Bachelor's Thesis
(Course Code: XB_40001)

Envirostick: Benchmarking Enviroment-based MVE Workloads For Game Networking Libraries

by

Benedict Rigler
(Student Number: 2744003)

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

August 20, 2024

Certified by ....
Jesse Donkervliet
PHD student
First Supervisor

Certified by ....
TBD
TBD
Second Reader

Table of Contents

Envirostick: Benchmarking Enviroment-based MVE Workloads For Game Networking Libraries

Benedict Rigler
Vrije Universiteit Amsterdam
Amsterdam, NL
B.rigler@student.vu.nl

ABSTRACT

The gaming industry continues to experience rapid growth, making the development of efficient multiplayer online games increasingly critical, especially for smaller developers who often depend on existing networking libraries due to limited resources. However, there is a notable lack of established benchmarks to evaluate the performance of these libraries, particularly in the context of virtual environment workloads typical of Minecraft-like games (MLGs). Existing research has not sufficiently addressed this gap, necessitating further investigation. This thesis tackles the problem by designing, implementing, and conducting real-world experiments with a benchmark specifically tailored to virtual environment workloads within the Opencraft framework. The study provides a comparative analysis of various gaming libraries, measuring key performance metrics to assess their operational efficiency and resource utilization. The results reveal significant performance disparities among the libraries, underscoring the importance of networking library selection in managing game development costs and server resources. The insights gained from this research offer valuable guidance for both researchers and practitioners, aiding in the informed selection of networking libraries for MLG development and enhancing multiplayer gaming experiences.

1 INTRODUCTION

The gaming industry is one of the largest entertainment sectors, boasting approximately 3 billion players worldwide with a revenue of around $200 billion [11]. A significant portion of popular video games today are multiplayer online games. Among these, Minecraft-Like Games (MLGs) have become particularly influential. MLGs feature expansive virtual environments where players can freely interact with and modify the world around them, often collaborating or competing with other players in real-time . MLGs make up a large aspect of video games, with the most popular MLG being Minecraft itself with over 300 million copies sold worldwide [8]. Minecraft still ranks as one of the most popular video games, even 13 years after its release [7]. In MLGs, one of the largest workloads on the network is the transfer of environmental information between the server and the clients that interact with this environment. In the case of Minecraft, the server models a world consisting of millions of individual, interactable blocks. Each block can have its own properties and state, that the server must manage and update. This large computational load significantly impacts performance, as the server needs to handle not only the storage and retrieval of block data, but also the physics, interactions, and changes resulting from player actions. In addition, the network load increases as the server communicates these updates to all connected players in real time, ensuring a synchronised and seamless gaming experience. The complexity and scale of such tasks can strain server resources, leading to performance degradation.

1.1 Problem Statement

While large game developers can afford to build their own infrastructure network for their games, game developers of smaller projects typically do not have the resources for this, in terms of both monitary and technical know-how. This means that most smaller developers opt not to build their network infrastructure but instead rely on existing libraries to manage the networking aspects. Despite the multitude of options available, there is little to compare these libraries beyond their basic feature, since there is no established benchmark yet. This thesis aims to develop this benchmark to offer a quantitative method for evaluating these libraries. Moreover, the resources that these libraries use and the service they provide have a large impact on the developer, as a more performance intensive network infrastructure will increase the cost of running the game [16].

1.2 Research Questions

In this study, the objective is to investigate and compare the performance of different networking libraries within the specific context of environmental workloads in online multiplayer gaming. To this end, we formulate and address the following three research questions (RQ) in this work.

RQ1 How to design a comprehensive benchmark to effectively evaluate the performance of various game networking libraries under environmental workloads?
RQ2 How to implement such a benchmark in practice?

RQ3 What are the performance characteristics of selected gaming libraries when subjected to the implemented benchmark, and how do these libraries compare in terms of their performance under environmental workloads?

1.3 Research Methodology

To create a comparison of these libraries, we will design and build a benchmark to test the multitude of such libraries. Moreover, this paper will implement a subset of popular libraries in this reference and draw a comparison of their performance. To create a comprehensive comparison of these libraries, we will design and build a benchmark to test a wide range of libraries commonly used in this domain. This benchmark will be carefully constructed to evaluate key performance metrics. Moreover, this paper will implement a representative subset of popular libraries, selected based on their prevalence and relevance in the current industry. Lastly, we will conduct an analysis of the benchmarking results to ensure the reliability and validity of our findings. This approach will provide an understanding of how these libraries perform relative to each other and will offer valuable insights to researchers and practitioners in the field.

1.4 Thesis Contributions

This work makes three main contributions:

C1 The design of a benchmark to test gaming networking libraries against environmental workloads. this benchmark will ensure a fair comparison of the performance of various gaming libraries.

C2 The implementation of that benchmark in Opencraft to test a subset of gaming libraries.

C3 Experiments on that benchmark to analyse the performance of a subset of gaming libraries and a discussion of their performance

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

2 BACKGROUND

In this section we provide an overview of the essential background information relevant to the this thesis

2.1 Minecraft-like Games

Minecraft-like games (MLGs) utilize a traditional client-server model, where the server handles most of the computational tasks. Users install client software locally, which connects to the server to interact with the game world. In this model, the server is responsible for maintaining the state of the game, managing the logic of the game, and ensuring that all clients receive the necessary updates to provide a consistent environment.

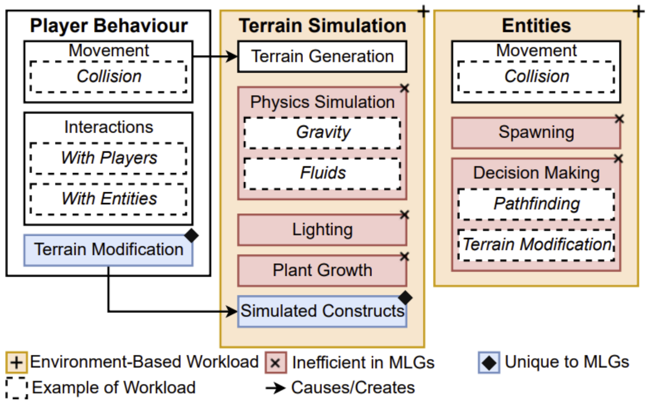


Figure 1: Workload components in MLGs, taken from Meterstick[3].

A key component in this architecture is the Network Manager, which handles the incoming connections from clients. It manages connection establishment, maintenance, and termination, ensuring secure communication channels through encryption layers. The Network Manager also encodes and decodes messages, exposing them to the game loop via a message queue for processing. This setup allows the game loop to update the game state based on client interactions and maintain synchronised gameplay.

The game consists of a virtual world hosted by the server, where players interact with the environment and each other. This virtual world is composed of a terrain, represented by a grid of immovable blocks, and various entities that can move freely. Entities include player-controlled avatars and non-player characters, each capable of interacting with the environment and other entities. Players can build structures, mine resources, craft items, and engage in combat, with the server continuously updating and broadcasting the game state to ensure a consistent experience for all participants.

