Vrije Universiteit Amsterdam

Master Thesis

VoxelBench: A Performance Benchmark for Distributed MVEs

Author: Alfred Daimari (2803310)

1st supervisor:	Daniele Bonetta
daily supervisor:	Jesse Donkervliet
2nd reader:	Francesc Verdugo

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

October 28, 2025

Abstract

Modifiable Virtual Environments (MVEs) such as Minecraft have become a central pillar of modern digital culture, creating a community of hundreds of millions of players and billions in revenue. However, despite their massive popularity, MVEs face inherent scalability limitations and can support only around 200 concurrent players within a single contiguous world. To overcome this barrier, recent efforts from both industry and academia have explored distributed or "sharded" architectures, exemplified by projects such as MultiPaper and Mammoth. Yet, the absence of standardized benchmarking tools makes it difficult to rigorously evaluate and compare these systems or to understand the tradeoffs between different sharding approaches.

Existing benchmarking frameworks for MVEs are insufficient, as they primarily target single-server architectures and lack the capability to measure performance in distributed settings. In this work, we introduce VoxelBench, a benchmark designed to evaluate the scalability and performance of distributed MVEs. VoxelBench captures system-, application-, and thread-level metrics, and introduces derived metrics such as chunk sharing ratio and average synchronization time to characterize the behavior of sharded MVEs.

We implement VoxelBench and use it to conduct experiments on Google Cloud Platform (GCP), evaluating the performance of a popular distributed MVE, MultiPaper. Our findings reveal that the choice of load balancing strategy plays a pivotal role in simulation performance. In particular, a proximity-based load balancer demonstrates superior performance compared to MultiPaper's default least-connected load balancer, underscoring the importance of spatial locality in distributed virtual environments.

Contents

1 Introduction **1**

1.1 Research Context 1

1.2 Problem Statement 2

1.3 Research Questions 2

1.4 Research Contributions 4

1.5 Plagiarism Declaration 5

1.6 Thesis Structure 5

2 Background **7**

2.1 MVE 7

2.2 MVE Operational Model 7

2.3 MVE Workloads 9

2.4 MVE Game Loop 10

2.5 Simulated Constructs 11

2.6 Distributed MVEs 13

2.6.1 Statically Sharded MVEs 13

2.6.2 Dynamically Sharded MVEs 14

2.6.2.1 Multipaper 15

2.6.2.2 Mammoth Minecraft 16

3 Design of VoxelBench **17**

3.1 Overview 17

3.2 Design Requirements 17

3.3 Design of Player Workloads 18

3.3.1 Exploration & Traversal (Walk) 20

3.3.2 Player-versus-Player (PvP) 21

3.3.3 Player-versus-Environment (PvE) 21

3.3.4 Building & Editing (Build) 21
3.3.5 Joint Workload 21
3.4 Design of Metrics 22
3.5 Design of Player Clustering 24
3.6 Design of Map Workloads 25
3.7 Unified Benchmark Model 25

4 Implementation of VoxelBench 29
4.1 Overview 29
4.2 Requirements 29
4.3 Player Emulation Models 30
4.3.1 Player Emulation Rationale 30
4.3.2 Walking and Exploration 31
4.3.3 Player versus Player 32
4.3.4 Player versus Environment 32
4.3.5 Building 33
4.3.6 Simulated Constructs 33
4.3.7 Player Spawn 33
4.4 Extensible Benchmark 34
4.4.1 Node Provisioning 35
4.4.2 Coordination Node Setup 36
4.4.3 Game Server Node Setup 36
4.4.4 Bot Node Setup 37
4.4.5 Exposing Metrics 37
4.5 Reproducibility 38
4.5.1 Benchmark Configuration 38
4.5.2 Seed Implementation 38

5 Performance Evaluation of Multipaper 39
5.1 Experimental Setup 39
5.1.1 AtLarge Cluster Host 39
5.1.2 Google Cloud Compute Host 40
5.1.3 Virtual Machines 40
5.2 Experimental Design, Motivation & Main Findings 41
5.3 Findings 42
5.3.1 2 CPU config works best for 100 players. 43

5.3.2 Multipaper achieves similar performance to paperMC. 45
5.3.3 Larger worlds perform better under exploration workloads. 45
5.3.4 Walk, PvP perform better under more nodes. PvE, Build perform better under more cores. 47
5.3.5 Multipaper cannot efficiently scale beyond 200 players. 49
5.3.6 Proximity load balancing of players improves performance. 53
5.3.7 Chunk sharing reaches 80% under the least connected load balancer. 53
5.3.8 Multipaper has negligible data syncing overhead between nodes. . . . 55

6 Related Work 57
6.1 Yardstick 57
6.2 Servo 57
6.3 Meterstick 58

7 Conclusion 59
7.1 Answering Research Questions 59
7.2 Limitations and Future Work 61

References 63

A Reproducibility 67
A.1 Abstract 67
A.2 Artifact check-list (meta-information) 67
A.3 Description 67
A.3.1 How to access 67
A.4 Experiment workflow 68
A.5 Experiment customization 68
A.6 Methodology 68

1 Introduction

1.1 Research Context

Modifiable Virtual Environments (MVEs) are hugely popular due to the fact that they allow players to modify the virtual environment itself—such as terrain, structures, and game logic unlike other games which are mostly limited to player aesthetics. One of the most successful MVEs is Minecraft, which is also the best-selling video game of all time, and had sold over 350 million copies by 2024 (1). It maintains a vast player base of approximately 200 million monthly active users (MAUs) and generates at least $400 million in annual revenue (1).

MVE's influence extends well beyond entertainment: in 2020, Mojang—the creators of Minecraft—launched Minecraft Education Edition, a gamified learning platform now used in more than 40,000 schools across at least 140 countries, where it helps teach subjects such as mathematics, history, and computer science (2).

The enormous popularity of MVEs has attracted growing interest from both academic (3, 4, 5) and industry circles (6, 7), with a shared goal of massivizing MVEs—scaling them to support large-scale, concurrent multiplayer experiences within a single contiguous world. Currently, however, monolithic architectures limit even the most optimized MVE servers to around 200 concurrent players (8).

The motivation to scale MVEs arises from both economic and creative incentives. From an operational perspective, horizontally scaling MVEs could reduce the costs associated with vertically scaled architectures used in most large-scale online environments. From a creative standpoint, enabling thousands of players to coexist within the same persistent world could unlock new forms of gameplay, social interaction, and educational use cases—such as teaching economic principles through real-time, interactive simulations (9).

Recent years have seen a surge of efforts to overcome these scalability barriers. Industry projects like Folia (PaperMC) (7), Multipaper (6), and Mammoth (10), as well as academic systems such as Dyconits (5), Kiwano (3), and Servo (11), have begun to decompose traditional monolithic MVEs into distributed, multi-server systems. These systems use different strategies to distribute computation and state: Folia introduces a multi-threaded model that performs simulations in parallel; Mammoth Minecraft (10) connects servers through WorldQL (12), a spatial database managing player state and world synchronization; and Multipaper (6) employs a peer-to-peer architecture, assigning exclusive ownership of small simulation regions (chunks) to individual nodes that coordinate when necessary.

1.2 Problem Statement

Scalability solutions for networked virtual environments have existed since the early 2000s(13), emerging alongside large-scale online games such as Second Life and World of Warcraft. Colyseus(14) introduced a distributed architecture for video games by distributing game state and computation across multiple nodes through a single-copy consistency model. Steed et al.(15) demonstrated that effective partitioning of a virtual environment's spatial domain can reduce inter-server synchronization costs and improve overall system scalability.

Building on these foundational ideas, more recent distributed MVE solutions such as Multipaper (6) and Mammoth (10) have begun to adapt these techniques to the unique challenges of modifiable virtual environments. However, unlike traditional MMOs where distributed solutions were developed, MVEs involve substantial user-driven modifications to terrain and game logic, resulting in workloads that differ from previous online virtual environments.

Also, existing MVE benchmarks, such as Yardstick(8) and Meterstick(16), were designed for single-server MVEs and cannot effectively stress test distributed or sharded MVEs. Moreover, they lack the necessary metrics to characterize key aspects of sharded MVEs—such as synchronization latency and inter-node data sharing. As a result, researchers and developers lack a systematic means to analyze performance trade-offs or validate the scalability claims of new sharded MVE architectures.

1.3 Research Questions

To address the challenges outlined above, we formulate the following research questions:

RQ1. How can we design a benchmark to effectively evaluate distributed Modifiable Virtual Environments (MVEs)?

Designing such a benchmark raises several key questions that must be addressed to ensure meaningful benchmarking.

First, we need to design a **unified benchmark model** that can represent different architectural components across different MVEs(See Section 3.7).

Second, we must determine **what workloads should our benchmark include**. We must include only those workloads that stress the distributed components of any MVE (See Section 3.3).

Third, we need to consider **how should players spawn and be distributed** across the world. Player placement directly affects load balancing, inter-server communication, and synchronization overhead. Understanding spatial distribution strategies and how players move and interact is essential for evaluating performance under realistic conditions (15) (See Section 3.5).

Fourth, it is important to ask **which worlds or map types should we include** since players behave differently depending on world topology, resource density, and terrain complexity (14)(See Section 3.6).

Finally, we must identify the **metrics required to characterize performance**. Traditional single-server benchmarks do not capture key aspects of sharded MVEs, such as game-state replication, synchronization latency, and inter-node communication costs (See Section 3.4).

Together, these questions guide the design of a benchmark that can systematically evaluate distributed MVEs which we answer in Chapter 3.

RQ2. How can we implement a scalable and extensible benchmark framework for distributed MVEs?

Designing a benchmark is only the first step; implementing it requires translating the design questions into a working system that can measure real-world performance.

First, we must determine **how do we implement an extensible framework that can support multiple MVE platforms.**

Second, we need to ask **how do we simulate realistic player workloads?** Our workloads should be able to perform a combination of actions such as walk, build, mine, etc based on real world distribution.

Finally, we need to ensure **reproducibility and scalability** of our benchmark. Our benchmark should be able to support the use of multiple nodes and support automated deployment and data collection for repeatable experiments.

These questions guide our implementation of our VoxelBench benchmark and we answer these questions in Chapter 4.

RQ3. What performance trade-offs and coordination challenges arise in dynamic, sharded MVEs, and how do they impact scalability?

Using VoxelBench, our objective is to analyze the scalability of the most popular distributed MVE implementation which is called Multipaper. Our evaluation is provided in Chapter 5.

1.4 Research Contributions

The thesis has the following contributions:

RC1. We design a novel benchmark tailored to evaluate the performance and scalability of distributed MVE servers. The framework introduces new workload categories, player distribution strategies, and performance metrics suited to sharded architectures.

RC2. We implement an extensible and scalable benchmarking system based on our design that supports multiple MVE architectures, providing automated provisioning, workload generation, and data collection.

RC3. We analyze the performance of Multipaper, the most popular distributed sharded MVE implementation and gather usable insights. We uncover tradeoffs and coordination challenges across different workloads and overall its scalability.

RC4. We contribute to the research community by openly releasing both our dataset and the benchmarking framework as open-source resources. This ensures that our results are transparent and reproducible, while also enabling future researchers and practitioners to build upon our work, compare alternative approaches, and extend the benchmark to new use cases.

1.5 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.

1.6 Thesis Structure

The remainder of this study is structured as follows. Chapter 2 introduces the background on MVEs. Chapter 3 presents the design and the rationale behind it. Chapter 4 discusses how we implemented our benchmark. Chapter 5 evaluates the performance of the most popular sharded MVE, Multipaper. Chapter 6 reviews related work. Chapter 7 discusses threats to the validity of the study and concludes the thesis.

2

Background

In this section, we discuss the necessary background on MVEs and distributed Minecraft. First, we discuss what an MVE is and give examples of workloads they run, the inner game loop and simulate constructs (SCs). We then end with a few distributed solutions that are currently available for the popular MVE Minecraft.

2.1 MVE

A Modifiable Virtual Environment allows players to actively shape the virtual world in real time. Users can modify individual objects (such as player apparel) and the virtual world's parts (like terrain), create new content by connecting components (Simulated constructs), and also interact with the virtual environment through programs. The latter three capabilities—component-based creation, programmatic interaction, persistent modifications to virtual environment—are unique to MVEs, making them highly dynamic and user-driven virtual worlds.

Using Minecraft as an example, we have further explained the operational model, workload, inner game loop and simulated constructs of MVEs below.

2.2 MVE Operational Model

An operational model of the MVE Minecraft was first proposed by Eickhoff et al.(2023) (16), as shown in figure 2.1.

1. Client ❶: The client performs two tasks. It takes user input and translates them into in-game actions. It sends them to the server for validation and then applies them to its local state.

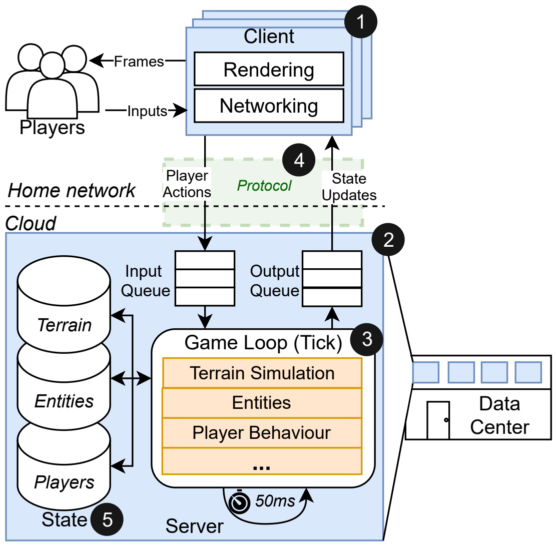

Figure 2.1: Monolithic Minecraft Architechture. *Source:[Eickhoff et al., 2023]*

2. Server ❷: The server is responsible for performing all simulations in the game. Its goal is to receive updates from the client, compute and apply game state changes based on input received, and then return the computed game state updates to each player.

3. Game Loop ❸: The game loop is the core component responsible for the simulations and world updates. It makes changes to the game state in discrete time steps called ticks. Minecraft provides the best QoS when an iteration of the game loop finishes within 50ms or 20Hz, in other words; when the tick is 50ms. When the tick is greater than 50ms, the server is considered overloaded. Minecraft servers that are a fork of Purpur (17) (most online servers) automatically shutdown if the tick goes beyond 2000ms.

4. Protocol ❹: Unlike other on-line multiplayer games that use UDP mainly for client-server communication, Minecraft uses TCP.

5. Persistent State ❺: Minecraft stores terrain, entity, and player information using storage and continuously updates them as updates come in. Minecraft stores terrain information in an NBT file format. The terrain data is stored in small sizes, called chunks. Each chunk stores its information in a single NBT file and consists 16x16x384 blocks. The file is compressed for efficient storage.

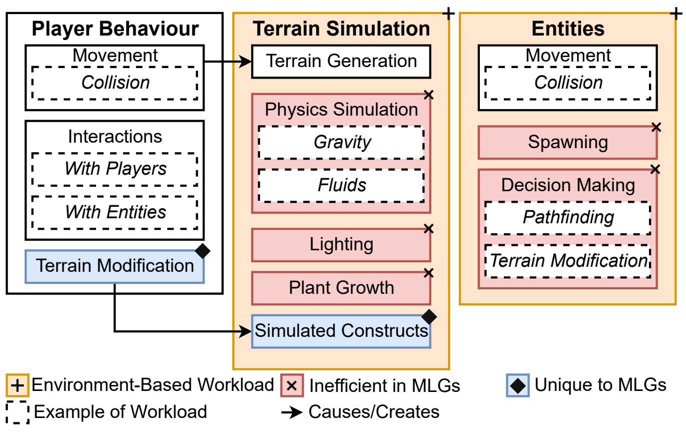

Figure 2.2: Minecraft Workload Components. *Source:[Eickhoff et al., 2023]*

2.3 MVE Workloads

A workload model, as shown in 2.2, was also proposed by Eickhoff et al.(2023) (16) that describes different workloads that are handled by the Minecraft game loop.

