Vrije Universiteit Amsterdam

Bachelor Thesis

# Lock-Step Simulation for Modifiable Virtual Environments (MVEs)

**Author:** Diar Kamberi (2738305)

*1st supervisor:* Jesse Donkervliet
*2nd reader:* Alexandru Iosup

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

August 20, 2024

## Abstract

The rapidly growing gaming industry, with its millions of players worldwide, increasingly demands scalable solutions to support massive multiplayer experiences. Games like Minecraft, which emphasize modifiable virtual environments (MVEs), face unique challenges in maintaining performance and consistency as player counts grow. The lock-step simulation approach, often employed in strategy games like Age of Empires and Civilization, offers a solution by ensuring synchronized state updates across all clients, thus maintaining fairness and consistency.

The problem of applying lock-step simulation to MVEs is important and challenging. Lock-step simulation is crucial for maintaining good consistency in game state, which is essential for a fair and immersive multiplayer experience. However, MVEs typically require high bandwidth usage and low latency due to frequent terrain updates, and the challenge lies in efficiently simulating only parts of the world on client devices. These devices must manage a large amount of state data, raising concerns about their capacity to handle the computational load without compromising performance. The core contribution of this work includes a design for a lock-step synchronization system tailored to MVEs, emphasizing *bandwidth efficiency*, *reduction in perceived latency*, and *good game state consistency.* Our results show a quadratic increase in bandwidth usage relative to the number of players, with near-identical bandwidth usage regardless of the complexity of player interactions. Latency experiments reveal significant impacts on update times, with higher latencies leading to noticeable increases in tick duration. However, the introduction of an *input offset of five turns* is shown to mitigate these delays effectively, improving the tick durations of the simulation by 4.2 times. *A promising approach* to distributed environments, such as Area Of Simulation (1) may further improve scalability in terms of player count, but is not a sufficiently explored direction of research.

# Contents

# 1

# Introduction

## 1.1 Context

The internet has seen immense growth over the last decade, where a considerable portion of network traffic is generated by video games, with a share of around 10% in 2021 (2). Low-latency, high bandwidth, scalability and support for various platforms are some of the key requirements in online video games. Real-Time Strategy (RTS) games, containing a large number of interactive objects (units), generate significant load on the network if using a traditional networking approach of communicating the state of the game for hundreds or thousands of units. To alleviate this problem, the developers of Age of Empires (AoE) (3) - a popular RTS game - communicate only the commands a user executed instead of the state, which is simulated in all the other clients. With this method, refered to as lock-step, it became possible to control 1500 archer units in an online video game, contributing to the online gaming landscape.

## 1.2 Problem Statement

The increasing demand for large-scale, interactive modifiable environments (2) introduces significant challenges in terms of managing resources, complexity and responsiveness of computer systems. MVEs often struggle with scalability and responsiveness due to the numerous entities and large world sizes involved, where few hundred players are supported in a virtual world (4). Offered as a service, this number can be even lower, seen in (5), where the simultaneous player count is limited to 10. This limitation hinders the development of complex, sustainable, multiplayer environments across fields such as education and entertainment, relating to the *manageability* and *sustainability* challenges indicated in the CompSysNL manifesto (6).

Lock-step simulation, a technique proven successful in online games like Age of Empires (3), offers a potential solution to scalability and bandwidth usage issues by distributing the computational load across the clients. By simulating the game state on each client rather than relying on a central server for all computations, such as in a peer-to-peer network architecture seen in Figure 2.2; lock-step simulation offers less bandwidth usage and simplifies maintaining good game state consistency across the clients.

On the other hand, lock-step simulation does not scale well in terms of player count, because each client needs to wait for all other clients to transmit their inputs. This indicates that with an increasing player count, the players with high latency slow down the simulation for all other clients. Furthermore, the latency requirements of such a system pose problems for the use of lock-step in the First-Person Shooter (FPS) category of games, which is why FPS games rarely use lock-step.

The potential benefits and inherent problems of lock-step simulation motivate us to explore its integration in MVEs. This study aims to adapt lock-step simulation to the context of MVEs and evaluate its feasibility and effectiveness as a distributed system. The findings of this study could propagate further research into developing a more scalable, hybrid approach such as Area of Simulation (1). Aligning with the AtLarge vision (7), this study embraces heterogeneity, decentralization and adaptibility by having localized game simulation, resource management and accommodating bad network conditions in Modifiable Virtual Environments, thereby promoting scalability and resilience in distributed settings.

## 1.3 Research Questions

In this section, we present the concrete questions that we are going to answer throughout this paper. The questions we address are as follows:

- **RQ1**: How to design a scalable and performant lock-step simulation system for MVEs?
- **RQ2**: How to implement a system that integrates lock-step simulation in MVEs?
- **RQ3**: How does lock-step simulation impact the scalability and performance of MVEs?

## 1.4 Research Methodology

To address our research questions, we follow common/accepted research methodologies in the field:

- **M1**: The research methodology used to answer **RQ1** is designing a system that conforms to the principles laid out in (8).
- **M2**: To answer **RQ2**, this thesis will employ a design science research methodology as presented in (9) to develop and evaluate an artifact, which in our case, is a prototype integrating lock-step simulation within an MVE.
- **M3**: For **RQ3**, we will design appropriate micro-level and workload-level experiments, quantifying the running system prototype developed using **M2**. To achieve fair and realistic results, we will follow the groundwork and best practices shown in (10, 11, 12).

## 1.5 Thesis Contributions

This thesis makes the following contributions to the field of MVEs and distributed systems:

- **Conceptual:** A framework for integrating lock-step simulation in MVEs, along with design principles for scalability and performance.
- **Technical:** Technical challenges and corresponding solutions for lock-step MVEs.
- **Experimental:** A prototype implementation of a lock-step MVE and its thorough evaluation in terms of scalability and performance.
- **Artifact:** Open-source release of the prototype implementation. Available in Appendix A

## 1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment

## 1.7 Thesis Structure

The structure of the thesis follows the AtLarge BSc Thesis template. Chapter 2 lays the necessary groundwork for understanding the concepts and processes shown in the paper. In Chapter 3, we describe the design challenges we address and the approach to solving these problems. Based on the foundation set by Chapter 3, Chapter 4 contains the implementation details of creating such a system. In Chapter 5 we evaluate our system in relation to the requirements set in Chapter 3. The later chapters 6 and 7 highlight related work in the topic and the conclusions drawn from the research.

# 2

# Background

There is not much research on the topic of MVEs, and even less so in terms of using lock-step simulation in MVEs. While deterministic lock-step is used in many games (3), there is a lack of academic research on the integration of lock-step in MVEs. In this chapter, we describe the core ideas underlying our research questions presented in Section 1.3.

## 2.1 Online game architectures

The architecture of an online game plays a crucial role in building a successful Massively Multiplayer Online Game (MMOG). Scalability, performance and cost are some of the key elements that influence this decision. In this section we discuss the two most common architectures for online games. Both of these architectures can be used to create lock-step simulation games, but make different trade-offs. We discuss these trade-offs in the remainder of this section.

### Client-Server architecture

In the client-server architecture, the server acts as an authoritative source of truth. The simulation runs on the server and the game state is synchronized between the server and clients. The clients are responsible for rendering the graphics, sending player actions and displaying the interface to the player. Since the game simulation runs on an authoritative server, as illustrated in Figure 2.1, it provides control over the game state. Therefore, enabling a consistent state saved on the server and cheat protection, since any invalid game state will be overridden in the next state synchronization.

However, the simple centralized architecture means a single point of failure. If the server becomes unavailable, the entire system is disrupted, losing all synchronization between

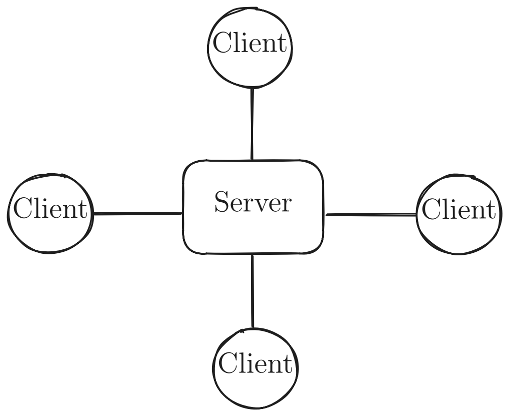

**Figure 2.1:** Client-Server Architecture Diagram

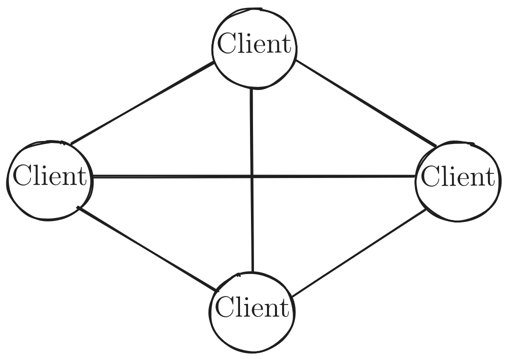

**Figure 2.2:** Peer-To-Peer Architecture Diagram

clients. Consequentially, attackers often target these game servers for this reason. Another problem is the issue of latency, introduced by the Round-Trip Time between the clients and the server in addition to the latency induced by the server processing the game state.