The Minecraft Virtual Environment (MVE). In this context, the environment refers to the intractable area with which the player engages during the game. This environment varies significantly between games. Some games utilise a static environment that is unmodifiable, where each client stores the environmental data locally, and only player information is shared among clients. In contrast, other games feature dynamic environments that players can modify, necessitating synchronisation between all players. In this paper, we will focus on dynamic environments, as they can substantially impact network performance, since this needs to be shared across all clients.

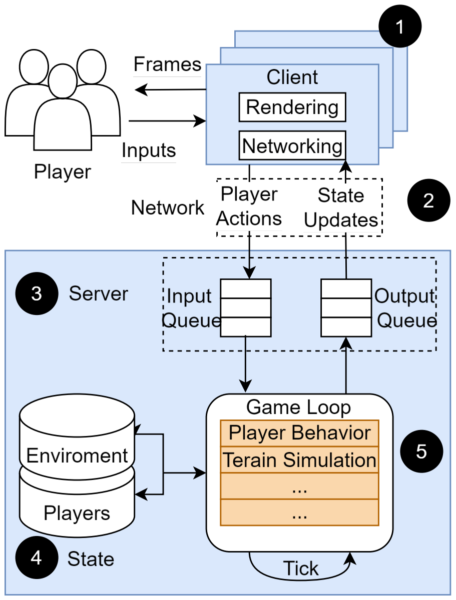


Figure 2: Overview of a simplified MLG.

Minecraft-like Games (MLGs) incorporate a virtual environment known as the Minecraft Virtual Environment (MVE). The MVE represents one of the most significant computational burdens on MLGs. This substantial impact is attributable to two primary factors. Firstly, MVEs constitute a considerable proportion of the workloads that MLGs must process and manage, as illustrated in Figure 1 (Workload Components). Secondly, many of these MVE-related tasks are particularly resource-intensive due to the inherent architecture of MLGs. In this illustration, the sections highlighted in red represent these resource-intensiv workloads.[3]

MLG system model. The MLG system model is quite simple, as it consists of a server and a client. Figure 2 represents this.

The client (❶) has two primary responsibilities. First, it converts player input into in-game actions, speculatively applying these actions to the local state while simultaneously sending them to the server for validation. These actions are sent over the network (❷) by the client-side networking script.

The server (❸) is responsible for executing all in-game environment simulations, maintaining the global state(❹), processing player actions, and sending state updates to system clients. It consists of a network input and output queue where incoming player actions are received and outgoing state udates are sent out.

The game loop (❺) performs simulations by applying state updates to the global state in discrete steps, or ticks, at a fixed frequency. If a tick takes less than the fixed frequency, the MLG waits until the next scheduled tick to start.

However, if any part of the server takes longer than a tick to process, the tick frequency drops, causing the server to enter an overloaded state. In this state, the game fails to meet its quality of service requirements, leading to game stuttering, visual inconsistencies, and increased input latency for players.[17]

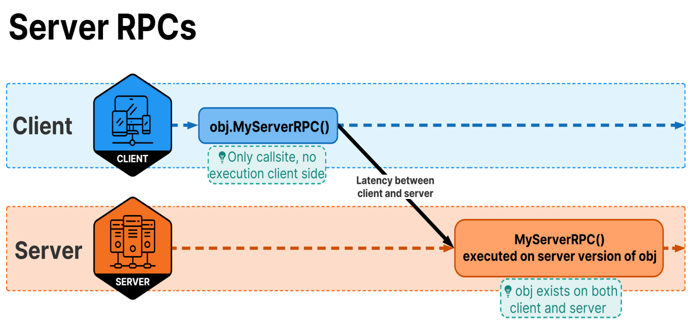

Server RPCs
Client
CLIENT
obj.MyServerRPC()
Only callsite, no execution client side
Latency between client and server
Server
SERVER
MyServerRPC() executed on server version of obj
obj exists on both client and server

Figure 3: RPC diagram taken from Unity's Netcode for game Objects documentation [15].

2.2 What is Netcode?

Game Networking, also known as multi-player networking or 'netcode', involves synchronising the state of the game and events among multiple players connected over a network. Netcode provides the necessary information to make a game multiplayer on the Internet, ensuring that all players have a consistent and shared experience. Its primary focus lies on addressing the complexities inherent in multiplayer online games, such as latency, jitter, and packet loss. These challenges can significantly affect gameplay and are mitigated by using advanced algorithms and techniques.

In addition to these core challenges, Game Networking is intricately intertwined and integrated with various game systems such as physics, animation, and game mechanics. For example, the netcode must ensure that the physics engine's calculations are consistent across all clients, so objects behave predictably in the game world. Similarly, animations must be synchronised to ensure that all players see the same character movements at the same time. Gameplay mechanics, such as hit detection and game rules, also rely heavily on accurate and timely data transmission to maintain fairness and responsiveness in multiplayer interactions.

Advanced netcode implementations may include techniques like server reconciliation, where the server periodically corrects the game state based on authoritative data, and client-side prediction, which allows players to see immediate responses to their inputs, even before the server confirms them. These mechanisms work together to provide a seamless and immersive multi-player experience, overcoming the inherent limitations of network environments.

Video games typically operate on a server-authoritative model, where the server holds the primary authority to modify the game state. Clients request changes through Remote Procedure Calls (RPCs), and the server processes these requests. The structure of these RPCs is shown in figure 3. In traditional video game architecture, the server manages a queue of these client-sent changes, updates the game state accordingly, and then sends the updated state to all connected clients. This model ensures that the server has the final say on the correctness of player movements and interactions, maintaining the integrity and consistency of the game environment.

3 BENCHMARK DESIGN

In this section, we present the design of the benchmark. First, a set of seven requirements is defined. Secondly, the high-level design of the benchmark is presented and the details that help fulfil these requirements. Thirdly, we discuss the proposed workloads to benchmark. Lastly, we discuss the metrics used to analyse the results of the benchmark.

3.1 Requirements

Here we describe the seven requirements for the benchmark. The first 2 are relevant for the use case of this benchmark, while the last 4 are based on existing guidelines.[5][18]

R1 **Validity of Workloads:** The workloads utilized in the benchmarking process should reflect real-world usage scenarios. Workload validity ensures that the performance measurements obtained accurately reflect the behaviour of networking libraries in practical, everyday situations, enhancing the relevance and applicability of benchmarking results.

R2 **Relevance of Metrics and Experiments:** The benchmarking system must be capable of accurately capturing key performance metrics related to networking libraries.

R3 **Fairness:** Fairness in benchmarking is crucial to ensure an unbiased assessment of networking libraries. The benchmarking process should provide a level playing field for all libraries, avoiding scenarios that may favour or disadvantage any particular library unfairly. Fairness promotes trustworthiness and reliability in benchmarking outcomes.

R4 **Clarity:** The benchmarking system should adopt a clear and structured approach to present results. Clear presentation of results aids in understanding and interpreting performance characteristics effectively, facilitating informed decision-making based on benchmarking outcomes.