1. Player Workloads: Players create workloads for the server through their actions. In Minecraft, players are allowed a wide range of actions that the server needs to compute. Collision detection is one of the workloads that is run most frequently by the game loop. The game needs to prevent players from walking through walls and broadcast player movement information to other players. Players can also interact with other players; one such example is PVP battles, where complexity of interaction can increase based on the weapons and armor that players have on. Players can also interact with in-game entities—entities are in-game non-playable characters (NPCS)—such as villagers and wandering traders to exchange items. One standout feature of Minecraft compared to other online games is the player's freedom to edit the terrain. This action is not only compute intensive, but also data intensive. When players move to new chunks fairly quickly, the server is required to uncompress the NBT chunk files and load them into memory. The changes a player makes to the terrain through creation and destruction also need to be synced and broadcast to other players. Players can also construct programmable elements in the game known as SCs in MVEs (Redstone in Minecraft). These dynamic elements significantly adds to the workload.

2. Terrain Workload: A significant portion of the workload can be generated by terrain simulation. MVEs and Minecraft present the players with massive world. Minecraft

allows a player to traverse 30 million blocks in any direction from the center. The world is split into chunks (16x16x384 blocks) and stored persistently using compressed NBT files. As players move around the world, the required chunk is loaded lazily from the storage. The four main components of terrain simulation are physics, lighting, plant growth, and SCs (16). The reason why terrain workload is so expensive is due to the reason that the game needs to compute simulations on multiple blocks. A simple act of maybe removing two or three blocks could bring an entire structure like a building to the ground and each falling block would have its physics simulated. Dynamic elements such as plant growth also add to the workload, by growing and changing in size reshaping the terrain. Simulating SCs require substantially more resources. Using SCs, players can build a fully functioning CPU, functioning train, etc. Simulating such complex instruments is done step by step and ends up substantially slowing the game tick rate.

3. Entity Workload: An entity is a non playable character. There are two ways that make entity simulation challenging for MVEs—spawning and pathfinding (16). MVEs compute spawn points dynamically. They need to check for cases where terrain might be obstructing their spawn location which further adds to the compute time. Second, entities use path-finding algorithms to move around the world. Since the terrain can be edited by a player at any moment, a path-finding algorithm needs to be used unlike static worlds where a simple overlay can direct entities. Dynamic path-finding further adds to the workload.

2.4 MVE Game Loop

The game loop ❶ as shown in figure 2.3 comprises of 3 main workload classes. The workload computed by each class is defined in section 2.3.

1. Player Handler ❷: The player handler is driven by the actions received from players. The workloads it handles is defined section 2.3. It performs a blocking get, if the received player actions are valid, it makes the changes permanent by writing it to State ❺ using a blocking write.

2. Terrain Simulation ❸: Terrain simulation is mostly independent from Player Handler. It comprises of workloads defined in 2.3. It performs a blocking read from the persistent state, in the event there is a change in terrain, it computes simulation rules for the new received state. Sometimes a simple update could affect multiple other

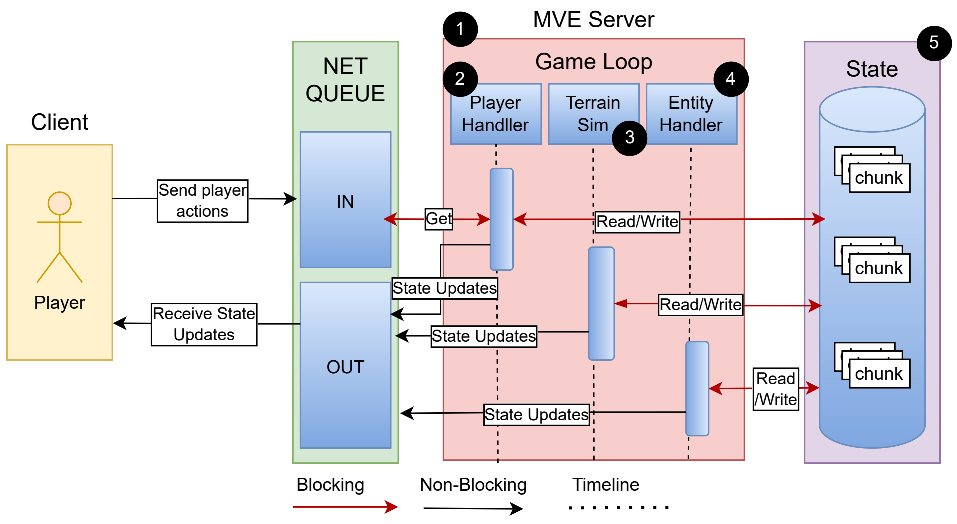

Figure 2.3: Minecraft Game Loop. Write blocking is done at chunk level.

blocks, e.g., removing a block could bring an entire structure down. In such cases, it performs a blocking write to the state for update the new terrain.

3. Entity Handler ❹: Entities are completely driven by the game state ❺. It performs a blocking read; based on its surrounding terrain, player data, it undertakes the next decision. One such example is reading its surrounding terrain to compute the next block using its pathfinding algorithm. Based on its decision, it performs a blocking write and updates the state.

2.5 Simulated Constructs

Minecraft and MVEs can contain stateful blocks/voxels. The authors of Servo (11) refer to a collection of such blocks as SCs or Simulated Constructs. Throughout this article we will be using the same name for such a collection. In Minecraft, you create SCs using special blocks such as redstone dust, redstone lamp, redstone repeater, redstone lever, etc. Each of the redstone block types has their own unique logic as to how they function. Using different kinds of stateful redstone blocks (which are Turing complete), players can create fairly complex SCs such as fully functioning computers as shown in Figure 2.4.

The developers at Mojang have given different redstone blocks certain properties in order to make them less computationally expensive, though it hasn't really been successful as showcased by Servo (11), which shows a huge boost in performance if SCs are computed

Figure 2.4: A fully functioning computer using Minecraft Redstone.

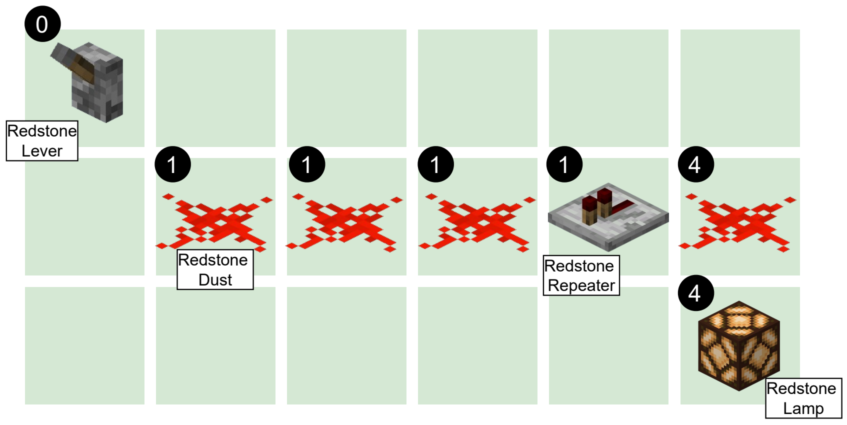

Figure 2.5: Minecraft Redstone signal propagation across ticks.

using serverless principles, but it still helps in spreading SC computation across multiple different game ticks. Figure 2.5 shows how Minecraft distributes the redstone computation in multiple ticks.

Minecraft redstone blocks work like cells in Conway's Game of Life, but unlike Conway's Game of Life, where a cell is influenced by both its live and inactive neighbors, only a live Minecraft redstone block influences its neighbors. Minecraft has special blocks such as redstone lever which a player can interact with in the game to make that block "live", or has "always live" blocks such as a redstone block.

In our Figure 2.5, a player first interacts with the redstone lever at tick ⓿; this interaction gets registered in the next tick. In the next tick ❶, the redstone lever influences the redstone dust next to it, which then influences its own redstone dust neighbour. This

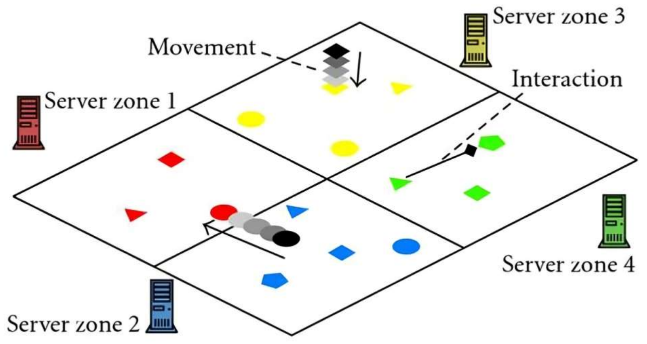

Figure 2.6: MVE Static Sharding Architecture. Source:[Steed et al., 2003]

chain of one block influencing another block is allowed to be chained up to 15 times in Minecraft, when the redstone "signal" becomes weak and subsides if not propagated by a redstone repeater. In our Figure, the signal reaches the redstone repeater in tick ❶, keeping it within Minecraft's chain limits.

Once the signal reaches a redstone repeater, here the signal is paused by Minecraft for one "redstone tick" which is equivalent to two ticks in the game loop. In tick ❹, the redstone repeater influences the redstone dust next to it which then influences the redstone lamp. In order to prevent cascading signals across the Minecraft game world, Minecraft only allows redstone interactions to occur in chunks that are loaded in memory.

2.6 Distributed MVEs

Today, many online MVE communities operate their own servers and seek affordable ways to accommodate more players. They leverage horizontal scaling, partitioning workloads across multiple servers to accommodate hundreds or thousands of simultaneous users. Today, distributed MVEs come in two designs, statically sharded and a more recent development which is dynamically sharded.

2.6.1 Statically Sharded MVEs

To create a workaround for vertical scaling limits, community-driven tools, such as BungeeCord (18), attempt to split a single game world statically between multiple servers. Each isolated server can then hold up to 150-200 players. Trying to scale vertically to accommodate more than 200 players on each server becomes prohibitively expensive. If there is only one player in each of these zones, the entire cluster is required to run and leads to substantial

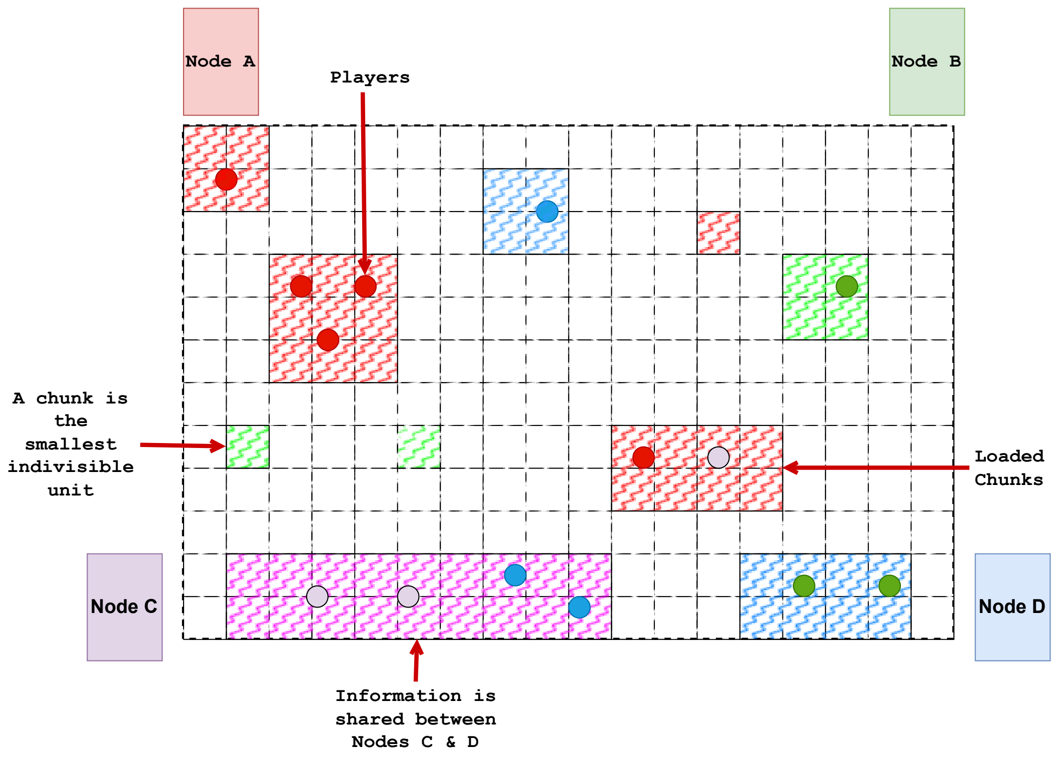
Node A
Players
Node B
A chunk is the smallest indivisible unit
Loaded Chunks
Node C
Node D
Information is shared between Nodes C & D

Figure 2.7: MVE Dynamic Sharding Architecture.

inefficiencies. Figure 2.6 gives an example. Using a static distributed approach, these online communities have been able to accommodate concurrent player counts of 5000 (19). However, this approach divides the world into isolated segments, preventing the experience of a seamless shared game space.

2.6.2 Dynamically Sharded MVEs

In Dynamic sharding, load is distributed across different nodes at the smallest indivisible level, which is called chunks. Chunks are small atomic areas in the world map. Through chunks, a single node can now hold areas from different locations in the map as opposed to a large zonal area. This leads to a more efficient design since if there are only a few players but in different areas of the map, one single node can handle it since it can load chunks from the needed areas. Figure 2.7 showcases dynamic sharding.

In recent years, there have been many developments within the open source Minecraft community, especially in relation to distributed Minecraft. We have two implementations of distributed Minecraft—Multipaper (6) and Mammoth Minecraft (10)—that scale hori-

Aspect	Multipaper	Mammoth
Synchronization	P2P	Publish/Subscribe
Chunk ownership	Exclusive	Shared
Write authority	Server	Hybrid
Scaling approach	Decentralized	Coordinated

Table 2.1: Architectural differences between Multipaper and Mammoth

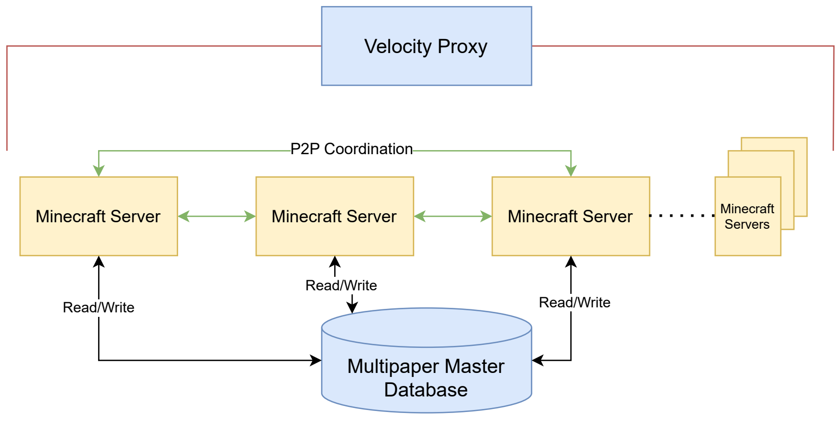

Figure 2.8: Multipaper architecture which uses P2P coordination.

zontally and where players can share one contiguous world. Both architectures differ from each other in 4 different aspects even though they have fairly similar looking components. Their differences are shown in Table 2.6.2.