These client-server architectures can be extended to fit your needs. For instance, we have distributed architectures and thin-client architectures. In distributed architectures, there are multiple servers connected with each other, and connected to the closest clients. This distributed approach shows the scalability potential of the client-server architecture. In a thin-client approach, the server can do all the processing, and the client only gets the results (frames), allowing for clients with insufficient computational power to still play the game.

### Peer-to-peer architecture

In a peer-to-peer (P2P) architecture, there is no central server, instead the clients are connected directly to all other clients which is illustrated in Figure 2.2. For $n$ clients, there are $n = \frac{n(n-1)}{2}$ unique client-to-client connections. The lack of a single point of failure, and low costs make this system enticing to game developers. P2P networks are inherently scalable. As more clients join the network, resources increase. Furthermore, direct connections between the clients means that the latency is reduced in comparison to server-client architectures.

Although there are many benefits to a P2P network, it offers its own array of issues. The absence of a central source of truth, means that clients are able to cheat, with no way to verify the actual game state. Broadcasting the game state is also required for every update due to no central game state simulation, increasing bandwidth. In addition, maintaining consistency between clients poses a complex configuration and implementation challenge.

Peer-to-peer networks can also be adapted to fit certain environments. To overcome the complexity induced by NAT addresses, a relay server can be deployed, with the purpose of forwarding the packets to other clients. This can be further expanded to a Hybrid P2P, where multiple nodes communicate with each other to help clients connect with one another, whilst maintaining the fault-tolerance of P2P.

## 2.2 Deterministic systems

Determinism essentially refers to a function/system that always gives the same output for a given input. Cross-platform determinism is notoriously hard to achieve. Reasons for this are the various sources of non-determinism. Floating-Point Units, present in most mainstream CPUs, have hardware acceleration optimizations that significantly speed up floating-point operations. However, these optimizations vary across different CPU architectures, leading to different results with the same inputs. Another source of non-determinism are compilers, which can re-order instructions or use different registers, resulting in timing differences or different order of memory accesses. Multi-threading, a parallelization technique with many benefits, can also be a source of non-determinism. Core scheduling can result in different order of operations if not handled carefully, leading to varying results. A consistent source of non-determinism are random number generators. After all, they are meant to be random, resulting in different results, even on the same machine.

Ensuring cross-platform determinism can be a tedious and complex endeavour. Developers facing this issue often have to rewrite libraries, or implement needed functionality from scratch with determinism in mind. The most common solution to floating point non-determinism is using fixed-point math instead. Fixed-point numbers have a consistent representation across devices since they can be treated as integers. Compilers should use consistent flags and avoid agressive optimizations, ensuring mostly similar assembly code across devices. Multi-threading is sometimes completely disabled to ensure determinism, since introducing many constraints on it will only result in a sequential implementation with little to no parallelism. Random number generators can be used as pseudo-random number generators, utilizing an agreed upon seed and having the exact same number of API calls to the generator, ensuring consistent results across devices.

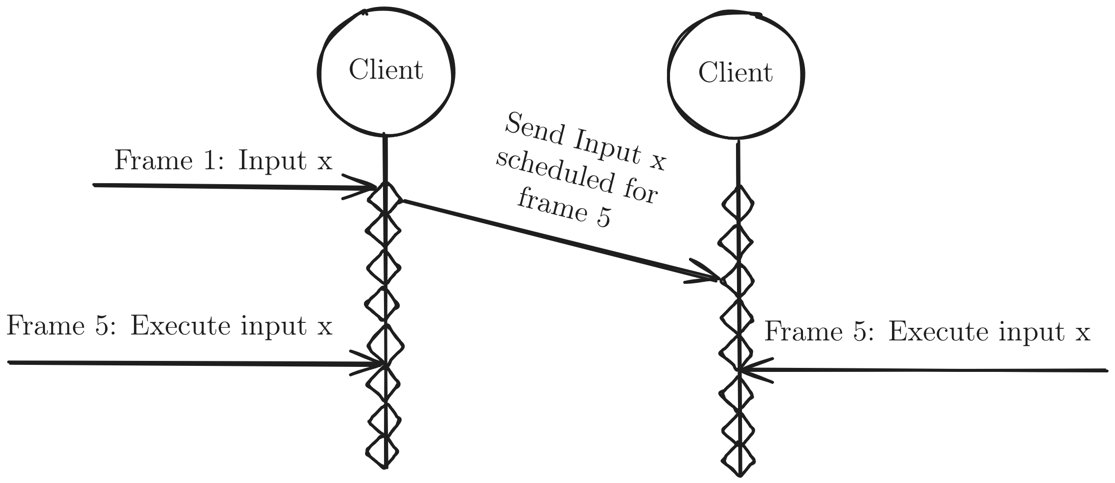

**Figure 2.3:** Lock-step with 4 frames input delay. (Diamond shapes represent simulation frames)

## 2.3 Deterministic lock-step with input delay

Lock-step is an architecture where each client simulates the game state, and the game state is synchronized by exchanging only the inputs of each player. This approach is scalable in terms of entity count and world size. To ensure that the game simulation is the exact same in all clients, the simulation needs to be fully deterministic, as outlined in Section 2.2. To achieve lockstep simulation, the simulation frames need to be separated from the rendering frames, since rendering frames depend on the computational power of the device. However, if there is significant latency, waiting for input from each player for each frame can make the gameplay feel laggy. To counteract this, a commonly used method is to schedule player inputs for some simulation turns in advance. This introduces some input delay in the local client, but masks the latency. An illustration of the execution is shown in Figure 2.3.

Using input delay can counteract minor latency issues, however, there are still some problems present with lockstep. In cases of significant latency where the client does not have all the inputs to simulate the frame, it has to stop and wait for synchronization. This is where the *lock* in lockstep comes from. Furthermore, since we need the input from all clients for a simulation frame, the simulation time is set to the slowest client. For instance, if a client has 100ms delay, each simulation frame on all clients is $100ms + processingtime$

## 2.4 Prediction and Rollback

Prediction and rollback are techniques used in online games to mask bad networking conditions and/or ensure a more responsive gameplay experience. Latency is present in any networking conditions. Prediction and rollback are two techniques that are employed to display a smoother experience despite this latency.

Server-authoritative architectures are heavily utilized in most mainstream virtual games. In a server-authoritative approach, a client has to request for a change to the game state, and the server then responds with the game state validating or invalidating the game state. The client renders the game state once it has received validation and the game state from the server. However, this reliance on the server to change the game state results in visible jitter due to the client updating its view to match the servers. With client-side prediction, the client immediately applies the changes to the game state, assuming that the server will approve it. This results in immediate feedback in the players game state, offering a responsive and smooth experience.

Rollback is a technique analogous to client-side prediction. In its simplest terms, it refers to reverting the game state to a point where it was valid. For example, when a client-side prediction turns out to be invalid, rollback can be applied to revert the game state to a certain valid state. After applying rollback, the client has to catch up to the real-time game state, which can be done by re-simulating everything up to the current point. This can sometimes introduce heavy delay in the rollback frame, so a different approach is to speed up the simulation until it reaches the current simulation state.

Modern multiplayer games traditionally employ a hybrid approach, using these techniques in conjuction to maintain smooth gameplay in addition to maintaining a consistent gamestate with the authoritative server. This combination allows for responsive gameplay while minimizing jarring visual anomalies. Other advanced methods, such as lag compensation and snapshot interpolation, further enhance the perceived smoothness and responsiveness of the game by accounting for network latency during input and predicting the movement of other players for rendering.

## 2.5 Serialization

Serialization, in basic terms, is the process of converting an object or data structure into a sequence of bytes that can be easily transmitted or stored. In terms of networking, it is often used to reduce the amount of data needed to be transferred and to increase interoperability between different machines. Let's say we have a client A and a client B. They both have a serialization and deserialization interface. If client A wants to send some data, it simply applies a serialization method on the data, which reduces the size of the data, converting it into a bit sequence, and client B simply reconstructs the data by applying the deserialization method. The reason for the reduction in a more compact size is because of the internal representation of the data in memory. To illustrate, binary data can be encoded as a single bit since it can take only the value 0 or 1, or ASCII characters can be encoded as 7 bits since the most significant bit is always 0. To conclude with, the main benefits that serialization provides are size reduction, interoperability and persistence.