R5 **Reconfigurability:**The benchmarking system should offer easy reconfigurability to enable swapping of underlying networking libraries. This feature allows for the efficient testing and comparison of multiple networking libraries without significant overhead or complexity in the benchmarking setup.

R6 **Ease of use:** The benchmarking system should prioritize ease of use, configuration, and setup procedures. User-friendly interfaces, clear documentation, and streamlined processes contribute to making the benchmarking system accessible to a wide range of users, ensuring that obtaining performance results for new systems remains straightforward and efficient.

3.2 Design Overview (R3, R4, R5, R6)

In this section, we present the design of the benchmark, the goal of which is to measure the performance of the network libraries with the environment workloads.

The benchmark comprises three distinct experiments; The initial experiment assesses the resource load, examining the network libraries under different loads. The subsequent experiment concentrates on server-side environment modification, evaluating how the net code handles a large amount of data coming into the client. The final experiment investigates client-side environment modification, specifically focusing on the impact of remote procedure calls (RPCs) facilitated by network libraries. Each experiment is carried out across each library using different parameters.

Figure 4 presents the high-level design of the benchmark. In our design, the user interacts primarily with the controller, which is responsible for running the set of experiments. The user configures the benchmark through the benchmark configuration (❶), allowing modification of aspects such as runtime and repetition and parameters relevant to the benchmarks. This setup makes it easy to add new Opencraft 2 builds in the future if more net code libraries are integrated into the benchmark additionally one can easily change inbuilt parameters to change the behavior of the experiments. This also has the goal of satisfying the requirement R5 by making the system easy to reconfigure.

After configuring the benchmark, the user initiates the benchmark run, and the Initiator (❷) is initialised. Its purpose is to execute the benchmark using the provided configuration. These arguments are passed when running the experiment and are interpreted by the argument readers (❸) in both the client(s) and server, setting up the behavior for both sides accordingly.

The server starts and runs with the given configuration until terminated. The client operates similarly but has a set timer for its run-time. Upon completion, the controller will terminate the server and proceed to the next experiment. To

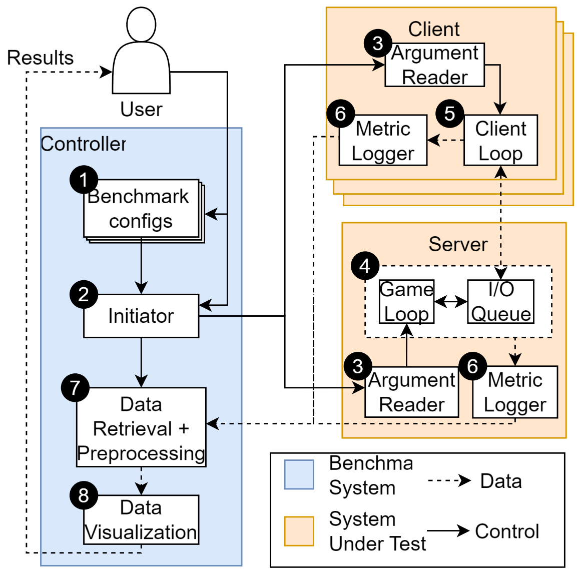


Figure 4: Overview of the benchmark system.

collect data, the server and client have a metric logger component (❼) that records relevant system metrics at regular intervals of half a second. This data is saved as a CSV file upon completion and read by the data retrieval component (❽) of the controller. The data is then processed, and new files with the relevant metrics are generated. Finally, the data visualisation component (❾) interprets the results and produces a set of graphs for the user to analyse and interpret, so satisfying R4.

This straightforward design ensures ease of use, thereby meeting requirement R6. The experiments will be implemented consistently across all libraries to ensure fairness, thus satisfying R3.

3.3 Benchmark Workloads (R1, R2)

This section discusses the different workloads used to test the benchmark and aims to address the requirements of R1 and parts of R2.

Workload: Resource load. The first experiment aims to measure the impact of network libraries on system resources and how this scales with increasing environment size. It will involve a single client and a server. The server will run with progressively larger networked objects in the form of blocks that will make up the virtual environment, and a player will connect and remain on the server for a set period for each world size. Information will be recorded on both server and client performance to assess how the increase in networked objects affects these libraries. This workload will be implemented in the starting server state as seen in figure 2 (❹). This measurement is crucial because modern MLGs must handle massive worlds with millions of objects. Understanding the impact of the number of objects and how each network library scales its resources is important, as these resources are costly.

Workload: Environment Modification. The second experiment focuses on terrain modification, aiming to analyze the performance of the networking libraries under a heavy load of environment updates. Specifically, it examines how the client-side connection handles a large volume of block updates. This experiment will consist of a single player receiving a large amount of state updates. In most MLGs, the server sends information about updated blocks and entities to the client, which can place a significant load on the clients netcode, especially in large worlds with many users or when large simulated constructs exist on the server that the server must simulate and update for the client. Since Opencraft 2 lacks many of these features, the benchmark will simulate this by sending new blocks to a single player at an increasingly rapid rate. This workload will be implemented in the servers game loop as seen in figure 2 (❺).

Workload: Client Environmental Load. The last experiment focuses on the sending and receiving of a large number of state updates in the form of RPCs from the clients modifying the MVE. Its goal is to examine the impact of multiple players concurrently modifying the environment around them. This experiment aims to simulate a busy server and client scenario typical in many popular Minecraft servers, assessing how networking libraries handle significant data traffic between server and client. This workload will be implemented on the client side 2 (❶) and will simulate a player placing and removing a block. This experiment involves one server being connected to by an increasing number of clients, each actively modifying the terrain.

3.4 Key Performance Metrics (R2)

This section discusses the different metrics used to analyse the results of the benchmark and aims to address the requirements of R2

RTT. Round-trip time (RTT), also known as ping, in networking, measures the time taken to receive a response after initiating a network request. In our context, it specifically refers to the time it takes between a dummy message being sent from the client and a dummy response being received. RTT measures how quickly the network library processes a message and sends a response. Therefore, RTT serves as a comprehensive metric for evaluating the network performance of each library and is measured in milliseconds. Typically a good RTT is between 40-50 ms, a tolerable RTT is between 50-100 ms and the RTT becomes problematic when it reaches above 100 ms [13].

Bandwidth Usage. Bandwidth usage refers to the amount of data transferred between network devices over a specific period. In our context, it measures the volume of data exchanged between the client and server during gameplay. Bandwidth usage is typically measured in bytes per second.

This metric is crucial for evaluating network performance and efficiency, as it directly impacts the user experience, especially in multiplayer games. Lower bandwidth usage indicates more efficient data transmission, potentially reducing latency and improving overall network responsiveness. It also allows for better scalability, as more players can be accommodated within the same network capacity.