2.6.2.1 Multipaper

Multi-paper, as shown in Figure 2.8 is a decentralized scaling approach. Multipaper runs a modified version of the open-source Paper (20) Minecraft game server behind a Velocity proxy (21). All clients connect to the Velocity proxy which routes them to one of the modified Paper servers. The server the player is connected to through the proxy is now responsible for the chunks that belong in the players view distance. The server requests for the necessary chunks from the Multipaper Master Database and is given full exclusive authority over it. If a chunk needs to be unloaded, the Minecraft server then writes it back to the Multipaper Master Database. If in case the required chunk has already been loaded by another database, Multipaper tries to migrate the player, but if that is not possible, it communicates the actions of the player to the server that holds the chunk. Servers

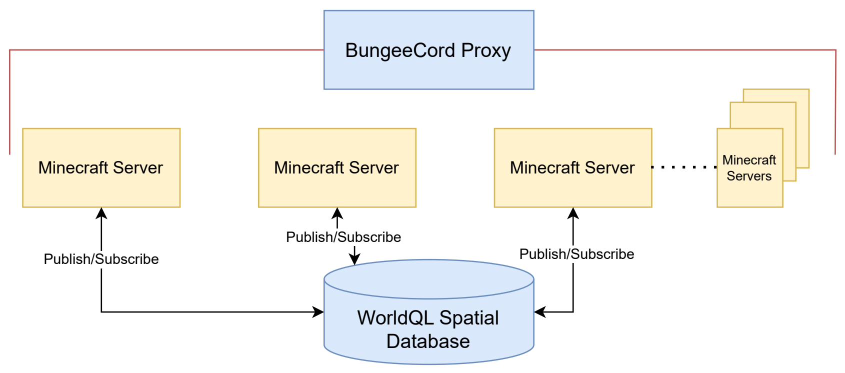

Figure 2.9: Mammoth Minecraft that uses publish/subscribe for coordination.

communicate directly with each other and do not need a messaging server since only one server can hold one chunk at a time.

2.6.2.2 Mammoth Minecraft

Mammoth, as shown in Figure 2.9, is a centrally coordinated scaling approach. Mammoth, like Multipaper also runs a modified version of the open-source Paper (20) Minecraft game server behind a *BungeeCord proxy* (18). All clients connect to the proxy, which routes them to one of the Paper servers. Unlike Multipaper, the server to which the player is connected is not exclusively responsible for all fragments within the view distance of the player. Instead, Mammoth allows multiple servers to access overlapping regions of the world. The chunk data are shared through a centralized spatial database called WorldQL (12). If a server requires a chunk, it loads it, but it also subscribes to changes using WorldQL for updates made by other servers. In the event of player movement or interaction across chunk boundaries, the player remains on the same server if possible, and other servers receive necessary chunk updates through the WorldQL database. Servers do not communicate directly with each other and only use WorldQL to maintain consistency.

3

Design of VoxelBench

In this chapter, **we address RQ1** by providing a design for a distributed MVE benchmark which we call ***VoxelBench***.

3.1 Overview

We design VoxelBench, a distributed benchmark to evaluate performance and design decisions in sharded Modifiable Virtual Environments (MVEs). Our contribution in this chapter is six-fold:

1. We analyze the design requirements for VoxelBench (Section 3.2).
2. We present the design of player workloads that target resource balancing and information management (Section 3.3).
3. We describe the metrics collected and derived to analyse distributed MVE performance (Section 3.4).
4. We discuss player clustering behavior (Section 3.5).
5. We discuss map workloads (Section 3.6).
6. We present a unified benchmark model (Section 3.7).

3.2 Design Requirements

We begin by outlining design requirements for our benchmark. Our requirements are motivated by **RQ1**.

R1. Workloads Targeting Distributed Design Decisions: The benchmark should only include workloads that stress distributed design decisions in MVEs. Each workload should be easily configurable and support realistic player behavior.

R2. Performance Metrics for Distributed MVEs: The benchmark should capture metrics beyond single-server performance. It should be able to analyze distributed overhead, replication costs, synchronization costs, etc.

R3. Representative Player Clustering: Player spawn and movement patterns must be configurable since spatial distribution directly impacts load balancing. The benchmark should support both uniform and clustered spawning.

R4. Multiple World Maps: Different world maps must be supported, as topology affects player behavior. Maps should vary in terrain complexity and size. For performance analysis to be more effective, the benchmark should mainly include popular maps.

R5. Unified Benchmark Model: The benchmark must model architectural components common across distributed MVEs under one unified abstraction. The unified abstraction simplifies metrics collection mapping to corresponding components, and creates a unified infrastructure provisioning system.

3.3 Design of Player Workloads

In this section, we design workloads that stress distributed design decisions in MVEs and satisfy **R1**. We also cover the rationale for including them for each workload.

Extensive research has been conducted on the distributed simulation of virtual environments (3, 14, 15, 22, 23). A taxonomy of these architectural designs is presented in Figure 3.1 which is given by Gonzalez et al.(2023) (13). Although much of this work does not directly address Modifiable Virtual Environments (MVEs), the underlying architectural principles remain highly relevant and are commonly applied in distributed MVE systems today.

For example, under resource balancing, Multipaper uses world partitioning (6) and Mammoth uses distributed hashes (10). Under information management, Multipaper uses a shared distributed world and Mammoth uses a shared centralized world.

Table 3.1: Overview of supported Benchmark Workload. Acronyms: W-Workload, C-Configuration, X-Map Location (x-axis), Z-Map Location (z-axis), α-Walk probability, β-Build probability, γ-PVP probability, δ-PVE probability, PVE-Player vs Environment, PVP-Player vs Player, SC-Simulated Constructs.

ID	Type	Parameter Description	Values
1	W	Input world complexity	High, Medium, Low
2	W	Connected players (count)	$25, 50, 75, \cdots$
3	W	Player emulation model	Walk, Build, PVP, PVE, SC
4	W	Clustering density	$1, 2, 3, \cdots, 25$
5	W	Area of Interest	$-30000000 < X, Z < +30000000$
6	W	Workload distribution	$\alpha + \beta + \gamma + \delta = 1.0$
7	W	Radius	$-30000000 \cdots, 0, \cdots, +30000000$
8	W	PVE mob type	polar_bear, zombie, ender_dragon,$\cdots$
9	W	Walk Destinations	$-30000000 < X, Z < +30000000$
10	C	View distance	10, 14, 18, 22, 26, 30, 32
11	C	Entity spawning	true, false
12	C	Video recording	true, false
13	C	Seed	random integer

In essence, distributed MVEs make key design decisions around two fundamental dimensions — resource balancing and information management. These two paradigms form the basis for the workloads that our benchmark aims to evaluate.

The workloads we have selected are based on three criteria:

1. The observation that players either perform a complex activity in a narrow area, for example, building a house, fighting another player, or move towards an area, for example, exploring the world (8).

2. The workloads target "MVE Workloads" as proposed by Eickhoff et al.(2023)(16) (See Section 2.3).

3. The workloads stress resource balancing and information management design decisions.

Based on our observations, **Exploration, PvP, PvE, Building, and Simulated Constructs (SCs)** workloads meet the above-described criteria. These workloads correspond to player emulation models in Table 3.1. In the subsections below, we describe how our

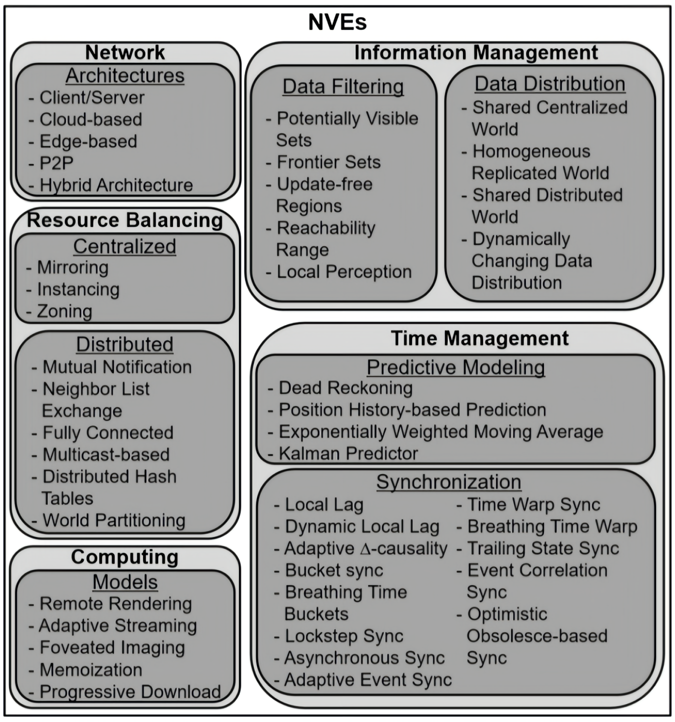

Figure 3.1: NVE Taxonomy. Components of an NVE and techniques included in them. Source:[Gonzalez et al., 2021]

workloads target resource balancing and information management (See Section 3.6 for SCs).

3.3.1 Exploration & Traversal (Walk)

In distributed virtual environment literature, there is substantial work that is concerned with how players are mapped to nodes (15, 24, 25). Exploration-based workloads are important for benchmarking MVE performance, as player movement influences how simulation regions are assigned to servers. Unlike single-server MVEs, a sharded MVEs must continuously determine region ownership between nodes and synchronize state across boundaries.

As players explore, the system must allocate ownership of newly visited regions, stream updates to peers, and also rebalance ownership when players leave or when regions become "hot" due to crowding. This creates a resource balancing challenge, where uneven player distribution can lead to overloaded nodes. Exploration therefore stresses the classic partitioning and mapping problem.

3.3.2 Player-versus-Player (PvP)

In sharded MVEs, combat scenarios highlight critical performance challenges. When players on different shards engage in PvP, slow damage resolution can lead to visible desynchronization and degrade the gameplay experience (13). Prior research on large-scale, low-latency game systems emphasizes the importance of localizing player interest sets and maintaining short dissemination paths for updates (23, 26). A PvP-focused player emulation model would therefore be valuable for benchmarking how a sharded MVE manages interest sets.

3.3.3 Player-versus-Environment (PvE)

Entities (Non-player characters) expose significant performance challenges. In distributed virtual environments, there is substantial literature comprising mainly of partitioning entities across nodes (27, 28, 29, 30). PvE generates entity-simulation workloads and raises cross-boundary consistency issues when entities traverse shard borders. As discussed in Section 2.3, entities introduce their own class of workloads, and high entity-simulation density can substantially degrade performance (16).

3.3.4 Building & Editing (Build)

Building and terrain-based workloads are a key compute-intensive dimension (16). In sharded architectures, we hypothesize that terrain edits could also create substantial update traffic and state propagation to peer servers that are dependent on its state.

Similar to the walk model, the building player emulation model is designed to benchmark how effectively partitioning algorithms distribute highly-edited terrain regions across the cluster while simultaneously minimizing communication overhead.

3.3.5 Joint Workload

The joint workload combines the four core workloads—Walk, Build, PvP, PvE—by distributing player behavior according to configurable probabilities ($\alpha + \beta + \gamma + \delta$). Each bot is assigned a behavior type per tick based on the distribution set, allowing simultaneous execution of exploration, building, combat, and entity interaction.

Table 3.2: Overview of performance metrics collected by VoxelBench. Acronyms: S-System, T-Thread, A-Application, D-Derived, V-VoxelBench, C-Collector.

Name	Type	Source	Description
RAM usage	S	S	RAM Usage
CPU load	S	S	CPU usage
Disk usage	S	S	R/W in bytes
Network usage	S	S	Sent/Received bytes
Lock contention	T	C	Thread lock contention (ms)
Conditional Wait	T	C	Conditional Wait (ms)
Thread Park	T	C	Thread blocked by scheduler (ms)
Thread Sleep	T	C	Thread sleeping (ms)
Player count	A	V	Current connected players per server
Player location	A	V	Player location in the world
Chunks loaded	A	V	Chunks loaded by each server
Disconnects	A	V	Server-wide disconnects, count
Tick length	A	C	Time spent processing each tick
Tick Frequency	D	C	Frequency of game loop updates
Relative Utilization	D	C	See Section 3.4
Average Sync Time	D	C	See Section 3.4
Shared Chunks %	D	V	See Section 3.4

3.4 Design of Metrics

In this section, we give an overview of how we satisfy the requirement **R2** which is metrics required to track performance in distributed MVEs.

For observability, we track a couple of performance metrics—both system and application level—which are shown in Table 3.2.

Collection of System-level Metrics: When the benchmark is running, each Node in our cluster executes an instance of the system monitoring sensor (Telegraf) and automatically collects and stores system-level metrics such as CPU load, RAM usage, disk usage, and networking usage. Each node monitors its own system usage and stores data locally on disk. Once the benchmark is done running, the data is copied from all nodes and stored on disk on the orchestrator node from where the benchmark was initiated.

Collection of Thread-level Metrics: VoxelBench captures several thread-level metrics: lock contention, conditional wait, thread park, thread sleep for the main game loop thread. In sharded MVEs, contention over remote resources may withhold the main thread from performing computation; hence, our benchmark needs to access thread-level information to get better insights. Tracking thread-level information along with the function call stack (Java Flight Recorder for Minecraft Java Edition) allows us to derive metrics related to syncing overheads between two sharded MVE game servers.

Collection of Application-level Metrics: VoxelBench captures several application-level metrics: number of connected players, player location, chunks loaded by an MVE game server, player disconnects, and tick length. The number of connected players and tick length are collected by the VoxelBench collector component that runs as a separate process. The chunks loaded by an MVE game server is tracked by a plugin that hooks into the game loop of the MVE server and logs all the world chunks loaded every tick. The chunk information is then written to disk by the plugin. The player disconnects and player location are tracked on the bot node by integrating the logging functionality into the player emulation model. All information is written to disk.

Derivation of Service-level Metrics: VoxelBench derives several service-level metrics from the system-level, runtime-level, and application-level metrics it collects.

From the game-loop data, VoxelBench derives tick frequency. From the tick frequency, it computes relative utilization. Relative utilization is discussed next as described by the Yardstick Benchmark (8).

Definition 3.1. The relative utilization of a server is defined as:

$$U_r = \frac{t_d - t_s}{t_e - t_s}$$

where U_r is relative utilization, and the variables t_d, t_s and t_e are interpreted as in Figure (create figure).

From the chunk loaded every game loop, we can derive the shared chunks % metric for the entire MVE cluster. The shared chunks metric is very important because it can help determine whether a cluster is sharing extensive world data between different game servers. Sharing huge % of chunk data could lead to sync issues and decrease tick rate. **The shared chunks % metric, which we introduce, is discussed next.**

Definition 3.2. The Shared Chunks % metric of an MVE cluster can be defined as:

$$SCP = 100 \times \frac{S}{L}$$

where SCP is shared chunk %, S is the number of shared chunks in the cluster and L is the total number of loaded chunks in the cluster. The derivation for S, L is given below.

