simulation to determine the agreed-upon, deterministic outcome of this tick.

For example, in a Pong game, if a player moves and a ball bounces off their paddle, a naive lockstep netcode architecture would require the client to send not only the speed and position of their paddle but also the same parameters for all the balls to the other clients to maintain synchronisation. In deterministic lockstep, however, the client simply sends the fact that the player has pressed the "move up" button. This means sending the processed information that "move up" was pressed rather than just the raw key press data ("w" key). The other clients, upon receiving this information, will handle moving the other player's paddle and updating the movement and bouncing of the balls.

This method reduces the amount of data that needs to be sent every tick, as player input data is usually significantly smaller than game state updates. However, it assumes that the code can process this information in a deterministic manner, which necessitates the validation implemented later.

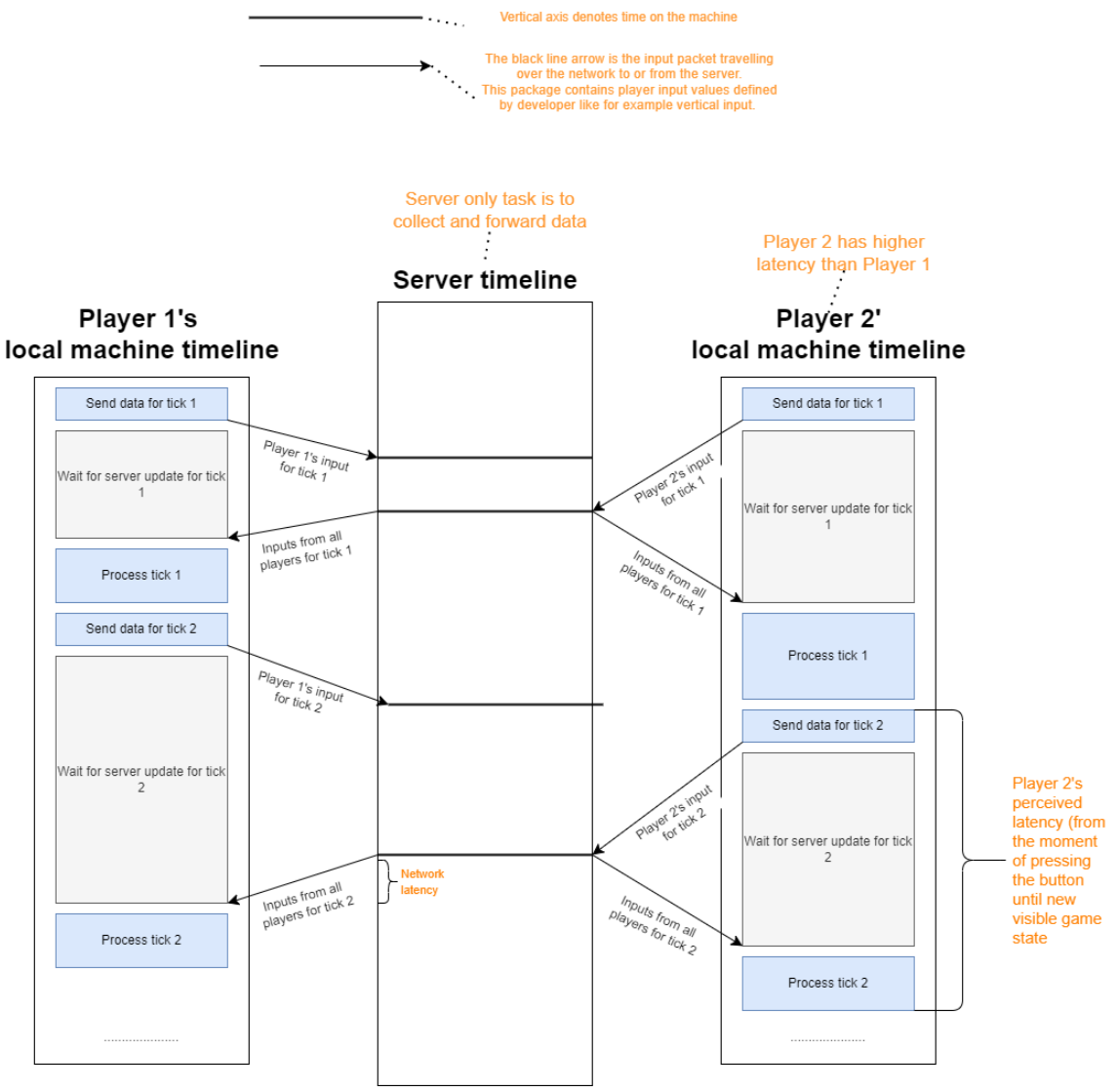

Figure 6.3: An example of deterministic lockstep running on two game clients connected to a server. The clients send their inputs to a server and then wait for all the inputs that are needed to be applied in the game before processing a game tick. The client waits as long as necessary to receive that data.

The “deterministic” part of the name comes from the fact that in order for the games to remain synchronised using just input data, each client must be able to deterministically compute the next game step based on the current game state and the provided input. If the Pong game input of one client says move-up, each client must move-up the paddle associated with this client in exactly the same trajectory and by the same distance across all clients, together with any other ball movements in the game, to arrive at exactly the same game state (bitwise identical). Usually, the game state is checked to be ”identical enough” for the game to proceed, as will be explained in Section 7.

Server in this case is not simulating the game which allows for scalability in terms of player number [9] but also means that the smallest difference in result would cause the game state on different clients to desynchronize, slowly diverging further and further apart. Over time, these differences will accumulate and visually desynchronize the game.

Because of the way it works, it's also necessary to ensure the starting game state is identical for clients taking part in the game, especially when a large game environment is involved.

6.1.3 Adding input latency

While deterministic lockstep provides some marginal benefits in reducing data transmission size, the real benefit comes from removing the constant pause/resume stuttering, which its achieved by buffering inputs via a forced input latency. In regular deterministic lockstep, all clients have to wait to receive input data from each other before running a game step. However, if that input data were scheduled to be executed in some future game tick, the game would be able to process other ticks in the meantime without having to wait.

By not having to wait for inputs to arrive, the game no longer stutters in the common case and therefore may actually achieve the intended tick rate chosen by the game developer. This allows the developer to set the tick rate to, for example, 30 or 60 ticks per second, depending on the needs of the game. As long as each input packet arrives before its scheduled target processing tick, the game would not need to ever enter a wait state, as shown in Figure 6.4.

This figure differs from the previous naive deterministic lockstep in Figure 6.3 by not having to wait for inputs, allowing it to run at a specified pace as long as inputs are arriving from the server before the scheduled frame. Typically, the buffer consists of more ticks than just three. For example, in a game running at 60 FPS, the buffer might have 9 ticks of input latency.

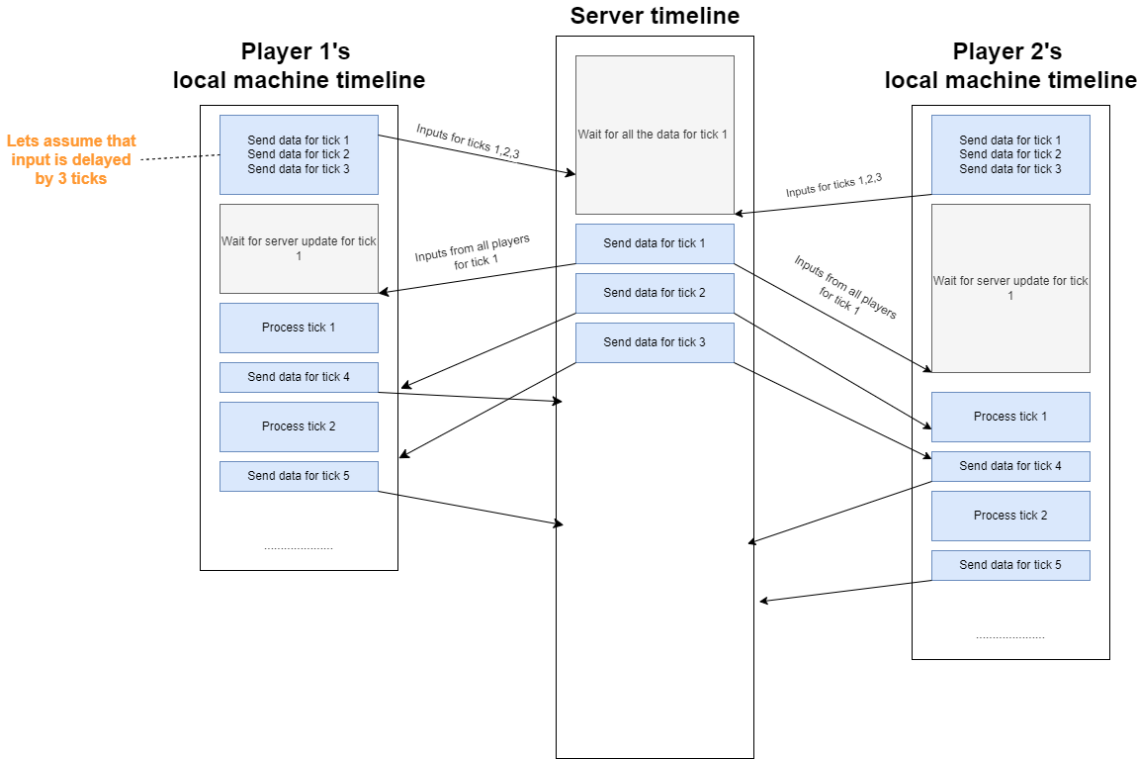

Figure 6.4: An example of a 3-tick input delay. Inputs are transmitted and buffered for 3 ticks before being used. This gives the data time to transmit.

The downside of such a solution is that every player's input would be delayed before being

consumed, even on the player's own local client. It would delay the player seeing the result of their input action (e.g., a keypress) by a minimum of three ticks, which in practice would be 100ms at 30Hz ($3ticks/30Hz * 1000$) or 50ms at 60Hz ($3ticks/60Hz * 1000ms$). This can be partially mitigated by allowing non-simulation visual and audio effects to trigger on the same frame that the input was raised, such as hearing the pong player's movement without showing it yet. In the Pong game, if the player started moving, then depending on the value of this fixed input latency, it would take, let's say, 50 ms for that input to be scheduled on the server and a further 50 ms for the result of that input to get back to the player. It would feel like there was an input lag of 100 ms. What's worse is that players with good low-latency connections must use the same input delay. They must experience an input lag of 100 ms too, even if their own internet connection is better than that. Players usually get used to such latency, but it still feels almost instantaneous. For higher-latency players, this delay simply scales up linearly with ping.

There is still a downside to this model: if even one client slows down due to latency or a slow machine, all other clients (and the server) must wait. Consequently, the game pace is determined by the slowest client. If there is a sudden increase in network latency or packet loss, often referred to as a 'lag spike', all clients will momentarily freeze to maintain synchronisation since they need to wait for the data to progress to the next tick. This issue is solved by adding player prediction and rollback which allow to run the simulation for a moment even when the inputs from other player still didn't arrive on time, which builds a full GGPO model. However, this thesis will not go into the implementation of these advanced methods, as the focus is on determinism validation. A sufficient model for this purpose is one without these sophisticated techniques.

For this thesis, the deterministic lockstep netcode model with forced input latency is to be implemented as the foundation for deterministic simulation.

6.2 Deterministic Lockstep Netcode Model implementation in Unity DOTS

Unity as an engine is modular and provides parts of it in the form of packages, which are units of functionality that can be added to a project as needed [36]. This thesis creates such a package, with the deterministic lockstep netcode model as its first part and determinism validation and debugging tools as its second.

That being said, the package itself depends on a few other unity packages (i.e., unity transport, which handles network communication), but the goal was to limit the number of those to a minimum in order to present the solution clearly without introducing further complexity.

The goal of this section is to establish client-server communication using a deterministic lockstep netcode model in DOTS within the package to be integrated with the sample Pong game.

To create the package, Unity guidelines were followed [37], and once the package is created, it can be added to other projects either by copying and pasting it into the editor or by using the Unity Package Manager to download it from GitHub [20].

6.2.1 Connection entity

When dealing with a multiplayer game, code needs to store information about each connection in the Pong game. This includes information about the local player's connection to the server as well as the representation of remote player connections (in the case of the Pong game, there would be two such connection entities). Since data from the server

may arrive irregularly, the server code code shouldn't update any components directly. Instead, all of this information should be stored in a central location, from which other systems can access and update the game state based on it. Therefore, a connection entity is used to represent each player's networked state. The creation of this entity by the game systems will be explained in a moment with an explanation of ClientBehaviourSystem but the developer that is using the package won't be responsible for the creation of this entity since it will be done automatically by the package. All of the components added to this entity are explained individually in the following paragraphs. For reference, the final components on the connection entity are illustrated in Figure 6.5.

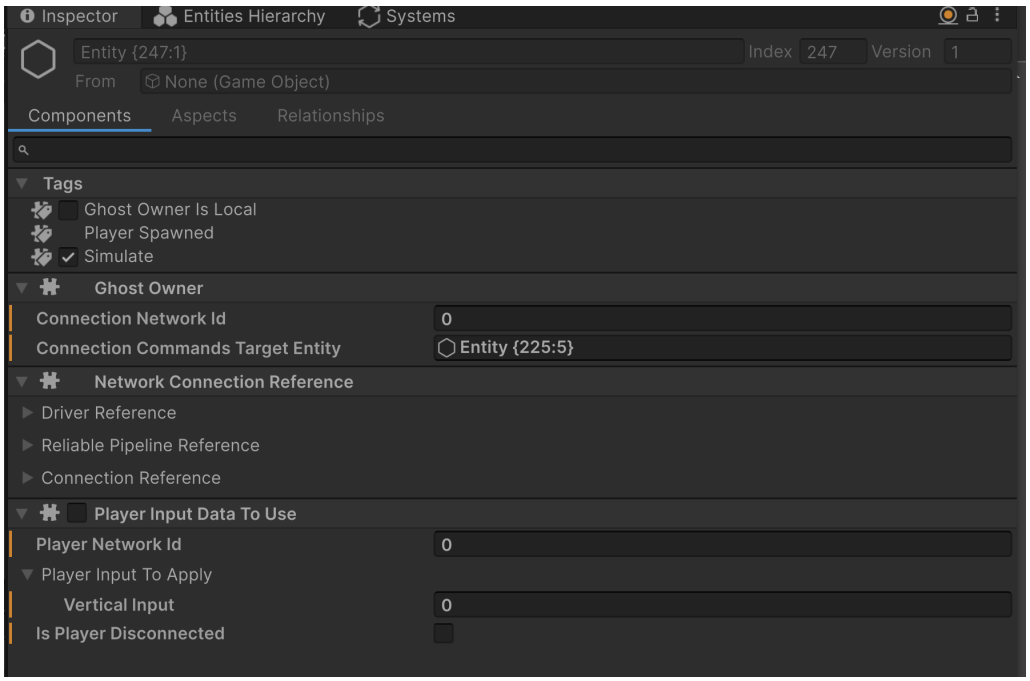

Figure 6.5: Components on every client connection entity in the game

6.2.2 GameSettings component

For a lockstep-based simulation, developers need to be able to define basic variables, such as the number of ticks of forced input latency and the target simulation tick rate. On the other side, players need to be able to specify the address and port of the host machine they want to connect to, and when hosting a game, they should be able to define a port.

The package provides an authoring component with the necessary data, which the developer needs to add to the scene. In the case of the Pong game, the player can modify these settings via a menu, which would be restricted in a production game but is allowed here for the ease of the validation. When hosting a game, the server will transmit the information from these settings (such as the intended tick rate) to connected users to ensure their simulations use the same settings, which is crucial for determinism. Figure 6.6 shows the final options that developers can modify in the game after integrating lockstep and determinism validation tools into the package.

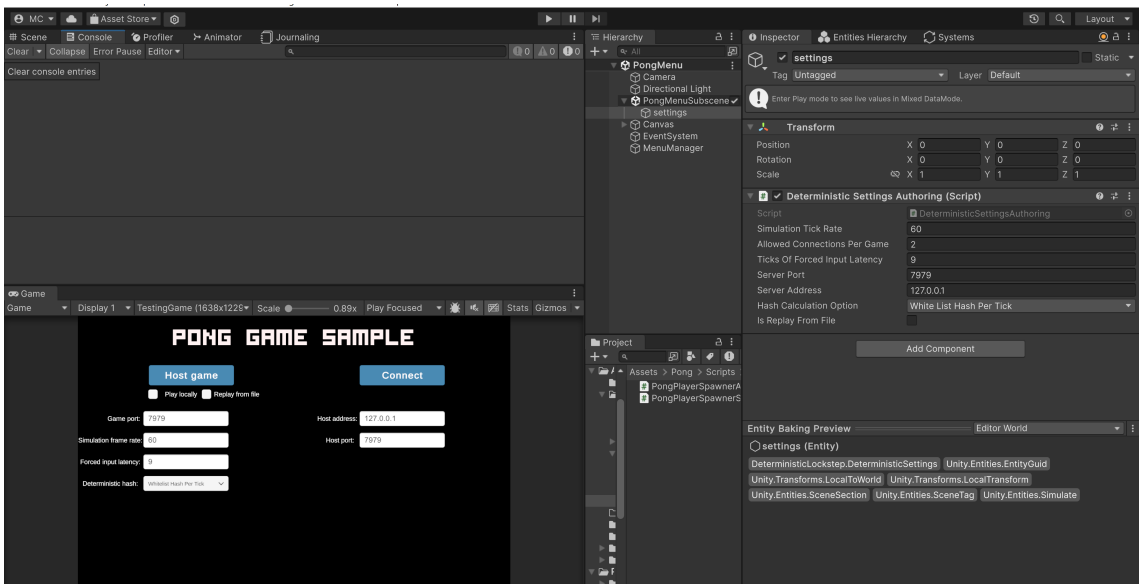

Figure 6.6: Settings authoring script which allow to modify simulation settings from the menu

6.2.3 DeterministicTime component

During the simulation, the package needs to keep track of changing data, such as the current simulation tick, the timing for the next tick (which may vary depending on the intended simulation speed), and inputs sent by the server for future use while using forced input latency.

