Network Synchronization of Physics Simulated Objects in Games

Ashank Rajendran
Rochester Institute of Technology
ar9666@g.rit.edu

Abstract - Synchronizing physics-intensive game objects in real-time multiplayer environments presents a significant technical challenge due to nondeterministic floating-point behavior, divergent hardware performance, multithreaded execution, and unpredictable network conditions such as latency and packet loss. This research investigates deterministic physics synchronization as a solution to these challenges by comparing two distinct implementations: a commercial deterministic engine using Photon Quantum and a custom-built deterministic physics system developed using a fixed-point arithmetic (FixedMathSharp) library and a lockstep networking model. A prototype Unity environment has been developed in which multiple clients interact with dynamic physics objects, including player-controlled capsules and continuously colliding rigidbodies. The Photon Quantum implementation serves as a baseline for deterministic client-side simulation, while the custom system aims to replicate deterministic behavior with greater transparency and control over the simulation pipeline. By comparing synchronization accuracy, bandwidth usage, computational cost, and perceptual player experience across these models, this study aims to provide a practical framework for developers selecting physics synchronization strategies. The findings are anticipated to reduce implementation uncertainty, help developers avoid unnecessary technical debt, and inform future design decisions for scalable, responsive multiplayer physics systems.

I. Introduction

Real-time multiplayer games increasingly rely on complex physics simulations to generate immersive and responsive interactive environments. Unlike simple state synchronization methods such as transmitting position, rotation, or animation parameters, physics-based synchronization requires consistent, deterministic simulation across multiple distributed clients or a high-bandwidth authoritative server model [1][2]. Maintaining a shared synchronous game state becomes especially challenging when simulations involve numerous dynamic objects, rigidbody interactions, or gameplay systems that depend heavily on real-time physics interactions.

These challenges stem from several well-known issues. First, floating-point arithmetic is inherently nondeterministic across platforms, compilers, and processor architectures, often resulting in divergent simulation states even when identical initial inputs are applied. Second, physics engines frequently employ multithreading or parallel solving techniques that introduce additional nondeterminism due to thread scheduling differences. Third, network-related issues such as latency, jitter, and packet loss, further complicate attempts to keep clients synchronized with one another or with a server. As a result, developers must carefully select synchronization strategies that balance accuracy, performance, bandwidth cost, and overall player experience.

Several architectural models aim to address these challenges. Server-authoritative simulation shifts all physics computation to a central server, simplifying consistency but increasing server load and raising scalability concerns [8]. Alternatively, dead reckoning prediction methods offer a lightweight, bandwidth-efficient approach, but may struggle during highly interactive or chaotic physics events [4][7]. However, such approaches may not be feasible for a distributed authority framework [2].

The deterministic simulation approach offers a potential solution by ensuring that identical inputs always produce identical outputs across all clients. This typically requires strict control over numerical precision, execution order, and simulation state. Fixed-point arithmetic is commonly used to eliminate floating-point inconsistencies, while lockstep networking models ensure that all clients advance the simulation in a synchronized manner using shared inputs.

This research investigates the feasibility, performance, and design trade-offs of two strategies for client-sided deterministic simulation for multiplayer. One approach uses a commercially available solution while the other explores the feasibility of implementing a similar architecture from scratch for more custom use cases.

Commercial solutions such as Photon Quantum provide fully deterministic simulation environments with integrated networking and physics systems, reducing development complexity but limiting flexibility and transparency. In contrast, custom deterministic implementations allow developers to tailor physics behavior and system architecture to their specific needs, at the cost of increased development effort and potential sources of error.

This research focuses on comparing these two approaches: a production-ready deterministic framework and a custom-built deterministic physics system using a FixedMath library and lockstep synchronization. By systematically evaluating accuracy, bandwidth consumption, computational load, and subjective gameplay responsiveness across these models, this study aims to provide a practical framework to guide developers in selecting synchronization techniques best suited for their game's requirements. Ultimately, this work seeks to reduce technical uncertainty and inform best practices in the development of scalable, predictable, and responsive multiplayer physics systems.

II. Methodology

This research adopts an experimental evaluation approach in which multiple physics-synchronization architectures are implemented and compared under controlled conditions. The overall methodology is divided into two stages: (1) development and analysis of a deterministic client-side simulation using Photon Quantum, and (2) higher-level design and planned implementation of alternative synchronization models to be evaluated in subsequent phases.

A. Photon Quantum Deterministic Simulation

The first phase of this research centers on the implementation of a deterministic multiplayer simulation using Photon Quantum, a commercially available deterministic networking and physics framework for Unity. This system employs a predict/rollback simulation model, making it an ideal starting point for evaluating deterministic approaches to synchronized physics [5].

The prototype developed for this phase consists of a Unity scene containing two player-controlled capsule characters placed within an enclosed room. Several spherical rigidbodies are added to the environment to create continuous bouncing and collision interactions, generating a dynamic physics workload. This configuration was selected because it produces unpredictable motion patterns and repeated contact events, which are useful for evaluating the consistency of deterministic simulation across multiple clients.

Photon Quantum's predict/rollback architecture enables deterministic synchronization by requiring each client to submit its input for a given simulation frame before the frame is advanced. Since only input commands are transmitted, the entire physics simulation runs identically on each participating client, provided that deterministic conditions are maintained [5].

Experiments were conducted under nominal network conditions without introducing artificial latency or jitter. The effects of adverse network conditions are planned to be explored in future work using Photon's network simulation tools.

Synchronization accuracy is assessed using a combination of qualitative and quantitative analysis. Qualitative analysis includes visual inspection of object divergence across clients to identify visible disparity, jitter or corrections. Quantitative analysis is performed with the help of Quantum's internal state-hashing tools by recording and comparing entity positions across clients over multiple simulation frames to compute positional error metrics and identify correction patterns.

This phase establishes the baseline characteristics of deterministic client-side physics: low bandwidth usage, strict dependency on identical client states, and sensitivity to any nondeterministic operations within the physics pipeline [8].

B. Custom FixedMathSharp Deterministic Simulation

The second system is a custom-built deterministic physics implementation developed using a fixed-point arithmetic library (FixedMathSharp) and a lockstep networking model. Unlike Photon Quantum which provides an in-built deterministic physics backend, this system required manual implementation of core physics behaviors such as gravity, motion, collision detection, and collision resolution.