Let $s \in S$ denote the set of servers and C_s denote the set of loaded chunks on a server s.

$$\text{Let } U = \bigcup_{s \in S} c_s \text{ be the set of all chunks loaded on a MVE cluster.}$$

$$\text{The number of chunks loaded is } L = |U|$$

$$\text{Let } n_c = |\{s \in S : c \in C_s\}| \text{ be the number of servers where a chunk } c \text{ is loaded}$$

$$\text{The set of shared chunks is } U_{\text{shared}} = \{c \in U : n_c \geq 2\}.$$

$$\text{The number of shared chunks is } S = |U_{\text{shared}}|$$

Using the collected thread-level metrics, we can now derive average sync time for each MVE game server. It is only calculated for the **Main Game Loop thread**. This metric is important because it helps us gain insights into the amount of time spent syncing and coordinating with other servers in the cluster instead of performing work. **The average sync time metric, which we introduce, is discussed next.**

Definition 3.3. The average sync time of a server is defined as:

$$\overline{S_r} = \overline{T_p} - \overline{(t_e - t_d)}$$

where $\overline{S_r}$ is average sync time, $\overline{T_p}$ is average duration of thread park events per tick and $\overline{t_e - t_d}$ can be interpreted as average tick wait duration as shown in Figure (create figure).

3.5 Design of Player Clustering

In this section, we describe player clustering behavior and satisfy **R3**.

Player clustering is strongly influenced by map design and points of interest (POIs) such as settlements, resources, or landmarks (14). We capture this behavior through two spawn modes in VoxelBench.

The first mode targets scenarios where there is a lack of data, where no positional traces or heatmaps are available. In such cases, bots are spawned at random positions within a user-defined radius R around the origin, with a density parameter D controlling how many bots appear at the same location.

The second mode addresses scenarios where the player clustering data is available. Here, instead of dispersing from the origin, bots are spawned at random radii around user-defined hotspot coordinates that correspond to these POIs, again with density D specifying local clustering.

Together, these two spawn settings allow us to reproduce POI-driven clustering behavior. The spawning and clustering are illustrated in Figure 4.3 and described in Section 4.3.7.

3.6 Design of Map Workloads

Table 3.3: Community maps with size, downloads, and complexity. Acronyms: SC-Simulated Constructs.

Name	Size [MB]	Downloads (in 1,000s)	Complexity (relative)
GreaterKCMetro	78.8	29	High
World of Worlds	81	423	Medium
Castle Lividus of Aeritus	36	330	Low
Redstone Mountain Village	13.5	19	SC

In this section, we explain map workloads and satisfy **R4**.

As described in Section 2.5, Simulated constructs (SCs) are non-player elements in an MVE that generate workloads for the MVE game server. SCs significantly impact the performance of a game server (11). For any distributed MVE game server to be usable, it should be able to handle the workloads generated by SCs. Hence, a benchmark that targets distributed MVEs needs to include SC workloads. To target non-player workloads, our benchmark includes popular MVE maps. For Minecraft, we have included a popular map that has 100s of SC elements called Redstone Mountain Village as shown in Table 3.3.

In our benchmark, we can target non-player workloads for Minecraft by running our Walk workload on the Redstone Mountain Village. With this and the above described player workloads, we satisfy the requirements of **FR.1** which is extensive player models.

3.7 Unified Benchmark Model

In this section, we design a benchmark model and satisfy **R5**.

Our benchmark provisions infrastructure along three node types that are Coordination, Game, and Bot Nodes to enable simplified provisioning and mapping metrics collection to

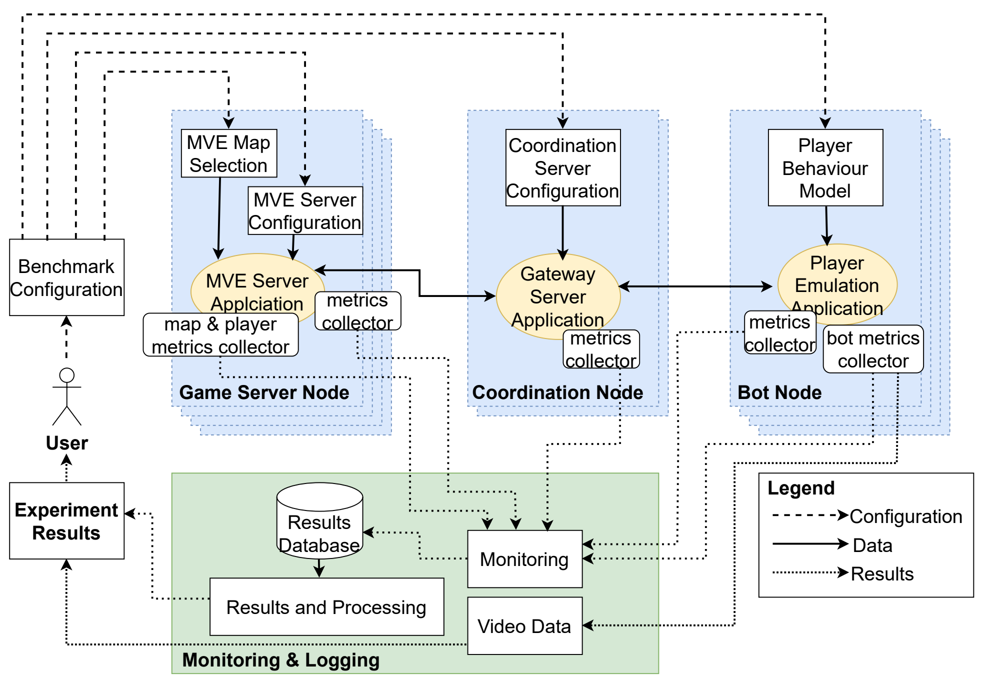

Figure 3.2: VoxelBench Benchmark Design

node types. The main motivation behind this is how community sharded MVE clusters such as Multipaper and Mammoth are architected as shown in Figure 2.8 and Figure 2.9 respectively. Coordination nodes not only comprise the proxy node, but also central co-ordination components like the Spatial Database in Mammoth and the Master Database in Multipaper. This is because for Multipaper, both the proxy component and Master Database run on one server.

As shown in Figure 3.2, the Coordination Node is responsible for enabling communication between Game Server Nodes and the Bot Nodes. The Coordination Node also stores IP information regarding how different Game Server Nodes should reach each other, and important databases that are needed for keeping the game state in sync. The Game Server Node is responsible for running the MVE game application, and the Bot Node emulates players.

Each of these Node types has its own Deploy, Run, Stop, Fetch, and Clean phase in the benchmark. In the Deploy phase, the MVE Server Application, Coordination Server Applications, and Player Emulation Application are installed along with their dependencies in the Game Server, Coordination, and Bot Nodes respectively. We also install a common metrics collector that tracks system information across all nodes.

Next in the Run phase, using the configurations provided by the user, the MVE Server Application, Gateway Server Application, and Player Emulation Application are run. After the benchmark has completed running, the applications are stopped and the data fetched in the Stop and Fetch phases respectively. Then all nodes are cleaned up for the next test in the Clean phase.

4

Implementation of VoxelBench

This chapter discusses the implementation of ***VoxelBench***.

4.1 Overview

We implement VoxelBench, a distributed MVE benchmark designed in Chapter 3, thereby addressing **RQ2**. In this chapter, our contribution is four-fold:

1. We describe the implementation requirements for VoxelBench and outline the motivations behind them (Section 4.2)
2. We present the implementation of player emulation models for generating workloads (Section 4.3).
3. We discuss the extensibility of the benchmark to other MVEs and implementation of scalable workloads and the node provisioning process (Section 4.4).
4. We describe the mechanisms for reproducibility and repeatability across experiments (Section 4.5).

4.2 Requirements

We begin by outlining requirements for our benchmark implementation. Our requirements are motivated by **RQ2**.

R1. Realistic player emulation models: The benchmark implementation must provide a broad set of player models representing common in-game behaviors and non-player models to generate extensive workloads. This includes converting activities such as Exploration, Building, PvE, and PvP into repeatable and configurable player models. The benchmark should also support behaviors such as players congregating around hotspots.

R2. **Extensible benchmark**: VoxelBench must support a wide range of MVE server implementations, versions and forks. It should also be straightforward to both configure workloads and deploy any target MVE server without worrying about infrastructure setup for benchmarking. Infrastructure provisioning process for virtual machines should be automated and configurable to enable quick benchmarking for any MVE.

R3. **Scalable workloads**: VoxelBench must implement scalable workloads that can be adjusted to simulate varying player counts. The implementation should easily scale across nodes.

R4. **Reproducible**: All randomized processes (spawn placement, waypoint selection) should be seeded. Seeds should be recorded with the experiment manifest. Each benchmark run should record seeds, workload parameters, world/map identifiers, bot setups, etc such that experiments can be repeated under identical conditions.

4.3 Player Emulation Models

This section describes the implementation of player simulation models and satisfies **R1**.

4.3.1 Player Emulation Rationale

Implementing a benchmark that reproduces realistic workloads involving thousands of concurrent players is currently unfeasible. Even for popular MVEs such as Minecraft, no established behavior models exist that would allow the construction of realistic workloads (16). Moreover, obtaining actual game traces is extremely challenging, since commercial game providers are protective of both player-trace data and system-architecture details (31). MineRL (32) is the most prominent public dataset; however, the dataset lacks multiplayer interaction and mostly contains task-centric gameplay such as navigation, diamond mining, etc. and short gameplay scenarios. All these factors limit its usefulness.

In contrast, a performance benchmark is feasible. Such a benchmark does not need to capture all contingencies of real-world play. Instead, it must be capable of stressing the

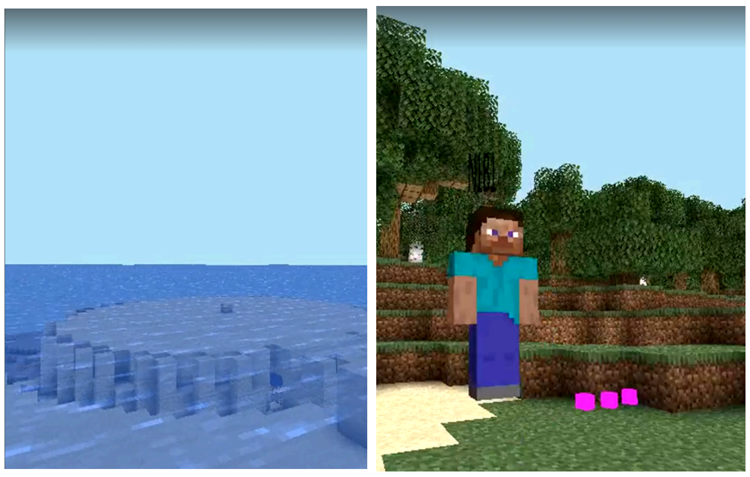

Figure 4.1: Left: Bot swimming, Right: Bot attacking another bot

system in ways that expose bottlenecks and scalability limits. This approach has already been demonstrated effectively in several single-server MVE experiments (5, 8, 16). Due to its successes in the literature, we decide to implement player emulation models to create workloads designed in Section 3.3.

4.3.2 Walking and Exploration

Walking and exploration model generates both player and terrain workloads. As the player walks around the environment, workloads such as collision detection, player information broadcast, etc. are generated. As the player enters different chunks, the game server starts uncompressing NBT chunk files and loads them into memory. In this model, after the bot is spawned at some random location within the given radius, the bot first sets up a constant variable that is a cardinal direction (east, west, north, south). It then generates a random destination in the chosen cardinal direction which does not change for the whole test. The destination blocks can be around 10 blocks away from the bot's location. Using a pathfinder module, the player walks, swims, or digs (breaks voxel blocks in its path) to its destination. On reaching its destination, it sets a new random destination to walk to again which is in the direction of the chosen cardinal direction. A first-person view of a walking bot is shown in Figure 4.1

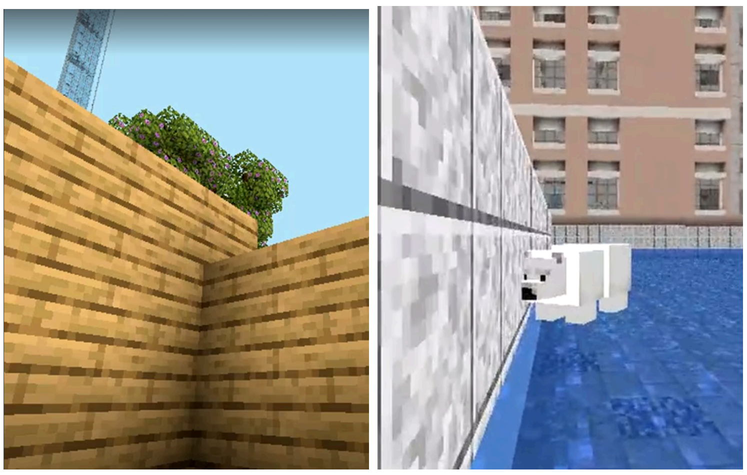

Figure 4.2: Left: Bot building a house, Right: Bot attacking a *polar_ bear* entity

4.3.3 Player versus Player

The PvP model (player versus player) generates player workloads. For this player model to work correctly, the *density* variable for the benchmark should be at least set to 2.

In this model, after a few bots have spawned at a random location them equip themselves with armor. The armor comprises a helmet, shield, chest armor, boots, and leggings. It also equips a sword. After equipping itself with weapons, it sends p2p chats to all other bots in its vicinity. The chat sent is "Fight me!". Whenever a bot receives a "Fight me!" message, it includes the sender bots' username into a foe list. From the list of foes, each bot identifies the closest bot to it and proceeds to engage them in combat. If a bot dies, it respawns in the same location to which it teleported at the beginning. It re-equips itself with armor and then proceeds to fight the nearest bot on its foe list. An image of a bot attacking another bot in first person is shown in Figure 4.1.

4.3.4 Player versus Environment

The PvE model generates entity workloads. By spawning entities or attacking entities, we force the server to compute entity simulation functions such as pathfinding, spawning, decision making (fleeing, attacking), etc.

The user decides which entity should be spawned. For this model to work, the underlying game server and the game world should support the spawning of that entity. After spawning at a random location, the bot equips itself with all 4 pieces of armor, a shield, and sword. Then it spawns an entity, tagged with a unique name. It proceeds to then engage and attack the entity. It respawns a new entity if it successfully terminates the spawned entity. An image of a bot attacking an entity *polar_ bear* is shown in Figure 4.2

4.3.5 Building

The Building model generates terrain workloads. The world's terrain is changed by adding new blocks. These changes have to be made persistent on the disk. Also, the chunks need to be compressed before storage, which further adds to the compute required. In our model, after spawning at a random location, it equips itself with *oak_ planks* using operator commands. It then proceeds to build a box home using the planks. On completion, it walks to a new destination and repeats the process. On average, it takes a bot around 4 minutes to build a home. Figure 4.2 shows a bot building a house using plank blocks.

4.3.6 Simulated Constructs

In our SC workload reference example for Minecraft, the benchmark executes workloads on the redstone-mountain-village map. Upon spawning, bots are programmed to randomly traverse the area, actively seeking out various redstone contraptions, which could include automated farms, complex piston doors, and other such mechanisms. All of these redstone contraptions are pre-activated before the bots begin their exploration, setting the stage for their interactions within the environment.

4.3.7 Player Spawn

In Virtual Environments, when a new player joins the map, the player spawns at the origin point. The player then gradually explores the world outward from the spawn point to other areas of interest (14). Player clustering in the game can range from densely populated around areas of interest to very sparsely populated.

Figure 4.3 portrays how we simulate this behavior. In our figure, ❶ represents the game world. Currently, popular MVEs like Minecraft have an origin ❸. For our benchmark, the user is expected to define a *radius*, as shown by Ⓡ. The circle ❷ in the Figure portrays the area where a bot can be teleported. When a bot joins the game world, the bot is instantly teleported to a random location within the circle ❷.

