Vrije Universiteit Amsterdam

Bachelor's Thesis

Net-Celerity: A Benchmark for Player Activity Analysis of Gaming Network Libraries

Author: Elena Stroiu (2731065)

1st supervisor: ir. Jesse Donkervliet
2nd reader: prof.dr.ir. Alexandru Iosup

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

August 14, 2024

Abstract

The gaming industry is one of the largest segments of the modern entertainment industry, with multiplayer games on high demand (1). Multiplayer online games can make use of readily available networking solutions that promise seamless gameplay experiences for large player bases. However, it is not always clear how different networking libraries compare to each other and which one will be more appropriate for a specific use case. This study creates a Net-Celerity benchmark to evaluate the performance of networking libraries, specifically for Unity games, within prototypes simulating multiplayer game scenarios on DAS-6, which was used to compare three networking libraries: Netcode for Entities, Mirror KCP, and Mirror Telepathy. Key findings reveal significant differences in performance metrics such as round-trip time (RTT), server-side resource consumption, and network traffic patterns. DOTS-NFE demonstrates higher RTT performance with a maximum observed decrease of 2.3x under increasing player loads compared to Mirror KCP and Mirror Telepathy. It is not affected by different player behavior, while M-KCP and M-TP have experienced a decrease of 2-4ms of RTT when player activity was introduced. However, DOTS-NFE also exhibits higher server resource demands and requires up to double the CPU and 18 times more memory at peak loads. In contrast, both Mirror KCP and Mirror Telepathy maintain stable resource usage across varying player counts, with slight increases observed as player numbers rise. Network traffic analysis indicates that DOTS-NFE has predictable data transmission rates compared to Mirror implementations, which can be caused by its entity-state synchronization and client prediction model, but requires further investigation for confirmation. These findings put emphasis on the trade-offs between performance and resource efficiency among networking libraries designed for multiplayer games.

Contents

1

Introduction

In this section we discuss the relevance of the chosen topic - benchmarking networking libraries, which problems developers face when using networking libraries, main research questions, methodologies used to tackle the said questions, and contributions the research makes.

1.1 Multiplayer Games in the Modern Era

The gaming industry is the largest part of the modern entertainment industry with more than 3 billion players and $184 billion in revenue for 2023 (1). Apart from entertainment, online gaming platforms, has proven effective in various domains such as education (2). This work focuses on multiplayer games, a prevalent and highly interactive category, with popular examples including Minecraft, Apex Legends, Fortnite, and Dota 2.

1.2 Problem Statement

Despite the wide of multiplayer feature in the field of digital entertainment and education, developers still face issues when choosing a networking library for their project:

1. **Performance ambiguity.** Developing multiplayer features involves distinct requirements and constraints depending on the genre of the game (3). Large virtual worlds or massively multiplayer online games (MMO) prioritize scalability and minimal overhead due to the large amount of data processed (4). In contrast, competitive games such as Counter Strike 2 and Valorant require a latency of less than 75 milliseconds (5). Developers often utilize existing netcode frameworks to mitigate the

labor-intensive nature of netcode development. However, integrating a network library can be risky if performance does not meet expectations, thus understanding and fair assessments of its properties, such as scalability, reliability, packet overhead, and syncing capabilities, is crucial.

2. **Benchmark implementation availability.** Several benchmarks, such as Yardstick (6) and Meterstick (7), evaluate Minecraft-like services and games, respectively. However, no public benchmarks that could be used by developers and researchers for networking libraries exist using similar principles. Furthermore, this gap highlights the need to evaluate the performance and scalability of these libraries. Yardstick captures system- and application-level metrics, while Meterstick focuses on performance variability through specialized workloads and metrics. This work introduces Net-Celerity, a benchmarking tool designed to address these needs by evaluating the performance and scalability of networking libraries.

3. **Workload availability.** In order to effectively analyze the performance of networking libraries realistic workloads must be utilized. Furthermore, in order to compare several networking libraries the workloads must be the same for objective conclusions. However, we discovered that majority of networking libraries do not provide any sort of workloads or benchmarking tools, and if they do, they highly differ from each other, making the comparison unfair.

Despite the widespread use of networking libraries, their capabilities and limitations remain inadequately measured and compared. This study systematically analyzes the performance of networking libraries in multiplayer games developed using Unity game engine, filling this critical gap. Our work aligns with our vision on Massivizing Computer Systems (8, P9, §6.3), informing the community about the current ecosystem and helping smaller companies reduce costs for the development of online games.

1.3 Research Questions and Methodology

The aim of this work is to provide a scientific instrument, Net-Celerity, that can help to fairly evaluate networking libraries. We ask three main research questions:

RQ1 *How do different network libraries perform compared to each other when benchmarked against new workload types within various game prototypes?* This question addresses requirement 1 and 3 by providing developers with overview of how different benchmarks handle different workloads and making it clear which ones will be more suitable for their specific needs.

RQ2 *What are the key performance metrics that effectively quantify the suitability of network libraries for different game genres and workload intensities?* In this question we address requirement 1 and 2, by asking ourselves which metrics Net-Celerity needs to utilize in order to provide insight to the performance of networking libraries.

RQ3 *How do widely used network libraries perform in real-world scenarios setting?* The third question extends on requirement 1 and focuses on evaluation of individual capabilities of networking libraries on a real world scenarios.

To satisfy the questions indicated above we developed Net-Celerity benchmark. We aim to provide a framework for research and practical usage in the evaluation of prototype-based networking libraries. In Section 5.2 we discuss the choices for the experiment setup and further evaluate the results in Section 5.1.

1.4 Thesis Contributions

Previous research provides insights into networking features for multiplayer online games (MLGs) and services (MLSs), but does not adequately address other game types and their netcodes. Addressing this gap, we make three contributions:

C1 We propose a benchmark design for game network libraries (Section 3). The benchmark fairly assess multiple network libraries with different workload types which are further discussed in Section 3.4.

C2 We implement the Net-Celetiry benchmark as illustrated in Section 4.1 which works with game prototype builds.

C3 We use our prototype implementation to conduct a series of real-world experiments, and derive insightful results from the analysis (Section 5).

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 Concepts and Models

In this section, we firstly provide an overview on the state-of-the-art networking libraries. Afterwards, we discuss the fundamental principles behind networking libraries in Unity. Finally, we provide a detailed explanation of the differences for two highly popular networking libraries: Netcode for Entities and Mirror.

2.1 Networking Libraries for Online Games

Networking libraries are used in the development of online games, providing the infrastructure necessary to arrange communication between clients and servers. These libraries consist of a collection of tools and protocols to manage data transfer, synchronization, and player interactions across the network, see Figure 2.1. The main goal and features of networking libraries varies widely as is discussed in Section 2.2 and Section 2.3, from low-latency and high-reliability connections to ease of integration.

Networking libraries act as mediators between the game implementation and the lower-level network protocols, for instance, such as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Since the underlying protocols operate on different principles (9) networking libraries need to take into account the existing costs and benefits and build on top a system that meets the needs of specific types of online games. TCP provides reliable, ordered, and error-checked delivery of data but can introduce latency. On the other hand, UDP does not guarantee the delivery of packets, but also does not wait for acknowledgment is more suitable for real-time applications.

A network library integrated in the game implementation handles various tasks through several components and processes. These include the transport layer or protocol that manages the transmission of data packets over the network, and session management, which

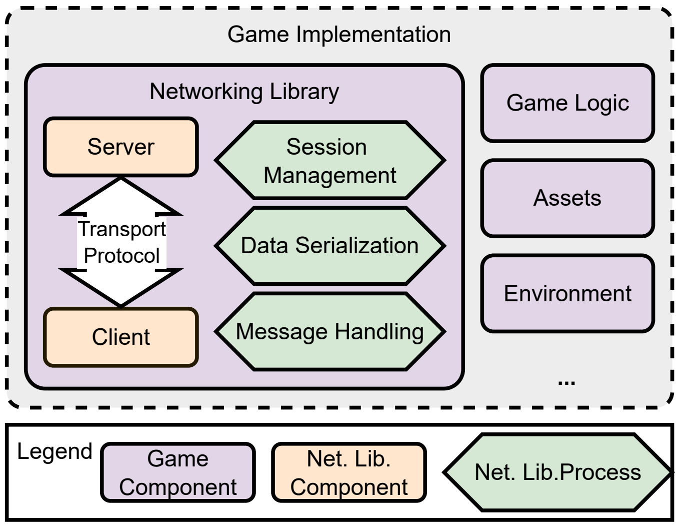


Figure 2.1: Networking Library System Model.

handles connection establishment, maintenance, and termination. Data serialization converts game state data into a format suitable for transmission. Message handling manages sending and receiving messages between clients and servers. The game server hosts the game world and synchronizes game states between clients which can be handled in different approaches depending on the networking library. Consequently, networking library manage majority of networking related mechanisms and provide the developers with necessary tools to integrate multiplayer functionality into their games or applications.

2.2 Unity Netcode for Entities

Unity's Netcode for Entities, which is primarily used in Unity's Data-Oriented Technology Stack (DOTS) architecture, is designed for large-scale multiplayer projects that employ data-oriented design principles (10). This netcode framework is transport-agnostic, allowing it to operate with various transport protocols, including UDP and WebSocket. Netcode for Entities supports both small-scale multiplayer games and large MMO games (11).