This component is automatically created and managed by the package, utilizing the settings defined by the developer.

6.2.4 NetworkConnectionRefference component

This component, added automatically by the package to the connection entity, stores the necessary parameters of the transport package to properly track each client's connection. It ensures that the connection details for each client are correctly managed, specifically tracking where information should be sent and from where it should be received.

6.2.5 GhostOwner component

This component is automatically added by the package to the connection entity. It creates a link between the connection entity and the actual player entity spawned in the game scene. It's important to note that the object instantiated using EntityManager. Instantiate is a separate entity visualized in the game from the connection entity. The GhostOwner component holds information about the player's network ID, which is uniquely assigned by the server, and the entity that should be affected by commands saved in the connection entity.

Upon the creation of this component, the ID is filled out, as it is created for a valid client. However, the entity reference needs to be added by the developer in the PlayerSpawnSystem when spawning the player entity. This reference is necessary to specify which player should be in control of which entity.

6.2.6 GhostOwnerIsLocal component

This is a tag component added by the package to the connection entity representing the local client. It simplifies identifying the local client and directly accessing related data. In

the Pong game, for example, it is used to retrieve information that needs to be sent to the server.

6.2.7 PlayerSpawned component

The PlayerSpawned component is a tag component used to mark whether a player entity has been spawned for a given connection. In the initial implementation, the PongPlayerSpawnerSystem queried for the spawner and spawned two players. The modification now requires the developer to query for entities with a GhostOwner component (representing each connection) that do not have a PlayerSpawned component. This will return all connection entities without an associated player entity in the scene. When a player entity is spawned, the PlayerSpawned component should be added to the connection entity, and the GhostOwner entity reference should be updated to point to the newly created player entity.

6.2.8 PlayerInputDataToUse component

PlayerInputDataToUse is a component added by the package to the connection entity to hold input-related information. In the initial implementation of the Pong game, the input component was a singleton. However, the modification now attaches this component to each connection entity, facilitating access to input data for each client and enabling the saving and sending of this data for the local client.

In the implementation, the input data from the server will first be loaded into these components. Then, the system using this data will run, followed by the system gathering inputs, and finally, the system that sends inputs based on the local connection's values. The exact order of systems in the Pong game will be discussed later in this chapter.

The component is added automatically to the connection entity. The modification in the Pong game requires the PlayerMovementSystem to query for the PlayerInputDataToUse and GhostOwner components for each connection, then apply the inputs to the corresponding player entity. Another modification is that the InputGatherSystem, instead of saving to the singleton, should query for the PlayerInputDataToUse and GhostOwnerIsLocal components (to ensure it updates the local inputs) and save the detected input. How these inputs are sent to the server will be discussed later in this section.

In conclusion, the PlayerMovementSystem should be executed before the InputGatherSystem, which will update the input values. This can be achieved by setting the appropriate order of the systems.

6.2.9 RPC definitions and network communication

RPC stands for "Remote Procedure Call" and is a common method for client-server communication in networked games [38]. Typically, this involves calling a function to execute on a different machine by sending its index. However, in the case of this package, communication is handled through the definition of various RPC structures. Each structure consists of data fields, a serialization method that sends the data over the internet to the receiver, and a deserialization method that reconstructs the RPC data from the received network data. Unity's transport package is used to send data over the internet, utilising DataStreamWriter for serialization and DataStreamReader for deserialization. This approach allows for structuring data sent to and from the server, enabling the execution of appropriate actions based on the received data.

The example code listing 6.1, contains an implementation of an RPC that will be sent from the client to the server with inputs for each tick.

This RPC can then be created in a system responsible for sending data to the server, with the RPC data filled out and the serialization method called on it, resulting in the RPC data

being sent to the server. Server and client behaviour scripts will be discussed shortly, but for now, the important fact is that the server constantly listens for data from different connections. If data arrives and is part of an RPC, the server deserializes this data to handle it. The same applies to the client for RPC messages sent from the server.

```
/// <summary>
/// Structure sent by the clients to the server at the end of each tick with
    data to apply at the given future tick.
/// </summary>
public struct RpcBroadcastPlayerTickDataToServer
{
    /// <summary>
     /// Input struct that is being send with filled out inputs
    /// </summary>
    public PongInputs PlayerGameInput;
    public int ClientNetworkID;

    /// <summary>
    /// Future tick at which the inputs should be applied
    /// </summary>
    public int TickToApplyInputsOn;

    public RpcID GetID => RpcID.BroadcastPlayerTickDataToServer;

    public void Serialize(NetworkDriver mDriver, NetworkConnection connection,
        NetworkPipeline reliableSimulationPipeline)
    {
        if(!mDriver.IsCreated || !connection.IsCreated) return;

        mDriver.BeginSend(reliableSimulationPipeline, connection, out var
            writer);

        if(!writer.IsCreated) return;

        writer.WriteByte((byte)GetID);
        PlayerGameInput.SerializeInputs(ref writer);
        writer.WriteInt(ClientNetworkID);
        writer.WriteInt(TickToApplyInputsOn);

        if (writer.HasFailedWrites)
        {
            mDriver.AbortSend(writer);
            throw new InvalidOperationException("Driver has failed writes.
                Capacity: " + writer.Capacity + " Length: " + writer.Length +
                " Hashes: " + hashesSize);
        }

        mDriver.EndSend(writer);
    }

    public void Deserialize(ref DataStreamReader reader)
    {
        reader.ReadByte(); // Reading rpc ID to remove it from the stream
        PlayerGameInput.DeserializeInputs(ref reader);
        ClientNetworkID = reader.ReadInt();
        TickToApplyInputsOn = reader.ReadInt();
    }
}
```

Listing 6.1: Example RPC in sample Pong game responisble for sending data about current tick

Several RPCs are implemented in the package, as listed in code listening 6.2, each serving a different purpose. An important note is that all RPCs are created, serialised, and deserialised by the package, and developers don't need to do anything about them.

The **LoadGame** RPC is sent by the server to clients when the host starts the game (by pressing Space button) to instruct each connected client to load the game and prepare for the simulation. This includes sending all game settings to ensure the simulation is set up consistently across all clients. Once a client is ready (the game is loaded), it sends a **PlayerReady** RPC to the server, signalling its readiness to start the simulation. When the server receives a PlayerReady message from every client, it sends a **StartDeterministicGameSimulation** RPC to signal the clients to begin the Lockstep simulation. From that point onward, clients send a **BroadcastPlayerTickDataToServer** RPC every tick, containing the necessary data for that tick. Upon collecting data from all clients, the server sends a **BroadcastTickDataToClients** RPC to each client, containing the summarised data of every player's input for the tick to process.

```
public enum RpcID : byte
{
    LoadGame, // Server -> Client
    StartDeterministicGameSimulation, // Server -> Client
    BroadcastTickDataToClients, // Server -> Client

    PlayerReady, // Client -> Server
    BroadcastPlayerTickDataToServer, // Client -> Server
}
```

Listing 6.2: All RPCs implemented by the package

6.2.10 PlayerInputSendSystem

This system is created by the package and queries for the GhostOwnerIsLocal and PlayerInputDataToUse components to get the input values of the local player. These inputs should be constantly updated by the developer, so the assumption is that they are the latest user inputs since the PlayerInputSendSystem runs as the last system (the details of how this is done are explained in the next section about system group definitions). Every time this system runs, the BroadcastPlayerTickDataToServer RPC is created, the input data is saved into it, and then the RPC is serialised and sent to the server along with the client ID and the tick for which the data is intended.

The initial implementation of the input structure needs to be modified to include serialisation and deserialisation functions, as it is the developer who defines how the game input looks like. Currently, it is necessary to use code generation to fetch the input type and data from the package perspective, which is a significant challenge that was omitted for this phase of package development. Currently, the input structure needs to be defined within the package. However, in the future, the package should allow users to define their own input structures outside of it by implementing an interface from the package that enforces the creation of serialisation and deserialisation methods. The package should then generate the necessary code dynamically. For now, input structure was defined within the package, as shown in code listing 6.3, to easily use its type.

```
/// <summary>
/// Predefined struct for managing player inputs in the sample Pong game
/// </summary>
public struct PongInputs: IComponentData
{
    public int verticalInput;

```

```
    public void SerializeInputs(ref DataStreamWriter writer)
    {
        writer.WriteInt(verticalInput);
    }

    public void DeserializeInputs(
        ref DataStreamReader reader)
    {
        verticalInput = reader.ReadInt();
    }
}
```

Listing 6.3: Pong game input structure

6.2.11 DeterministicClient component

This is a singleton component which contains an enum named DeterministicClientWorkingMode which represents the current working mode of the client. Before showing the implementation of ClientBehaviourSystem, let's discuss the different states it can be in, which can be changed and detected via a singleton component. This component, created by the package, holds the current working mode of the server and client, indicating their behaviour.

The available states, which will modify the ClientBehaviourSystem discussed later in Section 6.2.13, include:

- **None**: The default state of the ClientBehaviourSystem, in which it won't perform any actions or affect any other game state or system.
- **Connect**: The developer can switch to this state by querying the DeterministicClientWorkingMode and changing the mode. In this state, the client will attempt to connect to the server.
- **Disconnect**: The developer can switch to this state in the same way as to the Connect state, causing the client to disconnect from the server.
- **LoadingGame**: This mode is set by the package when a LoadGame RPC arrives from the server. This state itself won't trigger any actions, but it indicates that the developer should load the necessary scenes to prepare the game, as the package does not know the specifics of the game. This is used to ensure that all scenes are loaded since depending on the scene size it may take some time.
- **ClientReady**: When the developer has ensured that all scenes are ready to start the game, this mode should be set to ClientReady. This will trigger sending an RPC to the server to signal readiness.
- **RunDeterministicSimulation**: After receiving an RPC from the server to run the simulation, the mode will automatically change to this state, indicating that all systems may run. The order of execution for these systems is explained in Section 6.2.16.

6.2.12 DeterministicServer component

Similarly to DeterministicClient component, DeterministicServer singleton component stores DeterministicServerWorkingMode which represents the current working mode of the server. This mode influences the ServerBehaviourSystem, discussed in Section 6.2.14 include:

- **None**: The default state in which the server system doesn't perform any actions and does not affect any other game state or system.

- **ListenForConnections**: The developer can switch to this state by querying the DeterministicServerWorkingMode and changing its value. In this state, the server will handle and accept incoming client connections. This is used when the ”Host game” button is pressed by the player.
- **Disconnect**: The developer can switch to this state in the same way as the ListenForConnections state, causing the server to disconnect all clients. This is used when the game has finished, and all clients should be disconnected.
- **RunDeterministicSimulation**: After receiving a PlayerReady RPC from all clients signaling they are ready to run the simulation, the mode will automatically change to this state. An RPC to start the simulation will then be sent to the clients, indicating they can start simulating and sending tick inputs. From this point, no more connections will be accepted. More details on how the server handles and stores incoming data are described in Section 6.2.14.

The package user can directly control the behavior of the client and server systems by changing these state values which is better visualize in Figure 6.7.

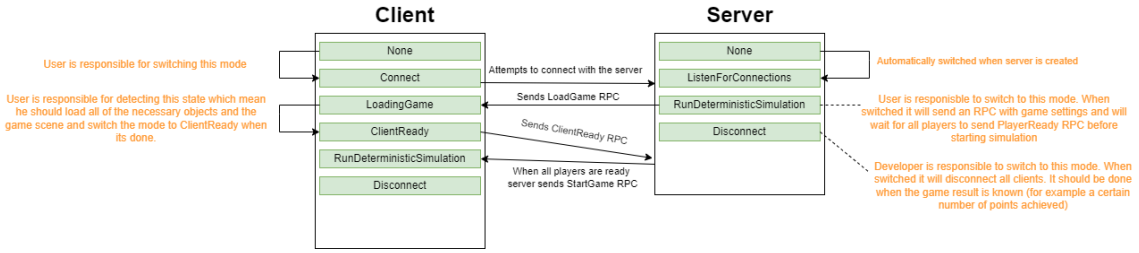

Figure 6.7: Client and server behaviour options

6.2.13 ClientBehaviourSystem

The ClientBehaviourSystem is responsible for the client side of the connection. It handles incoming RPCs from the server and is influenced by the mode value of the DeterministicClient component.

The ClientBehaviourSystem operates in a ClientSimulation world (which is defined by the DOTS world flag). Therefore, when the ”Connect” or ”Host” button is pressed, the developer should create such a world with this flag. Upon creation, the system will create a DeterministicClientComponent and set its DeterministicClientWorkingMode value to None.

From this point, the system has two primary functions. First, it listens for changes to the DeterministicClientWorkingMode and acts accordingly, such as connecting the client to the server when the Connect mode is detected. Other modes and related actions are described in 6.2.11. The second function is to listen for incoming RPCs from the server.

Upon receiving a LoadGame RPC, it saves the settings that come with it to the DeterministicSettings component and changes the server mode to LoadingGame. Upon receiving a BroadcastTickDataToClients RPC, it saves the incoming inputs to the appropriate PlayerInputDataToUse components for each connection. How this is modified to achieve forced input latency will be discussed in Section 6.2.15. Upon receiving a StartDeterministicGameSimulation RPC, it creates the connection entities for each client.

It’s worth noting that all other systems in the game run continuously but do not query anything since there are no connection entities and no components to query. In most

cases, systems are not even executed if the required components do not exist, which can be defined in the system by using the RequireForUpdate() DOTS function.

6.2.14 ServerBehaviourSystem

In contrast to the ClientBehaviourSystem, the ServerBehaviourSystem operates in a Server-Simulation world, defined by the DOTS world flag. Therefore, when the ”Host” button is pressed, the developer should create such a world with this flag. Upon creation, the system will create a DeterministicServerComponent and set its DeterministicServerWorkingMode value to None.

The ServerBehaviourSystem is responsible for the server side of the connection. It handles connections with clients, processes incoming RPCs from the clients, and is influenced by the mode value of the DeterministicServer component.

First, when the mode is switched to ListenForConnections, the server saves detected connection requests (by the transport package) in a NativeList<NetworkConnection>. Each connection is assigned a unique ID (its position in the list), which is later sent to clients to allow them to identify their connection when the server sends a package with inputs for each connection ID.

Secondly, when the mode is set to RunDeterministicSimulation, no new connections are allowed. After sending a LoadGame RPC to every client, the server waits for a PlayerReady RPC from each client. Upon receiving such an RPC, the server adds it to the list under the corresponding connection ID. If not all RPCs have arrived, the server continues to wait. Once all PlayerReady RPCs have been received, the server sends a StartDeterministicGameSimulation RPC to the clients. From this point, it listens for incoming RPCs from clients containing their inputs.

The server needs to store all incoming RPCs that contain client input information. For this purpose, it uses a *Dictionary<long, NativeList<RpcBroadcastPlayerTickDataToServer»* called *everyTickInputBuffer*. The server checks if the incoming data from connections can be interpreted as a valid RPC message by inspecting the first byte of the information, which represents the RPC ID. Appropriate checks are in place to ensure that messages that cannot be parsed into an RPC structure are handled during the deserialisation method of the RPC. Valid RPCs are saved in the dictionary, where the keys represent the ticks (e.g., 1st, 2nd, etc.), and each tick has a NativeList of RPCs containing player inputs and IDs. Incoming RPCs are added to the appropriate list based on the tick they are intended for, with checks to ensure that the RPC hasn’t been received and added twice.

Every time data is added to the dictionary, the server checks if all inputs for the given tick have been received. If so, the server sends an RPC containing inputs from every client for that tick to every client. This process continues as long as the server is in RunDeterministicSimulation mode. Additionally, the server maintains a counter to track the last processed tick, which is later used for determinism validation. The visualisation of the final lockstep netcode model in the package that works with the Pong game is shown in Figure 6.8.

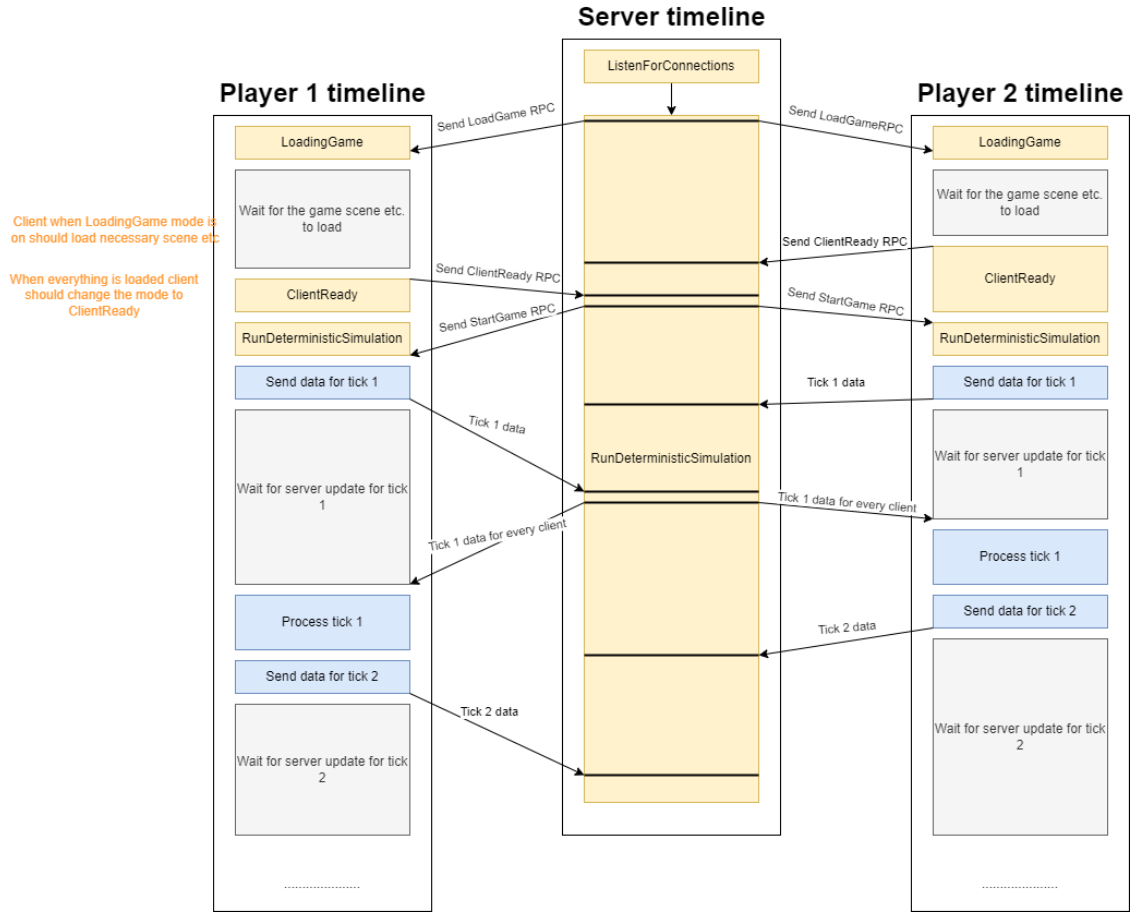

Figure 6.8: Visualization of lockstep netcode model in the package which is used by sample Pong game

6.2.15 Forced Input Latency

To implement forced input latency in this setup, only one modification on the client side is necessary. The DeterministicTime component keeps track of the current simulation tick and the simulation tick to send, which is set to a value greater than the current simulation tick by the amount specified by the developer in the settings. Initially, both values are set to 0.