Another configurable variable for player spawn is *density* Ⓓ. Using this configurable variable, the user can set the number of bots that would teleport to one random location. This number is guaranteed by the benchmark. Using these two configurable variables, the user can test for different conditions. Setting a large radius with low density would set up a

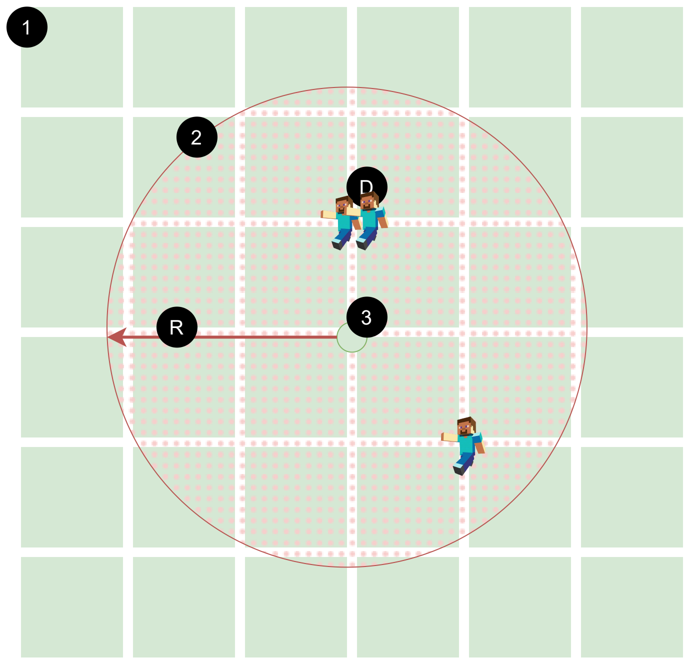

Figure 4.3: How voxelBench spawns bots in a Minecraft Map

sparsely populated world. Setting a small radius with high density would set up a densely populated world around the origin. We also allow setting up the same spawning behavior from multiple origin points, instead of a singular origin point which the game provides.

To create this functionality, we used two plugins that are compatible with different versions of Minecraft.

- Skript (33): Using Skript, a Minecraft script is installed such that when a bot joins it is converted into an operator so that it can teleport to any given location.
- EssentialsX (34): This plugin ensures our bots get safely teleported to valid locations. It's main use is preventing bots from getting teleported inside walls and getting terminated.

Through this, we satisfy the design requirements mentioned in Section 3.5.

4.4 Extensible Benchmark

This section describes how our benchmark can be easily extended to benchmark other MVEs (**R2**) and the implementation of scalable workloads (**R3**).

1. Section 4.4.1 implements a node provisioning system under the Unified Benchmark Model (See Section 3.7) to easily extend the benchmark to other sharded MVEs.

2. Sections- 4.4.4, 4.4.3, 4.4.4 give an example implementation of Multipaper Node setup with our benchmark, which are set up using Ansible. In our benchmark, the user needs to write ansible scripts corresponding to the node tags, and the benchmark runs the script on those exact nodes. Refer to Figure 4.4. Through Ansible scripts targeting specific node types, the user setup any distributed MVE.

3. Section 4.4.4 discusses the implementation of scalable workloads by scaling bot nodes.

4.4.1 Node Provisioning

Node provisioning is done through a configuration file, which also defines parameters such as memory and CPU allocation for each node type. The configuration file also tags the different nodes based on the node types and the bot node in the configuration file. An example configuration file for benchmarking Multipaper is given below. For quick local VM provisioning, libvirt-vagrant is used.

```
[coordinator]
memory=32768
cpu=2

[game-server]
memory=32768
cpu=2
total=8

[bot]
memory=16384
cpu=1
total=7
```

The benchmark user can easily add more node types through our node provisioning system and configure new components.

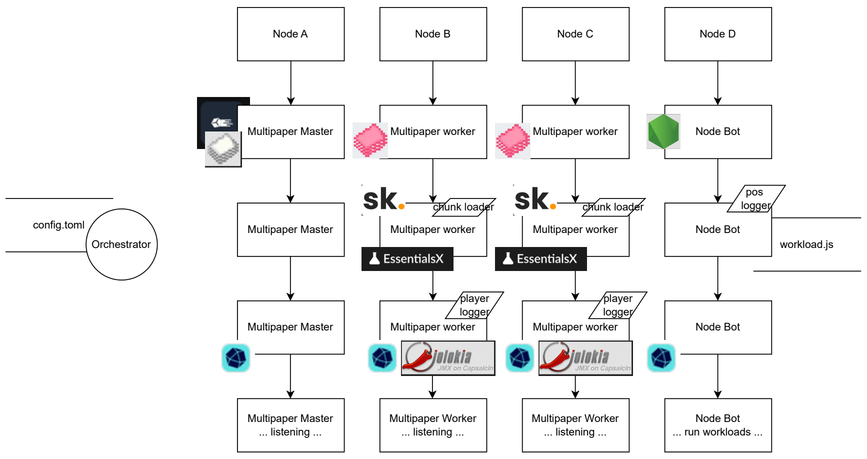

Figure 4.4: voxelBench configuring nodes under the benchmark model.

4.4.2 Coordination Node Setup

We first set up the required Java environment on the node. Following this, a dedicated Velocity working directory is created, where the essential Velocity configuration files are copied. A security step that involves setting up forwarding secrets to backend worker nodes is done next.

With secret setup, the next phase focuses on network and plugin configuration. This includes setting up the Velocity proxy server, configuring the load balancer to manage traffic effectively, and then integrating the RCON plugin for remote console access.

Finally, for configuration management, Dasel is installed. This tool allows easy editing of the Velocity server's TOML configuration files. The setup process ends when the world data (defined by the user) is copied into the master node.

4.4.3 Game Server Node Setup

Our Game Server Node setup begins with establishing a Java environment. Subsequently, we set up the dedicated worker directory.

Next, we download the MultiPaper worker JAR and execute it to generate initial templates. Plugin installation follows, including both the Skript, EssentialX, and our ChunkLogger plugins. We then configure multipaper.yml to ensure correct secret-based communication with the master node.

After accepting the EULA, additional templates are generated. Further configuration involves setting up server restart scripts, along with Bukkit, Pufferfish, and Spigot configurations. To automate operator setup, we install Skript scripts. Finally, the worker is initiated and run with Jolokia.

4.4.4 Bot Node Setup

Our bot node configuration begins with the installation of Node.js. We then copy over the necessary Node.js workload files. Next, an RCON client is set up to configure spawn locations. Finally, we install the required libraries for running a Minecraft client. The benchmark allows the user to create as many Bot Nodes as they want.

By running each bot on its own thread with no dependencies, we can easily deploy across multiple nodes, enabling simple scalability of workloads with the addition of more computing resources.

4.4.5 Exposing Metrics

In this section, we describe how we have implemented metric collection for our benchmark for Multipaper. The collected metrics are defined in Section 3.4.

System Metrics: System-level metrics, including CPU load, network activity, RAM usage, and disk I/O, are collected using Telegraf. The same program is run across all Gateway, Bot, and Game Server Nodes.

Thread Metrics: Thread metrics are collected using Java Flight Recorder for the benchmarking Multipaper. The monitored events are *jdk.JavaMonitorEnter, jdk.JavaMonitorWait, jdk.ThreadPark, jdk.ThreadSleep.*

Application Metrics: Player count is collected using the command *slist* which allows server operators to get all players on a game server node. The command is executed every second on one Multipaper's worker console to get a list of all players on the node using a named pipe.

Player location is readily accessible via the mineflayer-client on the bot node. We log each bot's coordinates every two seconds.

Chunks loaded is collected using our *ChunkLogger plugin.* It is installed on every MultiPaper worker server and queries the system every second for all currently loaded chunks. This chunk data is then logged to a file, which we retrieve once the benchmarking concludes.

The tick length is collected using Jolokia. Running each Multipaper server instance with Jolokia exposes the tick rate. Through jolokia, we log each and every game server instance's tick. Disconnects is collected through reading the logs of the gateway server node.

4.5 Reproducibility

This section explains benchmark configuration and the use of seed to satisfy the requirement **R4**.

4.5.1 Benchmark Configuration

This section describes how benchmarking is run for an MVE. As illustrated in Figure 4.4, VoxelBench follows a structured sequence to initiate a benchmark run.

Once the nodes are provisioned and dependencies installed, the orchestrator executes the specified workload on the bot nodes. The details of the workload to be run are provided by the user through a configuration file. An example of a test run in a configuration file is given below.

```
[benchmark]
world='WorldOfWorlds'
playerModel='pve'
density=5
radius=1000
players=25 # this is per bot node
pve_mob='polar_bear'
joinDelaySecs=5
```

On completion of a benchmark run, the orchestrator collects all performance, configuration, and monitoring data from each node and stores it locally for analysis.

4.5.2 Seed Implementation

All random processes in VoxelBench are initialized with a fixed seed to ensure deterministic behavior across runs. However, due to the nature of virtual environments, being non-deterministic, only player spawn locations and movement directions are guaranteed to be deterministic.

5

Performance Evaluation of Multipaper

In this section we use the Atlarge cluster and Google Cloud Compute to evaluate the performance of a sharded MVE, namely, Multipaper. This chapter answers **RQ2**. We begin with our experimental setup, followed by the main findings, motivation and experiment design for each of the experiments we ran, and then end with observations for each of the experiments.

5.1 Experimental Setup

Table 5.1: System Configurations

Name	Nodes	CPU	Memory (MB)
1x8	8	1	8192
2x4	4	2	16384
4x2	2	4	32768

For the experiments given below, two different node types were used. For small scale tests, one node at the AtLarge Cluster was used. The description of each host system and VMs is described in the following sections.

5.1.1 AtLarge Cluster Host

This host system is running Ubuntu 22.04.5 LTS, powered by an Intel Xeon Silver 4210R processor, with 20 cores running at 3.2 GHz and 256 GB of DDR4 memory. The RAM

Table 5.2: Main findings for experiments. Exp IDs (Experiment IDs) are linked to Section 5.2.

Section	Exp IDs	Main Finding
§5.3.1	1	2 CPU each across 4 nodes can best handle 100 players.
§5.3.2	1, 2	Multipaper achieves similar performance to paperMC.
§5.3.3	4	Larger worlds perform significantly better under exploration workloads.
§5.3.4	3	Walk, PvP perform better under more nodes. PvE, Build perform better under more cores.
§5.3.5	5	Multipaper cannot efficiently scale beyond 200 players.
§5.3.6	5	Multipaper can scale beyond 500 players using a proximity load balancer for players.
§5.3.7	1, 3, 5	Multipaper has high chunk sharing and can reach upto 80% for build workload.
§5.3.8	5	Multipaper's syncing overhead by the main loop thread is negligible and does not impact performance.

manufacturer is Samsung and is configured at 2400 MT/s. The SSD of the host system is Intel D3-S4510 SSD with 480 GB of storage.

5.1.2 Google Cloud Compute Host

This host system is called *n4-highmem-80* is running a Linux-based operating system, powered by an Intel Emerald Rapids processor, with 80 vCPUs (virtual CPUs) running with a base frequency of 2.1 GHz and a maximum turbo frequency of 3.3 GHz, with 640 GB of DDR5 memory.

5.1.3 Virtual Machines

The virtual machines were created using Vagrant. To set up the VMs, the vagrant-libvirt plugin communicates with libvirt to create the VMs and is configured to use the Linux KVM hypervisor such that we get very little overhead. The CPU mode for libvirt was configured to use "host-passthrough". Each VM is running a Vagrant box that can be downloaded from Hashicorp Cloud Platform under the name generic/ubuntu2204.

5.2 Experimental Design, Motivation & Main Findings

Using VoxelBench, we conduct 5 experiments on Multipaper. The main findings for different experiments are listed in Table 5.2. The motivation and design of the experiments are as follows:

1. **Core & Node Scaling:**

   Goal: Identify which vertical CPU configurations (core setup) are best suited for handling a fixed number of concurrent players.

   Method:

   - Keep the total core count constant across tests.
   - Vary the number of nodes and concurrent players to assess how distributing the same compute across multiple machines impacts performance.

   Insight Expected: Shows whether single, high-core servers or smaller multi-node setups yield better performance for the same total compute budget.

2. **Sharded Impact**

   Goal: Quantify the overhead of sharded clusters compared to standalone servers while scaling player count.

   Method:

   - Baseline: single standalone server (PaperMC, Multipaper) handling N players.
   - Sharded setup: cluster of servers, each handling N players (so total concurrent players = 2× baseline).
   - Compare performance metrics (latency, tick time) across both setups.

   Insight Expected: Reveals the coordination overhead introduced by sharding.

3. **Workload Impact**

   Goal: Determine how different player behaviors (workloads) affect performance, using two stable vertical CPU setups and keeping concurrent players fixed.

   Method:

   - Vary cluster configuration but keep concurrent player count fixed.
   - Vary workloads such as Walk, Build, PvE, PvP.

Insight Expected: Highlights how performance differs across workloads.

4. **World Impact**

   Goal: Examine how the underlying world complexity influences server performance under fixed concurrent players, walk workload and fixed cluster setup.

   Method:

   - Use a fixed number of concurrent players, walk workload and fixed cluster setup.
   - Change only the underlying world map.

   Insight Expected: Isolates the cost of world size on tick performance.

5. **Weak Horizontal Scaling**

   Goal: Study how horizontal scaling affects performance when concurrent player and node counts grow proportionally.

   Method:

   - Gradually increase both the number of nodes and the number of concurrent players at a fixed ratio (e.g., +1 node per +N players).
   - Compare performance and synchronization cost as the system scales outward.
   - Also compare performance using our custom proximity load balancer.

   Insight Expected: Demonstrates how efficiently multipaper scales under "weak scaling" conditions — where each node handles roughly the same load, but the player base expands.

5.3 Findings

Given below are our findings for each of the experiments mentioned in Table 5.2. In each plot, the red dashed line signifies the 50ms tick, representing the threshold for acceptable performance, as Minecraft's server logic is designed for a minimum update rate of 20Hz. The different node configurations are given in the Table 5.1. The gateway Node which contains Velocity Proxy that runs Multipaper Master as a plugin, runs 2 CPU with 32768 MB of RAM across all configurations.

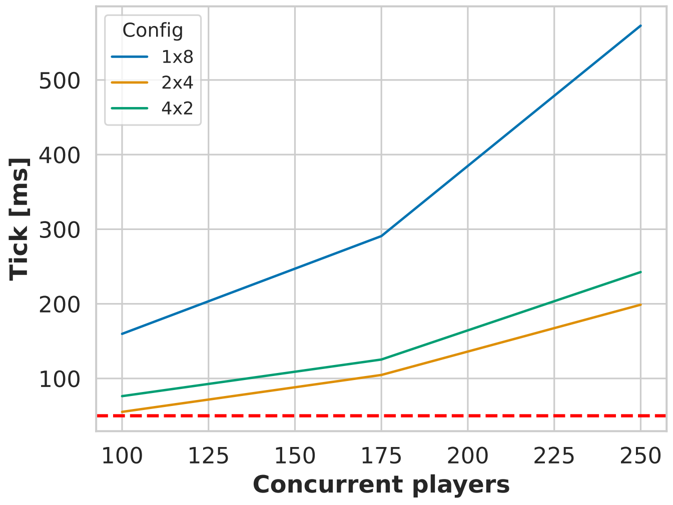

Figure 5.1: P95 Tick from 100 to 250 players

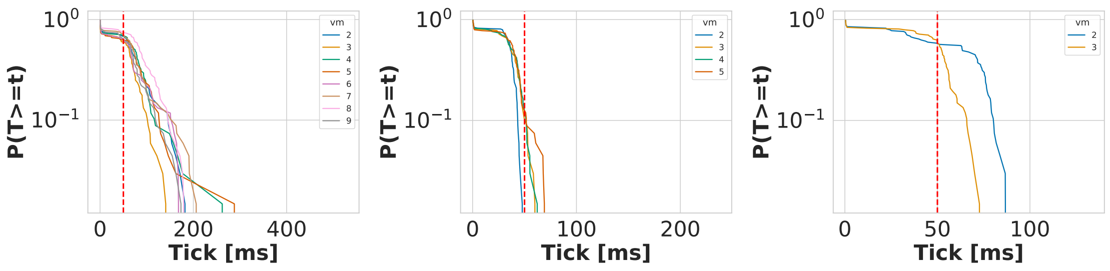

Figure 5.2: Tick distribution for 100 players: left: 1x8, middle: 2x4, right: 4x2.