Netcode for Entities (NFE) is initially designed to work within the Entity Component System (ECS), as illustrated in Figure 2.2. In multiplayer scenarios, this netcode library handles both server-side and client-side networking tasks by making use of UDP for synchronization and general communication between clients and servers. ECS structures, such

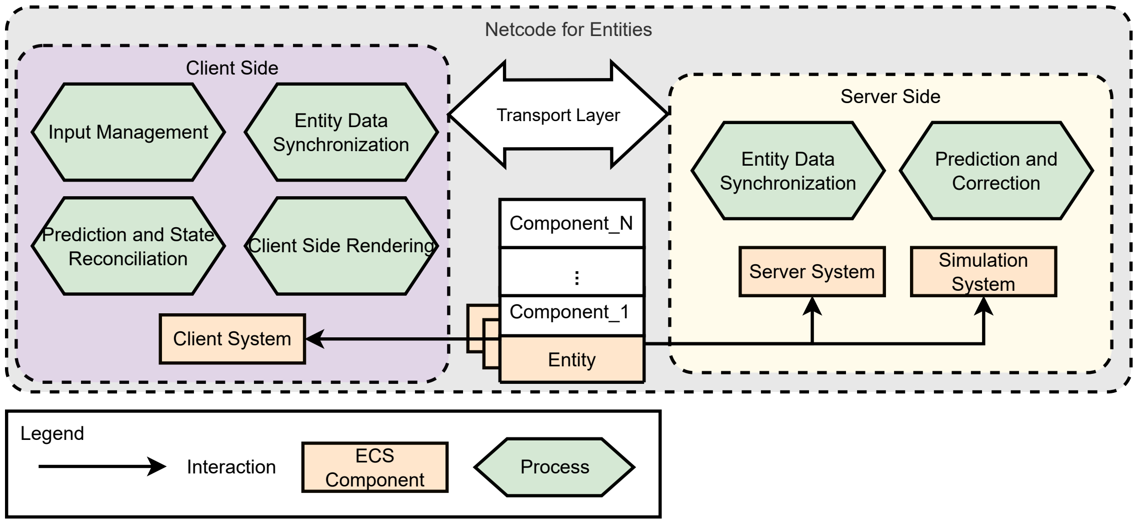


Figure 2.2: Netcode for Entities System Model.

as Entities, Worlds, and others, are serialized and transmitted over the transport layer. Moreover, the netcode provides mechanisms for client- and server-side predictions, data correction, and state reconciliation that reduce the the effects of latency and packet loss and consequently enhancing the player experience.

Synchronization within Netcode for Entities is achieved using Remote Procedure Calls (RPCs) (10). This allows game logic to be mirrored across the network, ensuring that actions performed on one client are correctly displayed on others. By default, the netcode synchronizes transform data, such as position, rotation, and scale, for Entities. However, developers can extend the synchronized properties list as necessary. Furthermore, Netcode for Entiteis uses custom serialization methods that optimize data preparation for network transmission, which further can be studied in the official documentation (11).

2.3 Mirror Networking

Mirror (12) is an open source game networking library for Unity that supports Unity versions from 2019 to 2022 LTS. It has been used in games like SCP: Secret Laboratory, Portals, and Inferna. One of Mirror's standout features is its integration with the KCP library, providing a reliable UDP transport layer (13). Additionally, it supports TCP through Telepathy (14) and WebSockets via Simple Web Transport (15). Originally developed from the UNet framework, Mirror has addressed many of UNet's issues and gained

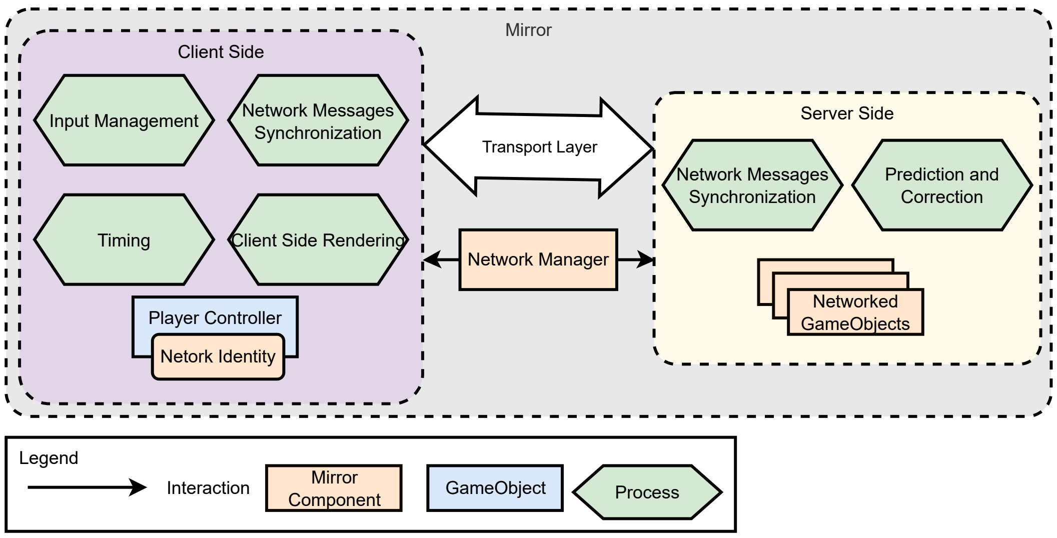


Figure 2.3: Mirror System Model.

popularity, with over 100,000 downloads annually (16).

Mirror is designed for ease of use and integration into existing single-player games (12). It is built on top of the LLAPI (Low-Level API) (17) provided by Unity and allows direct control over network functions and data serialization solutions. The functionality of Mirror is similar to Netcode for entities with some differences in architecture that are illustrated in Figure 2.3. Mirror makes use of GameObjects instead of Entities and has a Network Manager that orchestrates all the networking logic.

For the transport layer, Mirror offers several options, including reliable UDP through KCP (13), TCP via Telepathy, and WebSocket transport through Simple Web Transport. In addition, it supports LiteNetLib (18), providing developers with the flexibility to choose the most appropriate transport protocol for their needs. These transport components are configurable within the Network Manager, which also manages GameObjects with Network Identity components.

One of the significant advantages of Mirror is its extensibility. Developers can customize and enhance its networking capabilities using callbacks and events, enabling the implementation of advanced features such as custom packet handling, network optimizations, and latency compensation techniques. Compared to Unity's Netcode for Entities, Mirror offers more choices in transport layer. While Netcode for Entities typically uses UDP to minimize latency, Mirror can make use of TCP through Telepathy or KCP for reliable transmission and combine low latency with reliability. WebSockets is particularly useful for browser-based connections over HTTP.

Consequently, Mirror is a user-friendly option with flexible networking solution for Unity developers. Its integration with various transport protocols, ease of use, and extensibility make it a popular choice for less experienced developers and smaller multiplayer games.

3

Design of Net-Celerity

In this section, the design of the benchmark that uses Minecraft-like games for testing is established. We define a set of requirements that the benchmark has to follow, as well as general design.

3.1 Requirements

R1 *Ease of use:* the benchmark should have approachable and understandable design so that obtaining results would not cause user confusion. The structure of the components must be consistent and easy to navigate for the developers, without convoluted design choices. The simplicity of Net-Celerity makes it accessible to a broader range of users with varying levels of expertise. The primary challenge of this requirement is that ensuring ease of use is not a straightforward process and might require balancing simplicity with functionality. Consequently, a well-organized and consistent structure for the benchmark components, providing clear documentation, and avoiding convoluted design choices are necessary.

R2 *Fairness:* the benchmark should assess the performance of gaming network libraries fairly under the same conditions. Environment and the setup of the benchmark that is going to test provided prototypes must be identical in order to eliminate the possible outside factors. This requirement is important for making objective evaluations and avoiding biased outcomes. However, ensuring fairness requires in-depth understating and control of the benchmarking environment.

R3 *Clarity:* the chosen metrics must be displayed in clear and understandable manner without ambiguity which includes being straightforward and easily comprehensible.

Additionally, visualization scripts will be provided for performance implications of each metric. It is crucial that metrics are presented clearly and without ambiguity to ensure users understand the performance implications of the networking library. Furthermore, it enables developers to make well-informed choices based on the provided results. Choosing clear and understandable metrics involves careful consideration of how data is displayed. Avoiding ambiguity requires detailed explanation and development of visualization tools that can effectively represent performance data.

R4 *Relevance:* the chosen metrics must be indicative of the performance of the networking library and provide clear overview of relevant values. They should provide users with valuable insights about prototype's performance and load on the system. Relevance is essential for offering a comprehensive overview of how well the networking library performs, helping users in understanding the system's behavior across different scenarios. The main challenge is identifying and selecting metrics that are reflective of performance requires deep understanding of both the technical aspects of networking libraries and the practical needs of developers.

R5 *Realistic prototype:* the game prototype in use must be close to a real world example of the multiplayer game and have similar workloads for testing purposes. The game world must be configurable to be able to fit large amount of player, and player behavior must include actions similar to Minecraft-like games including first-person movement and the ability to interact with the environment by putting or removing blocks. The workload must be representative of the demands and challenges typically encountered in real-world multiplayer scenarios. Developers would benefit from realistic prototype making the results applicable. However creating a realistic prototype involves simulating complex behaviors and workloads similar to those found in real multiplayer games, requiring considerable time and effort.

R6 *Configurability:* the benchmark must configurable by the user, so that different loads that are relevant to different scenarios could be tested. Furthermore, new prototypes can be added and tested in the same manner as the provided prototypes. This will enable users to choose which scenarios to test the prototypes against and suit their specific needs. Adding configurability to Net-Celerity might conflict with previously indicated requirement: **R1** ease of use. Keeping the balance of configurability and simplicity requires additional effort and evaluation of several design approaches.

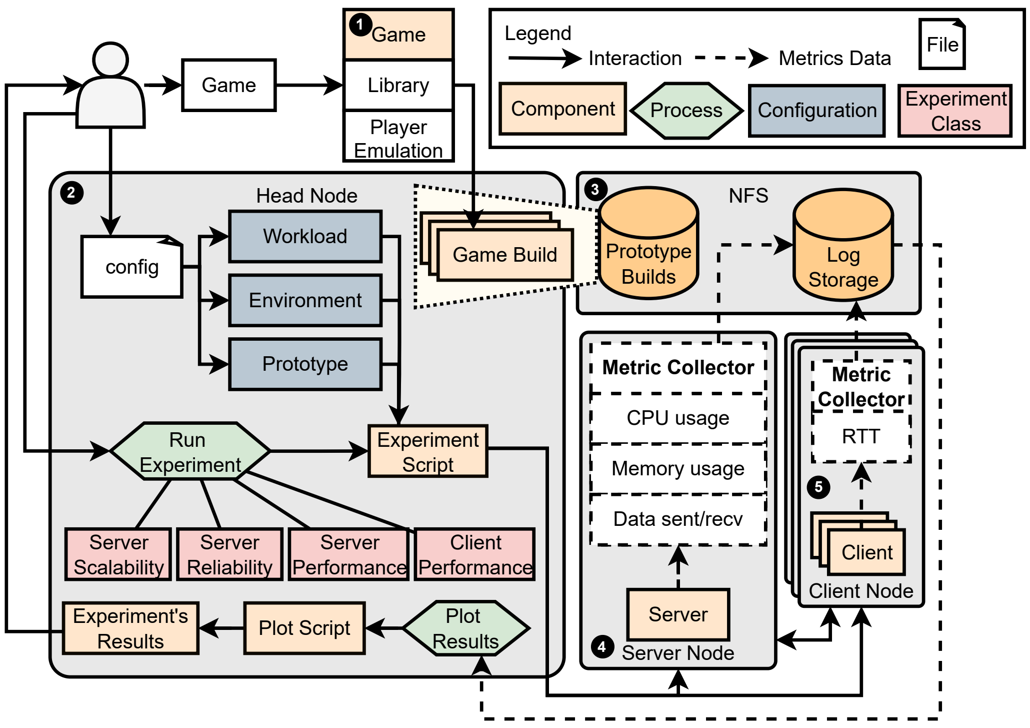


Figure 3.1: Benchmark Design Overview.