# 3

# Lock-Step Synchronization: A Design for Collaborative Virtual Worlds

In this section we present the requirements and high-level design employed to integrate lock-step simulation with MVEs. This comprises the base for our following work on the implementation and subsequent evaluation of such a system, therefore it is important that the requirements are clear and achievable in alignment with our research question R1:

> *" What are the key requirements of designing a scalable and performant lock-step simulation for MVEs? "*

## 3.1 Requirements analysis

We have identified the following **functional** requirements:

- **FR1** Deterministic simulation: The system must ensure that all game simulations running across the clients are deterministic. Otherwise, game state will be inconsistent across devices. See Section 2.2 for more details on sources of non-determinism and potential solutions.

- **FR2** Input synchronization: The system must synchronize inputs across all clients to provide a multi-player environment in which clients can interact with each other and the virtual world.

We have identified the following **non-functional** requirements:

- **NFR1** Input Delay: The system should introduce an input delay of at least 5 turns to mitigate the effects of latency up to 300ms. This delay ensures that all clients can receive and process inputs within a tick duration of 50ms. A tick duration of maximum 50ms provides a smooth gaming experience. For more details and a visual explanation, see Section 2.3. Furthermore, the input delay should be no more than 10 turns, otherwise perceived input lag reduces responsiveness of the system to a less than acceptable level.

- **NFR2** Frame Input Serialization: The system should serialize frame input data such that each serialized packet contains the minimum amount of bytes required to make the game work, in the case of MVEs, we found that not exceeding 4 bytes is a low enough size to allow for proper game operability as shown in Section 3.3. Proper serialization methods can reduce the size of data packets, which is essential for ensuring efficient bandwidth usage in lock-step games. Furthermore, serialized data is easier to handle and process on different platforms, improving compatibility and interoperability. For further details, see Section 2.5. Additionally, the system should be able to pack at least 3 serialized frames into a single packet to support redundancy in UDP channels, helping with overall bandwidth efficiency by requiring less re-transmissions.

- **NFR3** Scalability in World Size and Interaction Complexity: The system should be capable of supporting a minimum of 64 simultaneous players within a flat Minecraft-like world without experiencing significant performance degradation or high bandwidth usage. As the number of players or entities increases, the system should maintain a frame rate of at least 60 FPS. A scalable system ensures that as more players join the game, or as the virtual world becomes more complex, the system can still operate smoothly. This involves efficient resource management, optimized data transmission, and robust network architectures. Scalability is crucial for providing a consistent and enjoyable experience for all players, regardless of the number of participants or the complexity of the game world.

## 3.2 Design Overview

In this section we present a high-level design of our system in line with our requirements identified in Section 3.1. A diagram of a high-level design that fulfills the requirements is

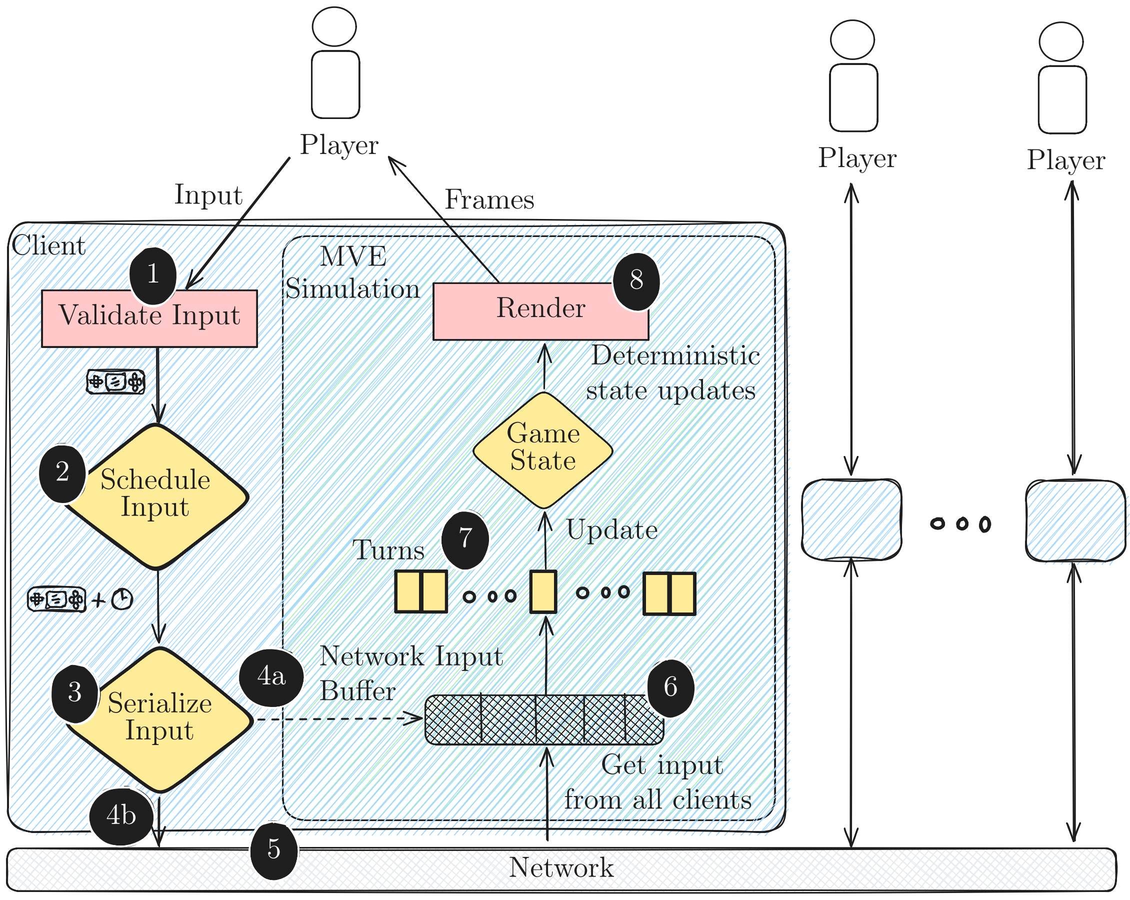

**Figure 3.1:** A High-Level Design for Lock-Step Simulated MVEs.

shown in Figure 3.1. At a first glance we see that every player interacts with the client of the game. It is also apparent that there is no explicit server component in step ❺. A lock-step architecture does not necessarily need a server to work. Either a peer-to-peer architecture or client-server architecture can be used to achieve lock-step, with their respective advantages and disadvantages as shown in Sections 2.1 and 2.1. The players interact with the game state by pressing inputs. In step ❶ the inputs are validated based on the constraints of the game to prevent cheating. Furthermore, since every client runs its own simulation of the game, the game is resilient against traditional cheating approaches.

The input from the client is then scheduled for a simulation turn in the future in step ❷. Note that simulation frames are decoupled from rendering frames, thus we refer to simulation frames as *turns* to avoid confusion. Introducing input delay can smooth out the gameplay under tougher network conditions, fulfilling requirement **NFR1**. The simulation turn to schedule the input depends on the latency of the system, ranging from zero turns all the way up to $n$ turns. $n$ should be a sufficiently low number to not introduce noticeable input delay on the client. This input is then serialized as in Section 3.3 in step ❸ to send it over the network and reduce bandwidth usage. Component ❷ and ❸ fulfill requirements **NFR2** and **NFR3**. For further details, look in Section 2.5.

After input serialization, in step 4b we send the input over the network. An optional approach 4a is to immediately queue the local input buffer in the input buffer. This step may be necessary based on the architecture we choose. If in ❺ we use a peer-to-peer architecture, we must queue the input in the input buffer since the other clients will not echo it back. In ❺, we have the liberty of choosing our networking architecture. This can range from a simple p2p system to a distributed server design. In this step, we must make sure that we can coordinate between the clients, handling connection, disconnects and naturally, distributing input over all the clients, therefore fulfilling requirement **FR2**. This functionality can be handled by a server or a master client.

When receiving remote input from the network, we queue it in the input buffer ❻. The input buffer contains all the inputs from the clients. Only when we have input from all clients for a specific turn can we simulate the state. This brings us to step ❼, where we update the game state deterministically. If we do not have all the input for a simulation turn, the clients will "lock" and wait until the input has been received. If we do not lock, we can desynchronize, where an input on a client may be executed on a turn, but the other clients execute it in the wrong turn. Following all the advice outlined in Section 2.2, we fulfill requirement **FR1**. In step ❽, the game is rendered based on the game state that we have locally, and these frames are shown to the player.

## 3.3 Frame Input Design

In our design, we are still lacking a valid frame input design. It is a crucial part of a lock-step system since this will be what each network packet between clients will carry. We present a frame input diagram in Figure 3.2. The proposed design tries to use the least bytes necessary to make a lock-step game. The explanations of each field are as follows:

- `Client ID`: Represents the identification number for each client. This is used to identify which client the input belongs to.
- `Frame ID`: Represents the identification number for each simulation frame. This is used to identify which frame the client input belongs to.