The same gameplay scenario used in the Photon Quantum prototype is recreated to ensure comparability in testing conditions. All physics calculations are performed using fixed-point arithmetic to eliminate floating-point divergence across different systems. The simulation is advanced in discrete, synchronized steps, with clients exchanging only player inputs for each simulation step.

This approach provides full control over the simulation pipeline, allowing detailed control of how determinism is achieved and where potential sources of divergence may arise. However, it also introduces challenges related to numerical precision, stability, and implementation complexity.

Experiments for this environment were conducted through local testing of two clients on the same device and introducing artificial network simulation latency.

Synchronization accuracy is evaluated by comparing entity states across clients over time, using custom logging and state comparison tools. Additional metrics include CPU usage, scalability, bandwidth consumption, and development overhead.

C. Comparative Evaluation Framework

Once all systems have been implemented, each will be evaluated using a consistent set of criteria to ensure comparability. These criteria include:

- Synchronization accuracy and determinism
- Bandwidth usage
- Perceived gameplay smoothness and responsiveness
- Client-side computational cost
- Implementation complexity and development overhead

By comparing a fully integrated deterministic framework with a custom-built solution, this study aims to highlight the trade-offs between ease of use and system flexibility, as well as the practical challenges involved in implementing deterministic physics for multiplayer games.

III. System Implementation

The system implementation consists of two independent multiplayer physics environments developed to evaluate deterministic synchronization approaches. The first system utilizes Photon Quantum, a commercial deterministic networking and physics framework, while the second is a custom-built deterministic simulation implemented using fixed-point arithmetic and a strict lockstep networking model. Both systems replicate an identical gameplay scenario to ensure comparability across experimental conditions.

A. Photon Quantum

1) Architecture Overview:

The first system was implemented using Photon Quantum, a commercially available deterministic networking and physics framework designed for real-time multiplayer applications. Rather than developing a physics engine and networking stack from first principles, this approach leverages Photon Quantum's integrated deterministic simulation pipeline, allowing the study to focus on system behavior, synchronization characteristics, and developer-facing constraints.

2) Project Setup:

The Photon Quantum project was implemented in a Unity project within its Entity Component System framework (ECS) using the provided SDK and tooling resources. Components and other runtime game state data types were declared using Quantum's Domain-Specific-Language (DSL) written to text files with the '.qtn' extension. The Quantum compiler parses these files and generates C# structs that integrate with the Quantum environment, eliminating the need for the developer to boilerplate code for type serialization. Four main types of structs were generated for use in the project, three of which were components used for identifying specific entities and storing custom values:

- PlayerComponent, used to identify which entity is a player entity
- PlayerLink, used to hold a reference to the client's owned player
- PlayerVisual, used to store the material properties of the respective player entity

Additionally, an input structure was also defined to specify what the valid inputs for the simulation are so that they can be serialized for network communication. Six different input types were defined for this project:

- Forward/Back
- Left/Right
- Up/Down

The simulation Unity scene and prefabs for the ball and player Unity GameObjects were stored in Quantum's resources folder as addressables as Quantum avoids hard references to assets wherever possible.

3) Gameplay Systems:

Gameplay logic in Quantum is determined by Systems keeping in line with the ECS framework. For this implementation, the custom systems defined were as follows:

- PlayerController, responsible for applying player inputs to movement
- PlayerSpawnSystem, responsible for spawning and initialization of player entities
- BallSpawnerSystem, responsible for generating spherical dynamic rigidbodies

These systems are driven by parameters defined in config files which correspond directly to each of these systems. All systems operate exclusively on fixed-point data and avoid non-deterministic APIs such as Unity's standard physics engine or time-dependent functions. This ensures that simulation results remain identical across all clients. These custom systems were added to the list of systems specified in Quantum's system configuration file to serve as entry points for executing game logic deterministically through strict execution ordering.

Other Core systems included in the system configuration file provided natively by Quantum include:

- CullingSystem3D, used to cull entities with transform components in predicted frames.
- PhysicsSystem3D, used to run physics on all entities with a transform and collider components
- EntityPrototypeSystem, used to create, materialize and initialize EntityPrototypes, which are serialized versions of entities.
- PlayerConnectedSystem, used to trigger events when players connect and disconnect

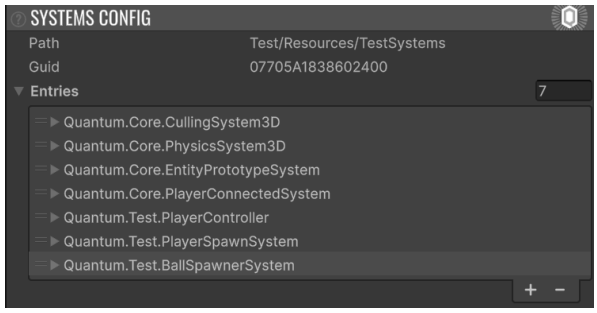

Figure 1. System execution order specified in configuration

4) Scene and Physics Configuration:

The simulation environment consists of two player-controlled capsule entities and fifty dynamic spherical rigidbodies enclosed within a bounded space. This configuration was selected to generate frequent collision events and complex interaction patterns, providing a rigorous test for synchronization accuracy.

Physics simulation is handled entirely by Quantum's fixed-point physics engine, which provides deterministic implementations of collision detection and resolution. A fixed timestep is used for simulation updates, ensuring consistent progression across all clients regardless of rendering frame rate. By eliminating floating-point arithmetic and enforcing deterministic execution order, the system guarantees reproducible physics behavior.

5) Networking Model:

The networking model follows a prediction/rollback simulation paradigm in which each client transmits only its player input for every simulation frame. The simulation advances deterministically on every client based on these player inputs. To maintain responsiveness under latency, the system locally advances simulation using input prediction. When delayed or corrected inputs are received, rollback and resimulation are triggered to restore consistency. This hybrid approach enables continuous simulation while ensuring eventual synchronization across all clients.

Photon Quantum utilizes a centralized, game-agnostic server component hosted on Photon's

infrastructure to coordinate simulation timing and manage input latency across clients. This server does not execute gameplay logic but instead maintains synchronicity within the environment. This mitigates the need to pause the simulation until inputs are received as they would in a lockstep system.