5.3.1 2 CPU config works best for 100 players.

In Figure 5.1, we compare three configurations: 1x8 (1 CPU across 8 nodes), 2x4 (2 CPUs across 4 nodes), and 4x2 (4 CPUs across 2 nodes). Among these, the 1x8 configuration performs the worst, while 2x4 delivers the most stable results. None of the setups, however, sustain the required tick rate once player counts exceed 100. Tick latency increases sharply with every 75 additional players, and in the 1x8 case, the rise is the steepest, quickly making the game unplayable.

Figure 5.2 highlights that 2x4 maintains a playable state for roughly 90% of ticks, whereas 1x8 and 4x2 both experience prolonged unplayable periods. The time-based performance shown in Figure 5.3 further confirms that 1x8 never reaches a tick rate below 50 ms during the entire test. Most tick spikes occur during the start of the run—when players join and the world loads. Both 2x4 and 4x2 stabilize within about 2 minutes, while 1x8 continues to fluctuate for nearly 2.5 minutes. Although 4x2 stabilizes the quickest, 2x4 remains the

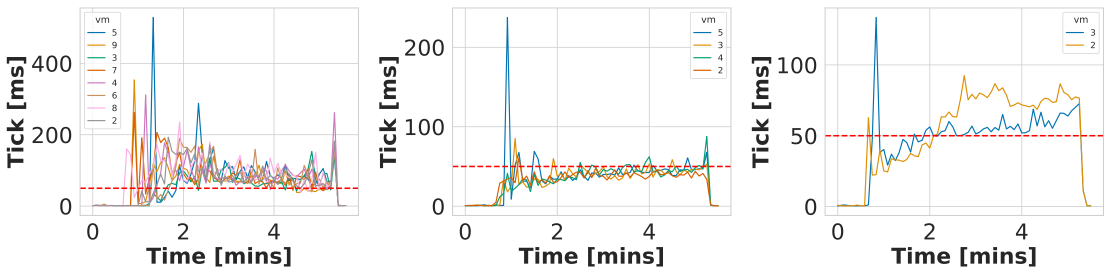

Figure 5.3: Tick timeline for 100 players: left: 1x8, middle: 2x4, right: 4x2.

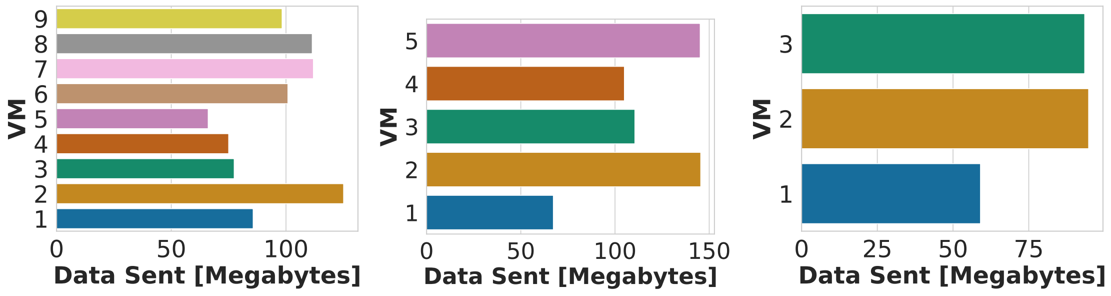

Figure 5.4: Data sent [MB] for 100 players: left: 1x8, middle: 2x4, right: 4x2.

most consistent overall once the game world settles.

Also, as shown in Figure 5.4, configurations with fewer VMs also reduce network overhead. The 4x2 setup sends out around 80 MB per VM over 5 minutes, compared to 125 MB for 2x4—about 45% less data. Meanwhile, 1x8 generates higher data transfer rates, particularly from the proxy node (VM 1).

Figure 5.5 explains why tick performance is worst in the 1x8 setup. In this configuration, nearly twice as many chunks are loaded compared to 2x4 and 4x2. Specifically, the average chunks per node are approximately 8500 chunks for 1x8, 5800 chunks for 2x4, and 6100 chunks for 4x2. Since we have kept the CPU core count the same across the cluster, 1x8 has to compute more chunks in comparison with 2x4 and 4x2.

In summary, tick performance is best in the 2x4 configuration, worst in 1x8, and unstable in 4x2. Beyond 100 players, all setups struggle to maintain the required tick rate.

Based on our results, we can conclude that 2 CPU configs can handle around 25 players whereas 4 CPU configs can handle 35-40 players. We can now use this ratio as the baseline for our weak scaling experiments, where the number of nodes is increased proportionally to these per-CPU player capacities (i.e., by +25-35 players for every +1 node increase).

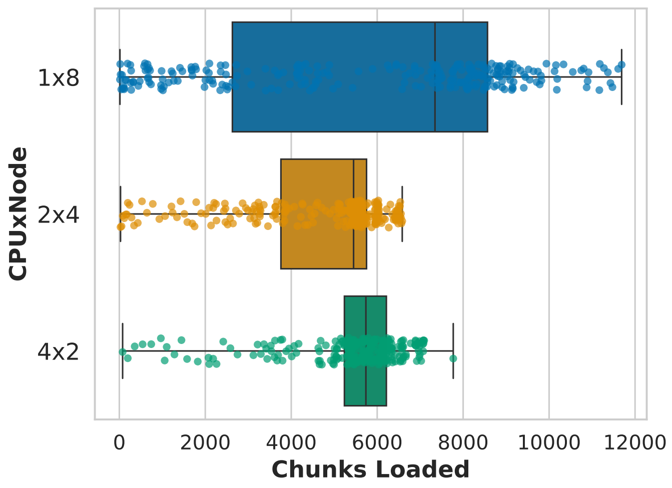

Figure 5.5: Chunks loaded (Queried at least every 2 seconds)

5.3.2 Multipaper achieves similar performance to paperMC.

MultiPaper demonstrates performance comparable to standard PaperMC under low player loads. Specifically, PaperMC, a single-worker MultiPaper cluster, and a sharded MultiPaper setup (with 25 players per node) all exhibit tick latencies in the range of 47–55 ms. This indicates that MultiPaper can achieve on-par performance with single-server setups upto 100 concurrent players. When the player count is increased to 50 players per node, MultiPaper incurs only an overhead of approximately 30%. However, at this load, both the single-worker MultiPaper cluster and PaperMC exceed the 50 ms tick latency threshold.

5.3.3 Larger worlds perform better under exploration workloads.

As shown in Figure 5.7, the GreaterKCMetro map, which is substantially larger than the other two maps, is the only configuration that achieves a p95 tick latency below 50 ms with 100 concurrent players. In comparison, the Castle Lividus map exhibits the poorest performance, with p95 latencies approaching 1 s—significantly higher than WorldOfWorlds, which records a p95 latency of approximately 72 ms. Overall, the Castle Lividus p95 tick latency is about 2100% higher than that of GreaterKCMetro, rendering smaller worlds such as Castle Lividus unplayable under high player concurrency.

The degraded performance observed in Castle Lividus can be attributed to terrain generation overhead. In this experiment, the spawn radius was configured to 1500, exceeding the total map size of Castle Lividus. Consequently, as players explore the world, the

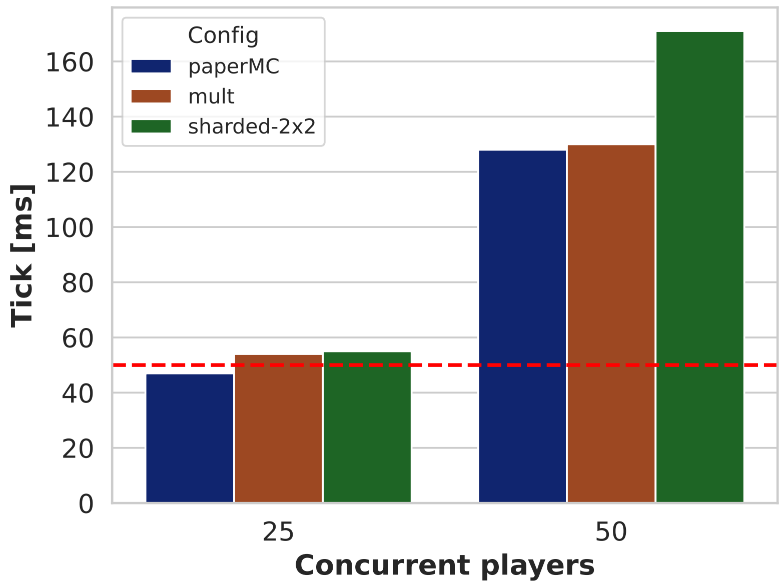

Figure 5.6: P95 tick for different single servers vs sharded. Mult refers to a cluster with a single Multipaper worker instance and 2x2 refers to a cluster with 2 nodes with 2 CPU each.

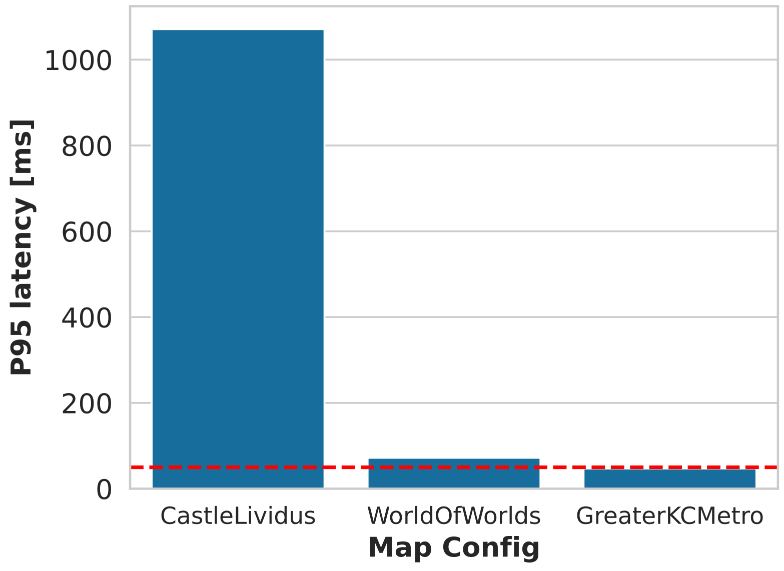

Figure 5.7: P95 tick for different maps for 100 concurrent players. Redline represents 50ms.

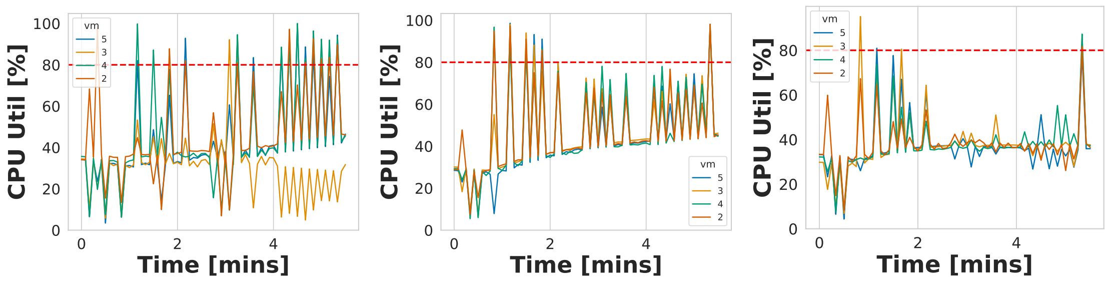

Figure 5.8: CPU utilization for different maps for 100 players. Left:CastleLividus, Middle: WorldOfWorlds, Right: GreaterKCMetro

server must continuously generate new chunks beyond the preexisting terrain. This effect becomes particularly pronounced with 100 simultaneous players, as the rate of chunk generation increases sharply, leading to sustained tick latency spikes.

In contrast, the GreaterKCMetro map avoids this issue, since the spawn radius lies entirely within its pre-generated terrain. When players traverse new areas, the game server retrieves stored chunk data (NBT files) from the central database instead of generating new chunks, resulting in lower latency and more consistent tick performance.

As shown in Figure 5.8, CPU utilization patterns further validate this finding. Both Castle Lividus and WorldOfWorlds show periodic utilization spikes; however, Castle Lividus consistently has usage spikes above 80%, reflecting the additional computational cost of real-time terrain generation.

5.3.4 Walk, PvP perform better under more nodes. PvE, Build perform better under more cores.

As shown in Figure 5.9, several performance patterns emerge across the different workloads and cluster configurations. The dataset compares two configurations—2x4 (2 CPUs per node across 4 nodes) and 4x2 (4 CPUs per node across 2 nodes)—under workloads of Walk, PvP, PvE, and Build, each tested with 100 and 175 concurrent players.

From the observed tick latencies, we note the following trends:

- Walk and PvP workloads achieve lower tick times in the 2x4 configuration, indicating that these workloads benefit from a higher number of nodes. This suggests that actions involving world traversal and player interactions scale more efficiently with increased distribution of main loop threads—the threads responsible for chunk computation and synchronization across nodes. Refer to Figure **??** In other words,

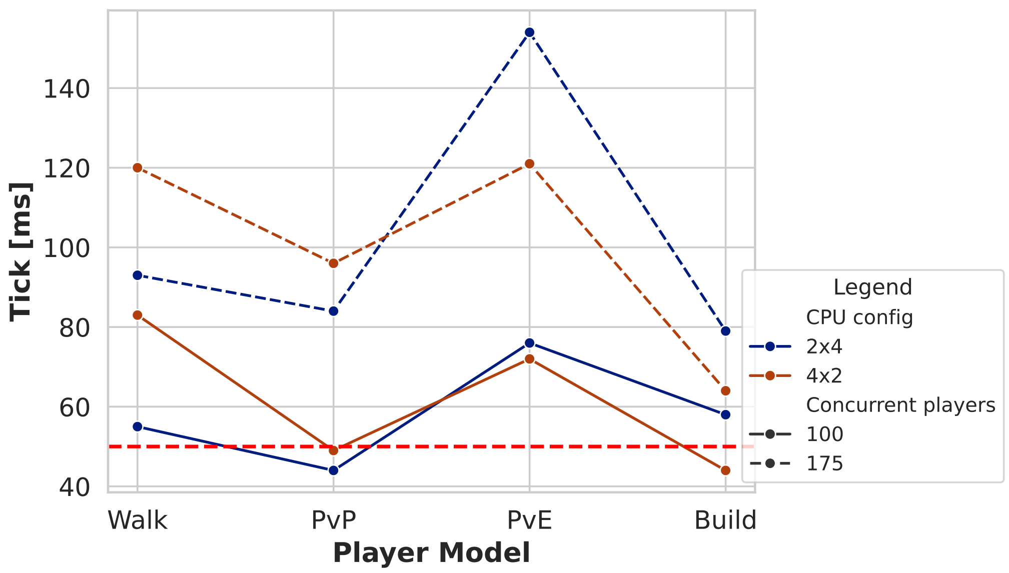

Figure 5.9: P95 tick for different workloads for 100, 175 concurrent players across 2 CPU each on 4 nodes and 4 CPU each on 2 nodes. Redline represents 50ms.

when players and combat simulations tick latency is lower. Refer to Figure 5.11 for more details.