CPU Usage . CPU usage refers to the percentage of the CPU capacity being utilized by a program or application at any given time. It is a important metric for evaluating the performance of networking libraries, as it reflects how efficiently the library processes data and manages network requests. The higher the CPU load compared to other networking libraries, the greater the additional load placed on the system by the networking library. CPU usage is typically measured as a percentage, where 100% indicates full utilization and 0% indicates that the CPU is idle.A higher CPU load will lead to increased system requirements, which, in the worst case, may incur costs for the client and game server provider, as they will need to pay for more expensive hardware. In a lesser case, it may cause a decrease in quality of service due to the reduction in performance.

Memory Usage. Memory usage refers to the amount of system memory (RAM) consumed by a program or application during its operation. It is typically measured in megabytes (MB). Lower memory usage generally indicates more efficient resource management, allowing the system to allocate memory more effectively for other tasks. This metric serves as a critical indicator of performance and efficiency for network libraries, as higher memory usage can impact overall system responsiveness and scalability. Additionally, the memory capacity of an MLG server is currently one of the primary factors influencing the cost of renting a private server. The general rule is that the more RAM a server has, the better the quality of service it can provide. This underscores the importance of minimizing additional memory consumption by a library; the lower the memory usage, the lower the cost of renting the server will be.

4 IMPLEMENTATION OF BENCHMARK

In this section, we describe the implementation of the benchmark and the prototypes that we will utilize to analyze performance through our benchmarking process.

4.1 Implementation of Game Prototypes

The Game Prototypes where written in C# in the Unity game engine. Unity is a cross-platform game engine developed by Unity Technologies, first released in 2005. It is designed for creating both 2D and 3D games and interactive simulations across multiple platforms, including desktop, mobile, console, and virtual/augmented reality devices. currently Unity is one of the most popular game engines being used today especially by smaller independent developers [19]. The prototypes can be found on Github.

Networking Libraries in this paper. Networking libraries simplify the creation of online games, as they abstract the difficulty and let the developer interact with a simple interface that manages the netcode aspect of their game. many libraries exist on the market with many of them being open source and free to use:

NFGO Net-code for Game Objects is a high-level networking library built for Unity by Unity to abstract networking logic. It is one of the most popular networking libraries as it is the default library hence is a preferred option, especially by people with limited networking know-how [14].

Mirror Mirror Networking is a system for building multiplayer capabilities for Unity games. It was created as a stable and easy to use open source Networking for Unity and was developed as an alternative to other libraries that were either expensive, unstable or a closed source black box. it is one of the most popular net-code libraries on the Unity platform [6].

Fishnet Fishnet Networking is another library that focusses on reliability, ease of use, efficiency, and flexibility. Fishnet is less known than Mirror, but is still a popular option among Unity developers [4].

OpenCraft 2. In this paper, we will focus on OpenCraft 2 as a base for our benchmark. OpenCraft 2 is a remake of Minecraft, developed using Unity and written in C#, unlike the original Minecraft, which was written in Java. It includes the core features of Minecraft, focusing on a cube-based environment with a simplified game loop, making it an excellent test-bed for research on Minecraft-like games. OpenCraft 2 operates on a similar system model as mentioned in Section 2.1. OpenCraft 2 was created by Jerrit Eickhoff in his master's thesis[2]. We chose Opencraft 2 primarily because its open-source approach allows us to modify the underlying netcode. Additionally, since it uses Unity's engine, there are many supported and well-known libraries to choose from, giving us a good selection of libraries to test.

Each game library was implemented within the existing OpenCraft 2 framework. Unlike the original OpenCraft 2 that used an entity-based approach to model the environment,

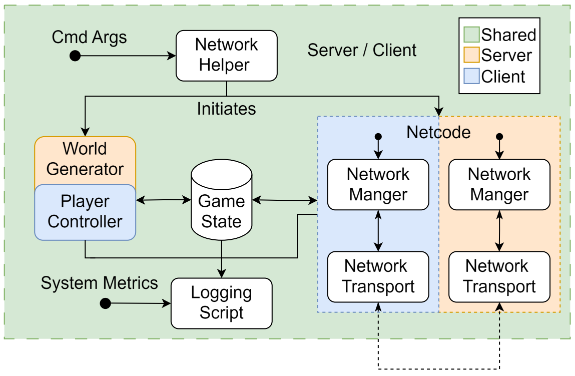


Figure 5: Simplified structure.

these implementations had to be made using GameObjects. This change was necessary because most existing game libraries do not support an entity-based system, and there are currently limited netcode libraries that do. Additionally, this shift in environmental representation would make comparisons more challenging. This aspect will be addressed in future research and further discussed in the conclusion.

As a result, we had to reconstruct much of the game loop for OpenCraft. While a significant portion of the code is based on the original, it differs substantially. We also removed features not relevant to our benchmark, as reimplementing these would not provide significant benefits for our study.

Instead of using the DOTS (Data-Oriented Technology Stack) system, we now use Unity's default object system, which means that every block is a networked object. While this approach has some performance disadvantages, it benefits our focus on netcode by placing a larger load on the network, allowing for more comprehensive testing of networking capabilities.

The New Game Loop. This implementation is based on the work of a BSc student who partially implemented the initial phase of the mirror system [12]. The revised game loop comprises several key steps. Initially, the Network Helper interprets command-line arguments and initiates the server. Upon activation, the server invokes the World Generator, which constructs the virtual environment based on the specified command-line parameters. Subsequently, the server enters a waiting state, prepared to accept incoming client connections. Once connected, clients receive the current game state from the server and render it for the user's interface. The clients then process player input and transmit these actions back to the server. Figure 5 illustrates the general flow of information within this system. The components of this game loop are explained in greater detail in the following sections.

Network Manager. The network manager is responsible for overseeing client connections, network configurations, and overall network behavior. The implementation of this component varies across different networking libraries.

Network Transport. The network transport is tasked with the fundamental operations of transmitting and receiving data over the network. This layer manages network connections, handling tasks such as establishing connections, maintaining them, and addressing disconnections. It functions analogously to a transport layer protocol but operates at the application layer, utilizing UDP as its underlying transport mechanism. The implementation of this transport layer is specific to each networking library.

Network Helper. This script is responsible for the initiation of the game and enables program execution via the command line interface. It serves as the entry point, initiating both server and client processes. Additionally, it configures the IP address and port for the connection.

World Generator. The world generator is a server-side script responsible for the generation and modification of the virtual environment. Server-side workloads are implemented within this component. The generated blocks are subsequently stored as part of the game state.

Player Controller. The player controller is a script that manages the game loop on the client side of the system. Client-side workloads are implemented within this component. The player controller processes player input, modifies the local environment, and then transmits Remote Procedure Calls (RPCs) to the server to update the game state.

Loggers. These components are responsible for collecting application-level metrics and recording them in logs, which are subsequently used for performance analysis. To record system metrics like frame time and memory usage we used the Unity profiling library, This allows you to sample relevant metrics at set time. Round-trip delay information was retrieved from the network managers, which maintain records of this data. The network managers continuously monitor and log the time taken for data packets to travel from the client to the server and back. The logging system was designed to minimize its impact on overall system performance while still providing comprehensive data for analysis. The collected metrics are stored in a structured format, allowing for efficient post-processing.