B. Custom FixedMathSharp Implementation

1) Project Setup:

The custom implementation system was created from the ground up following a more traditional lockstep networking model. This was implemented using Unity's traditional runtime environment rather than using their ECS-based framework, which Quantum does. Physics interactions are resolved using a custom collision system that utilizes fixed-math data types from the FixedMathSharp library, which is a deterministic fixed-point math library for .NET. Unity Netcode for GameObjects is utilized as the netcode layer for client network communication.

Unity's ScriptableObjects are used to store scene configuration data. InitialPhysicsData is a ScriptableObject used to store initialization data for every dynamic physics object that needs to be initialized during the first frame of simulation. Once the simulation begins this data is loaded from resources and is used to instantiate all non-static physics objects and populate their data. Any static physics objects that are not dynamic are configured directly in the scene in editor mode and are not spawned dynamically.

Another instance of the same ScriptableObject framework called ClientInitialPhysicsData is used as the initialization data for any joining client. When a connecting client requests to join the simulation, the host collects data of every physics object in the scene during that simulation frame, packages it, serializes it and sends it to the client. The client uses this data received from the host to initialize their simulation scene to be identical to the simulation state on the host device.

To assist with setting up the simulation scene, a custom editor tool was created to bake the data of all in-scene physics objects to the initialization data ScriptableObject. This allows developers to place their required physics objects in the scene during editor time in their required configuration and then use the bake tool to automatically write the data to file so that when they run the simulation in play mode, it loads the same configuration.

The project has also been set up with data collection functionality for analytic purposes. Data for each physics object logged to a SimulationData ScriptableObject during each simulation step. At the end of the simulation, this data is written to a .csv file to be used for data analysis.

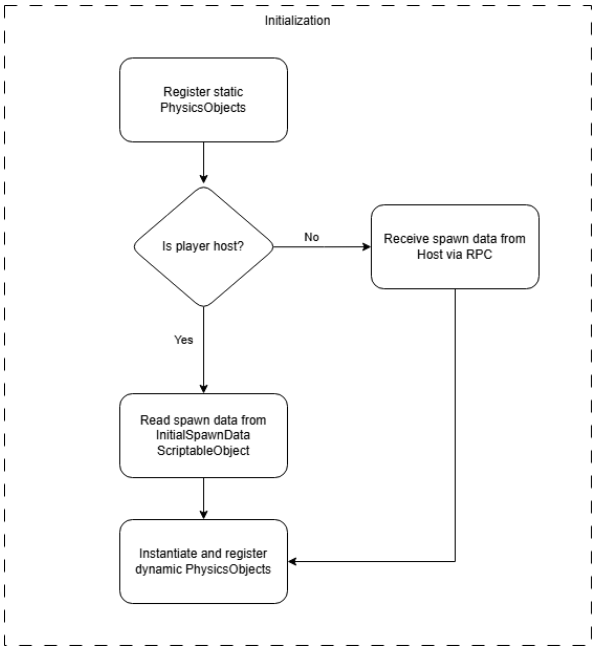

Figure 2. Scene initialization flowchart

2) Deterministic Physics Engine:

All physics data and computation is performed using a fixed-point math library known as *FixedMathSharp.* This ensures determinism across devices by eliminating floating point calculation variance across different devices that use different processing architectures.

Every GameObject used in the simulation contains a PhysicsObject component created to store physics data specific to that object. This component includes all the information that is required for physics calculations. Once the object is initialized, this data is registered to the PhysicsData class, which is responsible for performing the physics and collision calculations for every PhysicsObject. Aside from the data required for the physics calculations, PhysicsData also stores some additional data that has its own uses:

- *objectId***,** is used to store the unique ID of the respective PhysicsObject. This ID is generated deterministically to ensure that it is always the same on each client.

- *isPlayerControlled*, is a boolean flag used to indicate if a particular PhysicsObject is a player controlled object. This is used to determine whether the object should be registered as a player object during initialization in order to process any player inputs on it.
- *ownerPlayerId*, is used to identify which client the object belongs to if it is a player controlled PhysicsObject. This is used to determine which player inputs should be applied to which player objects.

The rest of the data is used purely to store its physics data values which will be used by the PhysicsData class during runtime for physics processing. PhysicsData stores the information of every physics object that has been registered to it and uses it to run the physics simulation. Data is stored in a Structure-of-Arrays (SoA) layout to group data together in contiguous memory to optimize CPU cache performance. The deterministically generated unique *objectId* of each PhysicsObject serves as the index for that object's data in the SoA. The data stored in this layout include position, velocity, mass, inverse mass, gravity scale, static state, and collider information.

For each simulation step that PhysicsData computes, a strict execution order is maintained and any and all collections involved are sorted to ensure determinism in ordering across devices. For each simulation step, processes are executed for all PhysicsObjects in a sequence of physics calculations. First, the effects of gravity are calculated and all velocity values are updated accordingly. Next, the positional data of all entries are calculated based on their velocity values and a fixed delta timestep value. Lastly, any collisions occurring between objects are resolved. This is done in two major steps. First, PhysicsData iterates through every pair of PhysicsObject collider data and checks if there is an intersection between the two colliders. If so, a *collisionPair* is generated that stores the indices of the two colliders involved. Next, the collision resolution is calculated. These two steps are processed iteratively multiple times during a simulation step to ensure accuracy and avoid small divergences in calculations. For this project, the process is performed iteratively 4 times.

There are two main collider shapes used for collision data, spherical colliders and box colliders. Collision resolution is processed differently depending on the configuration of colliders present in the generated collisionPair. There are three main configurations for collision resolution:

- Box collider vs. box collider
- Spherical collider vs. spherical collider
- Box collider vs. spherical collider

For each of these processes, the calculations can broadly be classified into a few steps.

First is position correction, which corrects the positions of each colliding body so that their colliders no longer overlap. This is using a mass weighted correction based on the objects' inverse masses. This means that objects of the same mass are pushed equally apart during the collision, while collisions with static bodies whose inverse masses are taken to be 0, result in only the dynamic object's position being changed while the static body remains stationary.

The next step is updating the velocities of each body based on the following impulse calculation formula:

$$J = -\frac{(1+e)(\vec{v}_{rel} \cdot \vec{n})}{\frac{1}{m_A} + \frac{1}{m_B}}$$

Where J is the impulse magnitude, e is the coefficient of restitution, $\vec{v}_{rel}$ is the relative velocity, $\vec{n}$ is the direction of the normal to the surface collision and m is the mass of the object [9].