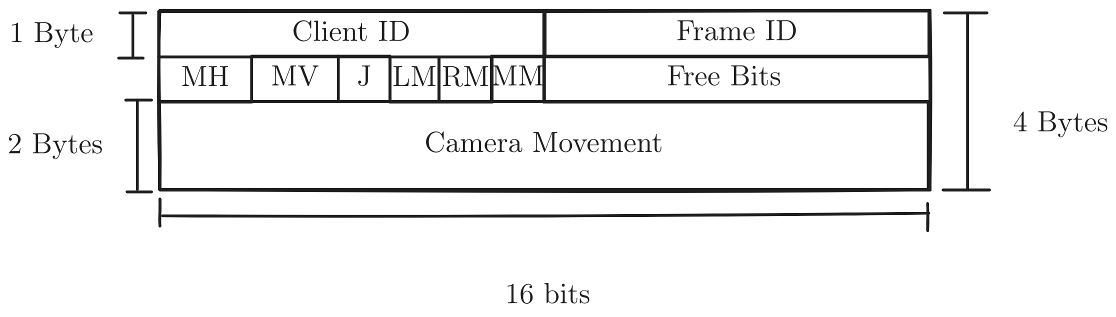

**Figure 3.2:** A design of an input frame for MVEs.

- `MH`: 2 bits to represent horizontal movement. 00 represents no movement, 01 represent right movement, 10 represents left movement and 11 is reserved for future use.
- `MV`: 2 bits to represent vertical movement. 00 represents no movement, 01 represent upward movement, 10 represents downward movement and 11 is reserved for future use.
- `J, LM, RM, MM`: 1 bit to represent a jump, destroy/place block and attack action. They can either be 0 meaning not pressed, or 1 meaning pressed.
- `Camera Movement`: 32 bits to represents camera movement

In this design, the ClientID and the Frame ID are arbitrarily set to 1 byte each. If needed, they can be extended to 2 bytes each. The movement vectors represent the minimum bits required to represent movement direction. An alternative is to use 3 bits for 8 movement vectors, however, it is useful to have the reserved for future bits in case more complicated movement is necessary. The next 4 bits are self-explanatory. They denote a certain action in the game, with the state of each being pressed or not pressed. The next 8 bits are for free use, relating to any actions needed in the game. They may correspond to NPC interactions or any other needed actions.

The last field of the Input Frame is the Camera Movement. While 4 bytes may seem excessive, it allows for $2^{32} = 2,147,483,647$ combinations, resulting in smooth camera movement. In some cases, it may even be necessary to include more bytes for smoother camera movement and ease-of-use.

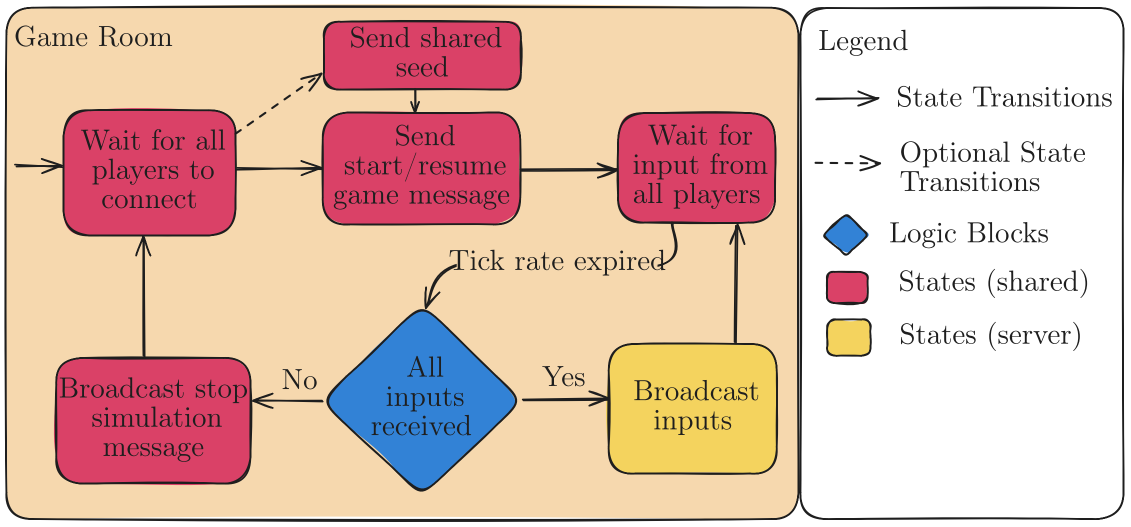
Game Room
Send shared seed
Wait for all players to connect
Send start/resume game message
Wait for input from all players
Tick rate expired
All inputs received
No
Yes
Broadcast stop simulation message
Broadcast inputs
Legend
State Transitions
Optional State Transitions
Logic Blocks
States (shared)
States (server)

**Figure 3.3:** Server/host state diagram for a Game Room.

## 3.4 Server/Host Design

In Figure 3.1, we have seen how our lock-step system works at a high-level. However, the synchronization part in ❺ between clients for a game world requires a more detailed design to provide an in-depth design of the system. A state diagram is shown in Figure 3.3, that depicts a game room and its processes. A game room, in concept, is similar for both a server and a host, which is why there are so many shared states between the two. Notably, there are no distinct states for the host (apart from the game simulation, which is a separate component from the game room). *Host* refers to a client who is a player and is responsible for the game room, seen commonly in p2p systems.

# 4

# Quantumcraft: A Prototype Implementation of Minecraft Using Lock-Step

In this section, we detail the prototype implementation based on the design outlined in Section 3.2. This involved making key decisions regarding the game engine, rendering, and strategies to ensure determinism. The resulting QuantumCraft prototype, an overview of which is shown in Figure 4.1, serves as a practical exploration of research question R2:

> *" What are the technical challenges and implementation strategies required to integrate lock-step simulation in an MVE? "*

## 4.1 Achieving deterministic simulation

A crucial part of the lock-step system is deterministic game state simulation. The high-level problems are mentioned in Section 2.2. An issue encountered early in the implementation is that most game engines/frameworks are not made to be deterministic. For example, the Unity physics engine of Unity is not deterministic across devices. In fact, not only the Unity physics engine, but also other popular physics engines are not deterministic. This is mainly due to performance problems encountered by using fixed point math. Aside from making our own physics engine, which is out of the scope of this thesis, we had to choose between using a deterministic implementation fork of the Open-Source BEPUPhysics engine, or using the Photon Quantum game engine (13).

Photon Quantum is a game engine that provides us not only with deterministic physics, but also handles the networking part of Lock-step. On the other hand, to use BEPUPhysics,

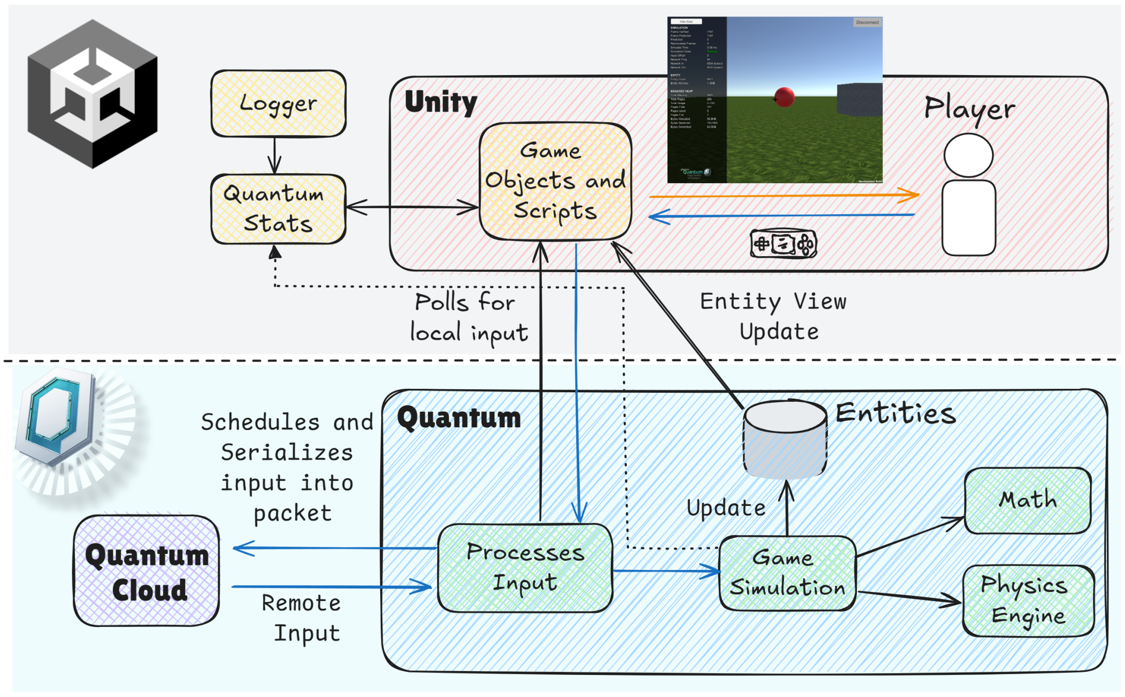

**Figure 4.1:** Overview of the QuantumCraft prototype implementation. Blue arrows indicate input path, orange arrow indicates rendered frames, arrowheads indicate interaction, dotted line for clarity purpose

we would need to make an interface between Unity, which we use for rendering, and the engine itself. This would not be trivial. Our final choice was to use Photon Quantum for its ease-of-use and the increased performance due to it internally using Entity Component System architecture.