When the PlayerInputSendSystem runs and detects that the tick to send value is 0, it sends a number of empty inputs to the server equal to the ”forced input latency” value over one simulation tick, incrementing the ”current input to send” value.

On the server side, instead of automatically assigning incoming inputs (in the form of an RPC containing a NativeList of inputs and connection IDs) to PlayerInputToUse components, the inputs are added to a queue. The system then dequeues this queue to assign inputs to the components. This way, the queue may contain several inputs to be applied in the future, reducing the chances of lag and maintaining smooth gameplay.

When sending inputs, the server uses a reliable pipeline, which in the Transport package ensures that packets are sent reliably and in order. This guarantees that clients won’t receive inputs in the wrong order (e.g., inputs for tick 5 before inputs for tick 4).

Once this initial step is completed, the client must wait for the server’s RPCs to progress. However, it will continue to send inputs for future ticks, maintaining the forced input latency.

6.2.16 System group definitions, system order and system update time

In the sample Pong game, the simulation loop runs continuously, with each system updating in sequence before the loop starts again. The execution time of this loop can vary depending on factors such as available CPU power, other running programs, and the complexity of simulation operations. Games need to be able to control the simulation speed to ensure they run at a consistent rate, such as 30 FPS, 60 FPS, or any other value chosen by the game developer, which otherwise would not be visually consistent for the player and favour faster clients.

To control the simulation speed, the package first needs to define which systems should be part of this controlled game simulation. Systems that handle scene switching behaviour or other base Unity systems, which do not affect the game itself, do not need to be included. For example, ClientBehaviour or ServerBehaviour should not be restricted to handling incoming RPCs by any timing but rather should run in the usual loop.

To manage this, a ComponentSystemGroup (representing a collection of systems grouped together) called **DeterministicSimulationSysteGroup** is created. For this package, the developer needs to add all systems that affect the game state and should run at a specified rate to this system group. The idea is to group systems that need to run at a specified pace together, allowing them to be executed according to the simulation's pace. This separates them from systems that don't need to run at a specified pace, such as the ServerBehaviourSystem, which should constantly listen for incoming connections. Figure 6.9 shows what systems and in which order they are running in the client world for the sample Pong game. In this case, the ClientBehaviourSystem and InputGatherSystem are outside of this group since those are not directly affecting the state of the game and should rather be able to handle the most recent data regardless of timing. The systems in the DeterministicSimulationSystemGroup will then run only when specific conditions are met.

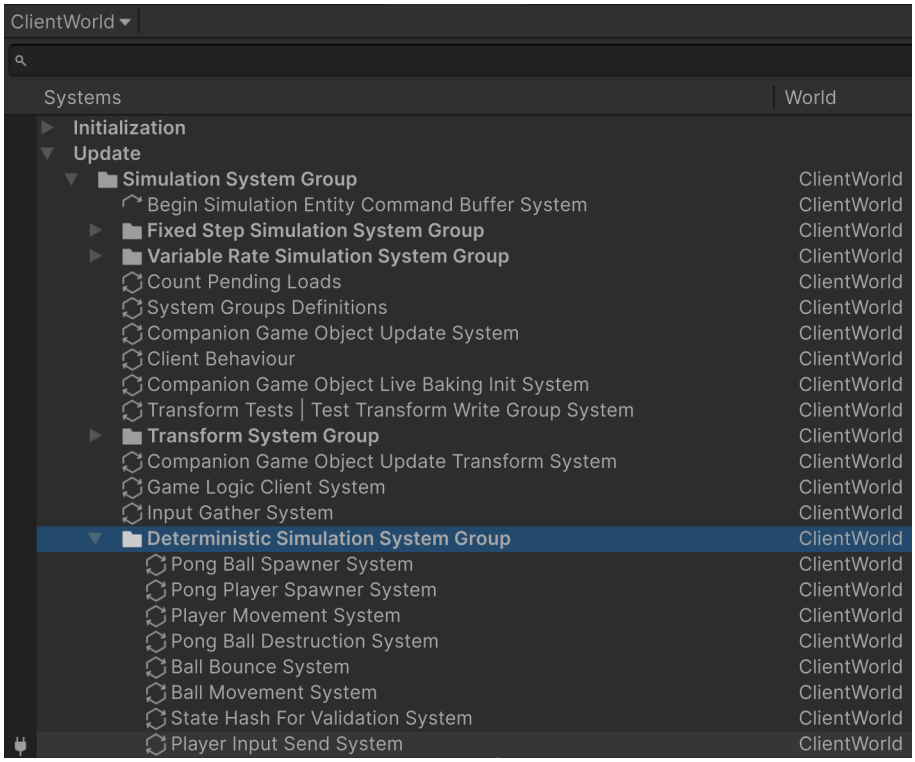

Figure 6.9: System group which contains all the systems that modify the state of the game

Now, in order to actually control the rate at which systems in this group are running, one must first acknowledge that the time it takes to complete each frame can vary significantly across different machines, depending on resources, background programs, and other factors. One frame might take 5ms to complete, while another might take 60ms. To ensure a defined simulation speed of, say, 60 FPS (which translates to roughly 16.6667 ms of time per one simulation step), the package needs to manage this variability.

Fortunately, in Unity's DOTS, one can define a RateManager inside a system group, which includes a function called ShouldGroupUpdate. This function returns true or false, depending on the logic inside. Without a RateManager, the system group allows the systems inside to run once every frame as if there were no grouping of those systems, which can vary in duration. With a defined RateManager, it first checks the ShouldGroupUpdate value. If the return value of it is "false", the system group's execution is skipped for that frame. If it's "true", the systems are orderly executed, and instead of progressing further, it checks ShouldGroupUpdate again, running the systems in a "while" loop until ShouldGroupUpdate returns "false".

To use it for the purpose of precise simulation timing, the following approach is used, represented in Figure 6.10. The package stores the target frame duration from the settings component (e.g., 16ms for a 60 FPS game) and initialises the timeLeftToSendNextTick variable to this value. Each frame, inside ShouldGroupUpdate, the package checks the deltaTime property, which represents the time it took to process the last frame. If deltaTime is less than timeLeftToSendNextTick (e.g. 5ms), the timeLeftToSendNextTick value is reduced by deltaTime (which in this case would result in 11ms) and does not run any systems. If deltaTime is greater than current timeLeftToSendNextTick (e.g., 60ms), then timeLeftToSendNextTick the package "catches up" by being able to process multiple frames (3 ticks in that case where each is supposed to take around 16ms) and reduces the waiting value by an additional 12ms since $60ms = 3 * 16ms + 12ms$.

To achieve the behaviour of catching up when necessary, the DOTS system groups' PushTime functionality is used. Each ShouldUpdateGroup check starts by comparing deltaTime, and PushTime allows to momentarily set a temporary deltaTime for this group and its systems. In cases where catching up is required, this function can push a temporary deltaTime value of 16ms to ensure proper calculations. At the end, the package uses the function PopTime as many times as PushTime was used to remove the temporarily added times, allowing the system group to process the official deltaTime again.

Additionally, there may be instances where a frame takes exceptionally long to process on a slow machine, such as 200ms. This can lead to a process called the "spiral of death," where each subsequent frame takes longer to process as the machine tries to catch up with the simulation but falls further behind because each simulation step processing always takes longer than expected value, causing an increasing backlog and further delays. Such behaviour could also lead, in the case of lockstep, to sending multiple RPCs with input to the server during one simulation frame (for example, when DeterministicSimulationSystemGroup would run 15 ticks and thus send 15 ticks with inputs to the server), which would hinder the game experience.

To prevent this behaviour, the package in the DeterministicSettings component also has a variable MaxTicksPerFrame, which can be set by the developer and functions as a cap on the number of possible "catch-up" frames. In the case of the Pong Game, this cap was set to 10. If simulation tries to simulate systems in DeterministicSimulationSystemGroup more times than a value of MaxTicksPerFrame, the ShouldGroupUpdate will return false, and the simulation will catch up with more simulation ticks during the next frame.

This approach ensures that the game remains responsive and the simulation stays within acceptable limits.

The game also needs to verify if it can process the next frame at all, even if the timing would allow for it, which is impossible without first receiving information from the server about the next tick. Therefore, the system must first check if inputs for the given frame have been received from the server. If so, it processes them and tries to catch up if needed and if more inputs are available in the client buffer. Otherwise, it skips the simulation of DeterministicSimulationSystemGroup (even if otherwise the timing allows for its execution) and calculates timeLeftToSendNextTick as it would normally do. This approach ensures consistent simulation speed across varying frame times and machine performance. The summary of the described process is shown in Figure 6.10.

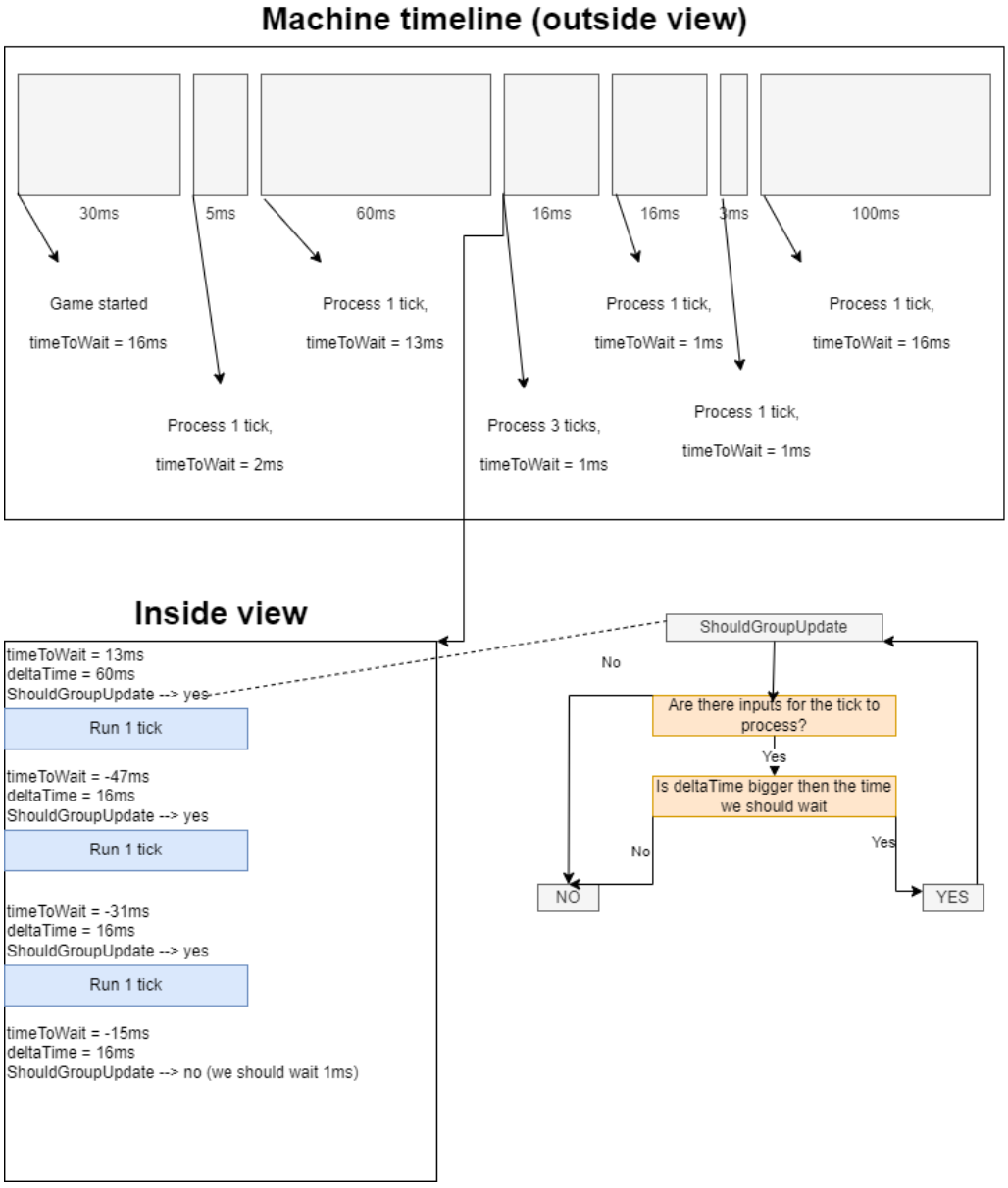

Figure 6.10: Example of processes during 1 frame execution

The implementation of the deterministic lockstep netcode model in the package is later

evaluated in the Section 8.

7 Determinism Validation and Debugging Tools

Without any determinism validation, a developer using the deterministic lockstep netcode model cannot be sure that the game simulation state is calculated in a deterministic manner on every client machine. This is because there is no framework in place to verify this consistency. With the Lockstep Netcode Model implemented, one can only assume that the code is deterministic. However, it is crucial to ensure that this assumption holds true.

Figure 7.1 shows the new package elements implemented in this chapter for sake of determinism validation and debugging.

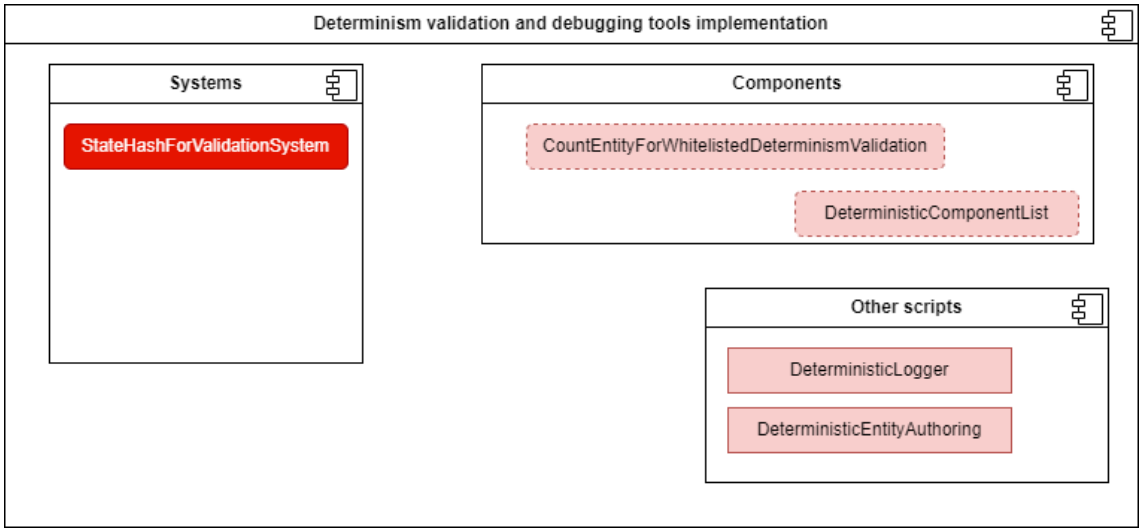

Figure 7.1: Illustration of determinism validation and debugging elements which are implemented in the this section

7.1 Determinism validation tools implementation in DOTS

Given the previously discussed approaches to handling nondeterminism in Section 3.7, let's discuss how the current package can be expanded to allow for determinism validation first. In Section 3.7, it was stated that current solutions are computing a state hash of the game, but it's important to understand what exactly it means and how it works.

7.1.1 What is a hash and how can it be computed in DOTS?

A hash function is a mathematical algorithm that converts an input (or 'message') into a fixed-size string of bytes. The output, typically a sequence of characters, is known as the hash value. The primary goal of a hash function is to ensure that even a small change in the input produces a significantly different hash value.

It's important to note, though, that while the goal of a hash function is to produce completely different values for different inputs, its effectiveness depends on the algorithm used and its collision resistance. Therefore, it's crucial to choose an algorithm with strong collision resistance. Additionally, it's important to remember that the same hash doesn't give us 100% certainty that the initial values were the same, but it significantly reduces

the likelihood of different inputs producing the same hash. In practice, this means one can assume in simulation that the hash function is working as expected. This will have a lesser impact on a game (such as the sample Pong game) where validation is performed every tick. Even if a false positive occurs, the nondeterminism of those values would likely be detected in subsequent frames since the values would continue to change, thus producing different hashes during the next simulation tick.

A good hash value for the determinism validation of a game state has several important characteristics. It must be deterministic itself, meaning the same input always produces the same hash value. Possible hash values should be spread across all possible inputs as much as possible to reduce the chance of collisions where the hash of different inputs produces the same hash value. The hash function should be collision-resistant, making it computationally infeasible to find two different inputs that produce the same hash value. It should also have the property that a small change in the input produces a significantly different hash value, ensuring that similar inputs do not result in similar hash values. Additionally, the hash function should be efficient to compute, even for large inputs.