The impulse is applied along the collision normal, affecting only the velocity component in the direction of n. The impulse is computed only when the relative velocity along the collision normal is negative, indicating that the objects are moving toward each other.

The inverse masses are precalculated and stored in the structure of arrays whenever a physics object is registered to it to avoid unnecessary recalculations. The coefficient of restitution is assumed differently for each collision configuration. For box vs. box collisions, we take 0, resulting in perfectly inelastic collisions. For spherical collisions we assume that the collision is perfectly elastic and take the coefficient of restitution to be 1. And finally, for box vs. sphere collisions, the coefficient of restitutions is taken to be 0.8 as this resulted in the desired level of bounciness for balls colliding with the ground.

After all the physics calculations are performed for each simulation step, the physics data for each entry is logged to the SimulationData ScriptableObject to be used later for generating data for the .csv file used in analytics

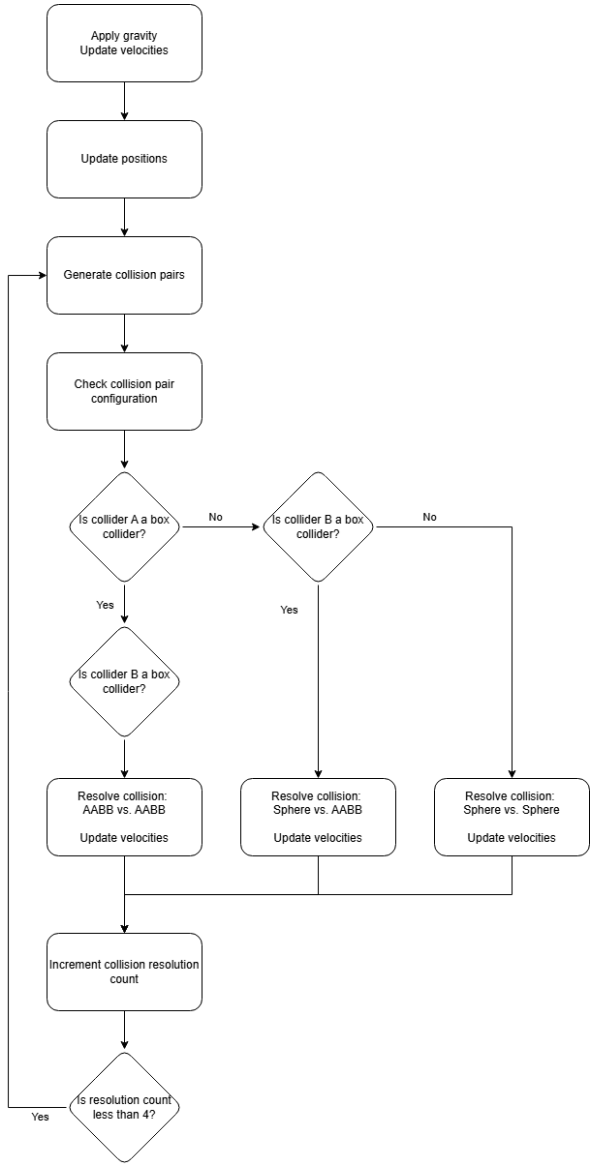

Figure 3. Physics simulation step

3) Simulation Loop:

The simulation is processed in discrete *ticks*. These *ticks* are not processed at a fixed rate. Instead, it checks every frame if input data from all players have been received before running a simulation step. Inputs are collected in a dictionary to associate each input with its respective *tick* number. Inputs are limited to one per tick to avoid dropping further inputs due to latency between processing the first received input and the delayed arrival of subsequent inputs due to latency. Thus, only the first received input is registered and sent per tick. For each tick, the system performs the following steps:

1. Collect inputs from all clients
2. Verify input completeness
3. Sort player inputs to maintain processing order
4. Advance the simulation by one timestep
5. Broadcast the next input frame

Because execution is strictly synchronized, the simulation halts if any required input is delayed.

On LateUpdate(), the actual Transform positional values of each corresponding PhysicsObject is updated to reflect its actual positional data value stored in the PhysicsData class.

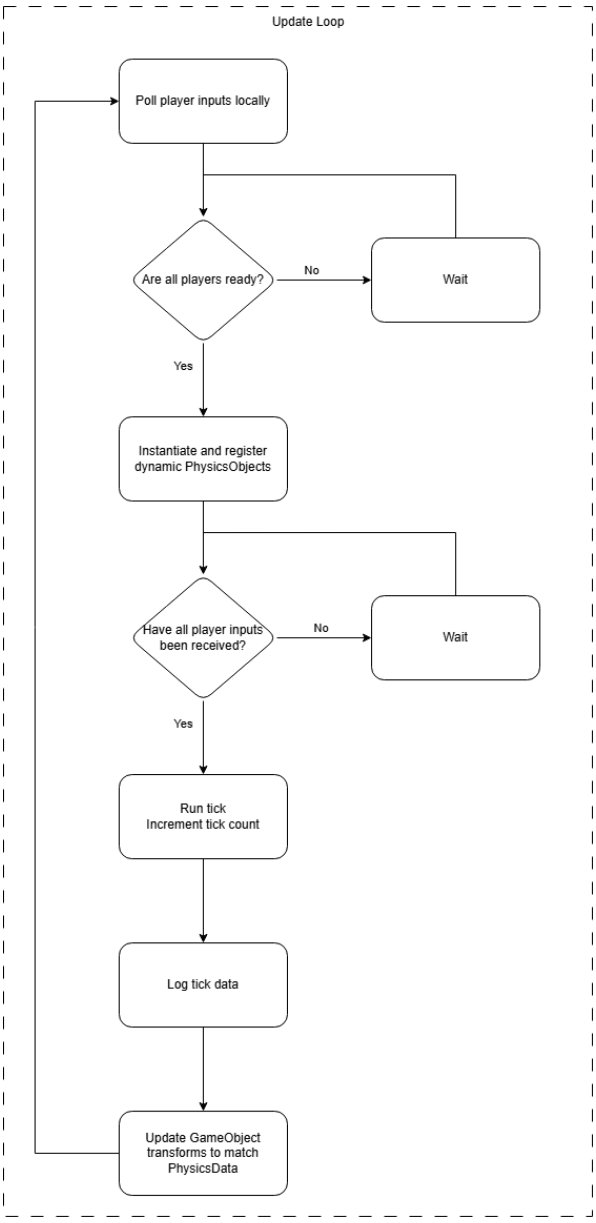

Figure 4. Simulation Update loop

4) Scene and Physics Configuration:

A near identical environment to the Photon Quantum system was recreated to ensure comparability. The scene includes two player-controlled PhysicsObjects and fifty dynamic PhysicsObjects balls within a bounded space. The player objects are initialized as boxes while the balls are initialized as spheres.