Game Objects (Game State). Each object in the game consists of a transform and relevant information. In the case of blocks within the world, they possess a network object (the nomenclature for which may vary slightly across different networking libraries). The role of the network object is to inform the network manager that this object requires synchronization across all connected users.

4.2 Implementation of Benchmark Workloads

Table 1: Values of the workload Parameters used in the benchmark. The workloads are Resource load (RL), Environment Modification (EM), RPC Impact (RI).

Workload	Parameter	Value
RL	World size (WS)	Small (S) (7.5k Game objects) Medium (M) (15k Game objects) Large (L) (40k Game objects)
EM	Tick interval (TI) Block Amount (BA) Tick interval change (TIC)	Float Integer Float
RI	Tick Interval (TI)	Float

Implementation of the benchmark controller was done in Python. We chose Python as it is good for bench-marking external programs because it offers powerful libraries like sub-process that allow the trivial execution of external commands. We implemented the workloads locally inside the server and client scripts. The controller indicates what experiments to run and with what parameters, thus passed to it through arguments passed at the start of the application. These allow the controller to easily run different benchmarks with a variety of parameters.

Workload: Resource load. The resource load benchmark was a straightforward addition, as it merely required the ability to set the world size before starting the experiment. To achieve this, the world generation script reads the command-line arguments, checks for the relevant flags, and retrieves the specified world size. These values then modify the parameters used to generate the world, resulting in the desired size. The world sizes were preset to avoid exceeding Unity's maximum object count, which is crucial, as it is difficult to determine when this limit is reached during bench-marking. The names and object counts of the worlds are shown in Table 1

Workload: Environment Modification. The environment modification experiment was implemented similarly to the resource load benchmark in the world generation script, but instead of modifying the world only at the start, it runs throughout the experiment. The script checks for the relevant flag and reads the arguments following it.

The Environment Modification experiment uses three parameters: The start tick duration, which sets the initial time interval between each step of the script; the number of objects to be placed per tick; and the change in tick rate, which decreases the tick duration with each step, causing the block generation to accelerate over time. These parameters make it simple to modify how the experiment runs through the controller, allowing for various setups of this experiment.

At each update of the system, the script checks if enough time has elapsed. If so, a set of blocks is placed above the player in a grid, resulting in a large object being generated above the world as the experiment progresses. After placing the blocks, the script reduces the tick duration by the change in tick rate and then waits for the updated tick duration before placing another set of blocks. The parameters this benchmark takes are described in Table 1

Workload: Client Environmental Load. The Client Environmental load experiment(also referred to as RPC Impact) is implemented within the player controller script, which handles player-related logic. Similar to the other two experiments, it takes parameters from arguments passed through the command line, looks for the correct flags, and reads the values provided. The client environmental load experiment, requires x and y values to represent where the player will place blocks, as well as a tick duration number, that sets the interval between ticks of the experiment.

This experiment works by tracking whether the player's block is placed or not, and then toggling its placement each tick to simulate player interaction with the environment. The tick duration determines the time between each tick. To place a block, the client sends an RPC that creates a new block at the specified coordinates and spawns it in the environment. To remove a block, the client sends a raycast above the block, pointing downwards to locate the block's object, and then removes the block.

This setup allows for the simulation of player interactions with the environment at regular intervals, providing a controlled way to test the performance and behavior of the RPC system under varying conditions. The parameters this benchmark takes are described in table 1

Additional benchmark tools in open craft. To ensure minimal graphics rendering load and compatibility with systems lacking a graphical user interface (GUI), the benchmark executes experiments with graphics disabled. This functionality was inherited from the original Open Craft 2 and is activated by passing the arguments -batchmode -nographics.

To manage the benchmark's termination after experiments are completed or after sufficient data is collected, a "close after" function is implemented within the network helper. It waits until the specified time is reached and then gracefully closes the application.

4.3 Recoding of System Metrics

To enhance the reliability and comprehensiveness of metric measurements important to the benchmark, we developed an additional system metric recorder. This script utilizes Psutil (Python System and Process Utilities) to monitor relevant

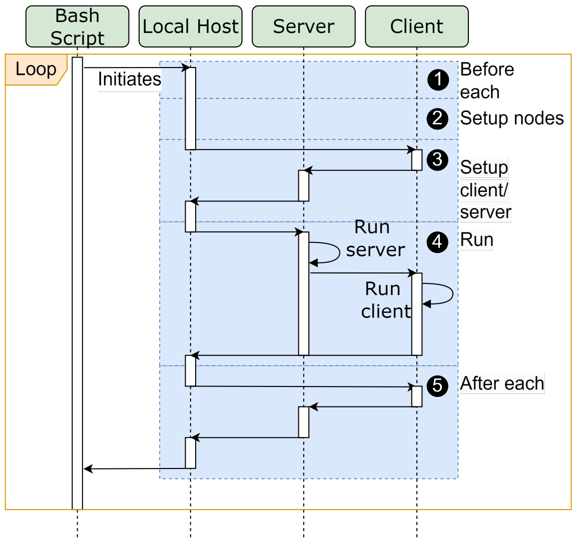


Figure 6: Ansible playbook time line.

metrics of the Opencraft process, providing an external perspective on system performance. By focusing on the Opencraft process, we can isolate its resource utilization from other system activities, offering a more accurate representation of the application's performance. This approach also allows us to observe the communication between the server and client components. Consequently, we added bandwidth measurement to this script, giving us important information about network usage. This is crucial for understanding how efficiently data moves in networked systems.

4.4 Implementation of Deployment system

To execute the benchmark in a distributed manner, we developed a system utilizing a combination of Python scripts and Ansible playbooks. Ansible playbooks, which are YAML files defining a set of tasks and configurations for remote host execution, facilitate the automation of the benchmarking process. The structure operates as follows: The benchmark is initiated by invoking a bash script, which first employs mini Anaconda to ensure the requisite environment is configured for the Python scripts in use. The bash script then proceeds to execute the Ansible playbooks, passing relevant benchmark settings. These settings are contained within YAML files that encompass instructions for each benchmark, including parameters such as player count and command-line arguments for both server and client sides.

Upon initiation, the Ansible playbooks execute the specified instructions. Figure 6 illustrates the sequence of instructions and their respective execution locations.

The process begins with the "Before Each" YAML file (❶), which ensures the existence of the output directory and establishes a series of variables pertinent to the entire experiment.

Table 2: Experiment overview (Abbreviations are defined in Table 1)

Section	Work-load	Focus	Metric recorded	Parameter values	Player count	Dura-tion
MF1	RL	Server performance	CPU Load & Memory consumption	WS = S, M, L	1	70s
	RI			TR = 0.2, WS = M	1-16	160s
MF2	RI	Server bandwidth	Bandwidth out	TR = 0.2, WS = M	1-16	160s
MF3	RI	Client RTT under server load	RTT	TR = 0.2, WS = M	1-16	160s
MF4	EL	Client network performance under client load	RTT, Bandwidth in	WS = S, TI = 0.4, BA = 300, TIC = 0.0040	1	220s

Subsequently, "Setup Nodes" (❷) is invoked to reserve nodes on the server and await their availability for use. Following this, the server and client are configured (❸), which involves creating temporary directories for log storage and transferring a copy of the Opencraft build to these directories.