3.2 High-Level Design Overview

The overview is illustrated in Figure 3.1 where the execution logic of the experiment, the components, and the processes are clarified. The design consists of several major components: game prototypes ❶, benchmark configurations and execution scripts that are located at the head node ❷, Network File System (NFS) ❸ that stores game prototypes and logs, and finally the metric collector scripts located in the server node ❹ and the client node ❺.

The game prototypes ❶ are created by the user using the desired networking libraries. The prototype must include basic player emulation in order for the Net-Celebrity to evaluate player activity. Each prototype build is stored in the NFS ❸ that can be accessed by other nodes for execution and metric collection. We chose the benchmark to utilize game builds that are meant to be playable, contributing to **R6** configurability and **R5** realistic prototype, which would not be possible if we chose to evaluate libraries within the unity editor instead.

The head node ❷ is responsible for the setup and running of the experiments, as well as providing the user with the results and relevant visualization. To satisfy **R6** configurability, we decided to make the experiment setup modifiable specifically workload, the selected prototype, and the environment in which the experiment will run. There are also several major classes of experiments that developers can benefit from: server scalability, server reliability, server performance, client performance. Server scalability and reliability is tested by increase the number of players and can be evaluated by evaluated introduced metrics collected from the server node ❹ and by checking how many players ended up spawning in the results collected from all the client logs. Server performance can be measured by conducting time-based experiments and evaluated with CPU usage, memory usage, and data transfer/receive rates. Finally, client performance can be evaluated individually or on average from the combined results log with the RTT metric. After the experiment is complete and the metrics data are retrieved, the visualization script is run and plots are stored in a separate folder, so that the user can evaluate the data relevant to them. The configuration and execution of the benchmark were designed in accordance with **R1** ease of use, **R2** fairness and do not require the user to modify the source code to set up the experiments using our prototypes. Furthermore, each experiment will follow precisely the configurations chosen by the user and will assess every prototype in the same conditions and using the same workloads.

The server is run on a dedicated server node ❹ and the clients on a dedicated client node ❺, or multiple nodes if configured. Those could be separate machines or a single one, depending on the user setup. Moreover, there are collector scripts for server metrics and client metrics that would monitor the data during the experiment and after the experiment is complete collect it and store in a separate results file that will be further used for plotting and visualization. The chosen metrics are described in Section 3.3.

Each game instance produces a log file that is stored on the NFS ❺ or if run on the local machine, at the specified storage location. Prototype builds are also stored on the storage system of the environment, which is necessary to run client and server instances. The minimum storage requirements for the benchmark would depend on which tests the user wants to perform, each prototype weighs approximately 200 Mb, and the weight of the log files is determined on the implementation of the individual prototype and the duration of the experiment and workload used.

For prototype builds which are stored in the NFS ❸, several approaches were attempted. Initially, concept design did not include separate builds for each prototype, and instead accounted for one unified build completely integrated in Opencraft 2.0, however, due to

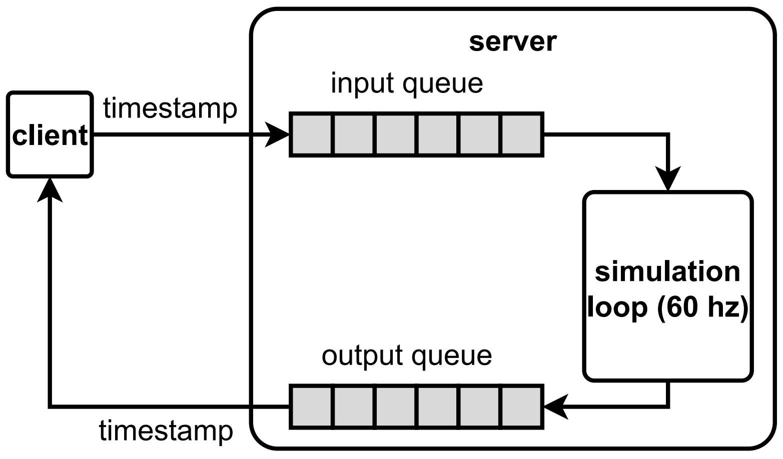


Figure 3.2: Round-trip time (RTT) measurement.

the complexity of integration of network gaming libraries in the same game instance, which was also discussed in section 2 and is apparent from Figures 2.3 and 2.2 it was decided to use an alternative concept. Instead, the benchmark operates with prototypes that use different networking libraries using the best practices specified individually. In order to satisfy **R2** fairness. With this approach, users can assess the performance of networking libraries and observe the effects of various software design choices on the performance of each prototype.

3.3 Metrics

Net-Celerity uses several metrics to provide constructive insight of the performance of networking library under various workloads, that are further described in Section 3.4. The metrics were chosen with respect to **R3** clarity and **R4** relevance and satisfy **RQ2** which questions which metrics would effectively quantify the performance of the networking libraries.

Round-trip time (RTT) is a measure of how long it takes a packet to travel from one location on a network to another and receive a response packet back. We measure RTT using build-in tools provided for chosen networking libraries, Mirror and NFE, and represent in milliseconds. This metric is influenced by the server's update intervals (12, 19) as well as the delay in queueing; see Figure 3.2. We configured both prototypes to update at a frequency of 60Hz, which translates to network updates occurring every 16 milliseconds.

Table 3.1: Metrics Used for the Assessment.

Name	Type	Unit	Description
Round-trip Time	Application	ms (milliseconds)	Round-trip time for messages sent between the client and the server.
CPU Usage	System	% (percents)	System-wide CPU utilization sampled by psutil library.
Memory Usage	System	Bytes	'Resident Set Size', which is the non-swapped physical memory a process has used.
Data Sent	System	Bytes	Number of bytes sent over network of every network interface installed on the system.
Data Received	System	Bytes	Number of bytes received over the network of every network interface installed on the system.

RTT messages may arrive during an update or while waiting for the next 16 ms interval, and the average wait time will be recorded to provide an accurate measure of latency. As a result, RTT offers insight on the effectiveness of data transmission. Moreover, it is highly sensitive to different types of player activity, such as movement, and real-time interactions, which we are evaluating.

CPU and memory usage are useful indicators of a server's performance under different workloads. These metrics are collected using the *psutil* library, which is run on a designated server node ❸. This library provides comprehensive system data, allowing us to monitor how the server responds to varying levels of demand. By analyzing CPU and memory usage, we can assess the efficiency and scalability of the networking libraries within the prototypes under test.

The amount of data sent and received over the network is also gathered using the *psutil* library. It is collected across multiple network interfaces, from *eth1* to *ethN*. The values collected by *psutil* are cumulative, which means that they either increase or remain constant over time but never decrease. Monitoring these values helps us understand the network load and bandwidth utilization done by each networking library which provides insights into their efficiency and impact on overall network performance.

3.4 Workloads

To effectively assess the performance of the system based on the metrics described in Section 3.3, specific workloads need to be tested. With primary focus on player activity, the workloads will involve player movement and actions including placing and removing blocks. In Minecraft-like games, players often explore the environment, build, fight mobs, or mine. Mirror and NFE prototypes allow us to focus on exploration and building aspects. For fair and clear comparison, two scenarios will be used: idle players and active players.

For the workload with idle players, each metric listed in Table 3.1 will be collected and subsequently evaluated by the benchmark. In this scenario, the environment and player movements, as well as player positions, will remain static. The benchmark will conduct multiple tests with an increasing number of concurrent players to observe how the system handles varying levels of player presence without dynamic actions.

For the workload with active players, a realistic scenario will be used where players will be spawned in a specific position and will perform certain activities, specifically jumping and placing a block underneath them. Each player will continue to perform the said activity until 10 blocks are placed. With such a workload, we hope to identify how each networking library deals with complicated behaviors and player activities depending on the number of players. The vertical movement of players was chosen because it will not cause the server to generate more chunks of environments and will remove workload unrelated to player activity, specifically terrain generation.

4

Implementation

4.1 Selection of Networking Libraries

Unity, being one of the most widely used game development platforms, offers a number of networking libraries to simplify the integration of multiplayer functionalities. Specifically, Unity provides two primary netcode solutions: Netcode for Game Objects and Netcode for Entities, which will be discussed in more detail in Section 2.2.

Netcode for Game Objects is designed for games with a moderate number of networked objects and less demanding networking requirements. It makes use of an object-oriented approach which is suitable for less complex projects where the overhead of managing numerous networked entities is not a significant concern.

On the other hand, Netcode for Entities utilizes a data-oriented approach, also called Unity's Data-Oriented Technology Stack (DOTS), which is intended for games that require high performance and scalability. This method is advantageous for projects with extensive networking needs, as it efficiently handles large amounts of data and numerous entities, thus improving overall performance.

In addition to Unity's built-in solutions, a wide range of networking libraries are available, providing developers with alternative options that can be adjusted to specific needs and further optimized. These libraries are summarized in the Table 4.1 in the decreasing order of popularity.

From the Table 4.1 we identified that Kimo's Connection Protocol (KCP) is the most frequently used open-source networking library. KCP is a lightweight protocol specifically designed for reliable UDP data transmission (13). It was initially developed in C and then adapted for C#. KCP is applicable not only for game development but also for other real-time applications where low latency and robust packet loss recovery are crucial, such as

Table 4.1: Unity Compatible Networking Libraries.