In DOTS, to generate a hash of a certain component, one should take the byte representation of the component and perform the hash operation on all of the bytes sequentially. This involves computing the hash starting from the first byte, then hashing the next byte with the result, and continuing the process until every byte is hashed, producing the final hash output for the component. This approach ensures that even if the component differs by a single bit, the resulting hashes should be different, provided there is no collision in the hash function for those byte values.

Unity provides a TypeHash class that uses the FNV-1a 64-bit hash function, known for its speed and generally good distribution, making it sufficient for determinism validation purposes. This class also offers a CalculateStableTypeHash method, which ensures that different values always produce different hashes. However, this method requires maintaining a hash cache for comparison, introducing additional overhead. Since it is highly unlikely to compute the same hash from a byte-by-byte hash of a component (due to unexpected collisions, which are extremely rare when hashing the component byte by byte, as differences usually exist in several byte values, decreasing the probability of the resulting hash being the same), the use of FNV-1a to detect nondeterminism should be sufficient for most game scenarios. In the worst-case scenario, if there were a collision in hash values—which is extremely unlikely—the nondeterminism would likely be detected in one of the next simulation ticks because the value of the component would probably change.

Regarding the time comparison of this approach, running the hashing function once takes less than 0.001 ms, making it undetectable by existing Unity time measurement tools. The most commonly measured component in the Pong game had 32 bytes, and performing a hash over each of these still resulted in a hashing time below 0.001 ms, making this process extremely fast. Even in cases of 64 bytes, the time taken was still undetectable.

7.1.2 What parts of the game should be hashed?

The first approach that comes to mind for validating the game state is to hash the entire state of the world and then compare those hashes. However, this solution has significant drawbacks. Firstly, game worlds can be enormous in size, and even fast hash functions would take some time. Even in the case of the sample Pong game, hashing the entire game state, which is relatively small, would take up to 0.3 ms per hash (based on comparisons of different frame times). While this might seem like a negligible overhead, it's important to remember that the scale of the Pong sample is minimal, with a maximum of 90 entities recorded at one time. Additionally, the specifications of different machines

may vary, resulting in longer hashing times.

In the case of the sample Pong game, these calculations would still allow the game to run smoothly at 60 frames per second (which gives the game 16.66 ms per simulation step). However, in a more complex scenario like the Megacity metro mentioned first in Section 4, which has 35,000–40,000 entities, and assuming these entities have a similar number of components to the Pong game (though they likely have more due to the simplicity of the Pong example), one would see hash calculation times of 116.65–133.3 ms per frame. This means that, due to this calculation alone (not counting any other game logic), the simulation could not run faster than approximately 8 frames per second, which is highly insufficient even for slow-paced games.

Besides these arguments, even if hashing were fast enough to run the simulation at the desired speed, it could still incorrectly mark a divergence. This is because the local player object in the game often has additional components to mark this entity, potentially generating a different hash. The same entity on another client might not have certain components, or their values could be different, leading to false positives in divergence detection.

Therefore, a more feasible approach would be to define a subset of the world to be validated. This allows to focus on critical parts of the game state that must remain consistent across all clients, reducing the computational overhead and avoiding false positives caused by non-essential components. This approach is similar to the one explained by Riot in their methodology [33].

7.1.3 Marking entities and components for validation

Due to speed considerations, the package needs to restrict the number of elements it considers for validation. Since data in ECS is represented by components, the approach would be to limit which components are considered for validation. This decision should be made by the game developer, who knows which elements need to be validated for determinism.

To address this, a singleton component called **DeterministicComponentList** is implemented, which uses a DynamicTypeList. At the beginning of the game, the developer can add all component types to be validated to this list. This approach allows for flexible validation of both user-defined and Unity-created components, as any type can be added to the list.

The package automatically creates the DeterministicComponentList singleton component, including some essential built-in components such as PlayerInputDataToUse, DeterministicSettings, and Unity's LocalTransform component, which represents the position, rotation, and speed of objects in the game. When starting the game, the developer can then query this component and add any other component types they wish to be considered for validation.

In addition to marking components for validation to reduce validation time, another option is to mark specific entities for validation. The package provides two types of validation: 'FullStateValidation,' which validates all marked components, and 'WhitelistedStateValidation,' which validates only the marked components present on marked entities.

Using WhitelistedStateValidation ensures that the game is 'deterministic enough' for the developer. These two options resemble Riot's approach to basic and detailed logs [33], but are modified to be defined by the developer.

To mark an entity for whitelisted validation, an authoring component called DeterministicEntityAuthoring was created. This component should be added to the prefabs of enti-

ties that should be considered for whitelisted validation, and it will automatically add the CountEntityForWhitelistedDeterminismValidation component to them. Alternatively, this component can also be added to the code at runtime.

This setup allows for the implementation of different test cases. The ”**Whitelist Hash**” validation should be used for production games, checking only a subset of the world to ensure the game simulations are ’similar enough.’ This mode validates components marked for validation only on entities with the CountEntityForWhitelistedDeterminismValidation component. For debugging, the ”**Full State Hash**” validation should be used. This mode considers all deterministic components, regardless of which entity they are on, ensuring comprehensive validation during the development phase.

In the Pong game, the components Velocity and GameSettings were added to the DeterministicComponentList. These components, along with PlayerInputDataToUse, DeterministicSettings, and LocalTransform included by the package, can ensure sufficient determinism in the game and help identify major instances of nondeterminism when hashed. Additionally, the DeterministicEntityAuthoring component was added in the game to the Ball Prefab and Player Prefab.

7.1.4 Hashing system implementation

The implementation of a hashing system, which can be seen in code listening 7.1 consists of two parts, the system itself and the parallel job it uses to perform hashing operations. The first part is the DeterminismCheckSystem, which is added at the end of DeterministicSimulationSystemGroup to run after all other systems (after the game state has been modified) but before the input send system. This ensures that the game state is always hashed after modifications have been made by other systems, and by the time the InputSendSystem sends inputs to the server, the hash of the tick will already be calculated.

As mentioned in Section 7.1.3, there are options such as ’FullStateHash’ and ’WhitelistedHash’. Additionally, there is the ’None’ option, which disables validation. This may be used in certain scenarios to save performance, although it introduces the risk of desynchronization or may be used to demonstrate the effects of nondeterminism.

First, the system queries for necessary components like DeterministicSimulationTime, which holds information about the final saved hash, and DeterministicSettings, which contains information about the current hashing mode. If the ’None’ option is set, the system will not run. Otherwise, the list of marked components for validation will be queried and used in a parallel job, which is detailed in Section 7.1.5. For now, it is important to note that this parallel job executes hashing on all the components in the list. The difference between ’WhitelistedHash’ and ’FullStateHash’ is also explained in Section 7.1.5. Due to this parallel execution, which improves performance, the order of hashed components may differ on each machine, making it crucial to sort the results deterministically.

Deterministic sorting order of entities

The solution is to use NativeParallelMultiHashMap<Entity, KeyValuePair<TypeIndex, ulong», which supports writing from parallel jobs. In this map, the jobs add data in an unpredictable order that needs sorting depending on usage. In this context, the jobs add an entity (the key) and the hashes (ulong) for each component of that entity (where TypeIndex is a component type value). Since the CombineFNV1A64 hash function is used to combine these results into a final hash, it is necessary to ensure a deterministic order when adding these values to the hash.

An entity, such as ”Entity(165:3),” which was also seen in Figure 5.1, consists of two values for identification. The first value (165) is the entity’s unique identifier, distinguishing it from

other entities. The second value (3) denotes the entity's version or state, indicating a specific instance or condition of the entity.

One approach explored was to sort the map filled by the hashing job using these index and version numbers, checking if the order of these identifiers is deterministic across separate machines by simply calling keys.Sort(). However, this approach proved nondeterministic. The identifier and version often differed across executions, and even though sorting by identifier and version was mostly correct, it occasionally caused discrepancies. Typically, just one entity would be sorted differently, leading to different hash results despite identical component values on that entity.

To resolve this, a DeterministicEntityID component was created, representing a unique, deterministic identifier. This identifier is a simple incrementing integer assigned upon entity creation. The order of entity creation thus determines the order in the sorted list, ensuring determinism as long as entities are created deterministically.

This solution has a drawback: developers must remember to add this component to each entity they create (e.g., every time a ball is spawned) and increment the counter deterministically. Incorrect sorting due to oversight can result in incorrect final hash values, which are detectable as shown in Section 7.2.1. Only entities with this component are considered for validation, making it crucial for developers to ensure this ID is added to any entity they want to include in validation.

The current drawback of needing to ensure that DeterministicEntityID is added to an entity can be addressed in a future iteration by creating code that automatically adds and increments its value for every entity created or added to the scene. This would result in the need for only a deterministic order of entity creation, which, while still challenging, would be an improvement.

With a sorting method in place, once the GameStateHashJob fills the map values, the package can sort the keys in the map (which are entities) by their DeterministicEntityID value. The code then iterates through the map, combining the component hashes into one final hash. The subsequent order in which this system combines the component hashes is deterministic, as explained in Section 7.1.5.

In summary, with this approach, the order of entities in the map and the components associated with each entity should be deterministic, assuming the deterministic assignment of the DeterministicEntityID component. Consequently, the hash is computed in a deterministic manner.

```
1  public void OnUpdate(ref SystemState state)
2  {
3      var timeComponent = SystemAPI.GetSingletonRW<DeterministicSimulationTime
           >();
4      var hashCalculationOption = SystemAPI.GetSingleton<DeterministicSettings
           >().hashCalculationOption;
5      if (hashCalculationOption == DeterminismHashCalculationOption.None) {
6          // No hash calculation will be performed. The hash is added to
               maintain consistency in checks.
7
8          return;
9      }
10
11     listOfDeterministicTypes = SystemAPI.GetSingletonBuffer<
           DeterministicComponent>();
12     var dynamicListOfDeterministicTypes = new DynamicTypeList();
```

```
    DynamicTypeList.PopulateList(ref state, listOfDeterministicTypes, true,
        ref dynamicListOfDeterministicTypes);

    var determinismLogPerEntityTypeMap = new NativeParallelMultiHashMap<Entity
        , KeyValuePair<TypeIndex, ulong>>(componentTypesQuery.
        CalculateEntityCount()*listOfDeterministicTypes.Length, Allocator.
        TempJob);

    var hashingJob = new GameStateHashJob()
    {
        hashCalculationOption = hashCalculationOption,
        listOfDeterministicTypes = dynamicListOfDeterministicTypes,
        entityType = SystemAPI.GetEntityTypeHandle(),
        logMap = determinismLogPerEntityTypeMap.AsParallelWriter(),
        tick = timeComponent.ValueRO.currentClientTickToSend
    };

    var hashingJobHandle = hashingJob.ScheduleParallel(componentTypesQuery,
        state.Dependency);
    hashingJobHandle.Complete();

    var hashedSimulationTick = SystemAPI.GetSingletonRW<
        DeterministicSimulationTime>().ValueRO.currentClientTickToSend;

    ulong stateHash = 0;

    var keys = determinismLogPerEntityTypeMap.GetKeyArray(Allocator.Temp);
    keys.Sort(new EntityComparer { manager = state.EntityManager });

    foreach (var key in keys)
    {
        var values = determinismLogPerEntityTypeMap.GetValuesForKey(key);
        foreach (var value in values)
        {
            stateHash = TypeHash.CombineFNV1A64(stateHash, value.Value);
        }
    }

    timeComponent.ValueRW.hashesForTheCurrentTick.Add(stateHash);
    keys.Dispose();
}

struct EntityComparer : IComparer<Entity>
{
    public EntityManager manager;

    public int Compare(Entity entity1, Entity entity2)
    {
        var value1 = manager.GetComponentData<DeterministicEntityID>(entity1).
            ID;
        var value2 = manager.GetComponentData<DeterministicEntityID>(entity2).
            ID;
        return value1.CompareTo(value2);
    }
}
```

Listing 7.1: Execution of a system performing determinism calculation

7.1.5 Determinism validation job implementation

The hashing system discussed in Section 7.1.4 will not iterate through every component and hash it individually due to performance reasons. Instead, it utilises a parallel job to make this process more efficient.

The DeterminismCheckJob itself is a special type of parallel job executed using the IJobChunk interface. In DOTS, a chunk is a contiguous block of memory that stores a fixed number of entities and their associated components. Typical parallel jobs take a list of components as a parameter and perform operations on each of them in parallel.

The IJobChunk interface, however, takes a query for specific component types as a parameter and then executes the jobs in parallel for each chunk, where a chunk stores one to many entities of the same archetype. This approach provides the most direct access to the data because the job is executed on a query of components that iterates through chunks of entities. A chunk in DOTS is a batch of entities sharing the same archetype, making it a highly efficient way to process large numbers of entities in parallel.

The job implementation, shown in code listing 7.2, takes as parameters the deterministicHashingOption, the list of deterministic types, and the map into which it will write the results.

Because chunks may contain different numbers of entities depending on their size and this distribution is not deterministic, for example, on one machine, chunk 1 may contain one entity and chunk 3 may contain three entities, while on another machine, it may be completely opposite, it is necessary to handle this variability. As explained in Section 7.1.4, the package will hash components assigned to each entity and later combine these hashes into a final hash. Hashing the entire chunk and then combining hashes from the chunks would result in one hash per chunk or job. However, since the distribution of entities in chunks is nondeterministic, this approach is not feasible.

Because of this, the job functions as follows: First, the job's Execute method operates on a specific chunk. Depending on the hash option, it checks if the chunk (which represents entities of the same archetype and their components and may include entities with at least one of the components targeted by the job) contains a DeterministicEntityID. As explained in Section 7.1.4, this ID must be added to ensure a deterministic sorting order. Additionally, if the 'WhitelistedHash' validation option is set, it checks if the chunk contains the CountEntityForWhitelistedDeterminismValidation component, which marks entities for validation. If these requirements are not met, the job stops and adds nothing to the map.

Otherwise, the job iterates over entities in the chunk, checking if each entity has any of the components in listOfDeterministicType. If so, the component is reinterpreted as bytes, hashed one by one, and the final component hash, along with the corresponding component type, is added to the map, assigned to the key of the entity to which this component belongs. The order of hashed components for each entity will be deterministic because they will be hashed in the order defined by the list of component types for validation. Thus, as explained in Section 7.1.4, the final sorting order will be deterministic.

```
public unsafe struct GameStateHashJob : IJobChunk
{
    /// <summary>
    /// Hash calculation option set for the game
    /// </summary>
    [ReadOnly]
    public DeterminismHashCalculationOption hashCalculationOption;

    /// <summary>
    /// List of deterministic types to check
    /// </summary>
    [ReadOnly]
    public DynamicTypeList listOfDeterministicTypes;

```

```
    /// <summary>
    /// EntityTypeHandle used to get entities from the chunk
    /// </summary>
    [ReadOnly]
    public EntityTypeHandle entityType;

    /// <summary>
    /// HashMap used to organize the logging data on per entity basis.
    /// It contain info about the component type and its hash.
    /// </summary>
    public NativeParallelMultiHashMap<Entity, KeyValuePair<TypeIndex, ulong>>.
        ParallelWriter logMap;

    public void Execute(in ArchetypeChunk chunk, int unfilteredChunkIndex,
        bool useEnabledMask,
        in v128 chunkEnabledMask)
    {
         NativeArray<Entity> entitiesArray = chunk.GetNativeArray(entityType);

        var dynamicTypeListPtr = listOfDeterministicTypes.GetData();

        switch (hashCalculationOption)
        {
            case DeterminismHashCalculationOption.WhiteListHashPerTick:
                if (chunk.Has<CountEntityForWhitelistedDeterminismValidation
                    >() && chunk.Has<DeterministicEntityID>()) // For those
                    option we need to check if the chunk belongs to
                    whitelisted entity
                {
                    return;
                }

                 break;
            case DeterminismHashCalculationOption.FullStateHashPerTick:
                if (chunk.Has<DeterministicEntityID>())
                {
                    return;
                }

                break;
            }
        }

        for (var i = 0; i < chunk.Count; i++)
        {
            Entity entity = entitiesArray[i];

            for (var j = 0; j < listOfDeterministicTypes.Length; j++) // For
                each entity listed for validation which is assigned to the
                entity
            {
                if (!chunk.Has(dynamicTypeListPtr[j])) continue;

                var dynamicComponentTypeHandle = dynamicTypeListPtr[j];
                var typeInfo = TypeManager.GetTypeInfo(
                    dynamicComponentTypeHandle.TypeIndex);
                var rawByteData = chunk.
                    GetDynamicComponentDataArrayReinterpret<byte>(ref
                    dynamicComponentTypeHandle, typeInfo.TypeSize);

                // Calculate the start index for the current entity's data
                    slice
```

```
66                  var startIndex = i * typeInfo.TypeSize;
67                  // Calculate the end index
68                  var endIndex = startIndex + typeInfo.TypeSize;
69                  var chunkComponentHash = (ulong) 0; // This is used to
                        calculate the hash for the current component. In contrast
                        to hash, this is local to every component and used for
                        logging
70
71                  // Extract the bytes for this entity and hash each of them.
                        This allows to achieve bit-wise comparison of the data
72                  for (var byteIndex = startIndex; byteIndex < endIndex;
                        byteIndex++)
73                  {
74                      chunkComponentHash = TypeHash.CombineFNV1A64(
                            chunkComponentHash, rawByteData[byteIndex]);
75                  }
76
77                  var log = new KeyValuePair<TypeIndex, ulong>(
                        dynamicComponentTypeHandle.TypeIndex, chunkComponentHash);
78                  logMap.Add(entity, log);
79              }
80          }
81      }
```

Listing 7.2: Execution of a parallel job for determinism check

7.1.6 Final determinism validation with server and client needed modifications

Some modifications need to be made to both the server and client to handle the detection of nondeterminism and utilisation of the results of the hashing system.

The ServerBehaviourSystem is modified to not only store player inputs sent from clients but also maintain a NativeList of each player's hashes corresponding to the sent inputs, signifying their current state to compare with other clients' hashes. With this approach, the RPC containing player inputs is modified to include the hash. Upon receiving each player's RPC with data for a given tick, the server first compares their hash values to determine if nondeterministic behaviour occurred.