All PhysicsObject states are updated deterministically using fixed-point calculations, ensuring identical simulation results across clients under ideal conditions.

5) Networking Model:

Unity's Netcode for GameObjects library was used as the netcode layer for the project. Only three types of data are sent across clients during simulation to maintain synchronization.

The first is the initialization data. When a client attempts to connect with a host while the host is mid-simulation, the host pauses its simulation, collects all the physics data of each body in the current simulation tick, and stores it in the ClientInitialPhysicsData ScriptableObject along with the current simulation tick count. This data is serialized and sent to the client using a Remote Procedure Call (RPC). The *MultiplayerManager* class is responsible for handling all RPCs and other network related events such as a player connecting or disconnecting. The client then initializes its scene according to the provided data so that it is in an identical state as the host, and then sends the second type of network data, which is a ready message informing the host that the client has finished setting up. Once this message is received the host resumes simulation and begins polling input data.

The former two data types are used only upon initialization of the client and as such, does not require packets to be optimized for storage. The third and final data type is the player inputs that are sent every tick. These inputs are stored in an *InputPacket* which contains the Vector3 input values along with additional information such as the client's *playerId* as well as the *tick* number that is intended for the ticks to be processed.

This model is structured so that simulation progression is constrained by input availability, ensuring synchronicity in simulation state across clients.

Artificial latency was introduced during testing to evaluate system behavior under non-ideal network conditions. The value of the latency was set to be approximately 50ms, with variance of 10ms, to mirror the conditions of the Photon Quantum simulation to attain a more accurate comparison between the two models.

6) System Constraints:

While the custom implementation achieves perfect synchronization under controlled conditions, it exhibits several limitations:

- No tolerance to packet loss or delayed inputs
- No rollback or state correction mechanisms

- Reduced responsiveness under latency
- Simplified physics model with limited realism

These constraints highlight the trade-off between strict determinism and practical usability in real-time multiplayer environments.

IV. Experimental Setup

The experiment was conducted using a Unity-based multiplayer scene simulated through Unity's Multiplayer Play Mode with two editor windows. The scene consisted of two player-controlled physics entities placed inside a closed room environment, along with fifty spherical rigidbody entities configured to continuously collide with each other and the environment. The simulation was executed with two connected clients participating in the same session.

All physics calculations were performed deterministically on both clients using Photon Quantum's fixed-point physics engine and the custom physics implementation using FixedMathSharp library respectively. No authoritative physics state or transform data was transmitted during gameplay. Instead, synchronization relied on the exchange of player input commands and deterministic simulations. Rollbacks were triggered only when confirmed inputs differed from locally predicted inputs for the Photon Quantum environment [5]. On the other hand the custom solution enforces strict lockstep simulation with no client-side prediction and rollback.

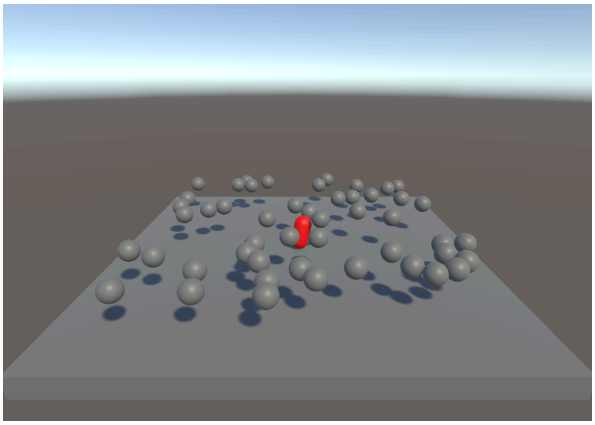

Figure 5(a)

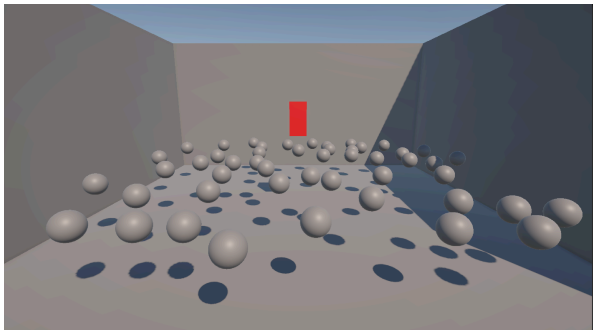

Figure 5(b)

Figure 5. Comparative views of the experimental scene setup for both implementations
(a) Photon Quantum setup.
(b) Custom Implementation setup.

V. Results

To measure synchronization accuracy, the world-space positions of all simulated entities—including both player objects and all fifty ball entities—were recorded independently on each client for 719 simulation frames. This resulted in over 30,000 individual positional samples with the custom physics implementation. Including resimulated frames would have required complex frame alignment across clients and provided limited analytical value, as resimulated frames typically converge to identical deterministic states. The dataset therefore reflects only forward-simulated frames, enabling direct and consistent comparison of entity positions across clients.

For each entity at each frame, the absolute positional difference between clients was computed. From this dataset, two primary metrics were derived:

- Mean positional error, representing average divergence across all entities and frames. Frames that were re-simulated as part of Photon Quantum's rollback mechanism were intentionally excluded from the data collection process for comparative symmetry

- Maximum positional error, representing worst-case divergence

These metrics were selected to quantify both overall determinism and rare correction events.

A. Photon Quantum Results

The data collection for the Photon Quantum simulation was recorded under fairly average network conditions. Average latency was measured to be 57ms.

1) Quantitative Synchronization Metrics:

Across the full dataset, the simulation exhibited a mean positional error of 0.000178853, indicating a high degree of synchronization consistency between clients. The maximum observed positional error was 0.4419556, occurring in isolated instances rather than as a persistent divergence.