Name	GitHub Stars/Favorites	Protocol
KCP (kcp2k)	13800	UDP
Photon (PUN2)	7554	TCP or UDP
Mirror	5611	TCP or UDP
MagicOnion	3400	TCP
GGPO	2900	P2P
LiteNetLib	2800	UDP
Enet	2500	UDP
yojimbo	2300	UDP
ForgeNetworking Remastered	1500	TCP or UDP
Telepathy	1100	TCP
RiptideNetworking	918	TCP or UDP
Networker	471	TCP or UDP
DarkRift 2	167	TCP or UDP

live streaming. Furthermore, because of KCP's efficiency and reliability, it was integrated into other networking libraries, such as Mirror, which is discussed in Section 2.3.

The focus of this study is to evaluate networking libraries that are applicable to real-world scenarios, where connectivity can vary significantly, and the number of concurrent players can scale to the hundreds. Consequently, Netcode for Entities and Mirror were selected for their capability in handling high number of concurrent player and changing network conditions. Specifically Mirror was chosen because of the ease of integration and the possibility to swap low-level netcode components as is discussed in 2.3.

To account for the differences in code structures required by various networking libraries, as discussed in Section 2, the implementation process was divided into two main parts: the integration of networking libraries - Section 4.2, and the creation of the benchmark - Section 4.3.

4.2 Integrating Networking Libraries

Opencraft 2.0 was designed using the DOTS data-oriented approach as ECS, which is fundamentally different from and not compatible with the object-oriented structure employed by Mirror. To ensure that both prototypes function in a comparable manner, several design decisions were made:

1. **Decoupling Assets:** Assets such as blocks, player entities, and scene generation must be decoupled from the NFE netcode. Where full decoupling is not possible, assets should be partially reused and recreated in a similar manner to ensure consistency.

2. **Functional Equivalence:** The game prototype must maintain the same functionalities across both networking libraries, except for the netcode itself, to ensure fairness in the evaluation as per **R2** fairness.

3. **Consistent Testing Conditions:** Each implementation will be tested against the same test cases and metrics outlined in Section 3, and the results will be compared. It is crucial that the builds of both OG and ECS prototypes are not in development mode, as this enables additional metric extraction methods via the Unity editor, which could unfairly affect the results. Furthermore, using the same builds that would be used in the end product is more valuable for real-world application and environment.

4. **Player Emulation:** Both prototypes must support player emulation to simulate realistic player interactions. This can be achieved by using input traces recorded from individual players, which are then replayed to create consistent and repeatable test scenarios. This method ensures that the performance evaluation accurately reflects typical player behavior and interactions within the game environment.

Following these design decisions and implementation steps, the Mirror prototype and DOTS-NFE prototype used in the benchmarking process were able to provide a comprehensive and unbiased comparison of the networking libraries. This allows Net-Celerity to highlight their respective strengths and weaknesses under same workloads, and consequently, ensures the reliability and validity of the benchmark results.

4.3 Implementation of Net-Celerity Benchmark

The benchmark integration must adhere to the design principles discussed in Section 3.2 and incorporate system metric collection scripts as indicated in Section 3.3. For the implementation overview see Figure 4.1.

The core implementation of the benchmark is executed using Python scripts, making use of both inbuilt and specialized libraries for various tasks. Specifically:

1. *psutil* is utilized for system metric collection, enabling the capture of detailed performance data such as CPU usage, memory consumption, and network activity.

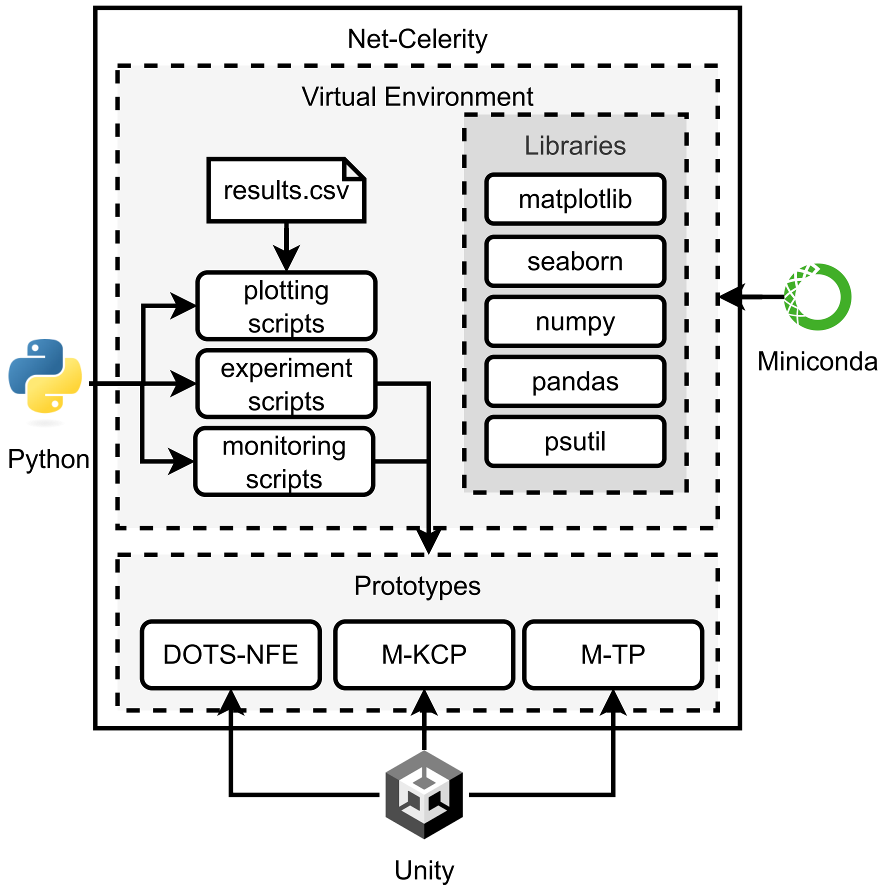


Figure 4.1: Net-Celerity Implementation Overview.

2. *numpy* and *pandas* are employed for data preparation. These libraries facilitate efficient handling and manipulation of large datasets, ensuring that the data is accurately formatted and ready for analysis.

3. *matplotlib.pyplot* and *seaborn* are used for data visualization. These libraries allows for the creation of comprehensive plots and graphs, which are crucial for interpreting the benchmark results and presenting them in a clear, understandable manner.

Data collected from the experiments are stored in specified CSV files. The configuration details, including paths to executables and logs, are managed in the *config.cfg* file. This configuration file allows the user to adjust the benchmarking parameters, making the setup adaptable to various testing scenarios.

The final plots are generated using the *plot_ results.py* script, which processes the collected data and produces visual representations of the benchmark results. These plots are then saved as PDF files in the *plots* folder, ensuring that the results are easily accessible and can be included in reports and presentations. Users can choose specific plots that can be plotted for analysis by calling a respective function in the script.

For a smooth experimental setup, it is recommended to use a virtual environment, such as *miniconda*. Using a virtual environment helps to manage dependencies and maintain a clean, isolated environment to run the benchmarks. This approach minimizes the risk of conflict between different software packages and ensures reproducibility of the experiments.

In addition, the setup includes automated scripts for initiating and terminating the benchmarks and logging relevant performance data. These scripts are designed to streamline the benchmarking process, reducing the manual effort required and increasing the overall efficiency and reliability of the tests.

By following this structured approach to benchmark creation, the performance of different networking libraries can be evaluated under controlled and repeatable conditions. This methodology not only provides an adaptable framework for comparing networking libraries within different prototypes, but also ensures that the results are accurate, reliable, and applicable to real-world scenarios.

Consequently, technical details of Net-Celerity can be found in the Section A.1. We keep detailed documentation of Net-Celerity and prototype implementation in accordance to **R1** ease of use.

5

Evaluation

5.1 Main Findings

We present here an overview of our main findings (MF), based on the experiments shown in Table 5.1. The findings answer **RQ1** and **RQ3** which focus on evaluating how various networking libraries perform compared to each other, and how a selected networking library performs in a real-world scenario, respectively. In the following sections, we discuss each main finding in detail.

MF1 The choice of networking library can significantly affect round-trip time (RTT) decreasing it from 1.6x starting with 20 players to up to 2.1x for 200 players, which is discussed in Section 5.3.

MF2 The selected networking library affects server-side resource usage, utilizing approximately 19 times more RAM and increasing CPU utilization by up to 5 cores for 200 players, see Section 5.4.

MF3 The networking libraries vary in the amount of data clients send to the server and display different patterns, which is further elaborated on in Section 5.5.

MF4 The performance of networking library can be affected by different player activity and user can experience the increase of 1-2 ms for 20 players up to 4-6 ms for 80 players, which is illustrated in Section 5.6.

5.2 Experimental Setup

In this section, the setup of the experiments conducted is discussed. Net-Celerity benchmark as well as DOTS and Mirror prototypes are hosted and tested on DAS-6 which is an

Table 5.1: Experiment Overview.

Prototype	Behavior	# Players	Dur. [m]
DOTS-NFE, M-KCP, M-TP	Idle, Active	20, 40, 60, 80	2
DOTS-NFE, M-KCP, M-TP	Idle	20, 40, 60, 80, 100, 120, 140, 160, 180, 200	2

optimal environment to evaluate the performance and behaviors of large-scale applications; see Table B.1. Furthermore, the summary of all the experiments conducted can be seen in Table 5.1.

Firstly, we evaluated how much clients does a single DAS-6 node can run by measuring CPU usage and RAM usage with the following amounts of players: 5, 10, 20, 40, 60, 80, see Figure B.3. According to the results after 60 player mark client node reaches around 95%-99% CPU usage, which will affect the experiments results. Consequently, we decided that 20 players per client node is the optimal amount for our experiment setup.

The experiments were conducted with intervals of 20, 40, 60, and 80 players with Idle and Active workload to evaluate the difference player activity may introduce, as well as intervals for 20, 40, 60, 80, 100, 120, 140, 160, 180 players with Idle behavior for scalability testing. These intervals were chosen to simulate various server loads representative of real-world scenarios. Each configuration provides insights into how different network libraries handle increasing player counts and behaviors.