- Build and PvE workloads, on the other hand, show improved performance in the 4x2 configuration, which provides more CPUs per node. These tasks are computationally more intensive at the node level. In PvE scenarios, the primary overhead comes from entity simulation, which is handled by separate threads distinct from the main loop. A higher per-node CPU count provides more available worker threads for concurrent entity updates. Similarly, in Build workloads, frequent block placements and updates generate significant inter-node data synchronization. More CPUs per node allow the server to better handle this increased data exchange through parallel worker threads. Refer to Figure 5.10 for more details.

In summary, Walk and PvP benefit from broader horizontal scaling (more nodes correspond to more main loop threads), while Build and PvE benefit from stronger vertical scaling (more CPUs per node correspond to worker threads having better access to compute resources. This distinction highlights that different gameplay mechanics stress different parts of the MVE's thread model—movement and combat depend primarily on chunk computation, whereas building and entity-heavy activities rely on local compute.

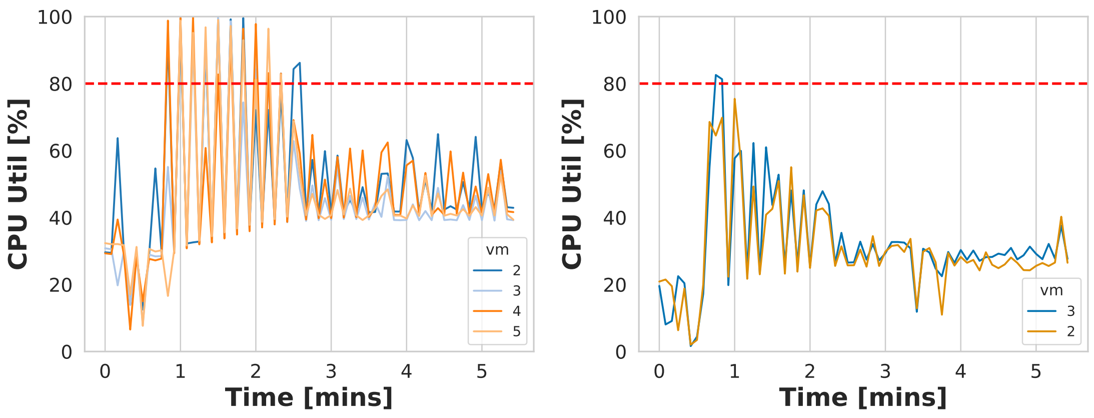

Figure 5.10: Overall CPU utilization for 175 concurrent players under Build workload, utilization is higher for nodes with 2 CPUs (2x4). Left: 2x4, Right: 4x2.

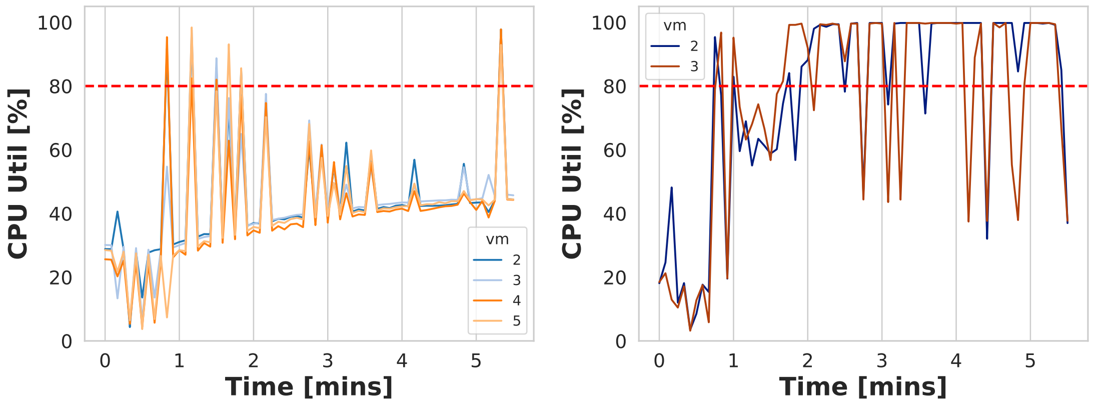

Figure 5.11: Main Loop CPU utilization for 100 concurrent players under Walk workload, utilization is higher in the cluster with only two nodes (4x2). Left: 2x4, Right: 4x2.

5.3.5 Multipaper cannot efficiently scale beyond 200 players.

As shown in Figure 5.12, Multipaper cannot scale horizontally. The p95 tick keeps on further increasing as the number of nodes increases. For 25 players per 2 CPU node, from 100 to 300 concurrent players, the tick rate rise is very steep. It increases by 56% from 71 ms to 110 ms. However, the tick rate is much better for 33 players per 4 CPU node setup. In this setting, it increases by 26% from 55 ms to 71 ms for 100 to 300 concurrent players. Our 4 node CPU uses 2x both memory and CPUs; by increasing resources by 100%, it achieves around a 40% boost in performance. For the 2 CPU case, Multipaper servers become mostly unplayable by the time we have 400 concurrent players. For the 4 CPU case, this is highly likely to occur by the 600 concurrent player case based on the trajectory shown in Figure 5.12.

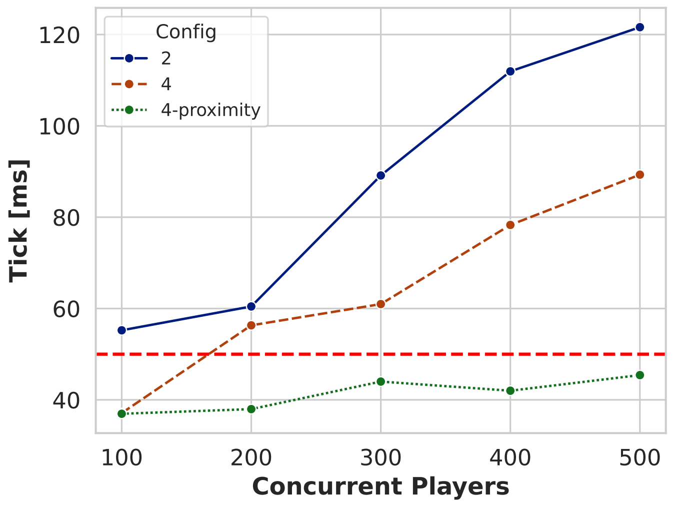

Figure 5.12: Concurrent player count under weak horizontal scaling. For 2 CPU, 25 players per node. For 4 CPU, 33-34 players per node. Proximity config uses a player proximity load balancer.

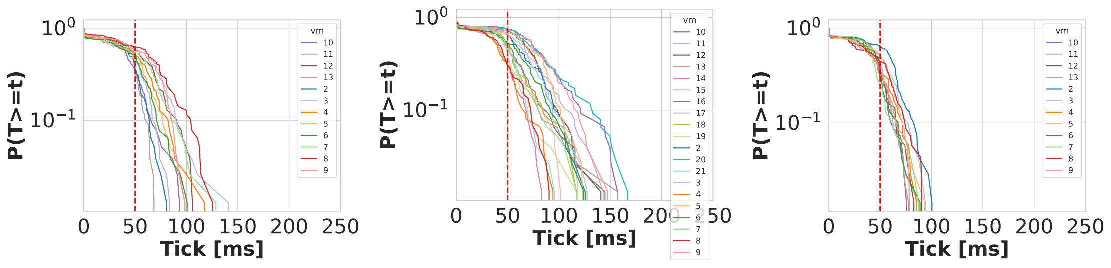

Figure 5.13: RCDF for 2 CPU and 4 CPU Node Setups. Left: 2 CPU each for 12 nodes (25 players per node), Middle: 2 CPU each for 20 nodes (25 players per node), Right: 4 CPU each for 12 nodes (33-34 players per node)

Also, CPU configuration also impacts tick performance across nodes. As shown in Figure 5.13, for 2 CPU each per node, the variance of tick rates among different VMs is fairly wide compared to the 4 CPU each per node setup. This variance also keeps on increasing as we increase the number of nodes.

In the case of 4 CPU nodes across 12 nodes, as shown in Figure 5.13, at p90, the maximum difference in tick rate between the VMs is 57 ms for VM-10 and 82 ms for VM-2, which results in a difference of about 25%. However, in the case of 2 CPU across 12 nodes, the maximum difference in tick rate between the VMs is 55 for VM-13 and 103 ms for VM-8 which is around a 47% difference. This difference further expands to around 60% at p90 when we increase the number of VMs to 20. The p99 tick also has significant variance among the vms for 2 CPU setups. Hence, it is important to also select the correct CPU

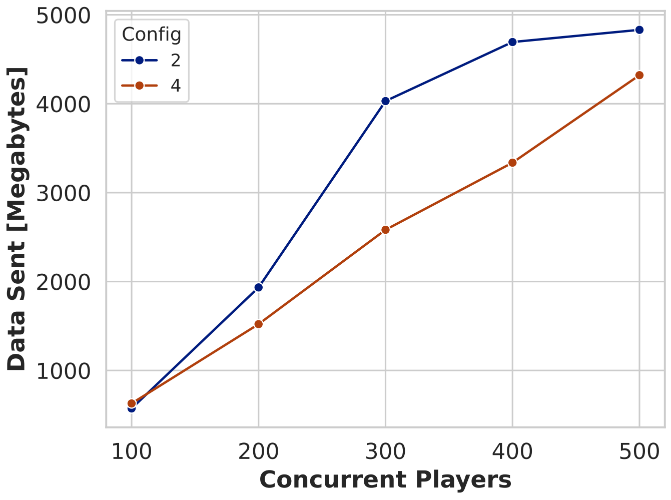

Figure 5.14: Data send [MB] for 100-500 concurrent players. For 2 CPU config, 25 players per node and 33-34 players for 4 CPU config.

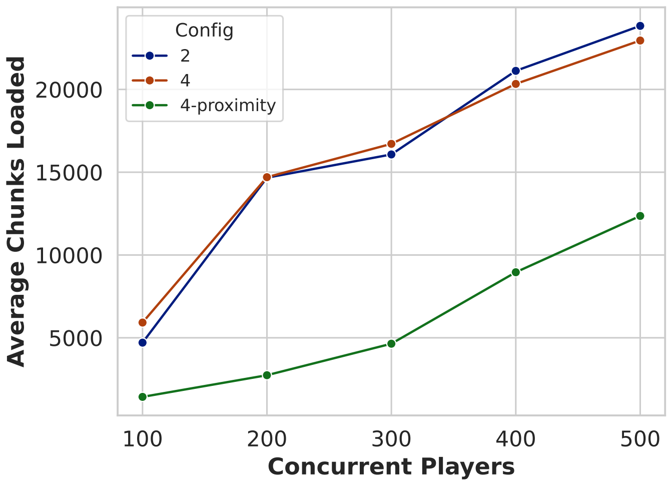

Figure 5.15: Average Chunks loaded (Queried at least every 2 seconds). 2 CPU setup manages 25 players per node. 4 CPU setup manages 33-34 players per node.

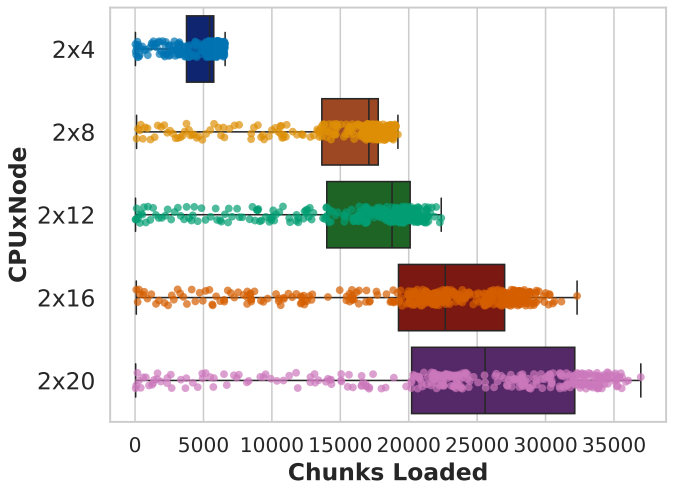

Figure 5.16: Average Chunks loaded (Queried at least every 2 seconds) for 2 CPU setup. Each node had 25 concurrent players.

configuration within the cluster.

As shown in Figure 5.14, total data sent grows approximately linearly with both the number of players and the number of nodes. However, once the cluster is large enough that a node's potential peer set is saturated—i.e., the maximum number of nodes a given node can exchange state with is capped by the configured players-per-node limit—the curve begins to flatten. Beyond this point, adding players yields little additional fan-out, so aggregate data sent saturates and thereafter increases only slowly.

The maximum number of nodes a given node can communicate with also impacts chunks loaded. As shown in Figure 5.16, from 100 concurrent to 200 players, the chunks loaded increase by at least 180% whereas when we double from 200 to 400, we only get an 80% increase.

As shown in Figure 5.15, our results demonstrate a clear distinction in the average number of chunks loaded across different scaling configurations. In the 2× setup, the average chunks loaded range from approximately 4,707 to 23,838, showing a broad increase as concurrency rises. Similarly, the 4× configuration exhibits a comparable growth pattern, ranging from 5,920 to 22,953 chunks on average. However, the 4-proximity configuration shows a markedly lower range, with averages spanning only from 1,431 to 12,359 chunks. This reduction—nearly 40–60% fewer chunks loaded compared to non-proximity configurations—illustrates the effectiveness of proximity-based player distribution in minimizing redundant chunk loading. Hence, the 4-proximity setup achieves significant efficiency by reducing shared chunk overlap between nodes, resulting in lower total chunk loads.

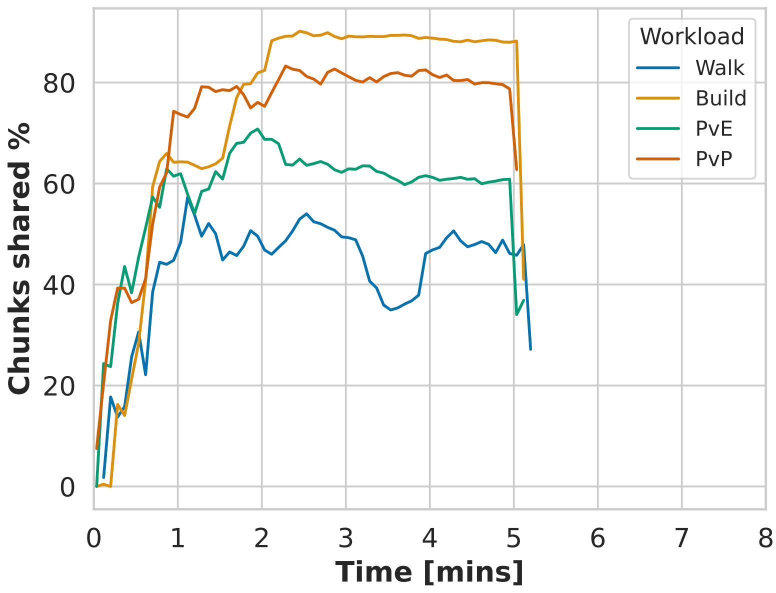

Figure 5.17: Chunk shared % across different workloads.