Out of the total recorded samples, only 160 data points exhibited a non-zero positional error. Of these, 107 instances were associated with the two player-controlled entities, identified by entity IDs 51 and 52. In contrast, the fifty ball entities demonstrated near-perfect determinism, with minimal observed divergence across the simulation duration.

These results suggest that Photon Quantum's deterministic physics engine maintains robust synchronization for passive physics objects, even under dense collision scenarios.

2) Error Distribution and Player Entity Divergence:

The observed concentration of positional error in player-controlled entities can be attributed primarily to input-driven resimulation behavior rather than numerical nondeterminism. Photon Quantum employs fixed-point arithmetic throughout its simulation pipeline, eliminating floating-point rounding errors as a source of divergence.

Unlike the ball entities, which are influenced solely by deterministic physics forces and collision responses, player entities depend on continuous user input that may be locally predicted and later corrected when authoritative input is received. When discrepancies occur between predicted and confirmed input, Photon Quantum performs a rollback and re-simulation of the affected frames. While this process ensures long-term determinism, it can result in brief positional corrections for player entities.

Additionally, player movement logic typically involves higher-level constraints such as velocity clamping, ground checks, and directional changes, which increase the likelihood of observable corrections compared to passive rigidbody entities. These corrections were transient and did not lead to sustained divergence, demonstrating the effectiveness of Quantum's deterministic rollback system.

3) Visual Results:

Visual inspection confirmed the quantitative findings. Ball entities appeared visually stable and consistent across clients, with no noticeable jitter or snapping. Player entities occasionally exhibited subtle micro-corrections during abrupt movement or collision events, though these corrections were generally imperceptible during normal gameplay.

The following figures are included to illustrate the experimental conditions and results between two player clients – Client A and Client B:

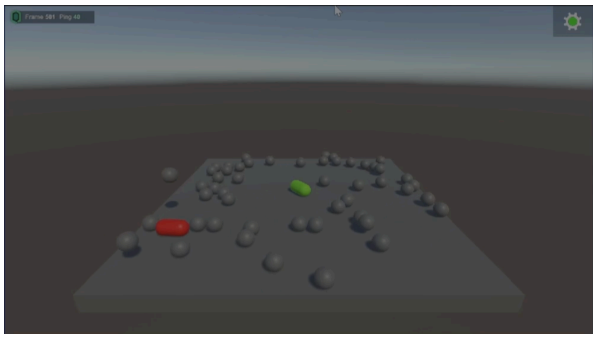

Figure 6(a)

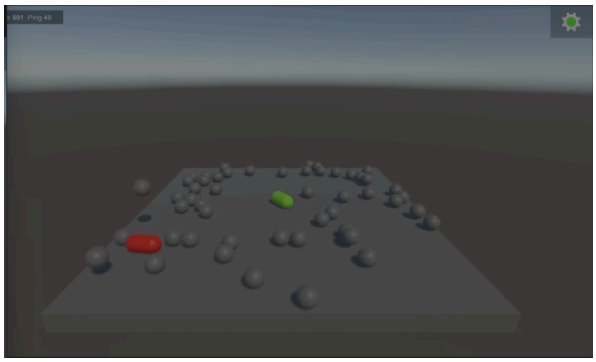

Figure 6(b)

Figure 6. Client-side views of the deterministic simulation at frame 501:
(a) Client A scene view.
(b) Client B scene view.

4) Perceived Gameplay Smoothness and Responsiveness:

The experiment was conducted with both clients connected on the same network under nominal latency conditions. Average latency was measured to be 57ms and peak latency was measured to be 65ms. Under these conditions, the simulation exhibited a high level of perceived smoothness, particularly for passive physics entities such as the ball objects, which showed no noticeable jitter or corrective artifacts.

Player-controlled entities remained responsive, with only subtle micro-corrections observed during abrupt movement changes or collision-heavy

interactions. These corrections were infrequent and did not significantly impact gameplay experience.

5) Computational Cost:

The deterministic simulation model distributes computational cost across all participating clients, as each client executes the full physics simulation locally. In this experiment, the measured mean CPU cost of the deterministic simulation step was 0.1898 ms per frame, representing less than 1% of the total frame time at an average simulation frame rate of 45 FPS (~22.22 ms per frame), indicating minimal computational overhead for the physics simulation itself.

Network activity remained relatively stable throughout the simulation, reflecting the low-bandwidth nature of the player input polling model, in which only input commands are transmitted. In contrast, average network latency (57 ms) had a more direct impact on perceived responsiveness, as simulation progression depends on timely input synchronization across clients.

6) Implementation Complexity and Development Overhead:

Photon Quantum significantly reduces the implementation complexity typically associated with deterministic multiplayer systems by providing built-in support for rollback, resimulation, and state validation. However, the framework imposes strict constraints on gameplay code, requiring all simulation logic to be deterministic and compatible with Quantum's execution model.

While the initial learning curve is non-trivial, the overall development overhead is lower than implementing a custom deterministic or server-authoritative physics system. This makes Photon Quantum a practical solution for small to mid-sized multiplayer games where synchronization accuracy and bandwidth efficiency are prioritized.

B. Custom FixedMathSharp Implementation Results

The data collection for the custom implementation simulation was recorded under local testing on the same device for both clients under artificially simulated network conditions of comparable latency as the Photon Quantum simulation.

1) Quantitative Synchronization Metrics:

Across the full dataset, the custom deterministic simulation exhibited zero measurable positional error between clients. For all recorded frames and entities—including both player-controlled objects and fifty dynamic rigidbodies—entity positions were identical across clients at every simulation tick.

As a result:

Mean positional error: 0.0
Maximum positional error: 0.0

This indicates perfect synchronization under the tested conditions, confirming that the fixed-point arithmetic and strict lockstep model successfully enforced deterministic behavior when no faults were introduced.

2) Error Distribution and Player Entity Divergence:

Under ideal, controlled conditions, no positional divergence was observed for any entity type throughout the simulation. Both player-controlled entities and passive rigidbodies remained perfectly synchronized across all ticks, with no distinction in error distribution between entity categories.

Unlike the Photon Quantum implementation, which exhibited minor transient divergence due to rollback and input correction, the custom system does not perform client-side prediction or resimulation. Instead, simulation advances only when all required inputs are received, eliminating the possibility of temporary inconsistencies.