The duration of the experiments was established after the analysis of the RTT variance 5.1.

$$S_{RTT} = \frac{\sum_{x \in RTT} (x - \frac{\sum_{x \in RTT} RTT}{len(RTT)})^2}{len(RTT)} \tag{5.1}$$

In Figure B.4, it is evident that both the M-KCP and DOTS-NFE prototypes stabilize their performance around 120 seconds. This period indicates that the server requires some initialization time before achieving optimal operation. During this initialization phase, the variance in RTT decreases, suggesting that the system is adjusting and optimizing its performance.

However, for M-TP, the data do not show a clear time frame for server initialization. Instead, the variance fluctuates without a distinct pattern. Consequently, the point at which the variance is midway between its maximum and minimum values was used as a

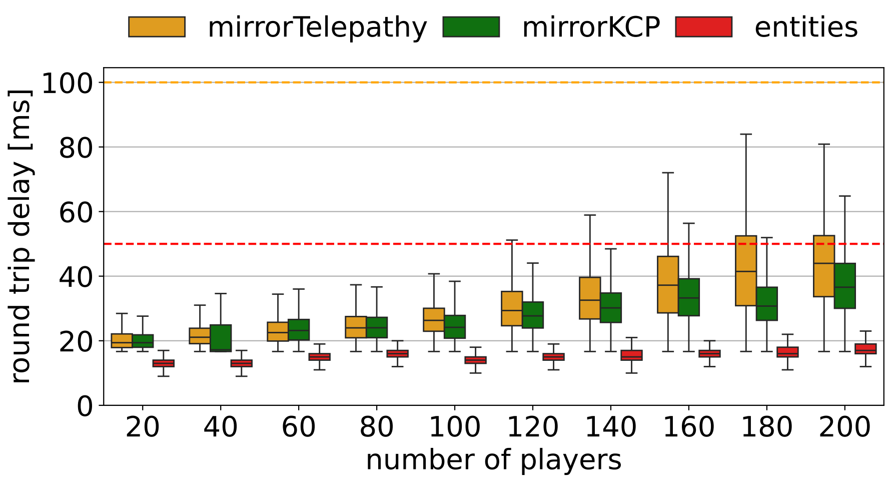


Figure 5.1: Round Trip Time of Prototypes for Extended Test.

reference for comparison, which is 120 seconds. For M-TP, although the variance does not settle as clearly, the chosen reference point provides a basis for comparative analysis.

5.3 Correlation Between Number of Players and RTT

The benchmark results reveal significant insights into the scalability of networking libraries, particularly in relation to round-trip time (RTT), a critical metric for real-time applications like multiplayer games.

Figure 5.1 presents box plots that illustrate the RTT performance of three networking libraries under varying player loads. The red and orange dotted lines represent the latency threshold noticed by users in different game genres. According to the studies, players of FPS (First Person Shooter) generally notice latency effects on 50 ms, and the accuracy decrease of 50% is apparent after 75 ms (5, 20). On the other hand, for third person RPG (Role-Playing Game) and racing games the noticeable latency for the players is around 100 ms (21, 22). Clearly, not all networking libraries perform within acceptable limits for user experience. Furthermore, notable differences are visible when comparing their scalability with increasing concurrent players.

Both DOTS-NFE and M-KCP performed within the set bound of 100 ms, however, at 120 players M-TP and 160 players M-KCP have crossed the 50 ms threshold. We mentioned that Telepathy is a TCP-based netcode and KCP is a reliable UDP implementation in Section 2.3 which might be the cause of the higher variance for M-TP with increasing players. Moreover, it was also apparent during experiment setup B.4, since only for M-TP variance fluctuations did not stop over time.

DOTS-NFE has better scalability, with its RTT increasing from approximately 12 ms (20 players) to around 19 ms (200 players). This represents an increase of around 1.6 times. In contrast, M-KCP and M-TP start with an RTT of about 19 ms (20 players) and increase to approximately 35 ms and 45 ms (200 players), respectively. This considerable increase of around times of 1.8-2.3 times indicates that these libraries experience more pronounced latency under higher player loads, potentially impacting the real-time gameplay experience.

The lower RTT observed with Netcode for Entities (NFE) compared to Mirror, despite potential differences in resource consumption, see Section 5.4, could be attributed to the distinct time synchronization approaches implemented by each framework.

Mirror operates on a state synchronization model where data is synchronized from the server to remote clients using SyncVars (23). This approach ensures that clients receive the updates necessary to maintain the consistency of the game state. However, Mirror does not synchronize data from clients back to the server unless using Commands, which means that it primarily focuses on maintaining state consistency across networked entities and does not have any way of predicting or optimizing in case of higher loads.

On the other hand, we discovered that NFE makes use of the authoritative server model with time synchronization mechanism controlled by the NetworkTimeSystem. This system calculates and adjusts server time estimates for clients based on round-trip times (RTT) and received snapshots. It's primary goal is to ensure that clients predict server states accurately which helps minimize issues that could lead to synchronization issues or higher RTT values. By managing prediction ticks and interpolation delays, NFE synchronizes client and server states.

The efficiency in NFE's time synchronization mechanism probably contributes to its lower observed RTT compared to Mirror. By accurately predicting and adjusting server ticks on clients, NFE reduces the need for frequent state updates, optimizing network bandwidth usage and client-server interaction.

For developers, this finding provides valuable insight into which networking library is best suited for their needs. Competitive games with large number of concurrent players per session, such as battle royale shooters, could benefit from very low latency that DOTS-NFE exhibited. Projects that might not require such optimizations and are made by a team with less experience could value the simplicity of integration of either M-KCP or M-TP, taking into account that they do not aim for best responsiveness and can sacrifice latency, which is a common practice in turn-based games.

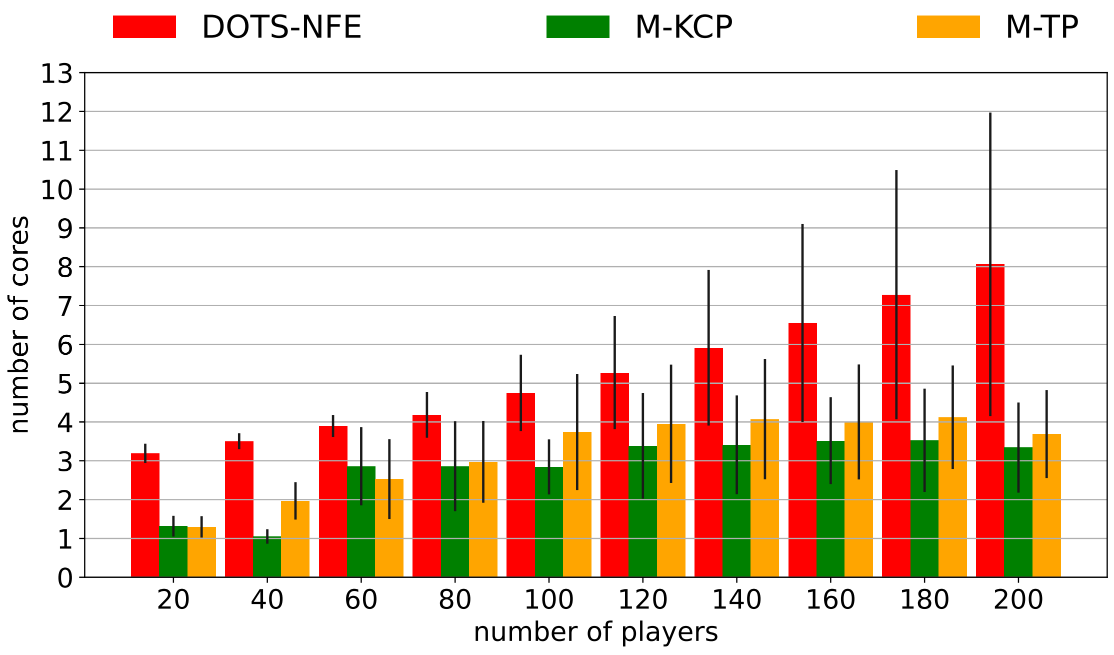


Figure 5.2: CPU Usage of Server.

5.4 Effects on Server-Side Resource Usage

Analyzing the resource utilization depicted in Figure 5.3 and Figure 5.2, we noticed that DOTS-NFE has higher demands on server resources compared to the Mirror prototype. Specifically, DOTS-NFE requires approximately twice the CPU usage and up to 18 times more memory when accommodating 200 players.

In contrast, both M-KCP and M-TP have relatively consistent resource requirements across varying player workloads, with slight increases in both memory and CPU usage observed as the number of players increases. This stability in resource consumption suggests efficient resource management within these networking frameworks.

The noticed difference between DOTS-NFE and both Mirror prototypes requires further investigation of potential factors. One possible explanation could be connected to the architectural differences between DOTS and traditional object-oriented approaches. DOTS optimizes data access patterns and emphasizes cache coherence, which can lead to more efficient data processing but may require higher computational overhead initially. In contrast, Mirror's object-oriented approach is simpler in design and may lower overhead, but could struggle with scaling efficiently under heavier loads.