5.3.6 Proximity load balancing of players improves performance.

As shown in Figure 5.12, if we do proximity load balancing amongst our players in Multipaper, we can consistently hit below 50 ms ticks. In proximity load balancing, we try to assign players that are close to each other in the game world to the same node. By assigning them to the same node, we reduce communication and chunk sharing. While in our experiments, the proximity-based load balancing was done through a controlled approach by manually directing players to the correct nodes, this shows the promise of gains just by changing our load balancer to use a player proximity technique. From 100 to 500 concurrent players, Multipaper achieved a very good playable p95 tick rate and was always below 50 ms. This also helps us conclude with certainty that chunk data coordination contributes the most to high tick latencies.

5.3.7 Chunk sharing reaches 80% under the least connected load balancer.

Across different workloads, MultiPaper exhibits distinct patterns in chunk sharing between nodes. During the Build workload, chunk sharing reaches around 85%, as players typically remain close together while constructing structures, clustering around a few zones. The least connected load balancer further exacerbates this issue since it randomly distributed players without taking player proximity into account.

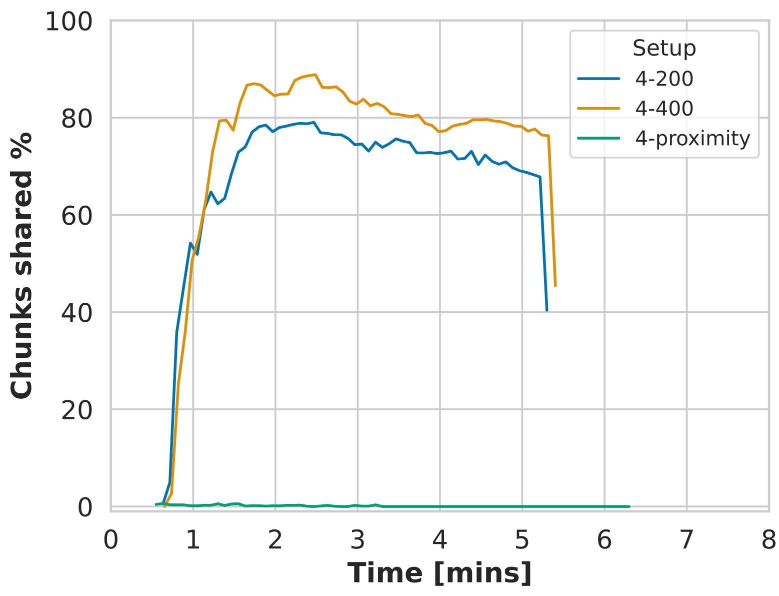

Figure 5.18: Chunk shared % across different node and load balancer setups.

In PvP scenarios, sharing is similarly high at 80%, since players often engage in combat within localized areas, forming dense interaction clusters. The PvE workload shows a moderate 60% chunk sharing, as players spread out while chasing entities, moving farther from the spawn area.

In contrast, the Walk workload demonstrates the lowest chunk sharing at 45%, reflecting highly dispersed player movement as participants freely explore and move randomly across the world. Refer to Figure 5.17 for more details.

In addition, as shown in Figure 5.18,when testing under weak horizontal scaling—with approximately 33–34 players per node—the chunk sharing behavior changes as concurrency increases. At 200 concurrent players, chunk sharing ranges between 70–80%. When scaling to 400 players, chunk sharing rises to 80–85%, indicating a slight increase in chunk sharing since we have more players which increases the overlap. In contrast, the proxy-based proximity distribution test exhibits 0% chunk sharing, as players are spatially partitioned to ensure minimal inter-node overlap. This configuration helps keep latency consistently below 50 ms, maintaining smooth gameplay while demonstrating the performance advantage of geographically aware player distribution.

Table 5.3: Average synchronization time across 2 CPU and 4 CPU configurations (500 players)

Configuration	**Total Sync Time (ms)**	**Average Sync Time(μs)**
2x20	192	59
4x15	112	22.97

5.3.8 Multipaper has negligible data syncing overhead between nodes.

Sync time is negligible across both experimental configurations. As shown in Table 5.3, for the 2×20 configuration, the total sync time of 192 ms over a 6.5-minute run ($\approx$ 4875 ticks at 80 ms per tick) results in an average sync time of approximately 0.039 ms ($\approx$ 39 μs) per tick. Similarly, for the 4×15 configuration, the total sync time of 112 ms over the same duration corresponds to an average of about 0.023 ms ($\approx$ 23 μs) per tick. In both cases, the synchronization overhead is well below 0.05% of the tick duration, indicating that inter-node coordination is effectively hidden within normal tick execution. This confirms that the system's distributed data synchronization mechanism introduces a negligible performance cost.

6

Related Work

In research, there has been significant work done in testing distributed architectures and creating benchmarks to evaluate the performance of Modifiable Virtual Environments (MVEs). However, most existing benchmarks specifically target non-sharded, single-server MVEs. While prior work has explored scaling techniques for multiplayer online games, little attention has been given to distributed MVE architectures, where the world state is actively shared and modified across multiple coordinated servers.

6.1 Yardstick

Yardstick (Donkervliet, van der Sar, and Iosup, 2019) (8) introduced one of the first systematic benchmarking frameworks for Minecraft-like services. It established a methodology to evaluate the performance of game servers under controlled workloads that emulate realistic player behavior. Yardstick provided insights into CPU utilization, latency, and throughput, helping identify bottlenecks in monolithic server setups. However, its design was primarily focused on single-instance deployments, limiting its applicability to distributed or sharded environments.

6.2 Servo

To address scalability limitations inherent to single-server systems, Servo (Donkervliet et al.,) (11) proposed leveraging serverless computing to increase the scalability of MVEs. The framework decomposed the simulation into fine-grained, stateless components executed within a serverless infrastructure. This approach showcased the potential of elastic scaling and reduced resource contention, demonstrating an architectural shift toward decentralized

MVE execution. Nevertheless, the model focused on modular task parallelization rather than consistent world-state distribution, leaving open questions about world partitioning across multiple servers.

6.3 Meterstick

Complementing these studies, Meterstick (Eickhoff, Donkervliet, and Iosup) (16) examined performance variability across cloud-hosted and self-hosted Minecraft-like environments. By benchmarking latency and performance fluctuations across heterogeneous infrastructures, Meterstick provided valuable data on deployment stability and resource variability. Although it improved understanding of hosting environments, it did not explore performance for distributed MVE systems.

7 Conclusion

This chapter concludes the research presented in this thesis by reflecting on the three research questions posed in Section 1.3. Each question is revisited in light of the design, implementation, and benchmarking of VoxelBench, our performance benchmark for Modifiable Virtual Environments (MVEs).

7.1 Answering Research Questions

Through a performance evaluation of the sharded MVE server MultiPaper, this thesis addresses *the lack of a benchmarking framework for sharded Modifiable Virtual Environments (MVEs).*

We contribute a methodology for the design of a distributed benchmarking framework, realized through the implementation of ***VoxelBench***, for the evaluation of scalability and performance of distributed MVEs.

RQ1. How can we design a benchmark to evaluate distributed MVEs? To design a benchmark capable of evaluating distributed Modifiable Virtual Environments (MVEs), this thesis developed VoxelBench, a benchmarking framework tailored for sharded MVEs. The design process involved identifying suitable workloads, determining player spawning strategies, and defining new performance metrics that capture the performance of distributed MVEs.

1. We selected workloads such as Exploration, PvP, PvE, Build and SCs that specifically target design decisions under the paradigms of resource balancing and information management.

2. We designed a metrics collection system that collects metrics in three different levels—system, thread and application. We also introduced two derived metrics—average sync time, shared chunk %— specifically for characterizing performance of distributed MVEs.

3. We then added player clustering to further enrich our workloads.

4. We underscored the importance of map workloads and selected popular maps for our benchmark.

5. We created a unified benchmark model to map metrics to node types and enable easier provisioning of compute resources across different MVE distributed architectures.

RQ2. How can we implement a scalable and extensible benchmark framework for distributed MVEs? Once we had the design of our benchmark following from answering **RQ1**. We implemented a benchmark that created workloads through player emulation models and was easily extensible to other distributed MVEs.

1. To create configurable workloads, we developed several player emulation models — including walking and exploration, building, PvE, PvP, and simulated constructs — to represent typical in-game activities. We also implemented a flexible spawning system that allows users to control player density and distribution, which helps reproduce both sparse and dense gameplay scenarios.

2. We implemented a node provisioning system that allows users to easily benchmark different MVEs by configuring nodes and resources through a simple configuration file. Using Ansible, we automated the deployment of game servers, bot nodes, and coordination nodes. This makes it easy to add or modify components for both single-server and sharded MVE architectures.

3. We then implemented seeded randomization for all stochastic processes within our benchmark and automated both provisioning and workload execution using a simple configuration file.

RQ3.What performance trade-offs and coordination challenges arise in dynamic, sharded MVEs, and how do they impact scalability? We evaluate our benchmark by generating usable insights through benchmarking a sharded MVE, namely Multipaper. From our benchmarking results, we found that certain workloads such as Walk and PvP benefited from more nodes, while workloads such as Build and PvE benefited from more cores. We were also able to conclude that Multipaper does not scale through the default load balancer, which is called the Least-Connected load balancer. Additionally, we were able to show that, through a proximity load balancer, Multipaper could scale.

7.2 Limitations and Future Work

Our evaluation did not account for skewed player distributions. Most bots in our experiments were arranged into clusters containing an equal number of players, resulting in a relatively uniform load across the virtual environment. Consequently, the effects of uneven player concentrations—such as those typically observed in real-world scenarios near popular landmarks or shared structures—were not reflected in our results.

Similarly, the performance gains of the proximity-based scheduler may be overstated, as our benchmarking sessions were of short duration and did not simulate long-term or highly skewed proximity patterns. Extended workloads or evolving distributions might reveal different scheduling dynamics and contention patterns over time.

For our build workload, each bot was assigned to construct an independent structure. This design naturally limited the number of terrain modifications within the same region. However, in real-world modifiable virtual environments (MVEs) such as Minecraft, players frequently collaborate while building structures, which could substantially increase the frequency of terrain edits and the resulting CPU and network load.

References

[1] Business of Apps. **Minecraft Revenue and Usage Statistics (2024)**, 2024. Accessed: 2025-05-13. 1

[2] Mojang Studios. **Minecraft Education Edition**, 2025. Accessed: 2025-05-13. 1

[3] Raluca Diaconu and Joaquín Keller. **Kiwano: Scaling virtual worlds**. *2016 Winter Simulation Conference (WSC)*, 2016. 1, 2, 18

[4] Raluca Diaconu Mathieu Valero and Joaquín Keller. **Manycraft: Massively distributed minecraft**. *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, **11**(1), 2014. 1

[5] Jim Cuijpers Jesse Donkervliet and Alexandru Iosup. **Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency**. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 1294–1304. IEEE, 2021. Accessed: 2025-05-13. 1, 2, 31

[6] MultiPaper Team. **MultiPaper: A Distributed Minecraft Server**, 2025. Accessed: 2025-05-13. 1, 2, 14, 18

[7] PaperMC Team. **Folia: Regionised Multi-threaded Paper Fork**, 2025. Accessed: 2025-05-13. 1, 2

[8] Jesse Donkervliet J. van der Sar and A. Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2019. 1, 2, 19, 23, 31, 57

[9] John M. Hogg. **How Massive Multiplayer Online Games Incorporate Principles of Economics**. *ResearchGate*, 2015. 1

[10] Jackson Roberts. **How we built an auto-scalable Minecraft server for 1000+ players using WorldQL's spatial database**. 2, 14, 18

[11] Jesse Donkervliet, Javier Ron, Junyan Li, Tiberiu Iancu, Cristina L. Abad, and Alexandru Iosup. **Servo: Increasing the Scalability of Modifiable Virtual Environments Using Serverless Computing – Extended Technical Report**, 2023. 2, 11, 25, 57

[12] **Spatial Database: WorldQL**. 2, 16

[13] Juan González, Fernando Boronat, Almanzor Sapena, and Javier Pastor. **Key Technologies for Networked Virtual Environments**, 2021. 2, 18, 21

[14] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. **Colyseus: A Distributed Architecture for Online Multiplayer Games.** 01 2006. 2, 3, 18, 24, 33

[15] Anthony Steed and Roula Abou-Haidar. **Partitioning crowded virtual environments**, 2003. 2, 3, 18, 20

[16] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games Extended Technical Report**, 2023. 2, 7, 9, 10, 19, 21, 30, 31, 58

[17] **Purpur Minecraft**. 8

[18] SpigotMC Community. **About BungeeCord**, 2023. Accessed: 2025-05-13. 13, 16

[19] DonutSMP. **DonutSMP**, 2025. Accessed: 2025-05-13. 14

[20] **Paper Minecraft Game Server**. 15, 16

[21] **Velocity proxy**. 15

[22] Nitin Gupta, Alan Demers, Johannes Gehrke, Philipp Unterbrunner, and Walker White. **Scalability for Virtual Worlds**. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1311–1314, 2009. 18

[23] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda†, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. **Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games**, 2008. 18, 21

[24] Pedro Morillo, Juan M. Orduna, Marcos Fernandez, and Jose Duato. **Improving the Performance of Distributed Virtual Environment Systems**. *IEEE Trans. Parallel Distrib. Syst.*, **16**(7):637–649, July 2005. 20

[25] Pedro Morillo Tena, Juan Orduña, Marcos Fernández, and José Duato. **A Method for Providing QoS in Distributed Virtual Environments. 2005**, pages 152–159, 01 2005. 20

[26] Frank Glinka, Alexander Ploss, Sergei Gorlatch, and Jens Müller-Iden. **High-Level Development of Multiserver Online Games**. *International Journal of Computer Games Technology*, **2008**, 07 2008. 21

[27] John C. S. Lui and M. F. Chan. **An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems**. *IEEE Transactions on Parallel and Distributed Systems*, **13**(3):193–211, 2002. 21

[28] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. **VON: A Scalable Peer-to-Peer Network for Virtual Environments**. *IEEE Network*, **20**(4):22–31, 2006. 21

[29] Haiyang Hu, Yizhi Ren, Xu Xu, Liguo Huang, and Hua Hu. **Reducing View Inconsistency by Predicting Avatars' Motion in Multi-Server Distributed Virtual Environments**. *Journal of Network and Computer Applications*, **40**:21–30, 2014. 21

[30] Roman Chertov and Sonia Fahmy. **Optimistic Load Balancing in a Distributed Virtual Environment**. In *Proceedings of the 16th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, May 2006. 21

[31] Maxime Véron, Olivier Marin, and Sébastien Monnet. **Matchmaking in multi-player on-line games: studying user traces to improve the user experience**, 2014. 30

[32] **mineRL**. 30

[33] **Skript hub**. 34

[34] **EssentialsX**. 34

Appendix A

Reproducibility

A.1 Abstract

This artifact appendix describes the experiment set up as well as the implementation of the benchmark.

A.2 Artifact check-list (meta-information)

- **Compilation: Gradle (java-21), maven**
- **Run-time environment: Ubuntu 22.04.5 LTS, Vagrant 2.49, Ansible 2.16**
- **Hardware: Intel Xeon Silver 4210R 3.2 GHz, 256 GB DDR4, google-cloud n4-highmem-80 node**
- **Execution: python 3.10**
- **Metrics: tick latency, tick rate, chunk shared %,**
- **Output: Execution logs, application metrics csv, bot video data, player positioning csv, system metrics csv, thread metrics csv**
- **How much disk space required (approximately)?: 50 GB**
- **How much time is needed to complete experiments (approximately)?: 20 mins to run a single workload**
- **Publicly available?: Yes**

A.3 Description

A.3.1 How to access

Github Project Link: voxelBench

A.4 Experiment workflow

As described in Section 5.2

A.5 Experiment customization

As described in Section 4.5

A.6 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`

...