Once the environment is prepared, the "Run" YAML file (❹) is executed. Its function is to initiate Opencraft and the system logging Python script. It first launches Opencraft on the server side, identifies its process ID (PID), and passes this information to the system logger. This process is then replicated on the client node, with the number of Opencraft instances launched corresponding to the specified player count. To mitigate system load, only one client instance incorporates system logging.

The playbook then enters a wait state until the experiment is completed. Upon completion, the "After Each" script (❺) performs environment cleanup. This involves transferring relevant logs to the output folder and removing temporary directories. Finally, it releases the node reservations and concludes the process.

This sequence is iterated for each experiment until all are completed. Subsequently, a data processing script is invoked to perform initial preprocessing of the data and relabel all logs to facilitate data visualization. These processed files are then transferred to the results folder.

5 EVALUATION

This section presents the findings from our real-world experiments using the benchmark to evaluate the performance of NFGO, Mirror, and Fishnet networking libraries. The goal is to address contribution C3 by providing an analysis of the results obtained from our experiments. Table 2 presents the experiment overview, outlining the experiments and workload pertaining to each main finding. Below the main findings of the experiments are described:

MF1 NFGO shows the best system performance under the workloads tested. NFGO demonstrates significantly lower CPU and memory usage compared to Mirror and Fishnet, both in the baseline scenario and under heavy environment modification stress.

MF2 Mirror uses substantially more bandwidth for environment modification. The bandwidth consumption analysis reveals that Mirror exhibits a much steeper increase in bandwidth usage as the number of players grows, along with greater fluctuations, compared to the more efficient Fishnet and NFGO libraries. This suggests that Mirror is less bandwidth-efficient, especially in scenarios with a large number of players.

MF3 Mirror's RTT suffers under heavy environment modification workloads. The client-side Round-Trip Time (RTT) analysis shows that NFGO offers the most consistent and reliable performance as the number of players increases, while Mirror's RTT exhibits rapidly increasing and highly variable behaviour.

MF4 Under write-heavy workloads, Fishnet trades off lower bandwidth usage for increased round-trip time. Fishnet's round-trip time reaches 140 ms, whereas NFGO and Mirror do not exceed 20 ms. However, Fishnet's bandwidth usage does not exceed 15 MiB/s, 4 and 7 times less than NFGO and Mirror, respectively.

5.1 Experimental Setup

In this section, we present the experimental setup used throughout our study. Using a benchmark, we assess and compare the performance of three networking library prototypes in each experiment, running independently in the same environment. The final benchmark with results discussed in this paper can be found on Github.

System Under Test. As mentioned, the systems under test in the benchmark will be three networking library prototypes. As discussed in the implementation section, they will consist of NFGO, Mirror, and Fishnet networking libraries. Our experiment will utilize these libraries, each implemented in their own Opencraft 2 prototype.

Experiment Environment. All our experiments use the DAS-5 multi-cluster system [1]. To run the experiments, we utilized three nodes:

One node is in charge of controlling the system and setting up each experiment. It creates the necessary temporary folders, moves the logs after each experiment, and performs final processing of the logs afterwards.

Another node is designated as the server node and hosts the server for each respective experiment. This node also

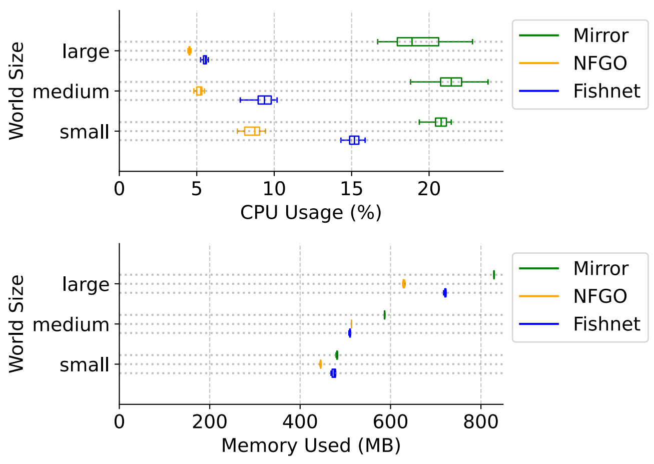


Figure 7: CPU Usage and Memory usage of Resource load workload on the server.(Vertical axis shows the world size. Box plot represents the quartiles)

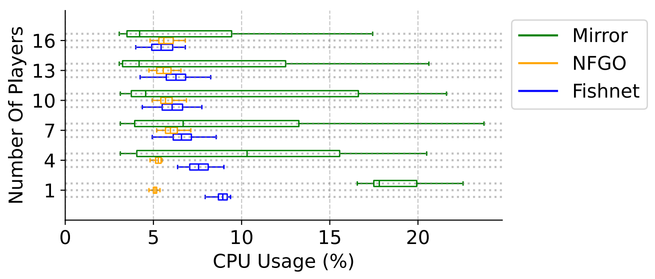
runs its own system performance script to monitor the server process.

The last node hosts the Opencraft clients. Depending on the experiment, multiple Opencraft instances will run on the node. Like the server, it also runs a script that monitors the performance of the Opencraft process.

5.2 MF1: NFGO demonstrates the best overall performance under resource load and client environmental load workloads

Looking at the results of the Resource Load experiment, we can observe substantial differences in the base system resource consumption of the tested networking libraries. Examining the data presented in Figure 7, we can see that NFGO demonstrates the best overall performance, utilizing significantly less CPU power and memory compared to the other two libraries, Mirror and Fishnet. This trend is further suported in Figure 8, which shows the resource load under environment modification-induced stress. Even under this increased load, NFGO continues to exhibit the lowest resource consumption. Unlike Mirror, which displays fluctuating resource utilization, NFGO maintains a more consistent load profile.

The consistently lower CPU and memory usage of NFGO, both in the baseline scenario and under environment modification-induced load, suggests that this library has a more efficient architecture and resource management strategy compared to its counterparts. This efficiency could translate to improved scalability, reduced server costs, and better overall system performance, particularly in resource-constrained environments or scenarios with high player counts.