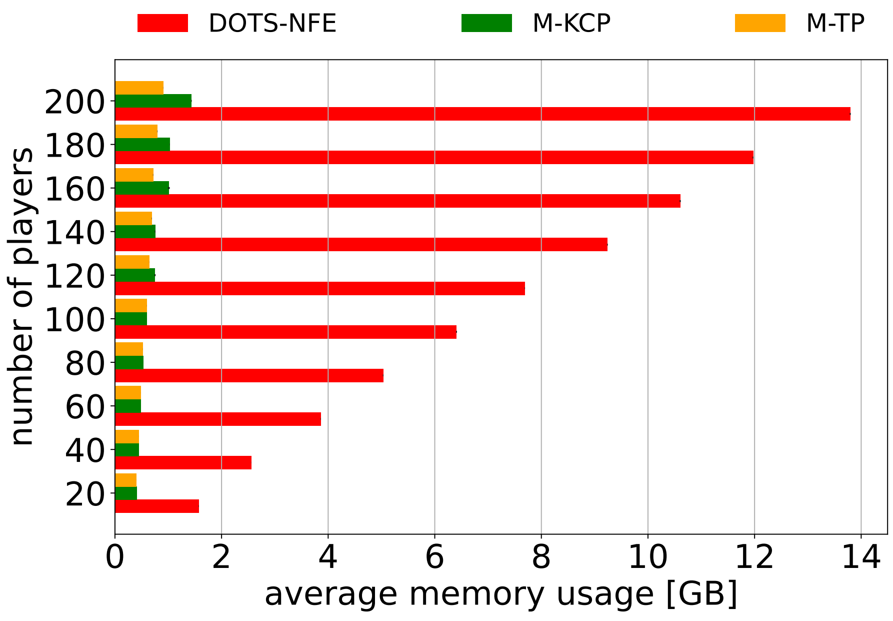
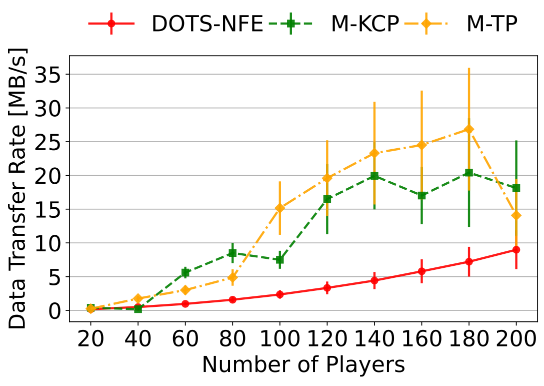


Figure 5.3: Memory Usage of Server.

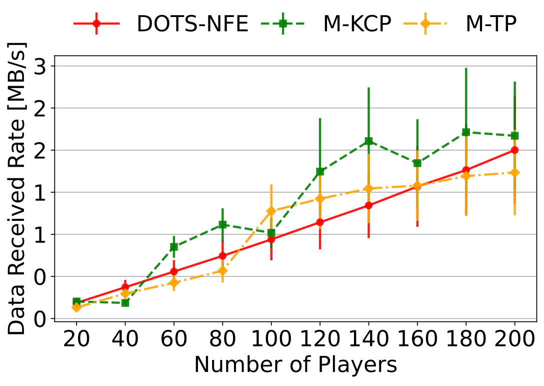
Despite these observations, we cannot provide a definitive answer to these differences in resource management. Hypothetically, differences in implementation techniques, memory management strategies, or the handling of concurrent operations between DOTS-NFE and the object-oriented Mirror prototype could contribute to these differences. We will further discuss in Section 7 what possible aspect can be investigated to answer those questions.

The actionable insight of this finding suggests that there exists a trade-off of performance vs. resource consumption. As we established in 5.1 in Section ?? that DOTS-NFE has lower RTT values compared to M-KCP and M-TP and combined with a higher server-side resource consumption, developers need to make a choice. If their resources are limited they will highly benefit from M-KCP and M-TP if they are ready to sacrifice RTT of players. On the other hand, if there are no restrictions for the server's resource consumption or they are comparable to our setup, we recommend the DOTS-NFE networking library.

In conclusion, while DOTS-NFE demonstrates superior performance in terms of real-time responsiveness and network latency, as highlighted in previous sections, its increased resource requirements underscore the trade-offs inherent in adopting more sophisticated data-oriented architectures for networked applications.



(a) Data Sent from the Server



(b) Data Received by the Server

Figure 5.6: Data Transfer Rates of the Server.

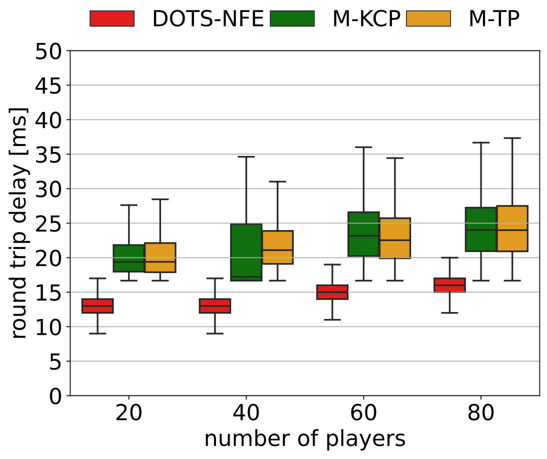
5.5 Differences in Networking Traffic

During the benchmarking analysis of DOTS-NFE, M-KCP, and M-TP, we closely monitored networking traffic patterns and observed distinct behaviors as player counts increased, see Figure 5.6. Although all three networking libraries show an increase in data rates with higher player numbers both sent and received by the server, DOTS-NFE has a lower increase in traffic sending rates, despite its higher system resource consumption which we previously discussed in 5.4. Furthermore, the increase for DOTS-NFE is predictable, meanwhile, M-KCP and M-TP do not have distinct patterns for the data transmission and received rates.

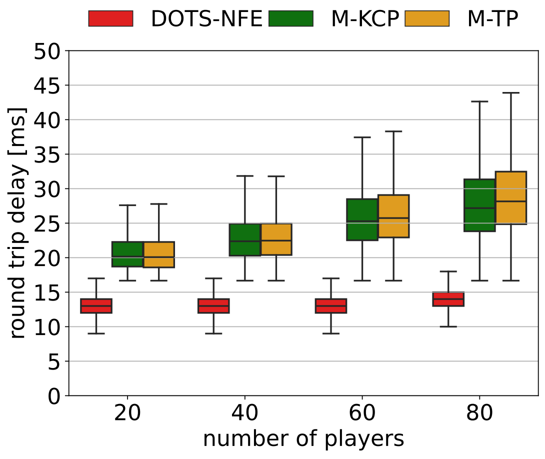
As discussed above, DOTS-NFE minimizes the amount of data that need to be synchronized between clients and the server. This efficiency in data management significantly reduces network traffic compared to the simpler but less efficient synchronization methods employed by Mirror. Unlike game object approaches, DOTS-NFE serializes and transmits only essential changes in entity state, and does not have to send redundant data of the whole game object. This strategy reduces the amount of data sent over the network, leading to lower transmission rates even under high player counts.

Furthermore, the stable nature of the data transmission and received rates can be also do to DOTS-NFE's client prediction implementation discussed in the Section 5.6. However, this must be further investigated for a certain answer.

Developers can benefit from this result, since it clarifies which networking libraries will have predictive sent/received rates for the server, especially for the cases when networking



(a) Idle Players



(b) Active Players

Figure 5.9: Round Trip Time of Prototypes vs. Player Behaviors.

infrastructure cannot support inconsistent transfer rates.

5.6 Effects of Player Activity on Networking Libraries

After benchmarking specific types of behavior - idle and active, see 3.4 for further details, we noticed that DOTS-NFE was not affected by the change of player activity, meanwhile M-KCP and M-TP had experienced slight increase of RTT for all numbers of players.

If we compare medians of the boxplots from Figure 5.7 to Figure 5.8 there is always an increase with at least around 2 ms as well as lower quartile. It becomes more pronounced at the 60 and 80 player marks. Furthermore, variance of the boxes for with active playrs is higher in comparison to the idle players, which is visible by the increase of sizes in whiskers.

Since the player activity behavior involved placing blocks, which are considered Networking Objects, and M-KCP and M-TP have lower scaling capabilities compared to DOTS-NFE, this can be the cause of a slight increase in RTT. To further confirm the cause of the gain in ms for players, we will propose additional experiments in Section 7.

Since player activity is a key component of a multiplayer project, this finding will enable developers to be better informed about the limitations of the networking libraries. Current results suggest a negligible increase even for 80 players, and thus there are no trade-offs to consider yet, however, it is important to take a note of current differences since there is a possibility that with larger amount of player activity M-KCP and M-TP will have higher impact on RTT, and thus if the game involves a high number of player action DOTS-NFE is the suggested option.

5.7 Limitations and Threat to Validity

In Section 5.2 we show why we chose a specific duration of the experiments; however, taking into account that both M-KCP and M-TP have a higher round-trip time variance, and, furthermore, M-TP has irregular variance over time, it is possible that the results are affected by our choice. Alternative approach would have been experimenting with different time frames and comparing the results to effectively estimate bias that might be introduced.

In the late stage of the experiment we were able to identify a threat to validity from the client nodes bottleneck, which is apparent from Figure B.1, however, we did not have the opportunity to also investigate the DAS-6 networking environment which also introduced bias. It could have been done by evaluating what are the data transmission rates when there is only server launched and no client, which would indicate if the machine itself transmits data. Additionally, it could have been evaluated if the transmission rates on the network done by not client/server activity is irregular, since it would then greatly affect the results.

Finally, player activity was emulated with input traces, which require mouse input and camera movement. Although there are no issues with emulating pre-recorded behaviors on clients with graphical mode, there might be inconsistencies with headless mode that we have not checked. It could have been done by running the emulation and then connecting as a user in graphical mode to see if the blocks were placed correctly. Alternatively, enabling users in graphical mode is also an option, since DAS-6 has GPUs available, which would be a valuable additional experiment for the future work.

6

Related Work

The discussion of related work is essential to evaluate the research done within the existing domain, identify gaps, and highlight contributions. In the field of online gaming, particularly for Minecraft-like games, scalability and performance are critical areas of investigation due to the massive player bases and complex virtual environments these games support.

Previous benchmarks and middleware solutions have aimed to address these challenges. For instance, the Dyconits (24) focuses on scaling MVEs by bounding inconsistency dynamically and optimistically. Dyconits partitions the game world into units, each with specific bounds, to manage data consistency and scalability more effectively. This system integrates with existing game code bases with minimal modifications, supporting up to 40% more concurrent players and reducing the usage of network bandwidth by up to 85% without significantly increasing latency.

Another significant effort was made by Yardstick benchmark (6) which provides a comprehensive performance analysis framework for Minecraft-like services. Yardstick captures the operational characteristics of these services, using popular community maps as input workloads and measuring both system- and application-level metrics. Through real-world experiments, Yardstick has revealed the scalability limits and parallelization inefficiencies of various Minecraft-like servers, including the official vanilla server, Spigot, and Glowstone.