## 4.2 Quantum/Unity interoperability

Photon Quantum is a game engine that works on top of Unity. It mainly uses Unity to get input and render frames. All relevant game assets have been placed in the Database used by Quantum, which makes them accessible as entities. This allows us to apply physics and control game objects. Furthermore, it creates an entity view for each entity prototype, which means it can be rendered by Unity. This allows us to create the Virtual Environment and interact with it.

Quantum polls Unity for new input in every simulation turn. An important detail here is that all floating point inputs should be converted to fixed-point to avoid non-determinism.

This input is then handled to Quantum for processing of the game state based on the input.

## 4.3 Integrating Quantum with MVE

After we chose to use Quantum as our final approach, we made a simple prototype game that we can use to conduct experiments on. To make our prototype implementation, we had three objectives in mind:

1. Make a voxel-based virtual environment with flat-terrain.
2. Make a player controller that allows the user to move around, place and remove blocks.
3. Reflect the changes to the game state across all clients.

### Terrain

To create our terrain, we created three block prototypes: stone block, dirt block and grass block; textured with free assets from the Unity asset store. Each block is a *1x1x1* cube with a collider. Positional data of blocks is expressed with integers, conveniently creating grid-like terrain.

With the block assets available, we made a simple terrain generator that gets called when the game starts. Since all clients make the same flat terrain, we have no synchronization issues. Alternatively, we could use random noise to generate terrain, granted that we use the same seed on all machines to maintain a deterministic simulation

### Player controller

The player's movement and camera control are governed by a system that operates on entities possessing the following components:

- `CharacterController3D`: Manages physics-based movement.
- `Transform3D`: Holds position, rotation, and scale data.
- `PitchYaw`: Stores pitch and yaw angles for camera control.
- `PlayerLink`: Links the entity to the corresponding player input

```
FPQuaternion.CreateFromYawPitchRoll(filter.PitchYaw->Yaw,
↪  -filter.PitchYaw->Pitch, 0);
```

**Figure 4.2:** Pitch and Yaw to Quaternion Conversion

### Camera Setup

During game initialization, a camera game object is attached to the player entity. This is achieved by setting the camera to be a child of the player object, ensuring synchronized position and rotation. Worthy of note, is that there is only one camera per game instance, attached to the player owned entity.

### Rotation Calculation

Player input, applied to the matching player entity through the `PlayerLink`, directly influences the pitch and yaw angles. The input's $y$ value adjusts the pitch, representing the up-down rotation of the camera. The $x$ value modifies the yaw, representing the left-right rotation.

Pitch is clamped between predefined maximum and minimum values (80 degrees in this case) to prevent the camera from looking directly up or down. This clamping ensures that the player's view remains within reasonable bounds, avoiding disorienting or unnatural perspectives. Yaw, representing the horizontal rotation of the camera (and thus the player's view), is calculated using modulo arithmetic to ensure that it wraps around. This means that if the yaw value exceeds 360 degrees (or $2\pi$ radians), it's effectively reset to a value within the 0 to 360 degree range. This wrapping behavior allows for continuous, seamless horizontal rotation without the yaw angle growing indefinitely.

These pitch and yaw values are then combined to form a quaternion, which encapsulates the player, and consequently the camera's final rotation, shown in Figure 4.2.

### Movement Calculation

The system first extracts the forward and right vectors from the player's current rotation quaternion. These vectors define the directions in which the player can move based on their orientation, shown in Figure 4.3.

The keyboard input's $x$ and $y$ values determine the intended movement along these forward and right directions. The movement vector is then normalized to ensure consistent speed regardless of the combination of input directions, shown in Figure 4.4.

```
var forwardDirection = filter.Transform->Rotation *
↪  FPVector3.Forward;
forwardDirection.Y = 0; // ignore vertical movement
forwardDirection = forwardDirection.Normalized;

var rightDirection = filter.Transform->Rotation * FPVector3.Right;
rightDirection.Y = 0;
rightDirection = rightDirection.Normalized;
```

**Figure 4.3:** Extracting Movement Vectors

```
var movementDirection = FPVector3.Zero;
// Forward and Backwards movement
if (inputVector.Y > 0) movementDirection += forwardDirection;
if (inputVector.Y < 0) movementDirection -= forwardDirection;
// Left and Right movement
if (inputVector.X > 0) movementDirection += rightDirection;
if (inputVector.X < 0) movementDirection -= rightDirection;
movementDirection = movementDirection.Normalized;
```

**Figure 4.4:** Calculating Movement Direction

This normalized movement vector is passed to the character controller, which applies physics-based movement, incorporating factors like collisions and gravity to move the player and camera in the 3D environment.

### State synchronization

To synchronize our game states through input, we use Quantum's networking solutions. It is easy to use, but a downfall is that we must rely on Quantum's server implementation. However, Quantum does offer a high degree of customizability for the lock-step configuration. The most relevant configuration parameters for the deterministic simulation are shown in the Table 4.1.

## 4.4 Summary

The resulting game is a small voxel-based game, where players can walk around and place/destroy blocks. The most relevant technical part of our implementation is that

| Parameter | Description |
|---|---|
| AggressiveSendMode | If the server should skip buffering and perform aggressive input sends (suitable for games with $\leq 4$ players). |
| InputDelayMax | Maximum input offset a player can have (frames). |
| InputDelayMin | Minimum input offset a player can have (frames). |
| InputDelayPingStart | Ping value (ms) at which Quantum starts applying input offset. |
| InputFixedSize | If input data has fixed length, enabling this saves bandwidth. |

**Table 4.1:** Relevant Quantum configuration for deterministic simulation.

the game is based on a lock-step deterministic simulation. All simulation is done on the client-side, and all clients must synchronize their inputs with each other to maintain a consistent game state.

"QuantumCraft" serves as a working prototype for exploring lock-step simulation in virtual environments, providing valuable insights and practical solutions for deterministic state synchronization, terrain generation, and player control. The experience gained from this implementation helps address research question R2, demonstrating both the feasibility and challenges of integrating lock-step mechanisms in MVEs.

# 5

# Real-world experiments and evaluation

In this chapter, we conduct real-world experiments on the prototype implementation, QuantumCraft, explained in Chapter 4. Our primary goal is to assess the impact of lock-step synchronization on the performance of MVEs, aiming to answer our research question R3:

> *" How does lock-step simulation impact the scalability and performance MVEs? "*

We design a series of real-world experiments, collecting relevant performance metrics, with a focus on network-related metrics such as bandwidth usage and network latency. By analyzing these results, we improve our understanding of the strengths and weaknesses of lock-step synchronization in the context of MVEs.

## 5.1 Main Findings

Here we present our main findings from running the experiments on the DAS-6:

- **Bandwidth Stability Regardless of Player Actions**: We observe that the bandwidth increases quadratically, and reaches 2 Mbps with 64 concurrent players, with near-identical results regardless of player actions.
- **Latency and Tick Duration:** The game reaches a 20 tick rate threshold at an added latency of 150ms.

| **Property** | **Value** |
|---|---|
| CPU | 24-core 2.8 GHz |
| Memory | 128 GB |
| Central Storage | 256 TB |
| Node Storage | 1.8TB NVMe |
| Storage Interconnect | 100G Eth |
| GPUs | 28x 4000, 3x A6000, A100 |

**Table 5.1:** DAS Node Machine Specifications

| MF | Section | Tick Rate | Player Count | Key Metrics |
|---|---|---|---|---|
| MF1 | 5.3 | 20, 60 | 4, 8, 12, 16, 20, 24, 32, 64 | NetworkIn/Out |
| MF2 | 5.4 | 20 | 64 | Latency, Tick Duration [ms] |
| MF3 | 5.5 | 20 | 64 | Latency, Tick Duration [ms] |

**Table 5.2:** Summary of experiments and relevant metrics

- **Impact of Input Offset:** Implementing an input offset of 5 turns can lead to 4.2 times decrease in simulation time with significant added latency, improving responsiveness of the game in bad networking conditions.