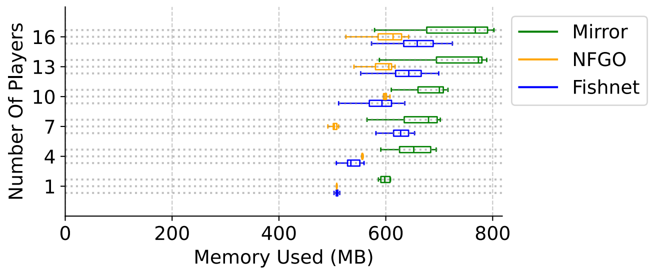


Figure 8: CPU Usage and Memory usage of RPC impact workload on the server.(Vertical axis shows the amount of players connected to the server. Box plot represents the quartiles)

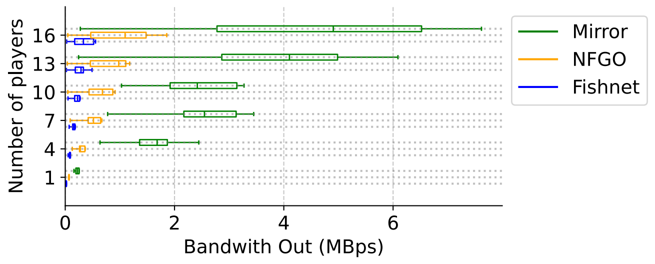


Figure 9: Outbound bandwidth usage of RPC Impact workload.(Vertical axis shows the amount of players connected to the server)

5.3 MF2: Mirror uses substantially more bandwidth for RPCs

When examining the results from the RPC experiment, as depicted in Figure 9, we can observe distinct trends in the bandwidth consumption of the tested networking libraries as the number of players increases.

Mirror exhibits a significantly more pronounced increase in bandwidth usage compared to the other two libraries, Fishnet and NFGO. As the player count grows, Mirror's bandwidth requirements escalate at a much steeper rate than its counterparts.

Furthermore, Mirror also demonstrates a larger degree of fluctuation in its bandwidth utilization. This inconsistent bandwidth profile could introduce challenges in resource

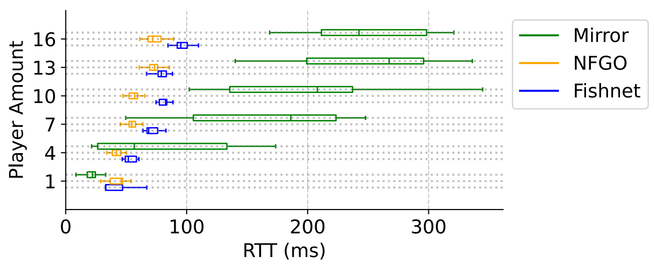


Figure 10: Client RTT of RPC Impact workload. (The vertical axis shows the amount of players connected to the server. A good RTT is between 40-50 ms, a tolerable RTT is between 50-100 ms and the RTT becomes problematic when it reaches above 100 ms [13].)

provisioning and potentially lead to suboptimal network performance under varying load conditions.

In contrast, the best performer in this benchmark is Fishnet, which consistently exhibits the lowest bandwidth usage among the three libraries. NFGO, while not the absolute lowest, also maintains a relatively modest bandwidth footprint, only slightly more intensive than Fishnet.

The large differences in bandwidth consumption patterns between the libraries suggest that Fishnet and NFGO have more efficient Netcode and data transmission strategies compared to Mirror. This efficiency could translate to improved scalability, reduced bandwidth costs, and more stable network performance, particularly in scenarios with high player counts or variable load conditions.

5.4 MF3: Mirror RTT suffers under RPC workloads

When examining the results from the RPC experiment on the client-side, we can observe distinct trends in the Round-Trip Time (RTT) of the tested networking libraries as the number of players increases, as depicted in Figure 10.

As observed in the previous sections, Mirror appears to perform quite poorly, with its RTT increasing rapidly as the player count grows. This trend is concerning, as higher and more variable RTT can negatively impact the user experience, particularly in real-time applications.

In contrast, both NFGO and Fishnet exhibit a more consistent RTT profile. The fluctuations in RTT are significantly lower for these two libraries compared to Mirror, which demonstrates a very high degree of variability in its response times.

In this specific experiment, NFGO emerges as the best performer, consistently outpacing Fishnet in terms of client-side RTT. This suggests that NFGO's networking architecture and implementation are more optimized for handling the

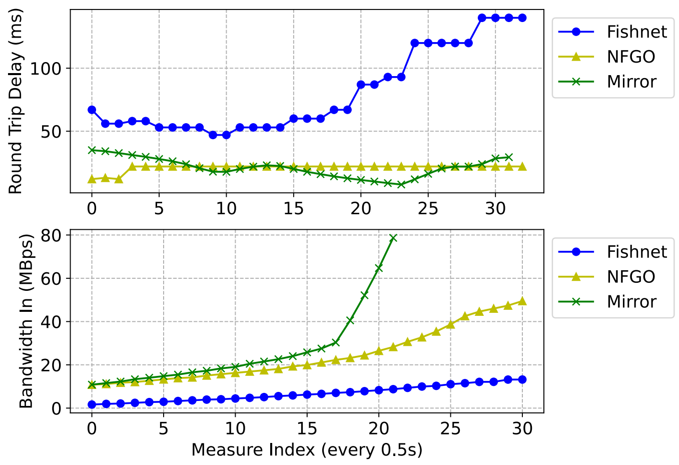


Figure 11: Client RTT(ms) and inbound bandwidth usage(MBps) of Environment Modification workload (Horizontal Axis represents the Measurement index where each measurement is 0.5s apart)

increased load and maintaining low, stable latency for the clients.

However, it is worth noting that when only a single player was connected, Mirror actually outperforms both NFGO and Fishnet. This observation indicates that Mirror's networking capabilities may be more efficient in low-load situations, but its performance degrades more significantly as the player count increases.

The client-side RTT results demonstrate that NFGO offers the most consistent and reliable performance as the system scales, making it a potentially better choice for applications that require predictable and low-latency network communication. Addtionaly, the high variability of RTT observed in Mirror's performance will significantly reduce the quality of service provided to client

5.5 MF4: Fishnet struggles with large amount of block updates

Finally, when examining the results of the Environment Modification benchmark, we can observe interesting trends in the Round-Trip Time (RTT) compared to the RPC benchmark, as shown in Figure 11. Unlike the previous observations, in this experiment, Fishnet appears to struggle the most with RTT, reaching a high of approximately 140ms, while NFGO and Mirror experience RTT in the 20ms range.

This finding suggests that while Fishnet may handle multiple requests well, a large amount of outgoing information can cause it to suffer significantly earlier than the other two libraries, NFGO and Mirror.

In contrast, the other two libraries, NFGO and Mirror, demonstrate much more promising performance, with both exhibiting very consistent RTTs that are minimally affected by the increasing block counts. This consistency is a desirable trait, as it ensures a stable and predictable user experience, even as the complexity of the environment increases.

An additional observation worth noting is the fact that, as observed earlier, Mirror's bandwidth usage struggles significantly. While its bandwidth consumption is comparable to the other libraries in the initial stages of the experiment, it grows considerably in the later stages, far exceeding the requirements of NFGO and Fishnet. Fishnet performs the best here with its bandwidth usage not exceeding 15 MBps, 4 and 7 times less than NFGO and Mirror, respectively.

This divergence in bandwidth usage, coupled with Fishnet's higher RTT under the Environment Modification workload, suggests that NFGO may be the most well-rounded performer, offering both consistent low-latency communication and efficient bandwidth utilization. This combination of attributes could make NFGO a more suitable choice for applications that require reliable and scalable networking capabilities, particularly in complex and dynamic environments.