However, this approach introduces a critical limitation: the system lacks any form of error detection, correction, or rollback.

As a result, while no divergence occurred under controlled conditions, the system is highly sensitive to faults. Any unintended discrepancy such as missed inputs, packet corruption, or nondeterministic code paths would not be corrected and could rapidly propagate, leading to complete desynchronization between clients.

3) Visual Results:

Visual inspection confirmed the quantitative findings. All entities, including both player-controlled objects and dynamic rigidbodies, appeared perfectly consistent across clients with no observable jitter, snapping, or corrective artifacts.

However, despite this perfect spatial consistency, the simulation exhibited visibly discrete updates due to

its low execution rate. Motion appeared step-wise rather than continuous, particularly during player movement and collision-heavy interactions.

4) Perceived Gameplay Smoothness and Responsiveness:

Although synchronization accuracy was perfect, perceived gameplay smoothness was significantly reduced compared to the Photon Quantum implementation. The simulation operated at an average rate of approximately 7 ticks per second.

Because the system is strictly lockstep, this effectively corresponds to ~7 updates per second, producing behavior analogous to a ~7 FPS simulation. This resulted in noticeably delayed input response, choppy and discontinuous motion, and reduced responsiveness during interaction.

Additionally, the artificially induced network conditions (latency, jitter, and packet loss) caused frequent delays in simulation progression, as each tick required all player inputs to be received before advancing.

These results highlight a fundamental trade-off. Strict synchronization guarantees come at the cost of responsiveness under non-ideal network conditions.

5) Computational Cost:

The computational cost of the custom deterministic simulation remained low throughout testing. Due to the reduced simulation rate (~7 ticks per second), CPU workload per second was significantly lower than that of higher-frequency real-time simulations.

However, similar to the Photon Quantum results, low computational cost in this context does not necessarily reflect greater efficiency. Instead, it is largely a consequence of reduced update frequency and network-bound execution constraints.

Thus, performance in this system is primarily limited by network synchronization rather than computational overhead.

6) Implementation Complexity and Development Overhead:

The custom FixedMathSharp implementation provides complete control over the deterministic simulation pipeline, including arithmetic precision, execution order, and networking behavior. This level of control allows for precise enforcement of determinism and full transparency into system behavior.

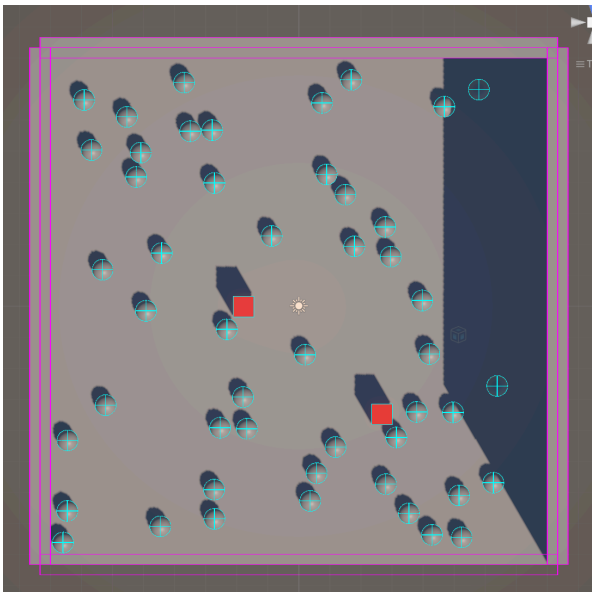

Figure 7(a)

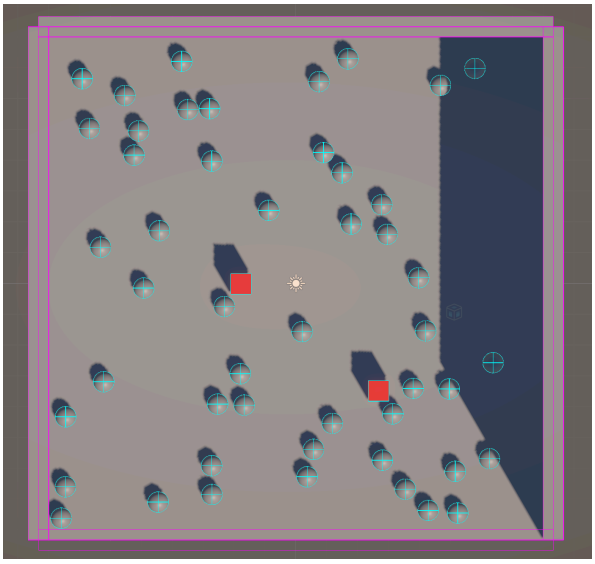

Figure 7(b)

Figure 7. Players' top-down editor views of the deterministic simulation at tick 793:
(a) Host editor view.
(b) Client editor view.

However, this approach significantly increases development complexity. Core systems such as physics integration, collision handling, and synchronization logic must be implemented manually, introducing additional sources of potential error.

Furthermore, the absence of higher-level systems such as rollback, prediction, and state validation places additional constraints on robustness. While the system achieves perfect synchronization under ideal conditions, it lacks resilience to real-world network inconsistencies and runtime faults.

In contrast to Photon Quantum, which provides built-in mechanisms for handling divergence and maintaining responsiveness, the custom implementation demonstrates the trade-off between full system control and practical robustness in networked environments.

VI. Comparative Analysis

A direct comparison between Photon Quantum and the custom FixedMathSharp implementation highlights a fundamental architectural trade-off between corrective determinism and strict lockstep determinism.

Photon Quantum employs a hybrid deterministic model based on input prediction and rollback resimulation. This design enables continuous simulation progression even under conditions of temporary input disagreement. As a result, it achieves superior perceived responsiveness and smoother gameplay behavior in dynamic interaction scenarios such as player movement and collision response. However, this approach introduces transient divergence and requires additional complexity in state history management, input reconciliation, and resimulation pipelines.

The custom implementation, by contrast, enforces a strict lockstep execution model in which simulation advancement is fully dependent on synchronized input availability across all clients. This guarantees perfect state agreement at every simulation step and eliminates all forms of divergence. However, it also introduces a direct dependency between network latency and simulation progression, resulting in reduced responsiveness and visible temporal discretization of motion under non-ideal network conditions.