## 5.2 Experimental Setup

To evaluate our system, we run all experiments on DAS-6 (14), on the VU cluster. A benefit of DAS is that the hardware is comparable to real-world cloud-based setups, leading to more reliable results, and allows us to simulate a large number of clients to replicate a real-world game scenario. Overall, the DAS is a good testbed to run our distributed experiments. The specifications of a DAS node on the VU cluster are shown in Table 5.1.

The Photon Quantum SDK limits us to using only 64 players per room, and 100 players in total with the free tier, although real-world online games typically support more than 64 concurrent players in a single world.

To run our experiments on the DAS, we run $n$ game instances for 5 minutes. This provides enough time for all players to connect, the system to stabilize, and obtaining accurate measurements.

Networking data and update time data has been taken from `Quantum Stats`, and the networking data has been verified by using netio counters from the system in `psutil` (15).

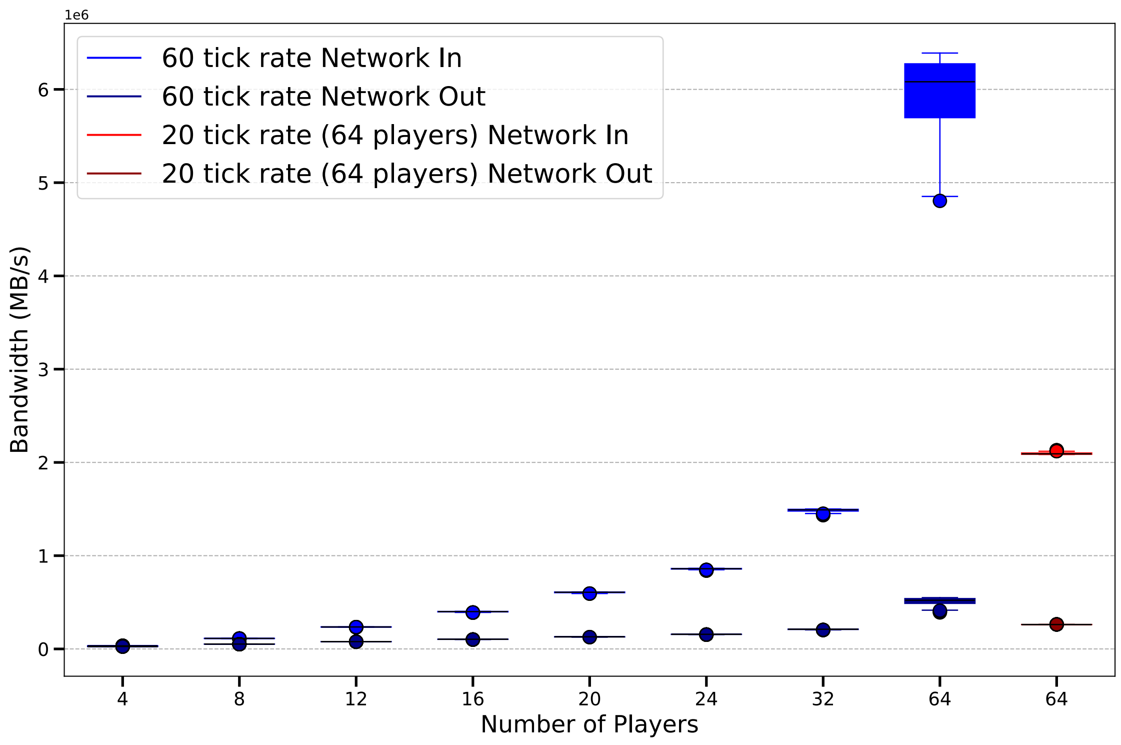

**Figure 5.1:** Boxplot of bandwidth as a function of player count. The NetworkIn and NetworkOut refer to the bytes going in and out measured on the client-side.

## 5.3 Bandwidth Results

In this section we present the results of the bandwidth experiments. We observe a quadratic increase of bandwidth usage in Network In, reaching a maximum of 6 Mbps when running 64 players at a tick rate of 60. Our result is depicted in Figure 5.1.

The figure shows the bandwidth usage of the game for an increasing number of players. The horizontal axis shows the number of players connected concurrently. The vertical axis shows the bandwidth usage, and the boxplots in the figure show how much network bandwidth the server uses for 20 and 60 ticks per second in blue and red respectively. The boxplots show the variability of the bandwidth usage collected for the duration of the experiment. Our results show that bandwidth reaches 6 Mbps for 64 players at a 60 tick rate, and that it grows quadratically with the number of players. Based on this result, we find that lock-step simulation does not bring any benefit directly to scalability in terms of player count. This is to be expected, and can be explained by the fact that each player still requires the input of all other players in the virtual world.

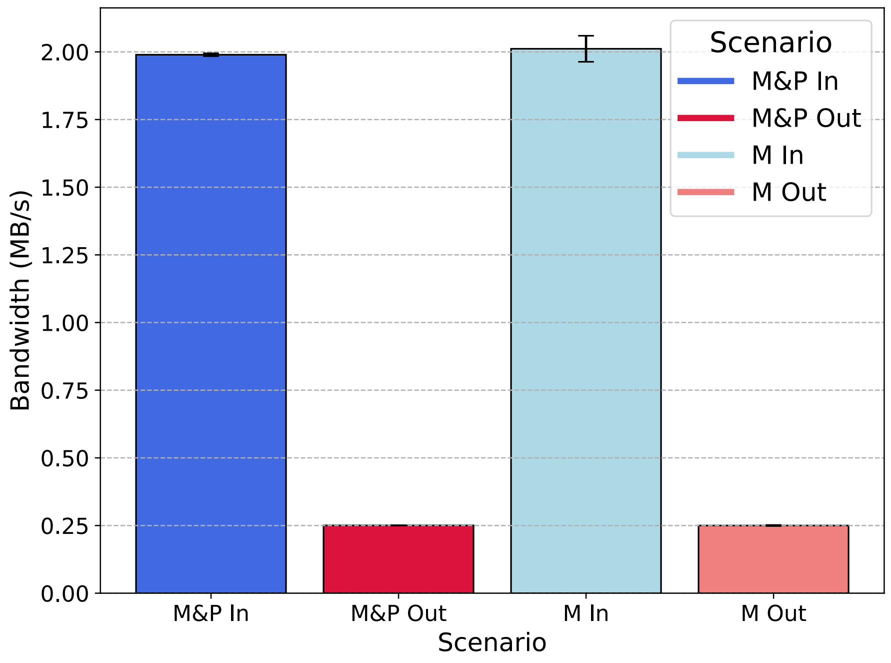

**Figure 5.2:** Barplot with error bar of bandwidth in relation to the different configuration for the player actions.

A keen observation is that bandwidth usage displays variability at a tick rate of 60 with 64 players. This is not to be expected, but can be attributed to the fact that our client implementation can't keep up sending 60 updates per second with 64 players in the game, resulting in decreased bandwidth usage at times.

When compared to results from (16), our system requires a comparable amount of bandwidth for 64 players as the reference system does for 200 players. However, the reference system uses optimal conditions, with minimum render distance and simple player interactions, where the author indicates that in a different setup there may be a significant difference in supported players.

Lastly, we observe that player actions do not have a noticeable influence on the bandwidth usage, as depicted on Figure 5.2. The figure shows the bandwidth usage of the game for different player actions. On the horizontal axis, the action is depicted, where `M&P` denotes Moving and Placing Blocks, and `M` denotes Moving. A further difference between these is that in `M&P` players are giving 10 inputs/second, contrary to 5 inputs/second in `M`. The vertical axis shows the bandwidth usage in Mbps. The error bars indicate the variability in the bandwidth usage for the duration of the experiment.