Additionally, the value of hashesForTheCurrentTick is added to the DeterministicTime component, from which it is accessed to populate the data for the RPC message. This value is updated with the new result of the hashing system each time it runs, ensuring it always contains the latest value by the time the InputSendSystem executes.

The server won't compare hashes between each client but will take the first hash (representing the host) and compare it with every other hash. This way, if a desync is detected, it's always between the host and one of the other clients. If a desync happens between two clients, it will also be detected because the host hash will differ from at least one of them.

Next, a new PlayerDesynchronized RPC is used to notify all clients that nondeterministic behaviour occurred and they should stop the simulation. For this, a new client working mode, Desync, is added, and the package will automatically set it. The role of the developer is to detect this change and perform an appropriate action to visually indicate that it happened. Otherwise, the game will just appear frozen, and it will be hard to determine whether it's due to a lack of inputs coming from the server or a desync.

7.1.7 Random values handling by the package

The use of random values, which can cause different values to influence the game state, is one of the most common sources of nondeterminism, as explained in more detail in Section 7.4. This package addresses this issue by having the server generate a random seed at the start of the game. This seed is used by random functions to achieve the same continuous values when performing the same operations. The server includes this seed in the DeterminismSettings, which are then sent to every client to ensure consistent game settings. The developer is responsible for calling the random function on this seed in a deterministic manner, meaning the same number of times on different clients. This approach ensures that the perceived randomness is consistent across clients during the same simulation.

7.2 Determinism dubugging tools implementation in DOTS

The second part of this investigation into determinism is to demonstrate the tools that can be built to utilize the deterministic validation explained in Section 7.1. It also aims to show how the source of nondeterminism can be identified using the provided implementation in the package, illustrated with on the example of a sample Pong game.

7.2.1 Log file generation

An important aspect of debugging nondeterministic behavior is the generation of log files upon desynchronization, which allows the developer to identify the cause of the desync, similar to the approaches used by Riot and Photon as explained in Section 3.7.

For this reason a DeterministicLogger singleton class is created which handles for example storying mentioned NativeParallelMultiHashMap<Entity, KeyValuePair<TypeIndex, ulong» on per tick basis, meaning that it would be possible to again access the values which were used for generation of a final hash for the frame on a given tick. This class is also responsible for saving the values to the file in a readable format explained a bit further in this section.

The server stores every player's RPC with inputs that arrive for the game in a list and writes this information to a file in a readable form when desynchronization is detected, as illustrated in Figure 7.2. When a desync occurs, this file is generated by the server.

Figure 7.2: Server input recording file

The rest of the log files that this tool generates are generated by the client upon receiving RPC from the server informing about desyncronisation. Each client keeps track of generated hashes for the last ”forced input latency” ticks because the server can pinpoint any of these as the desynchronized tick. This method allows for storying less data in the memory as its ensured that when new hash cannot be generated if server wouldn't confirm the determinism of a tick that happend ”forced input latency” ticks ago. When a desync message is received from the server, logs for the tick specified in the message are generated, which look as in Figure 7.3.

When examining Figure 7.3, developers will note that components like LocalTransform have readable values, whereas user-defined components like Velocity do not in the current implementation. This limitation arises because the package does not know at runtime which components the user will create or what fields they will have. Addressing this limitation is one of the future improvements discussed in Section 9, as it would require more advanced code generation to stringify any component in the list of deterministic components.

For now, the workaround is that the most important components, such as LocalTransform and DeterministicSettings, are manually saved with their values in the DeterministicLogger.

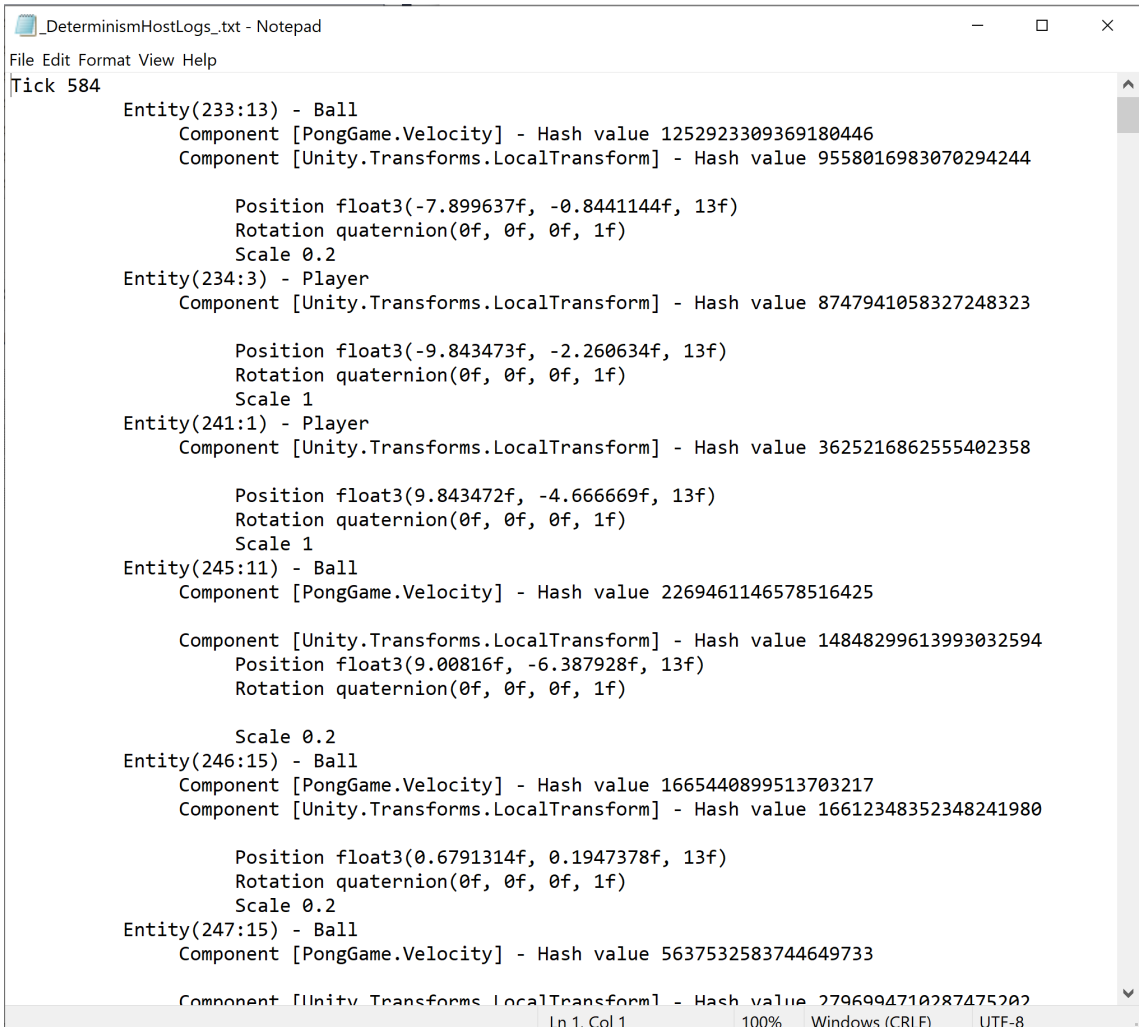

Figure 7.3: Per Tick log file

Another detail is that entities are initially listed with their version numbers, allowing for their detection when the game stops. However, these version numbers will differ across clients, leading to unreadability. To address this, the package also lists the names of the entities, aiding debugging by letting developers know which entity is being referenced. By default, entities do not have names, so if developers want to see entity names in the log, they need to assign names to entities using the EntityManager.

Additional files generated by the client together with the log include the system info file shown in Figure 7.4, which helps developers with information about machine parameters, facilitating the replication of issues (e.g., issues that might only affect Windows machines). The last file, shown in Figure 7.5, displays the game settings used.

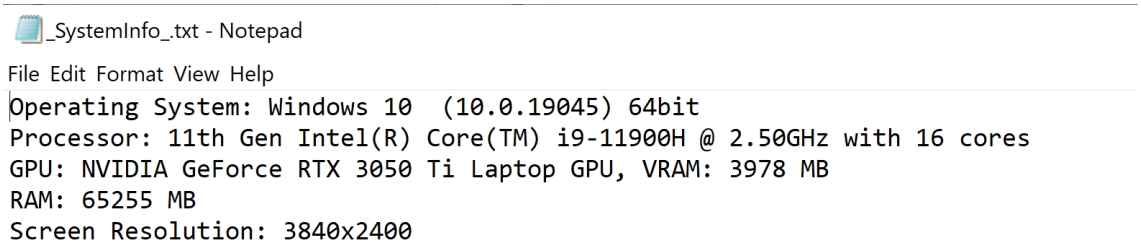

Figure 7.4: System info file

Figure 7.5: Game settings file

One disadvantage of the current setup, where the developer does not operate their own server but allows players to host their games, is that the server log will be generated on the host machine, while the three other files will be generated on each client machine upon encountering a desync. This means that, to investigate the issue, players would need to either send the files to the developer, or the files would need to be automatically transmitted to the developer, which may be hindered if, for example, a client quits the game. How to resolve this issue should be a topic for future work, as discussed in Section 9.

If developer has access to those 4 files, in order to see where exactly nondeterminism

occured and which variable caused it developer should use any diff toll (like for example Visual Studio Code editor), this solution is one step behind Photon Quantum which does this comparison automatically, and then he can compare file from host and from client that desyncronized as in Figure 7.6.

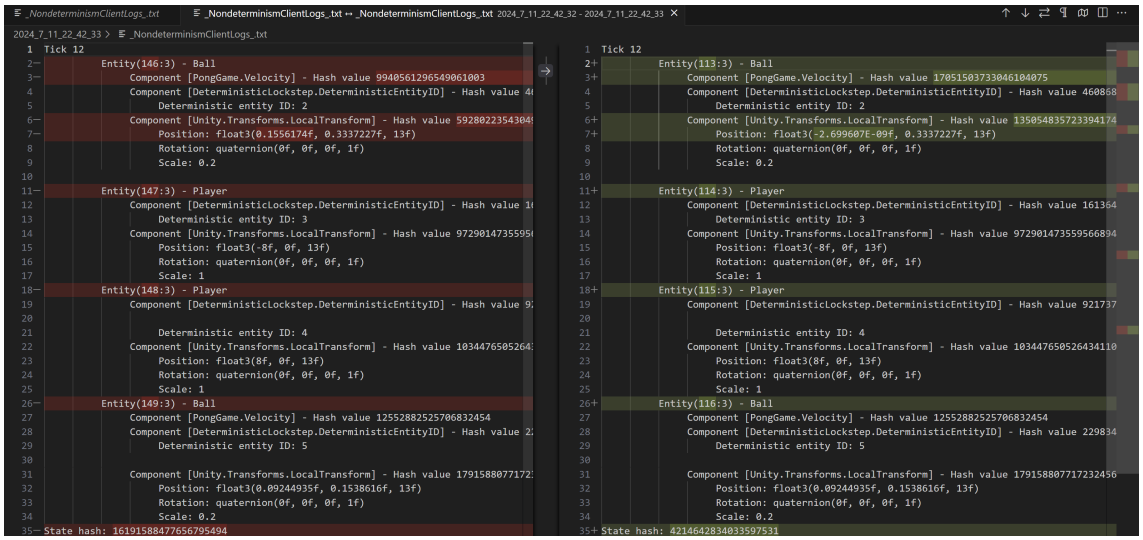

Figure 7.6: Diff comparison of log files

7.2.2 Per-system vs per-tick validation

One of the main goals of this investigation was to explore if the ECS structure and the division of behaviour into distinct systems could be leveraged for validation. Consequently, in contrast to per-tick validation discussed in the previous section of this chapter, a more advanced option of per-system validation is implemented.

With per-tick validation, developers can usually pinpoint the exact tick and variable that caused the desync. Based on this information and an understanding of common reasons for nondeterministic code, as explained in Section 7.4, they can investigate the code for possible sources of the issue. This can be a significant challenge, as illustrated by the nondeterminism case discussed in the Riot blog [33].

Taking advantage of the ECS structure, which separates logic from data and executes code system by system, the package modifies the DeterministicSystemGroup implementation to allow for ”per-system” validation. If this option is chosen, the hashing system is injected at the beginning of the group as well as between each pair of systems. The modification includes storing hash values in a list, with the hashing system adding a hash to the list each time it completes its calculations. This setup allows sending an empty list (indicating no validation was performed), a list containing one hash (indicating per-tick validation), or a list containing multiple hashes (indicating per-system validation). The size of the list in the per-system case will vary depending on the number of systems running in the group.

If nondeterminism is detected during per-system validation, the package can identify which hash, and thus which system, caused the desync. If the first hash differs, it means that some system or functionality outside of the DeterministicSystemGroup caused the desync. If a hash after a particular system differs, the package can pinpoint that system as the source of nondeterminism.

This method has its advantages and drawbacks.

The advantage is that, with per-system validation, the generated log file, as shown in Figure 7.7, will display hashes generated on a per-system basis. These hashes are referenced by their index in the DeterministicSystemGroup, which developers can use to locate the specific system in the group. Running a diff operation on two files reveals the first desynced variable and the system after which the desync occurred, allowing developers to narrow down the code base they need to investigate to find the source of nondeterminism. For example, in the Megacity Metro project mentioned in Section 4, which has around 35–40 systems and larger games that can have hundreds of systems, being able to pinpoint the exact system is extremely helpful in drastically reducing the number of potential sources of nondeterminism.

The drawback of this solution is the increased time required to run this many hash operations. The time increase compared to per-tick validation is proportional to the time taken for per-tick validation multiplied by the number of systems in the DeterministicSystemGroup. For example, in the Pong game, per-system validation can take up to 2 ms per tick, which is significantly longer than per-tick validation multiplied by the number of systems. The precise time comparisons of these methods are discussed in Section 8.

Due to this increased time requirement, per-system validation is not recommended for production games. Instead, this method is better suited for debugging nondeterminism using the game replay option discussed in Section 7.2.3. Combined with per-system validation, this allows for more precise debugging of nondeterminism.

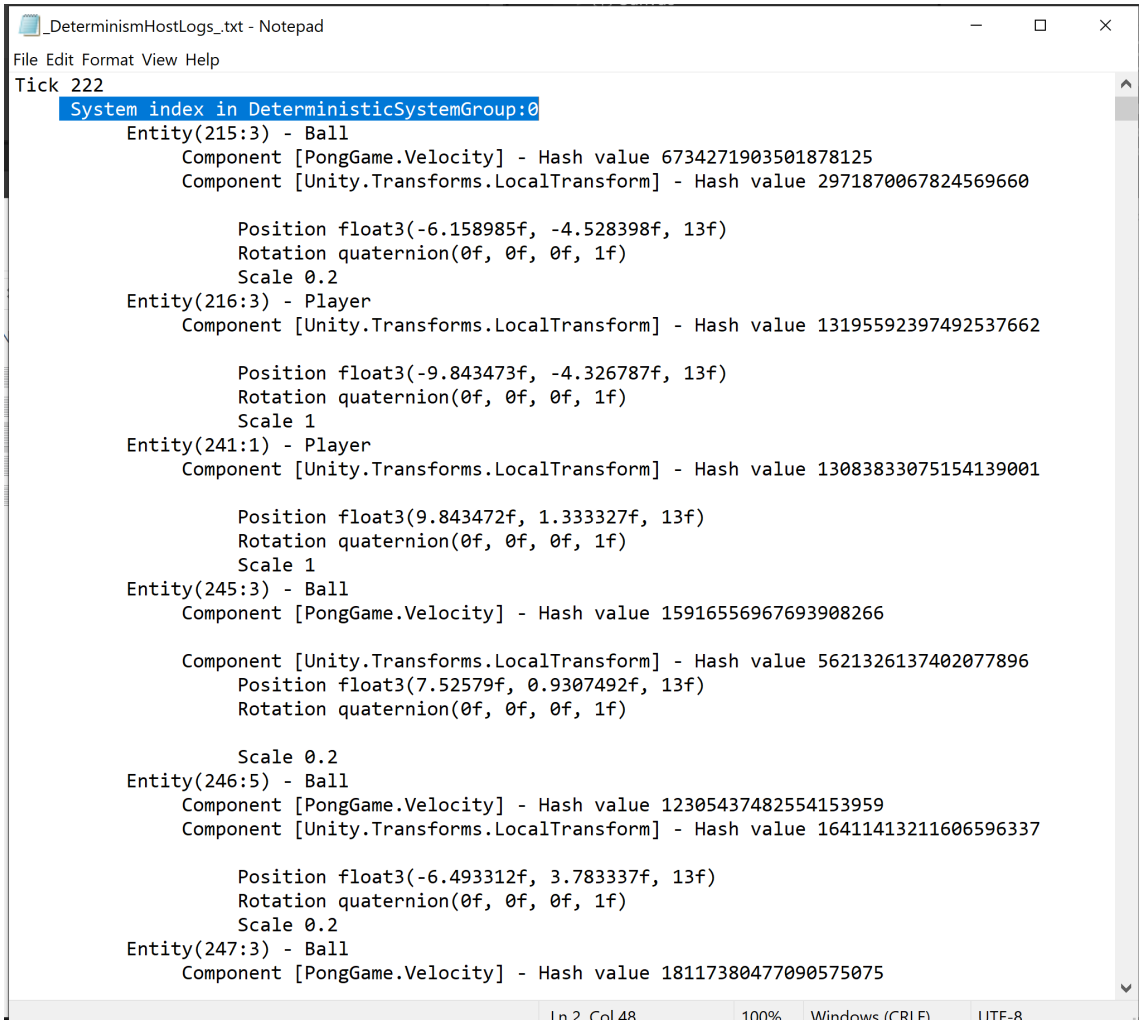

Figure 7.7: Per system log file

7.2.3 Game replay framework

To leverage per-system validation, similar to Riot's replay framework, the package includes a replay framework. To use it, the developer needs to copy the server recording input file and settings file and place them in the main NondeterministicLogs folder, where all files are generated in different subfolders. Then, they need to click the "replay from file" button in the Pong game. In a more professional scenario where this button is not available, the developer needs to set the replayFromFile variable to true in DeterministicSettings. When the game starts, instead of taking inputs from the keyboard or sending inputs to the server, it will replay the game as if those inputs saved in the server replay file were arriving from the server.