Net-Celerity builds on these foundations, but takes a different approach by focusing on prototypes and network libraries rather than Minecraft-specific implementations. Unlike Dyconits, which optimizes data consistency and scalability within a Minecraft-like environment, our benchmark evaluates the performance of different network libraries, Netcode for Entities, Mirror KCP, and Mirror Telepathy, within Unity prototypes. This allows us to analyze how these libraries handle RTT, CPU usage, and data reception under various workloads and player activity levels specified in Section 3.3.

Furthermore, while Yardstick provides a benchmark for Minecraft-like services, our work extends the scope of the benchmark to include the evaluation of network library performance in Unity, a widely used game development engine (3). This is particularly relevant, as Unity is increasingly used for developing a variety of multiplayer games beyond the Minecraft genre, even if the prototypes use Minecraft-like features as an example. Our benchmark measures key performance indicators offering a comprehensive assessment of how these network libraries scale with increasing player counts and activity levels.

In summary, while Dyconits and Yardstick have significantly contributed to understanding and improving the scalability and performance of Minecraft-like games, our research introduces a novel benchmarking framework that focuses on Unity-based, but not limited to, prototypes and network libraries. This expands the applicability of benchmarking tools to a broader range of multiplayer games, providing valuable information to developers working within the Unity ecosystem.

7 Future Work

In this study, we provided information on the performance of different network libraries under varying workloads using specified metrics. However, there still are further research to be done to cover unanswered questions and address identified limitations.

One area for future research is deeper investigation into the architectural differences between DOTS-NFE and the Mirror prototypes. Understanding these differences could provide clearer insight into the observed distinctions in server resource usage and network traffic efficiency. The process of deep analysis is quite time consuming but necessary for better understanding of technologies used. Existing experience with the networking library aids greatly with this task, and was something that was missing in our study.

Furthermore, extending the scope of experiments to include additional network conditions, workloads, and configurations would offer a more comprehensive evaluation of these network libraries. For example, testing under different network conditions or exploring the impact of the geographical distribution of servers on performance could provide valuable real-world insights for game developers and network engineers. Specifically 5.1 could benefit from higher player activity workloads to confirm the effects on the networking libraries. One possible workload could be introducing emulated behavior for the players to move around and continuously jumping and placing blocks under themselves to test for longer amount of time, instead of requiring input traces that will be limited to a specific amount and duration.

Moreover, conducting longer term studies to assess the scalability and stability of these network libraries over extended periods of time would also be beneficial. Although we had an attempt, see Section 5.2, the conditions did not allow us to conduct experiments for more than 15 minutes during the working day on DAS-6. Monitoring performance metrics over time could reveal interesting details about the long-term reliability and robustness of these libraries under sustained usage. We had already identified in 5.1 that M-KCP and M-TP have an unpredictable data transfer / receive trend that can be attributed to the total duration, so this can help prove or disprove this hypothesis.

Finally, utilizing user experience metrics and qualitative feedback from players in future evaluations would provide a more in-depth assessment of network library performance. Understanding how perceived responsiveness, gameplay smoothness, and overall player satisfaction correlate with the quantitative performance metrics measured in this study could help us validate the practical implications of using different network libraries in real-world multiplayer gaming scenarios. This can be done by introducing such metrics as jitter, frames per second (FPS) and other. The possible experiment could be done with existing input traces by, for instance, rerunning the experiments under different conditions and determining how many blocks ended up being placed.

8

Conclusion

In this study, we developed Net-Celerity, a prototype-based benchmark for network libraries, and have conducted a comprehensive evaluation of three network libraries, Netcode for Entities, Mirror KCP, and Mirror Telepathy, in prototypes resembling Minecraft-like multiplayer environments. Our experiments focused on measuring key performance metrics including round-trip time , CPU usage, and data reception under varying workloads and player activity levels.

Our findings indicate distinct performance characteristics among the tested network libraries. Netcode for Entities shows lower RTT compared to Mirror KCP and Mirror Telepathy under moderate to high player loads due to it's time synchronization architecture and prediction tactics. Meanwhile, all networking libraries stay under 100 ms RTT makes it still acceptable, M-TP and M-KCP have issues keeping up, and as a result crossed the threshold of 50 ms at 120 and 160 players marks respectively.

Our study highlighted the importance of considering architectural differences and network conditions when evaluating network library performance. Architectural details between DOTS-NFE and Mirror implementations can influenced resource utilization and scalability, underscoring the need for deeper architectural analysis in future studies.

Furthermore, we identified that player activity has no effect on DOTS-NFE prototype but causes a slight increase in RTT for M-KCP and M-TP. This can be attributed to Mirror prototype's sensitivity to the raise in number of Networking Objects which blocks placed by the players are also considered to be.

While our experiments provided valuable insights into immediate performance metrics, there are several possible for future research to improve our understanding in network libraries comparison process. Deeper architectural analysis could explain the observed differences and inform users about the optimizations specific network libraries might require.

Furthermore, extending experiments to take into account various network conditions and conducting longer-term studies would provide a more insightful assessment of scalability and stability that is closer to real-world scenarios.

As a result, our research contributes the Net-Celerity benchmarking framework designed for any prototype, expanding on the network library evaluations in addition to traditional Minecraft-like environments. By bridging the gap between performance metrics and real-world gameplay experience, our findings aim to assist developers in making informed decisions when selecting network libraries for multiplayer game development.

References

[1] Newzoo. **Newzoo Global Games Market Report 2023**, 2023. [Online; accessed 1. Feb. 2024]. iii, 1

[2] Samad M.E. Sepasgozar. **Digital Twin and Web-Based Virtual Gaming Technologies for Online Education: A Case of Construction Management and Engineering**. *Applied Sciences*, **10**(13):4678, jul 2020. 1

[3] Unity. **Unity 2022 Multiplayer Report**, 2022. [Online; accessed 8. Feb. 2024]. 1, 36

[4] Alexandru Iosup, Siqi Shen, Yong Guo, Stefan Hugtenburg, Jesse Donkervliet, and Radu Prodan. **Massivizing online games using cloud computing: A vision**. In *2014 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, pages 1–4, 2014. 1

[5] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. **The effects of loss and latency on user performance in unreal tournament 2003®**. In *ACM Conferences*, pages 144–151. Association for Computing Machinery, New York, NY, USA, August 2004. 1, 27

[6] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the International Conference on Performance Engineering, Mumbai, India, April, 2019*, 2019. 2, 35

[7] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games**. In *Proceedings of the International Conference on Performance Engineering, Coimbra, Portugal, April, 2023*, 2023. 2

[8] Alexandru Iosup, Alexandru Uta, Laurens Versluis, Georgios Andreadis, Erwin van Eyk, Tim Hegeman, Sacheendra Talluri, Vincent van Beek, and Lucian Toader. **Massivizing Computer Systems: a Vision to Understand, Design, and Engineer Computer Ecosystems through and beyond Modern Distributed Systems**. *CoRR*, **abs/1802.05465**, 2018. 2

[9] Shie-Yuan Wang, Hsi-Lu Chao, Kuang-Che Liu, Ting-Wei He, Chih-Che Lin, and Chih-Liang Chou. **Evaluating and improving the TCP/UDP performances of IEEE 802.11(p)/1609 networks**. In *Proceedings of the 13th IEEE Symposium on Computers and Communications (ISCC 2008), July 6-9, Marrakech, Morocco*, pages 163–168. IEEE Computer Society, 2008. 5

[10] Unity. **Unity Netcode for Entities | Netcode for Entities | 1.0.17**, 2023. [Online; accessed 2. Mar. 2024]. 6, 7

[11] Unity. **ECS for Unity**, feb 2024. [Online; accessed 22. Feb. 2024]. 6, 7

[12] **Mirror Networking – Open Source Networking for Unity**, 2024. [Online; accessed 2. Mar. 2024]. 7, 8, 15

[13] **KCP Networking Library**, 2020. [Online; accessed 2. Mar. 2024]. 7, 8, 19

[14] **GitHub - MirrorNetworking/Telepathy**, 2023. [Online; accessed 8. Feb. 2024]. 7

[15] **GitHub - MirrorNetworking/SimpleWebTransport**, 2021. [Online; accessed 8. Feb. 2024]. 7

[16] vis2k. **A Brief History of Mirror - Mirror**, 2024. [Online; accessed 8. Feb. 2024]. 8

[17] Unity Technologies. **Unity - Manual: Advanced operations: Using the LLAPI**, March 2024. [Online; accessed 5. Mar. 2024]. 8

[18] **LiteNetLib**, March 2024. [Online; accessed 5. Mar. 2024]. 8

[19] **Metrics | Netcode for Entities | 1.1.0-pre.3**, November 2023. [Online; accessed 2. Apr. 2024]. 15

[20] Xiaokun Xu, Shengmei Liu, and Mark Claypool. **The Effects of Network Latency on Counter-strike: Global Offensive Players**. In *14th International Conference on Quality of Multimedia Experience, QoMEX 2022, Lippstadt, Germany, September 5-7, 2022*, pages 1–6. IEEE, 2022. 27

[21] Mark Claypool and Kajal T. Claypool. **Latency and player actions in online games**. *Commun. ACM*, **49**(11):40–45, 2006. 27

[22] Pedro Casas, Florian Wamser, Fabian E. Bustamante, and David R. Choffnes, editors. *Proceedings of the 4th Internet-QoE Workshop on QoE-based Analysis and Management of Data Communication Networks, Internet-QoE@MobiCom 2019, Los Cabos, Mexico, October 21, 2019*. ACM, 2019. 27

[23] **Synchronization | Mirror**, June 2024. [Online; accessed 15. Jun. 2024]. 28

[24] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. **Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency**. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 126–137. IEEE, 2021. 35

Appendix A

Reproducibility

A.1 Abstract

Net-Celerity is a benchmarking tool that is used to evaluate networking libraries of prototype-based games. In this section, we will describe the details of the artifact, how to use it, and how to reproduce the results.

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

- **Data set:** input traces for DOTS-NFE and Mirror prototypes are provided.
- **Hardware:** DAS-6, see specifications in the Table B.1.
- **Experiments:** see Table 5.1.
- **How much disk space required (approximately)?:** 2 GB.
- **How much time is needed to prepare workflow (approximately)?:** 5-10 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1.8 - 2 hours.
- **Publicly available?:** Yes. `https://github.com/atlarge-research/Net-Celerity`
- **Code licenses (if publicly available)?:** MIT

A.3 Description

A.3.1 How to access

Net-Celerity is fully open source and is accessible on the github page: `https://github.com/atlarge-research/Net-Celerity`.