Our results show no identifiable difference in terms of bandwidth usage when the players

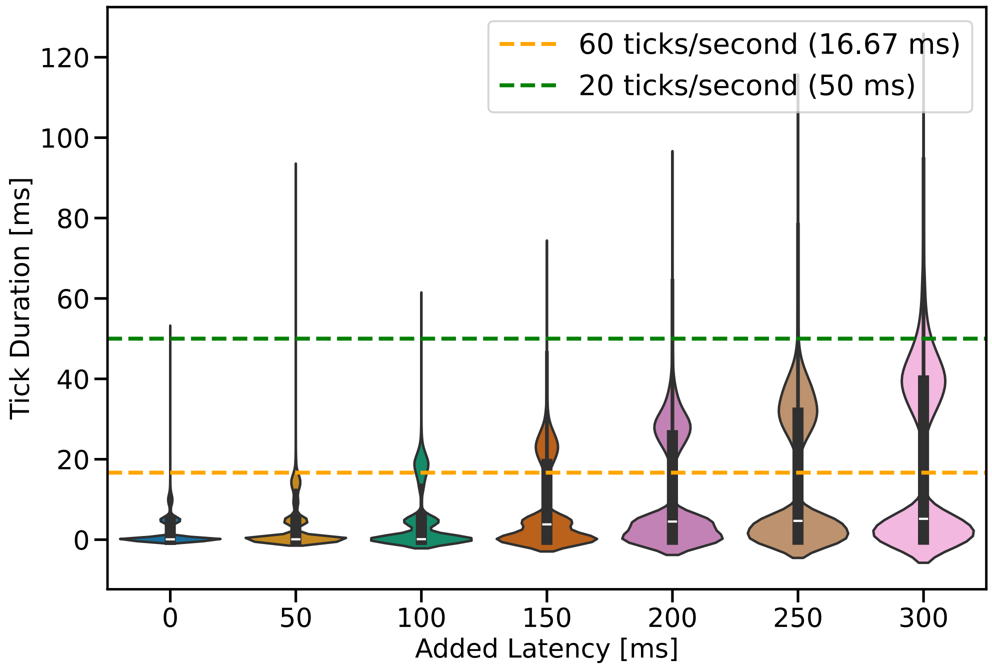

**Figure 5.3:** Violin plot of update times in relation to added latency.

are performing more complex movements and changing the virtual environment. Based on this result, we find that lock-step simulation efficiently uses network bandwidth and scales well in terms of player interactions. This is to be expected, because input is being transmitted at a consistent tick rate, with a fixed size, no matter the player action or world interaction.

Based on the evidence outlined above, we find that our approach supports up to 64 players at a tick rate of 20 regardless of the player actions, assuming a maximum bandwidth of 2 Mbps.

## 5.4 Latency Results

In this section we present the results of the latency tests. We observe an increase of Tick Duration in relation to the added latency, reaching the 20 tick rate threshold at 200ms of added latency. Our result is depicted in Figure 5.3 and Figure 5.4. The figure shows the tick duration collected from the clients during the experiment, for an increasing added

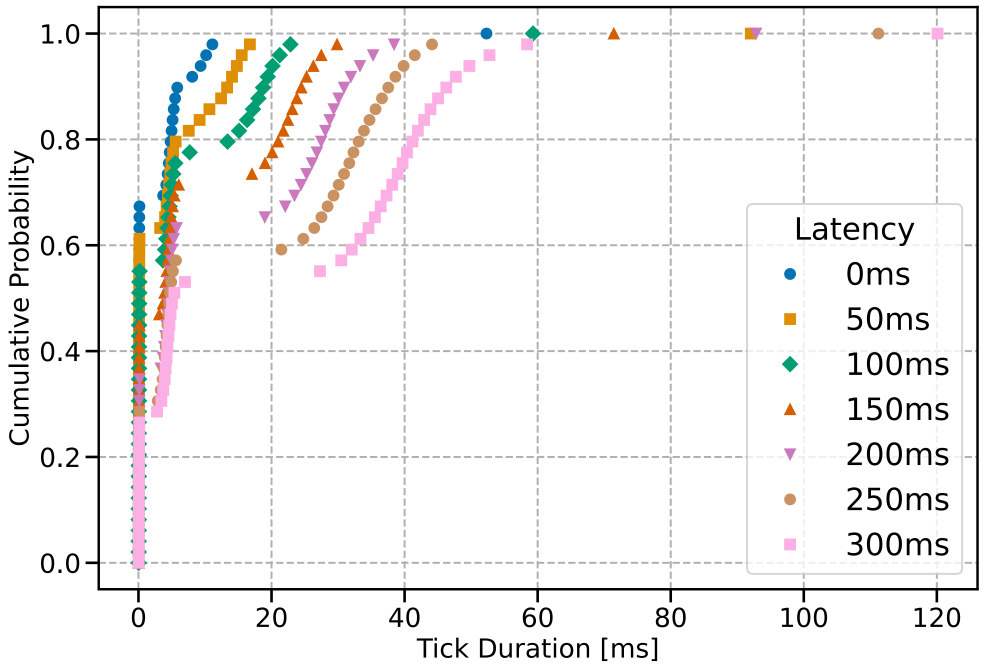

**Figure 5.4:** Empirical Cumulative Distribution Function (ECDF) plot illustrating the relationship between latency and update times in a lock-step synchronized system. The data is the same as in 5.3, with a sample of 60 values per latency configuration.

latency. The horizontal axis shows the added latency. The vertical axis shows the tick durations, and the violin plots show the density of the respective tick duration value. Our results shows that tick duration increases noticeably with latency, reaching a 20 tick rate threshold at 200ms of added latency, and reaching a 60 tick rate threshold at 100 ms.

This result is expected, because the clients need to wait for inputs from all other clients to arrive before simulating a turn. Based on this result, we find that lock-step simulation is sensitive to high latencies, and may result in an unplayable game if significant enough.

Additionally, we observe a gap in the frequency of tick durations between 20 ms and 18 ms. This is unexpected, but can be explained by the fact that the implementation replaces input with empty input if it is not received within a specified timeframe. Because we use a rollback window of 60 turns, any input that falls outside of this range is replaced with empty input, resulting in some low tick durations.

Based on the evidence above, we find that our approach can support a 20 tick rate threshold up to 200ms of added latency for 64 players.

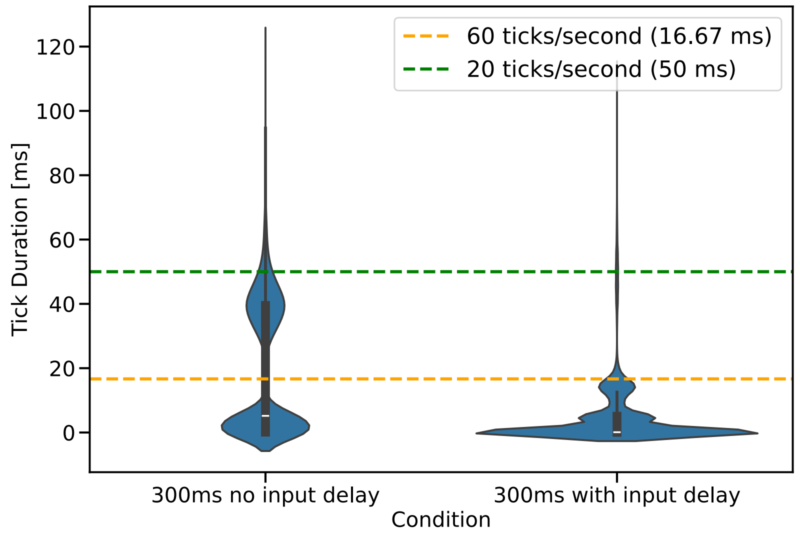

**Figure 5.5:** Violin plot of the update time in relation to using input offset of 5 turns as opposed to no input offset. On the left are depicted the update times without input offset, on the right are the update times with input offset.

**Table 5.3:** Update Time Statistics with and without Input Offset

| | Without Offset | With Offset | Ratio (Without/With) |
|---|---|---|---|
| Median Update Time (ms) | 5.18 | 0.09 | 57.56 |
| Mean Update Time (ms) | 20.16 | 4.72 | 4.27 |
| P-value from Mann-Whitney U test: 0.0000 (Statistically significant) | | | |

## 5.5 Input Delay Results

In this section, we discuss the impact of input delay on the tick duration. We observe a 4.2 times reduction in tick duration when an input delay of 5 turns is present with significant added network latency. Our result is displayed in Figure 5.5 and Table 5.3.

The figure shows the tick durations measured throughout the experiment, and whether input delay is present or not. The horizontal axis shows whether input delay is present and the added latency. The vertical axis shows the tick durations, and the violin plots show the density of the respective tick durations. Our result shows that most of the tick durations remain under 25ms when using input delay of 5 turns. Furthermore, Table 5.3 highlights this performance enhancement. The median update time with input offset is reduced to 0.09ms compared to 5.18ms without it, resulting in a ratio of 57.56. Similarly, the mean update time with input offset is 4.72ms, which is 4.27 times less than the mean update time of 20.16ms without offset. These statistics underscore the effectiveness of input offset in mitigating the impact of latency on update times, thereby improving the overall performance and responsiveness of the lock-step simulation.

Based on this result, we find that input delay can maintain a 20 tick rate threshold at 300ms added latency with an input delay of 5 turns. This is expected, and can be explained by the fact that adding input delay masks the network latency, effectively adjusting the simulation execution turn to be closer to the latency present in the network. Based on this evidence, we find that our approach can improve the responsiveness of the game even at high network latencies, resulting in 4.2 times reduction in tick duration at 300 ms added network latency with 64 players.