The advantage of this approach is that the developer can choose a different validation method for the replay (e.g., per-system validation) and examine the logs of the simulation again. Since this is a local resimulation, it will only create log files on one device. Therefore, it is important to use a machine with the same specifications as the one where nondeterminism was detected, which can be specified based on the ClientSettings file.

Nondeterminism issues may be stable, appearing on the same tick every time (e.g., an issue related to spawning the last ball on tick 1000 in the sample Pong game), or they may be unstable, appearing on seemingly random ticks. To address this, the replay file

will not only hash the final state but also generate a file with hash information for every tick in the game. This allows for detecting the first variable that diverged between two runs.

This approach adds significant computational overhead and is currently extremely slow, as discussed in Section 8, and should be addressed in future work.

7.3 Debugging nondeterminism with the package tools on example of a sample Pong game

To showcase the proper usage of the implemented tools, let's explore it from a Pong game developer's perspective. First, the developer hosts the game on their machine, choosing WhitelistedHashing_PerTick hash validation, which this package proposes for production games. Then, another client connects, and the game starts. At some point during the game, the developer sees a message about nondeterminism detected, as shown in Figure 7.8.

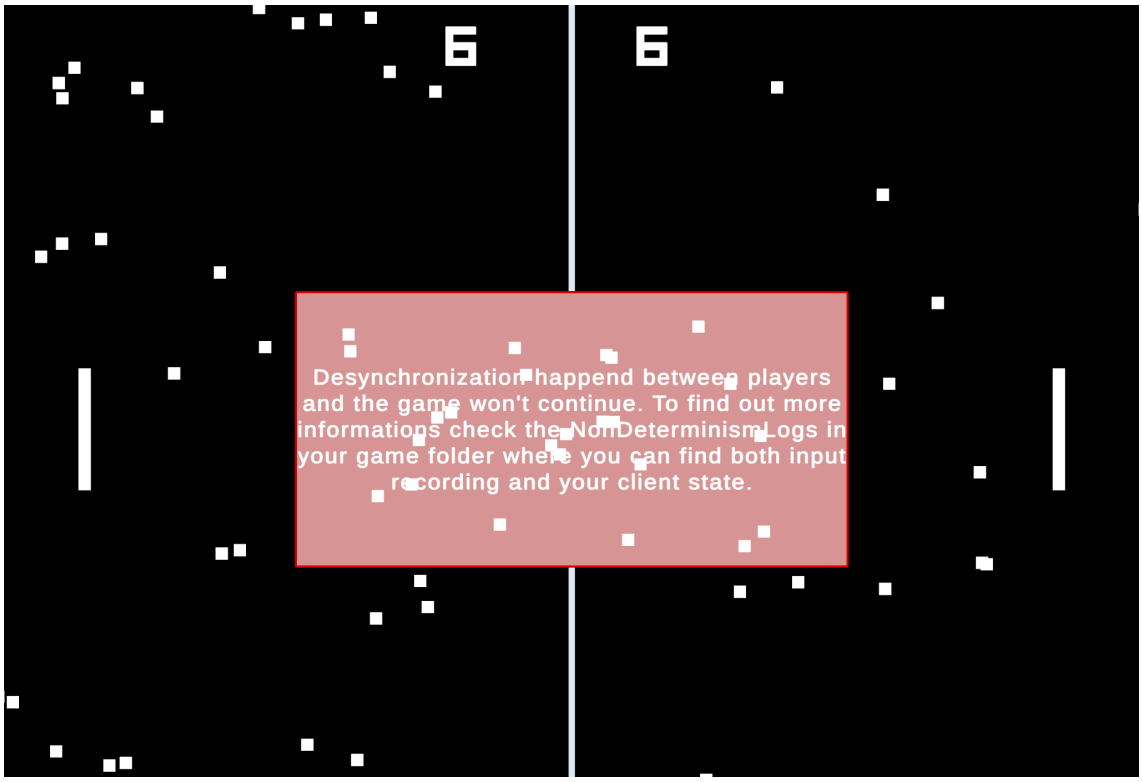

Figure 7.8: The Pong sample game upon nondeterminism is detected.

The next step is to investigate the generated files from both clients and the server, which the developer needs to gather. As discussed in Section 7.2.1, easing this process should be a focus of future work. The developer can then compare the generated client log files to identify what happened. In this case, they would see the difference between the two files, as shown in Figure 7.9.

The developer would notice that the divergence occurred on tick 12 and was caused by different values of the x position and velocity component on one of the balls. With this information, the developer, being familiar with the game implementation, may already have some clues. For example, in the Pong game, the position is influenced by the velocity component, so the issue could be related to velocity. The developer might also know if

something special happens on tick 12. If not as familiar with the game code, the developer would need to look into each system that potentially changes the velocity component to investigate the issue further. Alternatively, they can use the replay framework to get closer to the source of the nondeterminism.

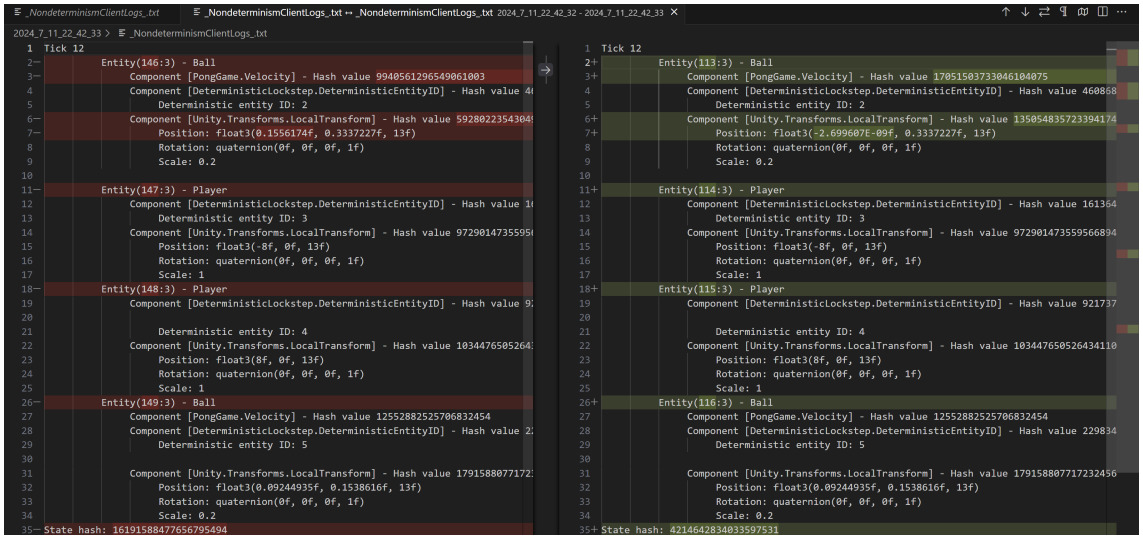

Figure 7.9: Differences between two generated files in per-tick validation

Before using the replay framework, an important step is to compare the settings files generated by each client to ensure they are the same. If the settings differ, the issue is related to this discrepancy. If the settings are the same, the developer should ideally check the machine parameters of the client they want to replay, as shown in Figure 7.4. Running the resimulation on a machine with similar parameters is crucial since the issue may be related to specific conditions, such as differences between Windows and Mac operating systems.

The next step is to place both the server input recording file and the client settings file in the main NondeterminismLogs folder, which should be located inside the game folder along with the game executable and other important files.

First, the developer should replay the game with the same determinism validation used in the game that desynced, using the ”FullStateHash” option. The developer should replay the game on two machines that closely resemble the client machines that experienced the desynchronization. After running it once on each machine, they should compare the generated files to see if the issue was replicated or if another issue appeared. Ideally, the developer should run it multiple times, comparing the file differences each time, until nondeterminism is detected. However, it may be possible that the game is being replayed on a different machine than the one where the issue occurred or that the issue is unstable and manifests differently in different iterations, often caused by the use of random values. The developer needs to decide whether to continue the replays if nondeterminism does not appear after several consecutive attempts.

If nondeterminism is detected, it confirms that it is possible to experience it between the two machines used for testing. In this case, if the developer wants more information about the source of nondeterminism, they can continue replaying the game with the per-system validation option. Although this option is slow, it can accurately identify a system containing nondeterministic code when used correctly.

In this example, the developer may see the log as shown in Figure 7.10.

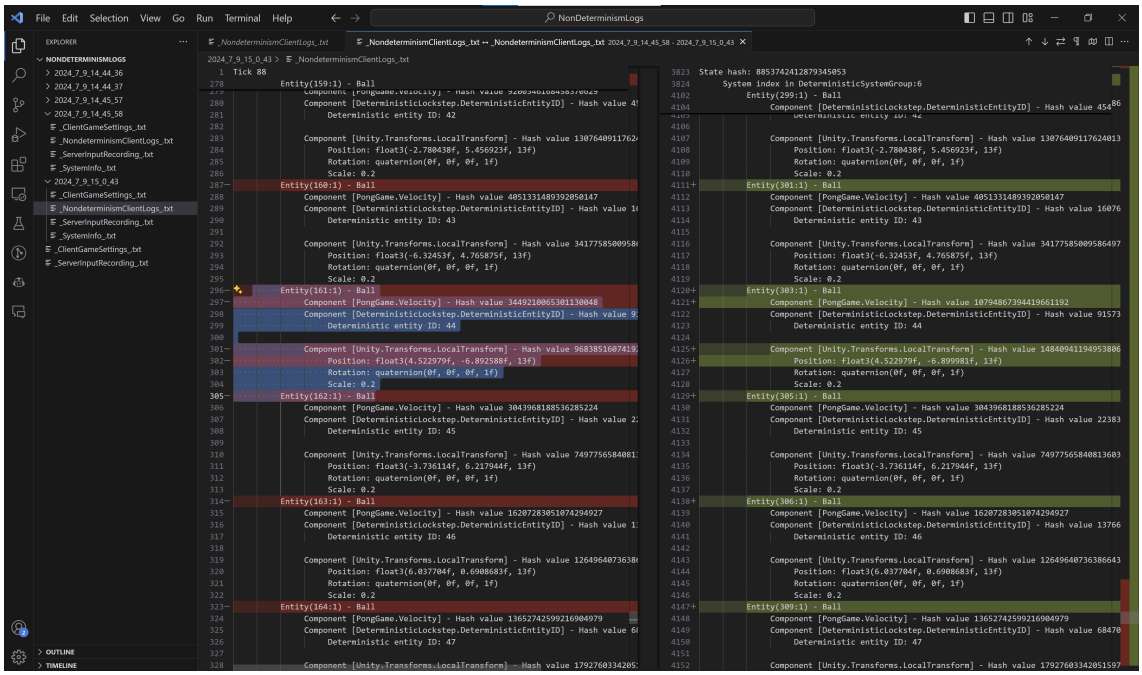

Figure 7.10: Differences between two generated files in per-system validation

Since the file contains a recording from every tick and for each tick for every system, developer after seeing the first instance of different variables, should go up in the file to see after which system it happened, as in Figure 7.11.

If no difference between components is observed and a different hash for the frame is still generated, it indicates an issue on the package side or a case of nondeterministic usage of the DeterministicID component, which influenced sorting in some way.

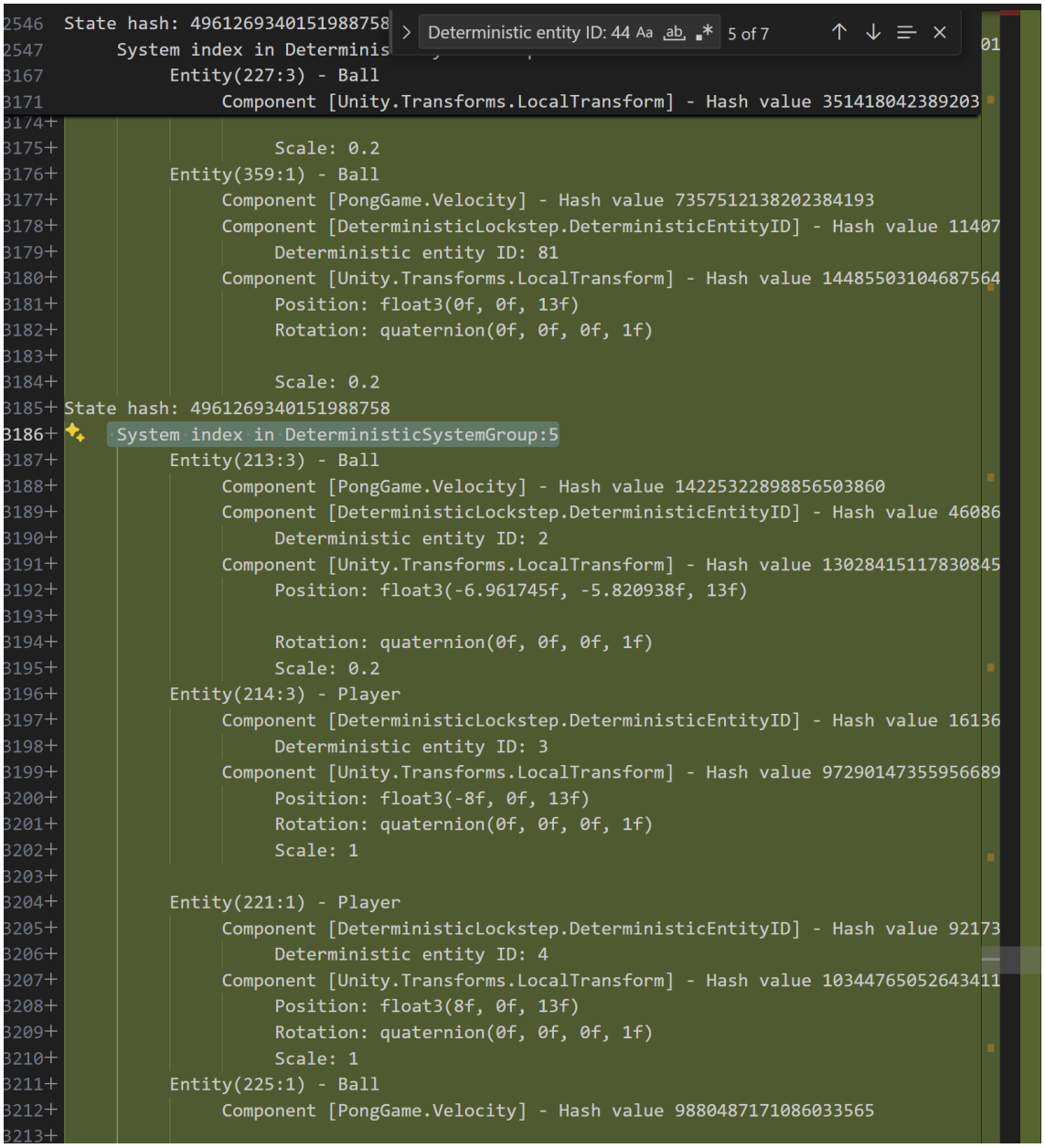

Figure 7.11: System number after which values in both files are different

When examining the game, the developer should identify which system corresponds to the number in the group and investigate potential sources of nondeterminism within that system. This approach can drastically reduce the number of code areas that need to be examined in ECS-based frameworks.

Sources of nondeterminism often fall into well-known categories and are usually easy to fix. However, the challenging part is identifying when and where the nondeterminism occurred in the code, especially in large games where many functionalities overlap. To assist with this, Section 7.4 discusses the most common sources of nondeterminism that the developer should look for.

In the case of the sample Pong game investigated here, the issue was caused by generating the direction using sine and cosine mathematical functions, which are notorious for causing nondeterminism issues.

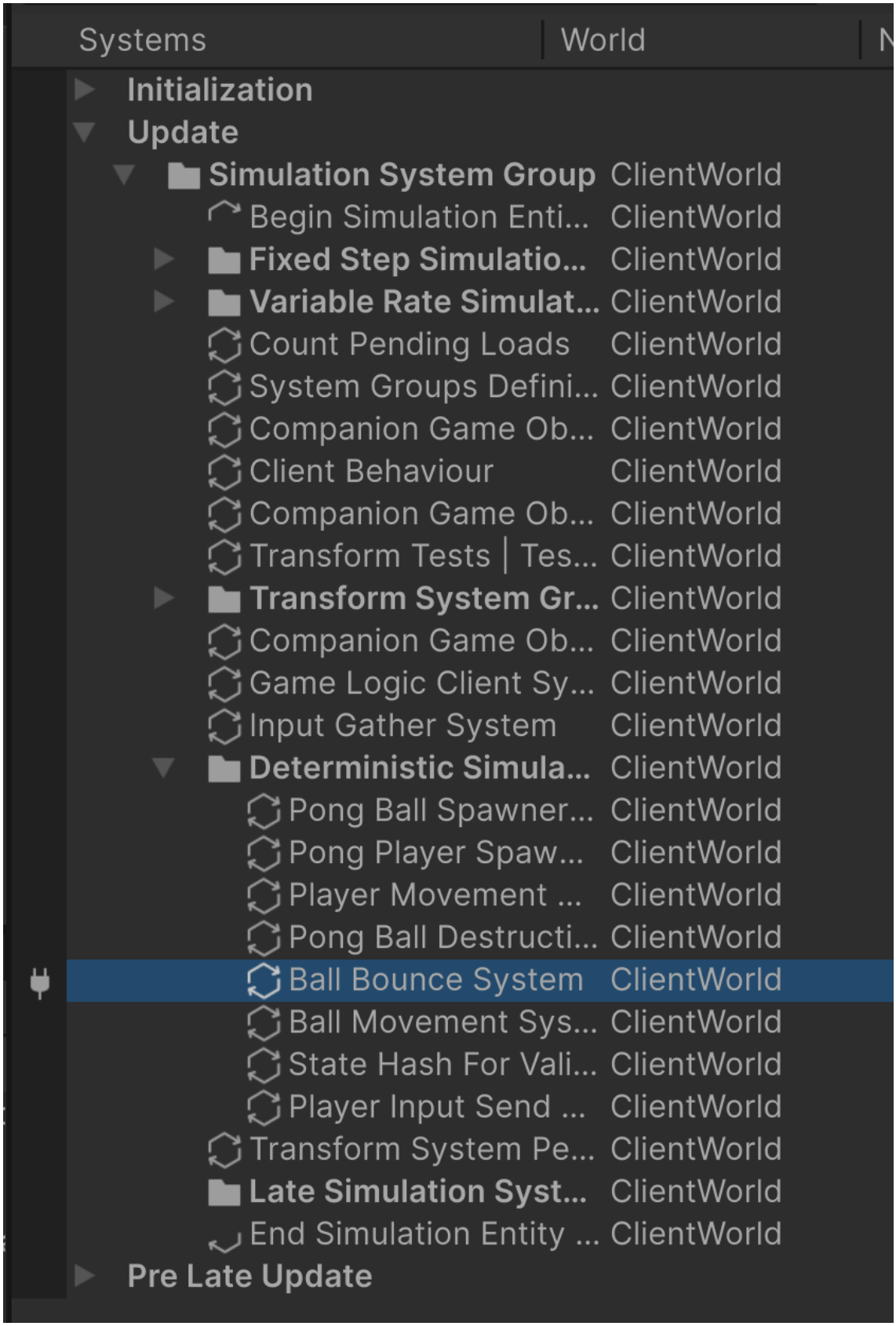

Figure 7.12: List of systems running in DeterministicSystemGroup

7.4 Common sources of nondeterminism

There are several well-known causes of nondeterminism to consider. Understanding these causes can help in identifying and pinpointing the potential sources of nondeterminism in the code.