Builds of prototypes used can be found here: `https://drive.google.com/drive/folders/1f9s32V-_nGZOcAA8TPbWQCMLxG817t8I?usp=sharing`

A.3.2 Hardware dependencies

Our experiment does not take into account the bias introduced by hardware, and thus is recommended to run on DAS-6 or system with specifications similar to those of DAS-6, which are indicated in Table B.1.

A.3.3 Software dependencies

Net-Celerity was designed with Linux systems in mind, it was tested on Rocky Linux 8.5 (Green Obsidian), and partially on Ubuntu 22.04.4 LTS- the builds and the script execution, but not the full duration of experiments.

A.4 Installation

A.4.1 Setting up Virtual Environment

In order to simplify the installation process, we use *miniconda3* virtual environment and store all dependencies in the *environment.yml* file. Make sure to install it first. To create conda environment do

```
conda env create -f environment.yml
```

and then activate it with

```
conda activate net-celerity-env
```

A.5 Experiment workflow

A.5.1 Editing config.cfg

Before running the experiments, config.cfg needs to be configured. The process is straightforward:

Path and execution related variables:

- *prototype_name:* the name of the prototype
- *prototype_logs:* location of the log folder for the prototype
- *prototype_server_command:* bash command to start the server
- *prototype_client_command:* bash command to start the client. The command can be further edited in prototype_experiment.sh in case the ip address to which the user needs to connect is not static and has to be configured once the server is initialized.

- *collection_script:* location of python script to execute log data collection.

Server/Client nodes related variables:

- *server_node:* name of a server node to ssh into. Can be only one.
- *client_nodes_number:* the amount of client nodes that need to be reserved.
- *client_nodeN*: name of the client node to ssh into. Replace N with a number. Add as many as necessary by starting with 1 and incrementing the number. The number of client nodes must be 1 to client_nodes_number for the script to work as intended.

A.5.2 Collection Script

The collection script is a Python script that is run at the end of the experiment to collect all relevant metrics and store them in one *<NAME>_results.csv* file. Currently it works by counting how many player scripts are available in the folder and then going individually through them and storing the round-trip time in a common *.csv* file with dedicated ID for the player. The headers of *.csv* are:

```
Player_ID;Total_Players;RoundTripDelay_ms
```

One drawback to this approach is that it is individual to a prototype since the logging is done with different fields. However, for future prototypes if the field names are the same as in one of the provided collector scripts, they can be reused.

In total we have three scripts: *mirror_collect_script.py* for M-KCP, *mirror_t_collect_script.py* for M-TP, and *entities_collect_script.py* for DOTS-NFE.

A.5.3 Monitoring Script

System metrics monitoring is done using *psutil* Python script and is the same for every prototype. It is attached to the server and monitors quite a few metrics. All of them can be seen in the *system_monitor.py.* They are later stored in the following manner:

```
system_logs/${prototype_name}/system_log_${num_players}p_${benchmark_duration}s.csv
```

A.5.4 Running the Experiments

All the experiments provided can be reproduced; however, it is quite time consuming. To analyze different player activities of a single prototype we would require: 2 * ((20 * 3 + 120) + (40 * 3 + 120) + (60 * 3 + 120) + (80 * 3 + 120)) = 2160 second or 36 minutes, where 3 is a time for spawning a player and 2 is how many player activities we use.

In order to reproduce the experiments and not change the *config.cfg* too often, we recommend going through all the workload within the same prototype and then move on.

To make it even simpler, it is possible to copy the bash commands from prototype_experiment.sh and into a separate script per prototype if there are several machines at your disposal to run concurrently. We provide three examples: mirror_experiment.sh for M-KCP, entities_experiment.sh for DOTS-NFE, and prototype_experiment.sh applicable to any prototype, and currently set up for M-TP. Once the values in *config.cfg* are configured, the scripts can be run.

A.6 Evaluation and expected results

The expected results are similar to the results discussed in Section 5, or can be evaluated individually `https://drive.google.com/drive/folders/1X1arvN_lzAPBwNt1CNTs1CAeNCinpzZa?usp=sharing`.

Each iteration will produce addtion to the common *<NAME>_results.csv* and several logs for server metrics. The specific structure is the following:

```
system_logs_workload1/
   DOTS-NFE/
      system_log_20p_120s.csv
      system_log_40p_120s.csv
      system_log_60p_120s.csv
      system_log_80p_120s.csv
   M-KCP/
      system_log_20p_120s.csv
      system_log_40p_120s.csv
      system_log_60p_120s.csv
      system_log_80p_120s.csv
   M-TP/
       system_log_20p_120s.csv
       system_log_40p_120s.csv
       system_log_60p_120s.csv
       system_log_80p_120s.csv
system_logs_workload1_extended/
   DOTS-NFE/
      system_log_20p_120s.csv
      system_log_40p_120s.csv
      system_log_60p_120s.csv
      system_log_80p_120s.csv
      system_log_100p_120s.csv
      system_log_120p_120s.csv
      system_log_140p_120s.csv
      system_log_160p_120s.csv
      system_log_180p_120s.csv
      system_log_200p_120s.csv
   M-KCP/
      system_log_20p_120s.csv
      system_log_40p_120s.csv
      system_log_60p_120s.csv
      system_log_80p_120s.csv
      system_log_100p_120s.csv
      system_log_120p_120s.csv
      system_log_140p_120s.csv
      system_log_160p_120s.csv
      system_log_180p_120s.csv
      system_log_200p_120s.csv
   M-TP/
      system_log_20p_120s.csv
      system_log_40p_120s.csv
      system_log_60p_120s.csv
      system_log_80p_120s.csv
      system_log_100p_120s.csv
      system_log_120p_120s.csv
      system_log_140p_120s.csv
      system_log_160p_120s.csv
      system_log_180p_120s.csv
      system_log_200p_120s.csv
system_logs_workload2/
```

```
   DOTS-NFE/
      system_log_20p_120s.csv
      system_log_40p_120s.csv
      system_log_60p_120s.csv
      system_log_80p_120s.csv
   M-KCP/
      system_log_20p_120s.csv
      system_log_40p_120s.csv
      system_log_60p_120s.csv
      system_log_80p_120s.csv
   M-TP/
       system_log_20p_120s.csv
       system_log_40p_120s.csv
       system_log_60p_120s.csv
       system_log_80p_120s.csv
workload1_results/
   DOTS-NFE_results.csv
   M-KCP_results.csv
   M-TP_results.csv
workload1_results_extended/
   DOTS-NFE_results.csv
   M-KCP_results.csv
   M-TP_results.csv
workload2_results/
    DOTS-NFE_results.csv
    M-KCP_results.csv
    M-TP_results.csv
```

The structure can be different, depending on what paths are specified in the *config.cfg*.

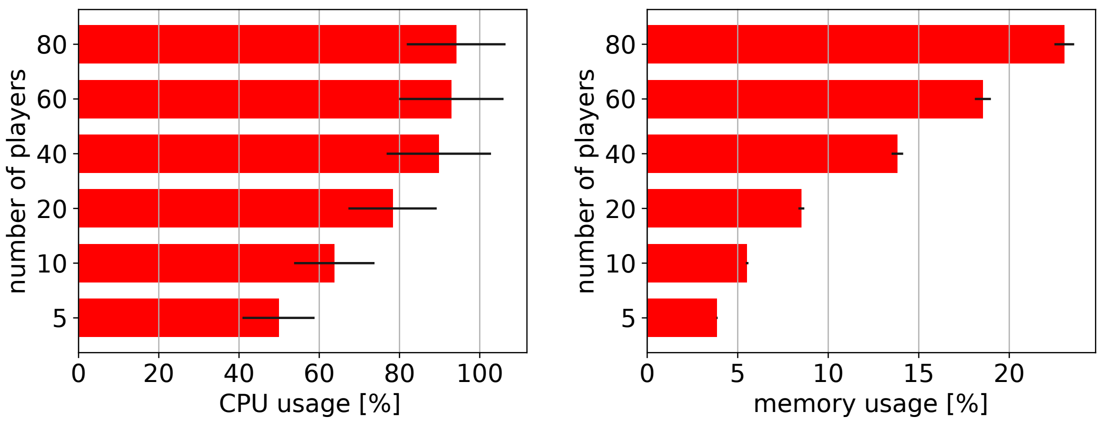
Appendix B

Additional Experiments

The plots of the additional experiments conduced for experiment setup which were discussed in Section 5.2 as well as specification of the system used.

Table B.1: DAS-6 Specifications.

Component	Specification
Nodes	34
Most common	single 24-core
Speed	2.8 GHz
Memory	128 GB
Central storage	256 TB
Interconnect	100G Ethernet
GPUs	28×A400, 3×A6000, A100



(a) CPU Usage of Single Client Node (b) Memory Usage of Single Client Node

Figure B.3: CPU and Memory Usage for Client Node. Whiskers Represent Error Bars.

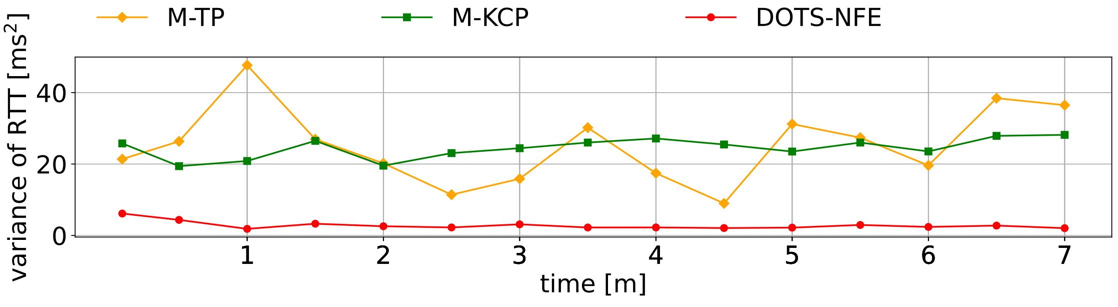


Figure B.4: Variance of Prototypes over Time.