# 6

# Related Work

This section reviews relevant research on the development and analysis of lock-step simulation in MVEs . The reviewed literature spans the areas of synchronization issues in multiplayer games, network programming challenges, and scalable architectures for MVEs. The historical context of network programming in multiplayer games, particularly the lessons learned from the development of Age of Empires, is documented in (3). This work highlights the practical aspects of implementing a networked game on limited bandwidth and processing power, using lock-step simulation to synchronize game state. The authors discuss the design and implementation challenges faced throughout the development of the game, providing valuable insight into the trade-offs, past mistakes and design considerations involved in creating such a system. Furthermore, it serves to show the commercial success of lock-step games in society. The scalability challenges of MVEs and potential solutions are explored in (1). It highlights the limitations of existing scalability mechanisms, such as the Area of Interest (AoI), which works for single-avatar environments but fails in multi-avatar settings. The authors propose a new mechanism, the Area of Simulation (AoS), which combines and extends AoI and EBLS to manage multiple areas of interest more efficiently. This approach allows for dynamic trade-offs between consistency and scalability, enabling a significant increase in the number of avatars and the size of the virtual world. The implementation of this architecture demonstrates improved scalability without overwhelming players' computer resources. Our research serves as a foundation for further research into hybrid lock-step approaches such as Area of Simulation.

# 7

# Conclusion

## 7.1 Answering Research Questions

**RQ1: How to design a scalable and performant lock-step simulation system for MVEs?** We answer this research question in Chapter 3 by making an encompassing list of requirements in Section 3.1 and then providing a high-level design in 3.2. Furthermore, we explore lower level details of design, by detailing an input frame format in 3.3 and a design for the game room in Section 3.4.

**RQ2: How to implement a system that integrates lock-step simulation in MVEs?** We answer this research question in Chapter 4, by building QuantumCraft, which is a prototype implementation incorporating the design of RQ1. Determinism is a large part of lock-step, requiring a deterministic physics engine and Fixed-Point math for deterministic calculations.

**RQ3: How does lock-step simulation impact the scalability and performance of MVEs?** We answer this research question in detail in 5. We found that lock-step simulation allows for good scalability in terms of player interactions, however, it does not scale much better with player count, with a quadratic increase in relation to player count. We observe that network latency has a noticeable impact on the tick duration of lock-step MVEs, reaching the 20 tick rate threshold at 200ms of added latency. We found that input offset can bring a 4.2 times improvement to tick duration in MVEs with lock-step at less than optimal networking conditions.

## 7.2 Limitations and Future Work

In this section, we outline the limitations encountered in our system and potential future work to extend upon the research. Despite the valuable insights gained from our experiments, some limitations impact the generalizability and accuracy of our findings:

### SDK Constraints

One major limitation is the restriction imposed by the Photon Quantum SDK, particularly the free tier's cap of 100 concurrent players and the room size absolute limit of 64 players. These constraints prevent us from testing scalability beyond this player count, which limits our ability to comprehensively understand how lock-step synchronization performs in environments with larger player bases. As a result, our findings might not fully reflect the system's behavior under extreme conditions with hundreds or even thousands of players. Future work would benefit from testing with larger player counts or using different platforms to validate scalability claims.

### Networking Focus and Neglected Metrics

Our evaluation is primarily focused on networking performance, specifically bandwidth and latency. Other critical performance metrics, such as CPU and memory usage, have not been addressed in this study. The computational overhead associated with lock-step synchronization, especially in scenarios with numerous players or complex simulations, could impact overall system performance. Future research should consider a comprehensive analysis of CPU and memory usage to provide a more complete picture of the system's performance and feasibility in resource-constrained environments.

### Real-World Simulation Conditions

Our experiments were conducted in a controlled environment using the DAS-6 supercomputer and may not fully replicate the conditions of real-world gaming scenarios. Variations in network infrastructure, client hardware, and other environmental factors could influence the performance metrics observed. Hence, while our results provide a valuable baseline, they may not perfectly reflect the performance of lock-step synchronization in diverse, real-world settings.

# References

[1] Siqi Shen, Shun-Yun Hu, Alexandru Iosup, and Dick Epema. **Area of Simulation: Mechanism and Architecture for Multi-Avatar Virtual Environments**. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, **12**(1), 2015. iii, 2, 31

[2] **Global Data Traffic by Content Type**. `https://www.statista.com/statistics/1312357/global-data-traffic-by-content-type/`, 2023. Accessed: [Date you accessed the website]. 1

[3] Mark Terrano and Sanjay Madhav. **1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond**. *Game Developer*, August 1997. 1, 2, 5, 31

[4] J. van der Sar, J. Donkervliet, and A. Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 243–253, 2019. 1

[5] **Minecraft Realms for Java**. `http://bit.ly/MinecraftService`, November 2020. Online; accessed 18. Aug. 2024. 1

[6] Computer Systems Research Group. **Computer Systems Research: The Science of Practical Computation**, 2024. Accessed: 2024-07-29. 2

[7] Alexandru Iosup, Dick Epema, Guillaume Pierre, Etienne Riviere, Alexandre Dumas, Andrey Guchin, Martijn Kuiper, Channabasavaiah H. Madhyastha, Ana-Maria Oprescu, Cleidson de Souza, Guillaume Thomas, Spyros Voulgaris, Timothy Wood, and Alexandru Uta. **The At-Large Vision: A Research Agenda for Software Systems at the Edge**. *arXiv preprint arXiv:1902.05416*, 2019. 2

[8] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent vanBeek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *ICDCS 2019*, pages 1765–1776, 2019. 3

[9] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. **A Design Science Research Methodology for Information Systems Research**. *Journal of Management Information Systems*, **24**(3):45–77, 2008. 3

[10] Gernot Heiser. **Systems Benchmarking Crimes**. `http://www.cse.unsw.edu.au/~Gernot/benchmarking-crimes.html`, 2019. Accessed 2020. 3

[11] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons Inc., NewYork, USA, 1991. 3

[12] John Ousterhout. **Always measure one level deeper**. *Commun. ACM*, **61**(7):74–83, 2018. 3

[13] **Photon Quantum**, 2024. Accessed: 2024-08-18. 17

[14] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. **A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term**. *IEEE Computer*, **49**(5):54–63, 2016. A draft version of this paper is available at http://www.cs.vu.nl/das5/das2016.pdf. 24

[15] Giampaolo Rodolà. **psutil: Cross-platform library for process and system monitoring**, 2024. Version 5.9.0. 24

[16] Raluca Diaconu, Joaqu'in Keller, and Mathieu Valero. **Manycraft: Scaling Minecraft to Millions**. In *2013 IEEE 19th International Conference on Parallel and Distributed Systems*, pages 52–59. IEEE, 2013. 26

# Appendix A

# Reproducibility

## A.1 Artifact check-list (meta-information)

- **Program: Unity**
- **Compilation: C#, Mono**
- **Run-time environment: Linux**
- **Hardware: DAS-6**
- **Execution: Headless instances, with no graphics**
- **Metrics: Network Latency, Bandwidth Usage, Tick Duration**
- **Output:**
- **Experiments:**
- **How much disk space required (approximately)?: 4 GB**
- **How much time is needed to prepare workflow (approximately)?: Around 15 minutes to set up 16 tabs, connect to DAS and prepare environment, 5 minutes to make new configuration + compilation time**
- **How much time is needed to complete experiments (approximately)?: 5 minutes per run, in total about 5 hours.**
- **Publicly available?: Yes**

## A.2 Description

### How to access

Available at `https://github.com/atlarge-research/QuantumCraft.git`

### Hardware dependencies

For experimental setup, a powerful computer, capable of running clients on separate nodes. For running clean build locally, need a graphics card to render.

### Software dependencies

To run clean build: None To edit the project: Unity at least Build 2022.3.22f1

## A.3 Installation

Install Unity at least Build 2022.3.22f1. Clone the repository, open the quantum_unity directory.

## A.4 Evaluation and expected results

To reproduce results, run the testing script, with 4 clients per node, at an interval of 2 seconds. Change the output path of the loggers in the Unity project to correspond to the desired directory. Expected to get "log_*" files with debug information and "stats_log_*" files with client statistics on the game simulation.

## A.5 Experiment customization

To customize the experiment, change the simulated network conditions by changing `UIMain.Client.LoadBala` and don't forget to enable it by setting `UIMain.Client.LoadBalancingPeer.IsSimulationEnabled` to true. To customize the player count, change the max players field in the GameSetup.qtn in quantum_code. Input delay and other relevant deterministic settings can be found in the DeterministicConfig asset of Quantum.

## A.6 Notes

Make sure to build the quantum_code visual studio solution first. Generate Asset DB resources for Quantum in Unity.

## A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html