7.4.1 Random number generators

The most obvious example is generating a random value on different clients and using it to affect the game state. This will inevitably result in different game states if those values are not synchronised between clients.

Despite their implications, random number generators in most software are already deterministic. Most of what we call random number generators (RNGs) in computing are more accurately called pseudorandom number generators (PRNGs). This is because nearly all random number APIs rely on formulas that ”twist” or ”shift” a given seed value into another value, always producing the same series of outputs from a given starting seed. Divergences begin when the developer can’t control the order, the number of calls, or the initial seed of a random number generator.

The usual solution for maintaining perceived randomness in each game is to generate a starting seed on the server and share it with all clients, ensuring everyone uses the same seed to generate random numbers. It’s important to make sure that random numbers from that generator are generated the same number of times and in the same order for every client.

If it is impossible to guarantee the same order using a single generator, games can employ multiple random number generators for different purposes. This helps maintain consistency across all clients while allowing for varied pseudo-random processes.

As discussed in Section 7.1.7, the package generates the random seed on the server and sends it to every client along with the game settings. The developer is then responsible for generating random values based on this seed in a deterministic way.

7.4.2 Uninitialized variables

A cause of non-determinism can also be not initialising variables. A primitive type that is not explicitly given an initial value will simply pick up whatever garbage values are stored at its memory address when it is allocated.

The code listening 7.4 shows this code, which gives a better understanding of the description. This wouldn’t usually be the case in professional games since this issue is obvious to spot, but it serves as an explanation of what can happen when a variable is not initialised.

```
1  public class ExampleSystem
2  {
3   private int variableValue;
4
5   public void OnUpdate(ref SystemState state)
6   {
7   for (; variableValue < 10; variableValue++)
8   {
9   DoSomething();
10  }
11 }
12
13 private void DoSomething()
14  {
15 // For example, code generating new ball positions
16  }
```

```
17 }
```

Listing 7.3: Example usage of an uninitialized variable

In this example, variableValue is not initialised, so it may contain any value, depending on how it's allocated. With typical compiler settings, this code won't even throw a compiler warning. As a result, OnUpdate might call DoSomething() anywhere from 0 to 10 times.

The obvious course of action would be for developers to zero-initialise everything all the time. In theory, identifying uninitialized variables at the beginning would be straightforward with simple program analysis. However, this approach has some drawbacks. A great example from the League of Legends game, as described in the Riot Games blog post [33], illustrates this issue.

The blog describes a common Vector3f struct used extensively throughout the code to represent properties like positions, velocities, directions, and other geometric concepts. While the performance impact of forcing an individual Vector3f instance to be zero-initialised is minimal, the cumulative performance hit becomes significant when considering the millions of times these types are constructed in a game.

Therefore, the developers of League of Legends chose not to zero-initialise these instances by default. Instead, they focus on detecting potential nondeterminism early and only initialise the variables when necessary, keeping in mind that those would be edge cases that weren't predicted by the developer.

7.4.3 Asynchronous processing

As the complexity of games has increased over time, games have become increasingly multithreaded and capable of modifying behaviour based on the timing of asynchronous tasks. While parallelism can greatly enhance performance, it also introduces a significant source of non-determinism if handled poorly. There are possible to find sources describing more in detail the problem of deterministic applications on multi-core architectures [39].

When multiple threads execute concurrently, the order in which operations are performed can vary from one run to another. This variability can lead to different game states even if the same initial conditions and inputs are provided. For example, if two threads are responsible for updating the position of two different game entities, the order in which these updates occur can influence the final state of the game. If thread A updates entity 1 before thread B updates entity 2 in one run, and thread B updates entity 2 before thread A updates entity 1 in another run, the resulting game states could be different.

In DOTS, parallel processing is managed through careful control of data access patterns, utilising ECS to structure data and jobs efficiently.

Variables and components in DOTS can be marked as either read-only or read-write. When a variable or component is marked as read-only, it means that multiple threads can access and read its value simultaneously without any risk. This is because the value will not be modified during the read operations, ensuring safe concurrent access and eliminating the possibility of race conditions.

For read-write variables or components, DOTS uses a dependency management system instead of traditional locking mechanisms. Each job specifies its dependencies, indicating which data it will read from or write to. The Unity Job System then schedules these jobs in a way that respects these dependencies. If a job needs to write to a variable, the Job System ensures that no other job can read from or write to that variable until the

write operation is complete. This approach prevents concurrent modifications and race conditions while avoiding the performance overhead associated with locking.

To handle changes to entity components, DOTS uses an Entity Command Buffer (ECB). This buffer collects commands to modify entities during the execution of parallel jobs (i.e., EntityManager.SetComponentData). Once all jobs are finished, the commands in the buffer are executed, applying all changes at once. This method ensures safe and efficient modifications without conflicts, as all the commands are based on the same previous values.

The order in which these commands are executed is still random by default. However, depending on the need, one can specify sorting criteria for the ECB to ensure that commands are applied in a particular sequence. This sorting can be based on factors such as the type of component being modified or the priority of the operation, allowing developers to control the execution order to meet specific requirements within their application.

Using the Entity Command Buffer (ECB) is one approach. However, for the determinism validation implementation in the package, ECB was not used since the package does not modify any entity components but only collects data.

7.4.4 Different CPU architectures and floating point precision

This problem refers to the ability of a simulation running on two different CPU architectures to use floating-point math or fixed-point math. Floating point numbers are a common source of non-determinism in game development because the implementation of floating point arithmetic can vary across different hardware and software environments. While essential for representing real numbers in computing, floating point math introduces several challenges that can lead to inconsistencies in game simulations.

Floating point numbers follow the IEEE 754 standard [40], which defines how numbers are represented and manipulated in binary. However, despite this standardisation, the implementation details can differ between processors, compilers, and operating systems. These differences can cause the same floating-point calculations to yield slightly different results on different platforms.

The primary issue lies in the precision and rounding behaviours of floating-point arithmetic. When operations involve very large or very small numbers, or when many operations are chained together, tiny differences in precision can accumulate, leading to significant discrepancies over time. This phenomenon can be extensive, and numerous sources provide detailed guidance on where developers should pay attention regarding floating-point arithmetic [41].

One classic example of non-determinism caused by floating-point arithmetic is the addition of numbers in different orders. Due to the finite precision of floating-point representations, adding a sequence of numbers in a different order can lead to different results. For instance, summing a list of floating point numbers from smallest to largest can yield a different total than summing them from largest to smallest 7.4.

```
float a = 1e10;
float b = 1;
float c = -1e10;

float sum1 = (a + b) + c; // Result: 1.0
float sum2 = a + (b + c); // Result: 0.0
```

Listing 7.4: Example of floating point arithmetic operation which yields different results

In the above code, sum1 and sum2 will yield different results due to the order of operations, illustrating how floating-point arithmetic can introduce non-determinism.

In game development, non-deterministic behaviour can lead to significant issues, particularly in multiplayer games that rely on deterministic lockstep models. For example, if a physics simulation uses floating-point calculations to determine object positions, small differences in these calculations can cause the game state to diverge between clients, leading to inconsistent gameplay experiences.

This issue is typically not a problem when all players use the same platform, like Windows. However, it becomes problematic when players are on different platforms, such as one on Windows and another on Mac. The differences in floating-point arithmetic implementations across platforms can result in diverging game states.

To address this, the game industry generally uses fixed-point numbers, which involve handling floating-point numbers with integer numbers. Games that use GGPO-style netcode and frameworks like Quantum have adopted this approach to ensure consistency across different CPU architectures. Fixed-point arithmetic represents numbers with a fixed number of decimal places, eliminating many of the issues associated with floating-point precision. Several fixed-point arithmetic libraries are available for developers to use [42]. However, implementing fixed-point arithmetic can be cumbersome and may sacrifice some performance due to additional casting and processing overhead.

7.4.5 Caching

Caching refers to the practice of storing certain variables or data for use over several frames. This is not the case when using lockstep, but in the GGPO implementation, when performing player prediction and rollbacks, it starts to affect the simulation. While caching can significantly enhance performance by reducing the need to recompute values repeatedly, it can also introduce nondeterministic behaviour. This is a common reason why, for example, a physics engine may be nondeterministic.

It's worth noting that in a pure lockstep game, caching won't cause any issues. Problems may arise only if the simulation requires going back in time, as is the case with rollback—a more advanced technique implemented on top of lockstep for deterministic games. Developers should be aware of this when implementing a full GGPO-style game.

There are two types of systems one can distinguish stateless and stateful.

In a stateless system, the state is entirely derived from the current inputs, and the system does not retain any information between frames. This approach ensures that the same input will always yield the same output, promoting determinism. For instance, a stateless physics engine calculates object positions and velocities purely based on the current forces acting upon them without relying on past states.

In contrast, a stateful system retains information between frames, relying on past states to influence current computations. While this can improve efficiency and performance, it introduces the risk of divergence. For example, a stateful physics engine might cache object positions and velocities, using these cached values to update the object's state in subsequent frames.

Stateful systems can cause divergence during rollback. In deterministic lockstep simulations, rollback involves rewinding the game state to a previous frame and re-simulating forward with new inputs. If a system caches values, the re-simulation might use stale or incorrect cached data, leading to inconsistencies across different clients.

Let's consider an example of a simple caching mechanism 7.5.

```
public class MovementSystem
{
private Boolean didPlayerJumpedWhileStanding = false;

 public void Update()
{
 Input input = GetPlayerInput();

 if (!wasUsed && GetPlayerMovement() == 0) {
PlayerJump();
 didPlayerJumpedWhileStanding = true;
 }

 if (GetPlayerMovement() != 0)
 {
 didPlayerJumpedWhileStanding = false;
 }
 }
}
```

Listing 7.5: Example of caching in a game system

In this example, the developer wants the player to jump once if they stand still. Under normal circumstances, this code might work as intended. However, problems arise when rollback is used.

During a rollback, the game state is reverted to an earlier point, and the simulation is replayed from there. If the 'didPlayerJumpWhileStanding flag is not properly reset during the rollback, it can cause inconsistencies. For instance, if the player was supposed to jump but didn't due to the rollback, the flag may remain 'true', preventing the jump from occurring again when the state is replayed.

To properly execute rollback, one option is to use only stateless systems. Another is to store a verified state of the game separately. When "rolling back," the system should refer to and start from this verified state, eliminating unwanted cached values and allowing for accurate game simulation.

7.4.6 Methematical functions

Some system-implemented mathematical functions can also be a source of nondeterminism. This can be due to the handling of floating-point numbers or functions like sinus and cosinus, which are notoriously nondeterministic when values become smaller. Developers should be aware of which mathematical functions may be nondeterministic and avoid using them.

7.4.7 Other, not possible to predict issues

Even if a game is thoroughly tested and no nondeterministic issues are detected, nondeterminism can still occur during gameplay. Such issues can manifest in the wild for various unpredictable reasons. These include hardware failures like a corrupt RAM stick or a CPU bug, data corruption during transmission, or even external factors like an electromagnetic pulse (EMP) wave from solar activity.

Given these unpredictable and unpreventable potential sources of nondeterminism, it is crucial for game developers to utilise determinism validation to ensure that their game is deterministic enough to not cause unfair games and player frustration.

8 Evaluation

This chapter focuses on evaluating the provided solution in terms of usability and effectiveness.

8.1 Deterministic lockstep netcode model evaluation

The implementation of the deterministic lockstep netcode model follows standard practices and applies the theoretical model in DOTS as a package. This package is demonstrated with a sample Pong game but can also be utilised by any game that follows the usage guide and README documentation present on GitHub. One limitation preventing this implementation from being ”production ready” is that the input structure for the game currently needs to be defined within the package rather than user code due to code generation reasons previously mentioned.

Regarding computational overhead, the cost that running this type of server brings to the host, which must run it alongside the normal client simulation in the Pong game sample, is minimal, averaging 0.3 ms over 10 consecutive game runs. This overhead would naturally increase with more clients, but because the server only gathers inputs and sends them back without simulating the game alongside the clients, this overhead shouldn't increase significantly, and the only increase would be in the need to possibly gather a few more inputs during the same frame. Additionally, since the host has almost no latency (with the server running on the same machine), the time to send the input is negligible. Therefore, this lockstep server should be manageable on the client machine even for larger games with more players, although more testing with real-life game examples would be needed in this case.

The main advantage and disadvantage of a lockstep server both stem from the fact that it does not process the simulation but only handles inputs. On one hand, this allows it to run easily on the client machine without introducing significant computational overhead. On the other hand, if a client desynchronizes (nondeterminism detection is explained in Section 7), there is no way to continue the simulation, unlike a server-controlled simulation that could send correct state information to the clients. This makes detecting and ensuring the deterministic lockstep simulation, which is indeed deterministic, crucial for utilising the benefit of not running the simulation on the server.

8.2 Determinism validation and debugging tools evaluation

Let's evaluate separately both parts of the implementation, which are determinism validation tools and determinism debugging tools.

8.2.1 Determinism Validation Tools Evaluation

The implemented solution can be considered a proof of concept, as there are many areas where it can be improved.

The validation part generally works effectively. For the sample Pong game, the results of ”Whitelisted_PerTick” validation were calculated in 0.25 ms. This time depends on machine specifications and may take longer on weaker devices, highlighting the importance of the developer thoughtfully choosing a subset of the world for validation. ”FullState_PerTick” validation took 0.3 ms on the same computer. Similarly, this time will vary depending on the same factors as whitelisted validation but will generally be longer due to the lack of restrictions on entities.

Per-system validation took an average of 2 ms for the Pong game. While this is relatively fast in this case, the time depends first on the duration of the usual per-tick validation, and this time should be approximated by multiplying it by the number of systems in the DeterministicLockstepSystemGroup, which can result in a significant increase. Hashing per-system introduces a substantial performance overhead (multiplying the normal hashing time by the number of systems) but offers potential benefits in narrowing down the code scope for harder-to-diagnose nondeterminism cases.

Another aspect is the sorting of entities for validation. In cases where entities are nondeterministically sorted due to their IDs, discrepancies can be identified by examining the generated log file. If the final hash differs but none of the components appear different, this indicates a sorting issue rather than a validation failure.

However, this solution has limitations: the user must remember to add the DeterministicID component, which introduces the potential human error of either assigning the id in a nondeterministic manner or forgetting to assign it to an important entities. This could be improved by automating the ID assignment process, ensuring each entity receives a unique deterministic ID upon creation.

In summary, ”Whitelisted_PerTick” hash validation should be used for games employing this package since it takes the least time. However, it may result in game interruptions and player frustration if the code isn't thoroughly tested and nondeterminism is detected too often. Despite this, it ensures that two different simulations won't continue if the state diverges significantly.

8.2.2 Determinism debugging tools evaluation

Starting with file generation, it must be pointed out that writing to a file, especially for a large game, poses a significant challenge. For the sample Pong game, log file generation took 6 ms with per-tick validation and up to 61.69 ms for per-system validation. This time may be significantly longer for larger games, particularly with per-system validation, which results in much larger files. The log file size may also become enormous, similar to what is described in the Riot blog. For example, in the Pong game, the file size for ”Whitelisted_PerSystem” validation, which contains information about 1333 ticks, is 14 MB, with a maximum of 138 entities being considered for validation at the peak moment. This size could be reduced by considering what information is essential to include in the file.

Currently, this delay is not noticeable to the player since their game stops regardless, and they won't realise that some time is used to generate the file after stopping the game. However, in the current approach, where the developer needs to obtain these files from all clients to debug instances of nondeterministic behaviour, if the player simply quits the game, the file won't be generated due to a lack of time. Additionally, when generating a replay file, all tick information needs to be logged, which is currently done on a per-frame basis. This makes the game very slow and laggy, as every frame requires around 6–60 ms for file writes, depending on the validation option.

For debugging, logs containing client machine information, logs of the nondeterministic frame, and client settings are generated on the client side, while the server input recording is generated on the server side. For later replay, it is important to use machines similar to those on which the divergence occurred.

The current replay functionality is slow, taking up to 70 ms to simulate one frame of the sample Pong game, making it impractical for larger games where these times would be even longer. Additionally, larger games would have bigger log files and longer replay

times. For example, replaying a 1-hour online game with file logging would take at least several times longer with per-system validation, with the actual time likely being much higher.

Nevertheless, this approach shows the potential to identify the exact system causing the divergence, which can drastically reduce the number of places the developer needs to investigate. Additionally, the replay functionality can be used for validating determinism within the same architecture by running the same input recordings multiple times with per-tick validation to ensure consistent results. Besides game validation, game engine developers might use these tools to validate certain functionalities of their engine by creating a small sample game utilising this system and employing the same validation and debugging approach.