An additional structural distinction lies in engine architecture. Photon Quantum is built on a DOTS/ECS-based framework optimized for parallelized simulation and large-scale entity processing. The custom system is implemented using traditional Unity runtime architecture using FixedMathSharp, avoiding DOTS dependencies entirely. While this limits access to engine-level optimizations, it increases portability and integration feasibility within traditional Unity projects that do not adopt ECS-based workflows.

Overall, Photon Quantum represents a production-optimized deterministic system prioritizing responsiveness under uncertainty, whereas the custom implementation represents a minimal deterministic baseline prioritizing correctness, transparency, and architectural simplicity.

VII. Discussion

The comparative results indicate that deterministic correctness alone is not sufficient to ensure high-quality multiplayer interaction. Instead, the primary determinant of perceived gameplay quality is how a system manages temporal inconsistency introduced by network latency.

Photon Quantum demonstrates that controlled short-term nondeterminism, when paired with rollback and resimulation, can produce a smoother and more responsive user experience. Although minor positional divergence is observed in player-controlled entities, these discrepancies are transient and converge rapidly due to corrective resimulation. This suggests that bounded inconsistency at the simulation edge is acceptable when it preserves continuous interaction.

In contrast, the custom FixedMathSharp implementation achieves perfect numerical consistency across all entities but exhibits reduced responsiveness due to its strict lockstep execution model. Because simulation progression is gated entirely on input synchronization, any network delay directly manifests as simulation stalling. This results in a more rigid and less fluid gameplay experience, despite the absence of any measurable divergence.

These findings indicate that perceived smoothness is more strongly correlated with simulation continuity than with strict determinism. In practical multiplayer systems, responsiveness mechanisms such as prediction and rollback may therefore be more critical than eliminating all forms of transient divergence.

VIII. Limitations

This study is subject to several limitations that affect both the scope and generalizability of the findings.

The custom FixedMathSharp implementation uses a simplified physics model restricted to sphere and axis-aligned bounding box (AABB) colliders. The system does not support rotational dynamics, torque, angular momentum, drag, or frictional material

variation. As a result, physical interactions are less expressive than those found in full-featured physics engines, limiting the realism of the simulation.

Additionally, the custom system does not implement rollback, client-side prediction, or state reconciliation mechanisms. While this ensures strict determinism under ideal conditions, it also introduces a structural inability to recover from network inconsistencies such as packet loss, jitter, or delayed input arrival. Any such disturbance would result in persistent desynchronization or halted simulation progression.

A further limitation arises from differences in engine architecture. Photon Quantum is implemented using a DOTS/ECS-based system designed for high-performance parallel simulation, whereas the custom implementation is built using standard Unity MonoBehaviour workflows. This introduces an inherent asymmetry in scalability and execution efficiency that may influence comparative performance characteristics.

Finally, the evaluation environment is limited to a relatively small-scale simulation with 52 dynamic entities and does not include large-scale stress testing or long-duration session analysis.

IX. Future Work

Future work should focus on extending the custom FixedMathSharp system toward a hybrid deterministic networking architecture suitable for real-time multiplayer gameplay.

A primary improvement is the integration of client-side prediction combined with rollback-based reconciliation. This would allow the simulation to remain responsive under latency while preserving deterministic correctness through periodic state correction. Such an approach would directly address the current limitation where simulation progression is strictly dependent on synchronized input arrival.

In addition, expanding the physics system beyond simplified collider types represents a necessary step toward practical applicability. Future iterations should incorporate support for rotational motion, torque, angular velocity, drag forces, and friction models. These additions would enable more realistic and expressive physical interactions while maintaining deterministic guarantees through fixed-point arithmetic.

From a networking perspective, hybrid synchronization models that combine lockstep determinism with localized prediction windows may provide a more balanced trade-off between responsiveness and correctness. Such systems could reduce perceived latency while maintaining global determinism over longer time horizons.

Finally, the use of a traditional Unity runtime architecture implementation presents a practical advantage in terms of integration. Unlike DOTS-based systems, the current architecture can be more easily embedded into existing Unity projects without requiring full ECS adoption. Future work may explore modularizing the deterministic layer to function as a drop-in networking and physics subsystem for conventional Unity-based game architectures.

References

[1] A. Abdelkhalek and A. Bilas, "Parallelization and performance of interactive multiplayer game servers," in Proc. 18th Int. Parallel Distrib. Process. Symp., Apr. 2004, doi: 10.1109/IPDPS.2004.1303003.

[2] A. J. R. Brown, "Distributed real-time physics for scalable and streamed games and simulation," Ph.D. dissertation, Newcastle Univ., 2020.

[3] S. Ferretti, "Interactivity Maintenance for Event Synchronization in Massive Multiplayer Online Games", Ph.D. Thesis, Tech. Rep. UBLCS-2005-05, University of Bologna (Italy), March 2005.

[4] S. C. McLoone, P. J. Walsh, and T. E. Ward, "An enhanced dead reckoning model for physics-aware multiplayer computer games," in Proc. IEEE/ACM 16th Int. Symp. Distrib. Simul. Real-Time Appl., Oct. 2012, pp. 111–117, doi: 10.1109/DS-RT.2012.22.

[5] P. Mieschke, "Deterministic lockstep in networked games," 2024. [Online]. Available: https://hdms.bsz-bw.de/frontdoor/index/index/docId/7107. Accessed: Sep. 21, 2025.

[6] S. S. de Oliveira, C. H. R. Souza, and S. T. Carvalho, "A multiplayer cloud gaming architecture for scalable physics," in Proc. IEEE Conf. Games

(CoG), Aug. 2024, pp. 1–8, doi: 10.1109/CoG60054.2024.10645556.

[7] L. Pantel and L. C. Wolf, "On the suitability of dead reckoning schemes for games," in Proc. 1st Workshop Netw. Syst. Support Games, Apr. 2002, pp. 79–84, doi: 10.1145/566500.566512.

[8] H. Viitanen, Deterministic and synchronous computation between client and server in mobile games, Jul. 23, 2025. [Online]. Available: https://aaltodoc.aalto.fi/handle/123456789/138185. Accessed: Sep. 21, 2025.

[9] Newcastle University. "Game Engineering - Newcastle University." Accessed May 3, 2026. https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/4collisiondetection/#d.en.854591.