6 RELATED WORK

Closest to our work are the benchmarks of Yardstick and Meterstick [17][3]. Yardstick, an MLG benchmark, demonstrates the limited scalability of MLGs. The authors employ Yardstick to evaluate the scalability of various MLG services. However, they do not extensively explore the network side of MLGs; instead, their focus lies primarily on the performance impact on systems running mlg environments and focus solely on player-based workloads. Additionally, Meterstick focuses on analyzing the performance variability of MLGs. While it explores this aspect concerning environmental workloads and player workloads similar to Yardstick, Meterstick does not investigate the network aspects of MLGs. While there is some research done on networking libraries, most focuses on the feature set of the libraries. For example, there exists a useful spreadsheet created by Unity community members to compare the feature sets of networking libraries[9]. However, there are only a few limited benchmarks and they focus on extreme conditions, which are not representative of actual workloads and lack scientific rigor. An example of this is Netick, a benchmark that tests a library's ability to handle poor network conditions[10].

7 CONCLUSION

Online multiplayer video games constitute one of the largest entertainment sectors and continue to grow rapidly. With many developers seeking to outsource their netcode, the abundance of options makes it challenging for them to select the best networking library for their applications. In this work, we present a tool to analyze the performance of these libraries concerning virtual environment workloads.

We make a threefold contribution to better understand the behavior of networking libraries in relation to MVEs. First, we propose a novel design for this benchmark and outline how it aims to evaluate the performance of networking libraries. Second, we implement this benchmark in OpenCraft 2 to assess the performance of various networking libraries concerning MVEs. Lastly, we use this benchmark to analyze the performance of a subset of networking libraries.

Our findings reveal differences in the performance characteristics of the networking libraries analyzed. Some libraries exhibit strengths in different areas; for instance, NFGO performs well for large MVEs with fewer players, while Fishnet shows the best performance for larger player counts. Interestingly, we also find that some of the most popular libraries, such as Mirror, while widely used, demonstrate the worst performance.

In future work, we aim to analyze the quality of service of these networking libraries concerning client experience, as this is an area not adequately covered by this paper.

REFERENCES

[1] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 5 (2016), 54–63. https://doi.org/10.1109/MC.2016.127

[2] Jerrit Eickhoff. 2024. *Polka A Differentiated Deployment System for Online and Streamed Games, Meta-verses, and Modifiable Virtual Environments.* Master's thesis. Delft university of technology.

[3] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. 2023. Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games. *ICPE* (2023).

[4] FirstGearGames. 2024. Fish-Net: Networking Evolved - Introduction. https://fish-networking.gitbook.io/docs.

[5] Raj Jain. 1991. techniques for experimental design, measurement, simulation, and modeling.

[6] Mirror. 2024. Mirror Networking. https://mirror-networking.gitbook.io/docs.

[7] newzoo. 2024. Most popular PC games by monthly active users. https://www.theverge.com/2023/10/15/23916349/minecraft-mojang-sold-300-million-copies-live-2023.

[8] Ash Parrish. 2023. Minecraft has sold over 300 million copies. https://www.theverge.com/2023/10/15/23916349/minecraft-mojang-sold-300-million-copies-live-2023.

[9] Punfish. 2024. Free networking solution comparison chart. https://discussions.unity.com/t/updated-free-networking-solution-comparison-chart/899755.

[10] Punfish. 2024. Netick 2 - unity-network-library-benchmark-on-bad-network-condition. https://github.com/StinkySteak/unity-network-library-benchmark-on-bad-network-condition.

[11] Felix Richter. 2021. Infographic: Gaming: The Most Lucrative Entertainment Industry By Far. https://www.statista.com/chart/22392/global-revenue-of˘selected-entertainment-industry-sectors.

[12] Elena Stroiu. 2024. *Net-Celerity: A Benchmark for Player Activity Analysis of Gaming Network Libraries.* Bachelor's Thesis. Vrije Universiteit Amsterdam.

[13] Bandwidth Place Team. 2023. A Guide to Ping and Latency in Gaming. https://www.bandwidthplace.com/article/ping-latency-in-gaming.

[14] Unity Technologies. 2024. About Netcode for GameObjects. https://docs-multiplayer.unity3d.com/netcode/current/about/.

[15] Unity Technologies. 2024. Sending events with RPCs. https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/messaging-system/.

[16] telstra. 2024. New survey outlines network infrastructure challenges and priorities for video game companies. https://www.telstra.us.com/en/news-research/articles/network-infrastructure-challenges-and-priorities-for-video-game-companies.

[17] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. 2019. Yardstick: A Benchmark for Minecraft-like Services. *ICPE. ACM, 243–253* (2019).

[18] Reinhold Weicker. 2002. Benchmarking. In Performance Evaluation of ComplexSystems: Techniques and Tools, Performance.

[19] Zenva. 2023. What is Unity? – A Top Game Engine for Video Games. https://gamedevacademy.org/what-is-unity/.

A ARTIFACTS

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

- **Program: Unity**
- **Compilation: C#**
- **Run-time environment: Linux**
- **Hardware: DAS 5**
- **Metrics: CPU Usage, Memory consumption, Round Trip Time, Bandwidth usage**
- **Output: Graphs and files, example results included**
- **How much disk space required (approximately)?: 10GB(due to size of prototypes)**
- **How much time is needed to prepare workflow (approximately)?: 30min + time taken take to compile unity prototypes(could be 15min per prtotype)**
- **How much time is needed to complete experiments (approximately)?: approx 3h**
- **Publicly available?: yes**

A.2 Description

How to Access. The code can be found on GitHub: Benchmark and OpenCraft 2 Prototypes.

Hardware Dependencies. The software should be run on a server that allows for access to separate nodes with Prun.

Software Dependencies. Required software includes Ansible playbooks and Unity to compile.

A.3 Installation

Opencraft 2. To create the builds, open the projects with Unity and build them for Linux. The builds are on different branches, so this process needs to be repeated for each build. Unity and the required build tools should be easy to install, with plenty of information available online to assist.

Benchmark. Clone the repository on the server and place the builds in your local scratch folder. You can customize the location, but you would need to modify the 'build_location' in the 'experiment.yml' file.

A.4 Experiment Workflow

Once the setup is complete, you can run experiments by first ensuring the output folder is empty and then running the './run.sh' bash script with the desired folder name as an argument:

```
./run.sh Results-Date-7-7-24
```

Results will be stored in that folder. Additionally, to visualize results, first modify the input folder to your new data and then run the 'Data_Visualization.ipynb' notebook, which outputs the important graphs in the 'Graphs' folder. It also contains a large number of other graphs, but these were not used in the paper, so they are not automatically saved.

A.5 Evaluation and Expected Results

To reproduce the results as seen in this paper, run the benchmark with the values described in the implementation section. Results discussed in the evaluation of the paper are in the 'Results-Das-18-7-24' folder.

A.6 Experiment Customization

To modify experiment values, edit their respective '.yml' files. More information about the parameters can be found in the paper.

A.7 Methodology

Submission, reviewing and badging methodology:

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

…