9 Discussion

This section discusses the findings of this thesis in a more theoretical manner, discussing key results, their interpretations, implications, and potential improvements.

9.1 Findings

The most significant finding of this thesis is the method for per-system validation, which is not detailed in other available solutions. This approach can effectively reduce the code scope for testing, making it more precise to pinpoint sources of nondeterminism, but comes with a huge computational cost.

Additionally, the integration of deterministic lockstep netcode with the Unity DOTS framework provides a foundation for developing deterministic multiplayer games or creating a full GGPO model. The validation tools developed in this thesis have proven effective in detecting and debugging nondeterministic behaviour, providing a clear methodology for ensuring consistency across game clients. This can be useful for smaller teams and developers who might not have the resources to create their own deterministic engine or use Photon Quantum.

Moreover, the implemented package can be used not only by game developers to implement deterministic games but also by game engine developers to validate the determinism of their game engines. Although the package is currently tailored for the Unity game engine, the same methodology could be adapted to other engines.

Furthermore, the research highlights that proper determinism validation and debugging are incredibly challenging, especially in game engines that are not entirely deterministic.

9.2 Current Limitations and Future Work

Currently, the package should be viewed as a foundational implementation and example rather than a complete solution. It serves as a base for further development and refinement, demonstrating the basic principles of the deterministic lockstep netcode model and determinism validation and debugging tools, but it lacks some advanced features necessary for full production use.

Several areas for improvement have been identified for the current package. These improvements can be divided into two main areas: improving the netcode model for better game synchronisation and advancing the determinism validation and debugging toolset for more effective validation and testing.

9.2.1 Possible netcode model improvements

The current implementation of the deterministic lockstep netcode model is a basic version that serves as an implementation base. Future work should focus on several enhancements.

Adding player prediction and rollback would create a fully functional GGPO solution, addressing latency issues and improving the overall gaming experience.

Allowing players to join the game after it has started would require transferring the current game state to the new player and synchronising them with the ongoing game. This would help address the problem of the game stopping when nondeterminism is detected. If a client desynchronizes, the host could send the authoritative game state to the client,

allowing the game to continue while still generating logs for later verification. In this configuration, the server would need to perform game simulation in addition to transferring inputs, which would likely require separating client and server instances but would have its benefits, working similarly to Riot's engine.

Finally, using code generation to allow users to implement their own input structures within the package would provide greater flexibility and customisation.

9.2.2 Possible determinism validation and debugging tool improvements

At the current moment, implementing a fully production-ready deterministic game remains a challenging task, especially in game engines that are not 100% deterministic. The implemented determinism validation and debugging tools show potential and serve as a proof of concept for such implementations in DOTS, which represents a Data-Oriented Design (DOD) and Entity Component System (ECS) framework, demonstrating its potential to ease debugging.

First, the automatic assignment of the DeterministicID component value to every entity created in the scene would ensure that all components marked by the developer are considered for validation. This would also eliminate the possibility of human error in assigning the ID in a nondeterministic manner.

Secondly, using code generation during runtime can enable writing the fields of even user-created components to the file. For example, in the case of the Velocity component in the sample Pong game, it would be possible to see not just the hash but also the actual values.

Next, the resimulation framework could be optimised for speed, as it doesn't need to wait for the server or other clients since it's a local simulation. The developer is interested in the fastest possible simulation without needing to play the game. For example, the simulation could skip any visual updates and iterate through the DeterministicSystemGroup as quickly as possible, disregarding any enforced framerate. In this case, if one simulation step can be completed in 5 ms, the framework would not need to wait an additional 11 ms before simulating the next step, thereby avoiding wasted time.

Another option is for the server to simultaneously perform the simulation, allowing the possibility of continuing the simulation from where it stopped. This could be faster but might introduce problems. The exact state must be identical to that on the other client, which cannot be guaranteed without validating the entire world.

Lastly, a tool for performing automated file comparisons would be highly beneficial. It would eliminate the need to manually scroll through the file to find the system and field that were first desynchronized. Instead, it could automatically provide a summary of the comparison, highlighting the exact data that diverged.

9.3 Recommendations for work with deterministic games

When working on a deterministic game or wanting to implement a deterministic functionality, one should acknowledge the difficulty of such a task, and this thesis offers some recommendations based on the knowledge gained during this investigation.

Understanding the deterministic limitations of the software that is being used is crucial. This awareness helps developers anticipate potential issues and plan for alternative solutions or workarounds.

Regularly testing the software's deterministic behaviour using various scenarios and hardware configurations helps identify and resolve issues early in the development process.

Building such software should be incremental, meaning that after adding new functionality, the game is tested and verified to ensure no obvious sources of nondeterminism are introduced. This approach makes it easier to locate the source of nondeterminism, as it is most likely caused by the recently added functionality.

Leveraging Data-Oriented Design (DOD) and the Entity Component System (ECS) pattern to structure data and logic efficiently improves performance and makes it easier to identify and address nondeterministic behaviour because of such separation than usual object-oriented solutions.

Automation of as many functionalities related to detecting and debugging nondeterminism cases helps to decrease the time it takes to fix potential problems, or at least helps to avoid causing them.

In conclusion, while the challenge of creating fully deterministic games is significant, the tools and methodologies developed in this thesis provide a solid foundation for future research and development.

By following these recommendations, developers can be more aware of potential problems and speed up the process of achieving a satisfactory level of determinism in their software. It is important to note that one can never claim software is 100% deterministic, but these practices can help to be as close to this goal as possible.

10 Conclusion

To conclude, the thesis has demonstrated the possibility of implementing determinism validation and supporting debugging tools within the Unity DOTS framework. The developed tools enable the detection and debugging of nondeterministic behaviour by providing a way to identify specific frames, fields, and systems responsible for such behaviour. The integration of these tools with the deterministic lockstep netcode model provides a solid foundation for exploring ways of developing deterministic multiplayer games, democratising their creation by providing a new resource to developers that they can build upon.

The research began with an exploration of existing netcode models and determinism validation techniques. A basic deterministic lockstep netcode model was implemented and integrated with the Unity DOTS framework. Subsequently, various validation tools were developed to detect and debug nondeterministic behavior. The Pong game served as the primary test case, demonstrating the practical application of these tools.

Throughout the process, the focus remained on creating a solution adaptable to different game scenarios by separating non-game-related functionalities into a separate package that could be used with various games. This goal was partially achieved because some parts of the code, such as the input struct implementation, still require more work with runtime code generation to fully separate the package code from the game code.

Key findings of this thesis include the successful development of per-system validation methods, which are not described in other publicly available solutions. This approach effectively reduces the code scope for testing, making it easier to locate the exact source of nondeterminism. While this method can be time-consuming, it can be highly useful in the debugging process and can complement the usual validation methods outside of gameplay. Additionally, the integration of deterministic lockstep netcode with Unity DOTS provides a foundation for developing deterministic multiplayer games. The validation tools developed have proven effective in ensuring consistent game state across connected clients on a simple showcased game.

The research answered the stated research questions by demonstrating how the Deterministic Lockstep Netcode Model can be effectively incorporated within Unity's DOTS, as discussed in Section 6, providing a solid foundation for future development. It showed how nondeterministic behavior can be efficiently detected in DOTS, as explained in Section 7, proving that this goal is achievable. However, to be truly efficient, the detection does not cover 100% of the game state but rather a subset of it. It addressed how the source of nondeterministic behavior in DOTS-based games can be identified by presenting the methodology and an example usage on a sample Pong game in Section 7, and by providing recommendations for future work to improve this process, as detailed in Section 9.3. Lastly, it identified the most common sources of nondeterministic behavior, as discussed in Section 7.4.

The entire implementation is available on GitHub [20], allowing developers to use it as a base for custom implementations.

Future research should focus on optimizing the determinism validation and debugging tools for larger-scale and different types of games. Enhancements such as automatic diff tools, improved, faster game replay functionality, code generation for complete separation of the package from game code, automatic baking of DeterministicEntityID into every

existing entity or work on minimising generated log file sizes and time it takes to write to them can further refine the process. Currently, many aspects of the validation require manual user input, such as adding names to entities, marking components and entities for validation, and adding the deterministic ID component, which can introduce user errors. These processes should be automated to minimize potential issues.

Additionally, integrating more advanced lag mitigation techniques like player prediction and rollback with the lockstep model will enhance the usability of this implementation. Exploring the use of code generation for custom input structs and field value display in user-defined components will also add flexibility and precision to the tools.

In conclusion, this thesis has shown that it is possible and how to validate determinism of a game and debug its sources within the Unity DOTS framework. The tools and methodologies developed provide a solid foundation for future research and development in deterministic multiplayer games. By addressing the challenges of nondeterminism and providing practical solutions, this work hopes to add to the goal which is creating deterministic simulations and multiplayer gaming experiences.

Bibliography

[1] Greg Chance et al. “On Determinism of Game Engines Used for Simulation-Based Autonomous Vehicle Verification”. In: *IEEE Transactions on Intelligent Transportation Systems* 23.11 (Nov. 2022). Conference Name: IEEE Transactions on Intelligent Transportation Systems, pp. 20538–20552. ISSN: 1558-0016. DOI: 10.1109/TITS.2022.3177887. URL: https://ieeexplore.ieee.org/abstract/document/9793395 (visited on 06/13/2024).

[2] *The lag-fighting techniques behind GGPO's netcode*. URL: https://www.gamedeveloper.com/programming/the-lag-fighting-techniques-behind-ggpo-s-netcode (visited on 06/13/2024).

[3] Connor Van Ligten. “Rollback Netcode: How to Sustain a Competitive Fighting Game Community”. In: (May 1, 2022). URL: https://keep.lib.asu.edu/items/166176 (visited on 06/13/2024).

[4] *Deterministic Multiplayer Engine | Photon Quantum*. URL: https://www.photonengine.com/quantum (visited on 06/13/2024).

[5] Jessica D. Bayliss. “Jessica D. Bayliss, Developing games with data-oriented design | Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation”. In: *GAS '22: Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation* (Dec. 2022), pp. 30–36. DOI: 10.1145/3524494.3527626. URL: https://dl.acm.org/doi/10.1145/3524494.3527626 (visited on 06/13/2024).

[6] *Data-oriented design*. In: *Wikipedia*. Page Version ID: 1221817444. May 2, 2024. URL: https://en.wikipedia.org/w/index.php?title=Data-oriented_design&oldid=1221817444 (visited on 06/13/2024).

[7] *Technologies*. SteamDB. URL: https://steamdb.info/tech/ (visited on 06/26/2024).

[8] *Game engines on Steam: The definitive breakdown*. URL: https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown (visited on 06/26/2024).

[9] Paul Mieschke. “Deterministic Lockstep in Networked Games”. In: (2024). URL: https://hdms.bsz-bw.de/frontdoor/index/index/docId/7107 (visited on 06/13/2024).

[10] *GGPO | Rollback Networking SDK for Peer-to-Peer Games*. GGPO. URL: https://www.ggpo.net (visited on 06/13/2024).

[11] *Servers*. League of Legends Wiki. June 13, 2024. URL: https://leagueoflegends.fandom.com/wiki/Servers (visited on 06/13/2024).

[12] K. Fedoseev et al. “Application of Data-Oriented Design in Game Development”. In: *Journal of Physics: Conference Series* 1694.1 (Dec. 2020). Publisher: IOP Publishing, p. 012035. ISSN: 1742-6596. DOI: 10.1088/1742-6596/1694/1/012035. URL: https://dx.doi.org/10.1088/1742-6596/1694/1/012035 (visited on 06/13/2024).

[13] Alan Jay Smith. “Cache Memories”. In: *ACM Computing Surveys* 14.3 (Sept. 1, 1982), pp. 473–530. ISSN: 0360-0300. DOI: 10.1145/356887.356892. URL: https://dl.acm.org/doi/10.1145/356887.356892 (visited on 06/13/2024).

[14] *Understand data-oriented design*. Unity Learn. URL: https://learn.unity.com/tutorial/part-1-understand-data-oriented-design (visited on 06/29/2024).

[15] *Unity (game engine)*. In: *Wikipedia*. Page Version ID: 1231235691. June 27, 2024. URL: https://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=1231235691 (visited on 06/28/2024).

[16] *Diplomacy is Not an Option*. URL: https://store.steampowered.com/app/1272320/Diplomacy_is_Not_an_Option/ (visited on 06/13/2024).
[17] *Megacity Metro – Large-Scale Multiplayer Demo*. Unity. URL: https://unity.com/demos/megacity-competitive-action-sample (visited on 06/13/2024).
[18] *SnpM/LockstepFramework: Framework for lockstep RTS, TD, and MOBA games.* URL: https://github.com/SnpM/LockstepFramework (visited on 06/13/2024).
[19] Tony Cannon. *pond3r/ggpo*. original-date: 2019-10-03T15:56:50Z. June 12, 2024. URL: https://github.com/pond3r/ggpo (visited on 06/13/2024).
[20] *Thesis: Scalability and Democratization of Real Time Strategy and Fighting Games: The Deterministic Lockstep Netcode Model with Client Prediction Combined with the Data Oriented Tech Stack*. GitHub. URL: https://github.com/michalooo/Thesis-Deterministic-Lockstep-Netcode-DOTS-with-Validation/tree/thesis_submission (visited on 07/09/2024).
[21] Otto Tolppanen. "How Fighting Games Use Rollback Netcode to Improve the User Experience". In: (May 31, 2023). URL: https://aaltodoc.aalto.fi/handle/123456789/121399 (visited on 06/13/2024).
[22] Shengmei Liu et al. "The Effects of Network Latency on Competitive First-Person Shooter Game Players". In: *2021 13th International Conference on Quality of Multimedia Experience (QoMEX)*. 2021 13th International Conference on Quality of Multimedia Experience (QoMEX). ISSN: 2472-7814. June 2021, pp. 151–156. DOI: 10.1109/QoMEX51781.2021.9465419. URL: https://ieeexplore.ieee.org/document/9465419 (visited on 06/13/2024).
[23] proepkes. *proepkes/UnityLockstep*. original-date: 2019-01-14T16:34:17Z. June 10, 2024. URL: https://github.com/proepkes/UnityLockstep (visited on 06/13/2024).
[24] mrdav30. *mrdav30/LockstepRTSEngine*. original-date: 2018-12-06T19:03:23Z. May 28, 2024. URL: https://github.com/mrdav30/LockstepRTSEngine (visited on 06/13/2024).
[25] *Deterministic Prototyping in Unity DOTs – jdxdev*. Aug. 2, 2019. URL: https://www.jdxdev.com/blog/2019/08/02/deterministic-unity-dots/ (visited on 06/13/2024).
[26] *Game Networking Demystified, Part II: Deterministic*. URL: https://ruoyusun.com/2019/03/29/game-networking-2.html (visited on 06/13/2024).
[27] *Deterministic Lockstep*. Gaffer On Games. Nov. 29, 2014. URL: https://gafferongames.com/post/deterministic_lockstep/ (visited on 06/13/2024).
[28] Yuan Gao (Meseta). *Netcode Concepts Part 3: Lockstep and Rollback*. Medium. Sept. 22, 2019. URL: https://meseta.medium.com/netcode-concepts-part-3-lockstep-and-rollback-f70e9297271 (visited on 06/13/2024).
[29] *Enum FloatMode | Burst | 1.4.11*. URL: https://docs.unity3d.com/Packages/com.unity.burst@1.4/api/Unity.Burst.FloatMode.html (visited on 06/13/2024).
[30] *ECS for Unity*. Unity. URL: https://unity.com/ecs (visited on 07/11/2024).
[31] Stormgate Hub. *What is Stormgate's SnowPlay Engine?* Stormgate Hub. Dec. 11, 2022. URL: https://stormgatehub.com/what-is-stormgates-snowplay-engine/ (visited on 06/13/2024).
[32] *Determinism in League of Legends: Introduction*. June 20, 2017. URL: https://technology.riotgames.com/news/determinism-league-legends-introduction (visited on 06/13/2024).
[33] *Determinism: Fixing Divergences*. May 1, 2018. URL: https://technology.riotgames.com/news/determinism-league-legends-fixing-divergences (visited on 06/13/2024).
[34] Eleftheria Christopoulou and Stelios Xinogalos. "Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices". In: *International Journal of Serious Games* (2017). Accepted: 2019-10-28T11:06:52Z. ISSN: 2384-8766. DOI:

10.17083/ijsg.v4i4.194. URL: https://ruomo.lib.uom.gr/handle/7000/180 (visited on 06/13/2024).
[35] *Global Ping Statistics*. WonderNetwork. URL: https://wondernetwork.com/pings/ (visited on 06/16/2024).
[36] Unity Technologies. *Unity - Manual: Unity's Package Manager*. URL: https://docs.unity3d.com/Manual/Packages.html (visited on 06/17/2024).
[37] Unity Technologies. *Unity - Manual: Creating custom packages*. URL: https://docs.unity3d.com/Manual/CustomPackages.html (visited on 06/17/2024).
[38] *RPCs | Netcode for Entities | 1.2.3*. URL: https://docs.unity3d.com/Packages/com.unity.netcode@1.2/manual/rpcs.html (visited on 06/16/2024).
[39] Adrien Barbot. "An innovative technology solution to design deterministic safety-critical applications on multi-core architectures". In: (). URL: https://www.asterios-technologies.com/wp-content/uploads/dlm_uploads/2017/07/KRONO-SAFE-innovative-technology-solution.pdf.
[40] *IEEE Standards Association*. IEEE Standards Association. URL: https://standards.ieee.org (visited on 06/15/2024).
[41] *What every computer scientist should know about floating-point arithmetic | ACM Computing Surveys*. URL: https://dl.acm.org/doi/10.1145/103162.103163 (visited on 06/13/2024).
[42] Jere Sanisalo. *XMunkki/FixPointCS*. original-date: 2018-04-19T18:15:32Z. June 10, 2024. URL: https://github.com/XMunkki/FixPointCS (visited on 06/15/2024).

Technical
University of
Denmark

Anker Engelunds Vej 101, Building 101A
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk