Vrije Universiteit Amsterdam

Bachelor Thesis

Benchmarking Deployed Modifiable Virtual Environments across the Cloud-Edge Continuum

Author: Victor Gavrilovici (2670026)

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

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

August 21, 2024

Abstract

Minecraft could be considered one of the most prevalent video games of our time, with over 300 million copies sold as of October 2023. Its uniqueness comes from its role in pioneering Modifiable Virtual Environments (MVE) in video games, allowing players to change or expand the game's world how they see fit. Games are deployed on highly heterogeneous hardware and environments. This makes evaluating their performance a difficult and time-consuming task. One of the inherent challenges with MVEs is that online games with such environments scale poorly to many players, as they have the task of simulating all the player and non-player interactions, as well as communicating these changes to the player, in a way that is efficient enough to satisfy the latency constraints of an online game. Many deployment options exist for online games. This is most noticeable with deployment strategies across the compute continuum, which allow for highly configurable hardware and network setups. In this work, we propose (or use?) the Continuum testing framework to create a benchmark for the performance of online MVE games running under various computing models to assess and compare their performance.

Highlights and insights from results, with numbers:

Contents

CONTENTS

1

Introduction

Video games have risen to become one of the most popular forms of entertainment, overtaking already established billion-dollar industries, such as movies and sports in the US (1). The recent global pandemic has also increased general interest in games, for example when Twitch, one of the leading live-streaming services for games, experienced an 83% year-over-year uprise in viewership (2). A European survey on the attitudes towards video gaming during the COVID-19 lockdown performed from April to June 2020 on more than 10000 respondents aged 11-64 revealed that playing video games made people feel less isolated, less anxious, and happier. Video games, especially online multiplayer experiences that provide a means for people to connect and socialize, proved to be a popular method for reducing isolation-induced stress (3) in the context of lockdown and social distancing.

A great example of the kind of work that brings popularity to the industry is Minecraft, which is currently the most-sold game of all time, with over 238 million copies (4). At its inception, the game presented a unique creative experience - it provided players with a virtually infinite, fully modifiable, procedurally generated world to explore: players are encouraged to explore, find items and places, or build different things as they wish. Furthermore, the Education Edition of this game allows teachers to assist the students during the learning process. By using computer science lessons, the students may construct their own digital computers or history lessons to explore, for instance, UNESCO world-heritage sites (5).

Another advantage particularly related to Minecraft is its capacity to unlock borders and to have an influence in different authoritarian or repressive societies. For instance, Minecraft provides the opportunity for people to enter an uncensored library, thus giving access to usually censored articles in their country of origin, where there is no free access to information since "Websites are blocked, independent newspapers are banned and the press

is controlled by the state" (according to Christian Mihr, managing director for Reporters Without Borders in Germany) (6).

Minecraft's creation has led to the inception of a new genre of games centered around one similarity - a Modifiable Virtual Environment (MVE). We describe this characteristic according to Donkervliet et al: an MVE is a real-time, online, multi-user environment that allows its users (i.e., players) to modify the virtual world's objects (e.g., player apparel) and parts (e.g., terrain) (7).

This thesis focuses on designing and prototyping a serverless approach to running persistent instances of a Minecraft-like environment.

1.1 Problem Statement

There are multiple challenges related to online games that allow full modification of their world. Scaling the MVE experience to many players is difficult because of the unique freedom that such a simulation allows. An online game has to simulate terrain modifications, player-to-player interactions, the behavior of creatures, and transmit this information to all players, all in real-time. All these compose a challenging set of constraints. Furthermore, we see that MVE games are deployed in a wide variety of environments, depending on the use case, but it is unclear how these environments exactly affect the game's performance and scalability. In this work, we show an approach for tackling this problem by using the Continuum benchmarking framework to create an Online MVE game benchmark that supports automated emulation of diverse hardware and network infrastructure configurations.
.

1.2 Research Questions

RQ1: How to design and implement a benchmarking platform for analyzing MVE games' performance on the Compute Continuum?

The continuum poses a unique challenge to game developers, due to the variety of available hardware configurations to choose from, as well as the constraints their applications pose. Previous work does exist in the field of Online MVE Games, as well as in the Computing Continuum sphere, however, a combination of these brings a new perspective into debate. The heterogeneous space of the Computing Continuum, with its wide spectrum of hardware configurations and performance options, makes it a challenge to get valid data about how software can perform in these environments.

RQ2: How can we evaluate an MVE game's performance in various environments on the Compute Continuum?

Evaluating the performance of the prototyped system is an essential goal of the project. Many metrics can be taken into account, and evaluating the relevant ones, such as bandwidth, network latency, or CPU resources, is the key challenge of this question. A comparison between potential performance gains and potential disadvantages (such as performance inconsistency) is needed to assess the system properly.

1.3 Thesis Contributions

Conceptual contribution 1: We designed real-world experiments to evaluate the performance of MVE games under various computing environments (Section 4).

Technical contribution 2: We extend the Continuum (8), a state-of-the-art framework for edge-computing research, to perform benchmarks on MVE games (Section 3).

Artifact contribution 3: We created open-source software for running these experiments, with all artifacts available at `https://github.com/Fugro-VictorG/continuum/tree/Continuum_Merge_Main`.

Experimental contribution 4:

1.4 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 chapter, we describe the concept of the compute continuum, including some of its associated computing models, and we explain how online games with MVEs usually work and present the ones we use in our experiments.

2.1 Compute Continuum

A basic overview of the Compute Continuum (9) is provided in Figure 2.1.

Cloud computing is a widely used computing paradigm, known for its on-demand resource availability, ease of scalability, and competitive pay-as-you-go pricing model. This enables users to create a large variety of servers and infrastructure services (e.g. storage, networks) while avoiding the capital expenses of acquiring and maintaining them (10).

A commonly used computing model that uses cloud computing has endpoints (component 1), which are relatively resource-constrained (mobile devices, IoT actuators, PCs), that offload their computational workloads to the cloud (component 3) for more efficient processing.

The cloud model is typically achieved by centralizing computing resources in the form of large data centers spread over the globe. This implies that, on average, there is a significant geographic separation between user devices and cloud clusters, which leads to latency in communication between these. This poses a roadblock for latency-constrained workloads used by recent technologies such as cloud gaming (11), Internet of Things (IoT) devices (12), Augmented Reality (AR) (13), or self-driving vehicles (14), among others, which are also constrained in processing power, and could effectively use computation offloading.

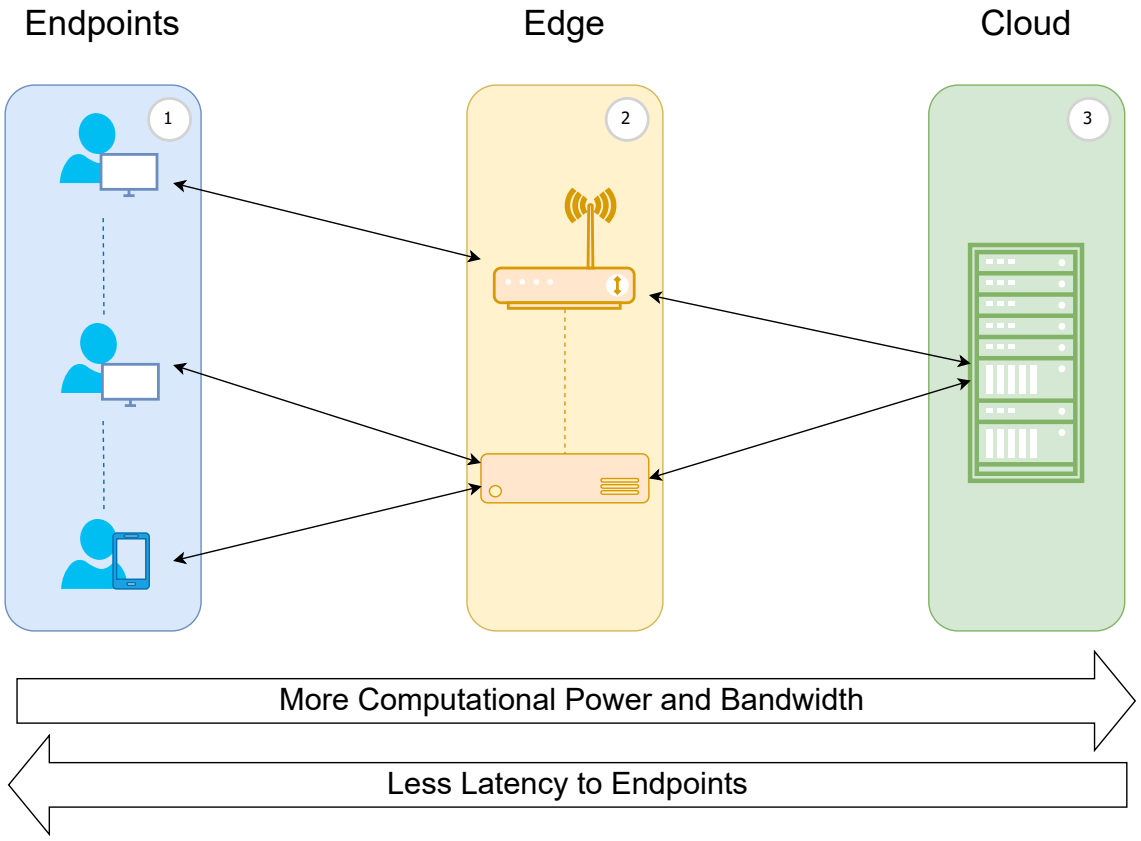


Figure 2.1: Compute Continuum Theoretical Diagram.

To satisfy these low latency requirements, the paradigm of edge computing brings computational resources, storage, and networking closer to the "edge" of the network, near the data sources and end-users (component 2). The proximity to the user devices results in faster response times, as well as satisfying potential privacy constraints by keeping data close to the source. The concept of edge computing can be traced back to the 90s when the concept of Content Delivery Networks (CDNs) was introduced. They are comprised of nodes close to the user, on the edge of the network, which cache web content, among other uses (15).

With edge computing situated as a middle point between the end user and the cloud infrastructure, it follows that applications can spread their workloads across a spectrum of cloud, edge, and end devices, depending on their requirements, that we can call the **compute continuum**. However, this new space lacks the standardized practices that already exist, for example, in the case of cloud computing. There can be many factors to take into account for software developers or infrastructure providers when it comes to developing or supporting continuum-based applications.

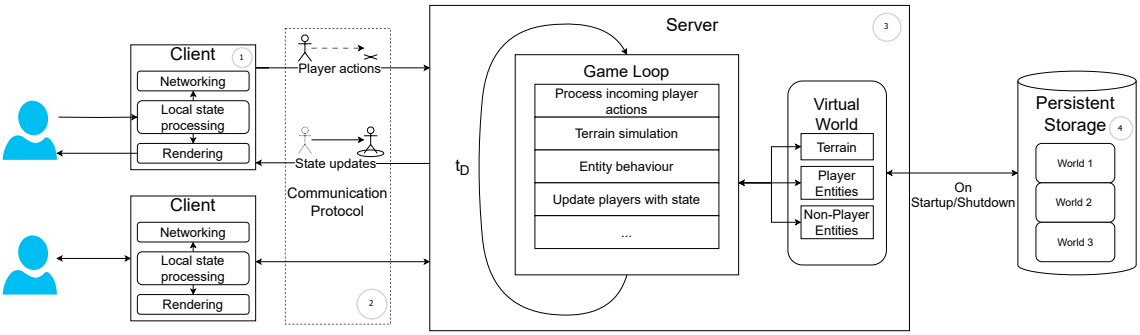


Figure 2.2: A System Model for Modifiable Virtual Environments (MVEs).

Fog computing is a computing model that brings cloud computing's capabilities to the edge, to networks of devices such as routers and gateways in closer proximity to the endpoints. Its core principles involve low latency and widespread geographical distribution with many heterogeneous nodes.

Mist computing can be seen as an extension of fog computing, pushing computation even further to the edge, involving end devices, such as sensors which previously only handled data acquisition. This leads to an autonomous, self-aware system, based on peer-to-peer communication between its devices (16, 17). While introducing complexity by having the endpoints be aware of each other and the overall state of the system, by bringing computation to the extreme edge of the network latency is minimized.

To allow researchers and developers to better understand the benefits and disadvantages of the compute continuum, a benchmarking framework (8) was created based on the reference architecture of Matthijs et al. (9). In our work, we expand this system, the *Continuum Benchmarking Framework*, to create and run our experiments.

2.2 Online MVE Games

This work commonly references games with modifiable environments. A server-client multiplayer system is a common way for online games to be implemented. Games with Modifiable Virtual Environments usually follow this pattern as well. Figure 2.2 shows a system model for MVEs (18). Each player connects to a centralized server using a game client on their local machine. Typically servers are hosted in remote data centers operated by third parties, for cost efficiency, and reliability, and to reduce chances of downtime. Depending on the need, however, one may choose to simply deploy the server on a personal machine, or use a server-as-a-service solution such as Microsoft's Minecraft Realms, which handles server hosting and world storage for a monthly fee.

The game client (component 1) receives inputs from the player and outputs rendered video, based on the current state of the game. Movements and other inputs from the player speculatively modify the state locally on the client, before the client notifies the server of the changes. The decisive source of truth for the game's state comes from the server, and for a player's perspective to stay synchronized with the rest, the client should receive continuous state updates from the server.

The client and server instances communicate using a common established communication protocol for the game (component 2). This also allows clients to have different implementations. For example, the game Minecraft has seen many community-created adaptations and modifications in the form of different server and client implementations, which all use the same communication protocol.

The game server (component 3) handles the game loop, which includes receiving the players' actions, processing them, and simulating all dynamic components of the environment, such as the non-player entities' actions, terrain physics, and world lighting. These operations are applied to the current state stored in-memory on the server (component 4), which results in a new state that is then sent back to all the players. One iteration of the game loop can be called a *tick*. One important metric in analyzing server performance would be to observe the *tick rate*, which is the amount of ticks processed in a second. Since the server is responsible for keeping the players' state up to date, a low tick rate could mean that the server is overloaded and cannot process the game loop fast enough. This leads to the player observing delays between inputs and their effects in-game, or *stuttering* due to the server's state being inconsistent with the client's.

Generally, an MVE game server would have a tick rate of at least 20Hz to provide the players with a smooth experience. Moreover, a dip in the server tick rate may not necessarily always lead to an effect on player experience. Depending on the implementation, game clients have methods of state prediction (19, 20), or other more complex ways of compensation (21, 22).

Once the server is shut down, the state should be saved to some form of persistent storage (component 5) for the changes to the world and players to be saved. On start-up, the server can also retrieve the needed state from this means of storage.

PaperMC (23) is a modified version of the Minecraft server, able to support the use of third-party server modifications. It is also very widely used by popular server providers as the recommended server choice for Minecraft.

Opencraft is an open-source server made for MVE game research, based on the code of Glowstone - an open-source variant of the Minecraft game server. Its main addition to

Glowstone is represented by the Dynamically Managed Consistency Units, or Dyconits (5). This is a system aimed at reducing the resource usage of the server and therefore allowing it to support more players simultaneously. This is done by permitting some level of inconsistency between the game clients and the global state of the game server. The server would send only a subset of the global state to clients, depending on the player's *interests*. For example, a player could receive regular updates on data about the nearby areas and players and less consistent updates on far-away players and areas.

In summary, we introduced in this chapter the compute continuum and the Continuum Benchmarking Framework. We also explained what online MVEs are and which ones we will be using for our experiments.

3

Integration of Continuum and MVEs

In this chapter, we first present an overview of the Continuum Framework (Section 3.1). Then, in Section 3.2 we explain the metrics we are interested in analyzing, and how they are collected. Section 3.3 presents the applications under test and describes how we implemented our testing workloads. Finally, in Section 3.4, we describe the changes we made to the Continuum Framework itself to allow our benchmark.

3.1 Continuum Overview

Our benchmark is an extension of the Continuum Framework (8), which adds new important workloads and metrics for MVEs and modifies Continuum's design to support these changes.

The Continuum Framework's input has the form of a configuration file (component 1, Figure 3.1), in which the Continuum user selects the parameters of the desired infrastructure, software deployment, resource management options, and the application to be benchmarked with any relevant parameters it offers (see Section 3.3).

Concerning the infrastructure options (component 2, Figure 3.1), at the highest level, the framework emulates Virtual Machines (VMs) or *nodes*, as they are referred to in the software. They are either deployed locally using the QEMU emulator supported by KVM and Libvirt, or on the cloud, using Google Cloud Platform's Compute Engine VMs or Amazon Web Services EC2 instances. The VMs' performance can be configured in CPU cores, memory size, and disk read/write speeds when emulating locally or by the available VM/instance tiers offered by the cloud providers.

Another key component of infrastructure provision is having a highly configurable network setup. The framework offers three levels of nodes for which network parameters

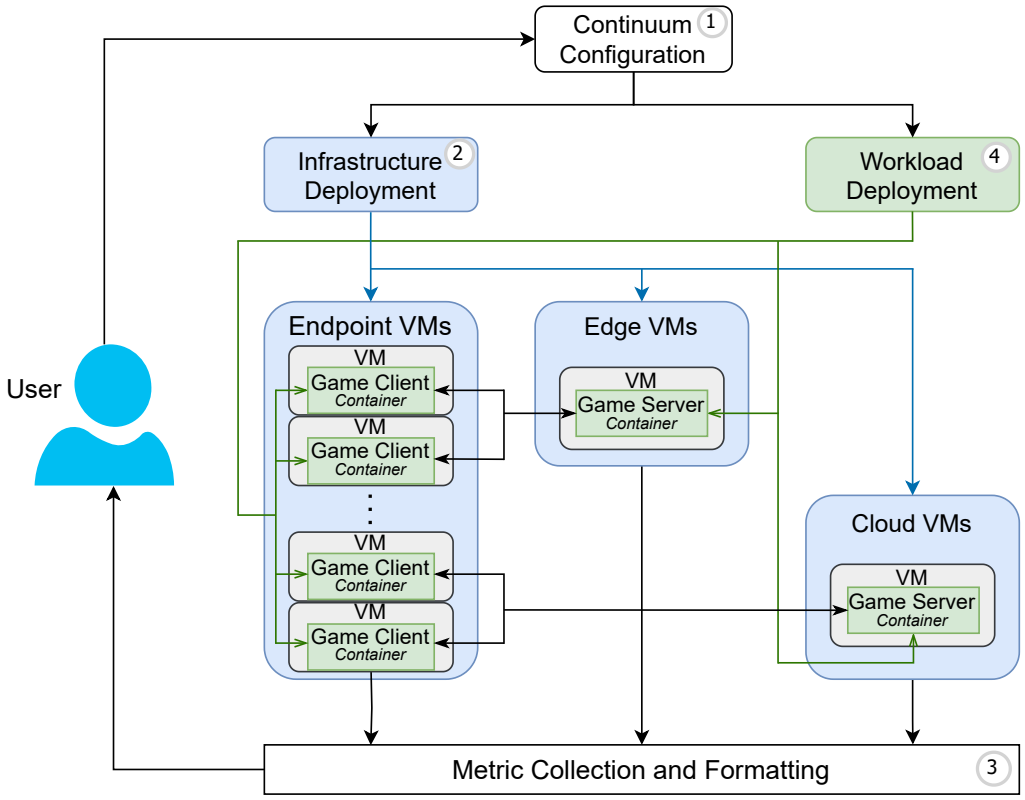


Figure 3.1: A system model for the Continuum Benchmarking Framework.

can be changed - endpoint, edge, and cloud. These three levels are connected by an emulated network, which is configurable for each inter-level connection in throughput and latency. The same network settings for edge-to-edge and cloud-to-cloud nodes can also be configured.

After the infrastructure is provisioned, the framework uses Ansible Playbooks to install prerequisite software for the benchmark (component 3). Optionally, resource managers Kubernetes or KubeEdge can be installed. On top of this, OpenFaaS can also be used for testing serverless functions.

Continuum also supports the usage of the MQTT protocol with the Eclipse Mosquitto message broker, for lightweight and scalable messaging appropriate for lower-power devices such as endpoint and edge nodes. Furthermore, monitoring and visualization applications Prometheus and Grafana can optionally be installed also for metric collection and visualization.

The application to test is then deployed (component 4). The endpoint nodes use Docker

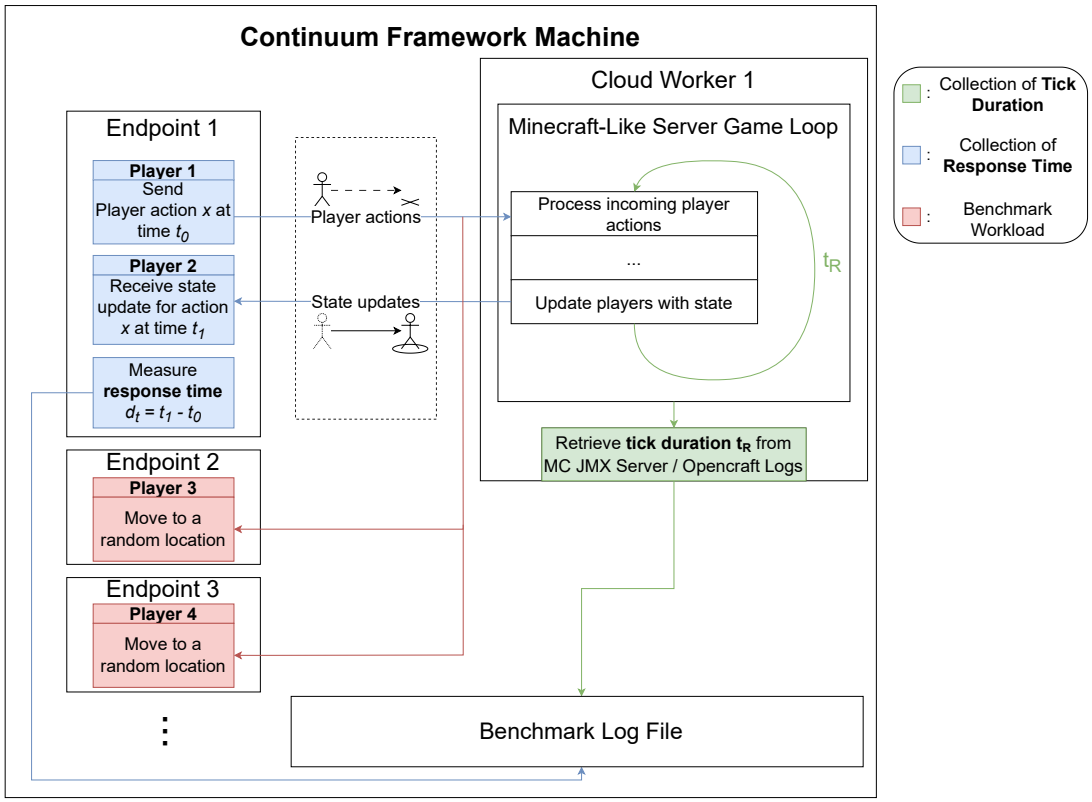


Figure 3.2: Overview of metric collection

containers for this purpose, while the *worker* nodes (cloud and edge nodes) offer flexible deployment means via Ansible files, allowing the use of Kubernetes, Docker containers, or other custom solutions supported by the application.

Once application execution is finished, Continuum aggregates logs of the worker and endpoint nodes, usually comprised of data from the standard output and errors. From these logs, user-defined Python methods gather and format the metrics desired for the application for displaying or further processing.

3.2 Metrics

In Figure 3.2, you can see an overview of our metric collection in the context of our MVE game. During our benchmarks, we collect the following metrics for analysis:

3.2.1 Tick duration

A commonly used metric for evaluating server performance is the tick rate, which defines how many times per second the server can process the game loop. Its importance is due to the potential negative effects on the player's experience if the tick rate doesn't exceed a certain threshold. The alternative metric for tick-related measurements is the tick duration, which represents the length of time taken to process a tick. During our testing, we obtain the tick duration metric in different ways for each type of server:

- The Opencraft server includes a particularly useful feature: tick-related measurements can be logged to a text file, exposing data such as the duration of a whole tick, the duration of the world processing part of the game loop, or the amount of time spent sending messages to clients. We retrieve the duration of each tick from this log file.

- For our tests that use PaperMC as the server, we had to use a different approach to obtain this metric. The vanilla Minecraft server can expose the tick duration by using *Java Management Extensions* (JMX). JMX is a built-in Java technology that allows remote monitoring of JVM-based applications. By enabling JMX monitoring in the server's properties, the server automatically exposes the tick duration in milliseconds. We use a tool created by a fellow student to specifically extract the tick durations from a game server's JMX agent and store them in a text file. We use a tool created by a fellow student (24) to extract the tick durations from a JMX server and store them in a text file.

3.2.2 End-to-end response time

This metric represents the time taken from the moment a player's action is taken until the state update (which includes the player's action) is received by another player from the game server. This is relevant, because for the effects of the action to be observed by the player, it must first be processed by the server, and the state update sent back to the game client. If the time taken for this is too long, the player will only see his, and other's actions after a noticeable delay.

We can expect the value of this metric to be the result of 3 main factors:

1. The time taken by the state update to traverse the network between the client and server. We refer to this value as the *network latency*, and it plays a large part in affecting this metric, as it can be highly variable and based on multiple factors, such

as the geographic location of the server and client, or type of connection of the client (WiFi, Ethernet, cellular). Studies have also clearly shown that latency has a high impact on an online game's Quality of Experience (25).

2. The time taken by the game server to process and send back a game state update containing the effect of the player's action. Once the server receives the player's action, the effects of their action must be applied to the game state. The duration of this may vary depending on the complexity of the action and how it affects other parts of the state.

   Typical MVE games of the kind we experiment with in this work send out updates to all clients once every 50 milliseconds, so the server's overhead in response time depends on the timing of the player's action in relation to the next update to be sent out by the server. For example, if the processing time for a player's action is 10ms, then the server could take from 10 up to 60ms to send an update back to the client if the player's action gets received right before a state update is sent.

3. The time taken by the client to process the incoming state update and output this information to the player. While a regular game client would be responsible for interpreting the state update into visual output for the player, in this work we use only simulated clients which do not provide graphical output. This results in the client's overhead only being the time taken for internally processing the data.

How this measurement is recorded is shown in Figure 3.2, and it is represented by the difference $\Delta t = t_2 - t_1$. Parameter t_1 represents the time at which we issue the bot's *block digging* and *block placement* commands, and t_2 the time at which the "*block updated*" event is fired by the Mineflayer library that corresponds with the first issued action.

3.3 Workloads

For our experimental evaluation, we simulate players using the Mineflayer API (26), which emulates a game client by communicating with the server via the Minecraft protocol, used both by PaperMC and Opencraft.

A player simulated by the software can be referred to as a *bot*. The behavior of these bots is programmatically defined with Mineflayer. This, along with the selected or generated world, represents our workloads used in the benchmark.

We use two main emulation components in our workloads:

3.3.1 Player movement component

Several bots independently pick random positions to go inside a square area of a pre-defined size around the world's *spawn point*, where players first appear when entering the world. We implemented two "patterns", *Linear Join* and *Fixed Join*, in which bots connect to the server, to compare different kinds of server use. These were largely inspired by the Yardstick (27) benchmark's workloads.

Linear Join: Every 120 seconds, 10 bots connect to the server, until the defined total number is reached.

Fixed Join: Every second, 5 bots join the server, until the defined total number is reached.

3.3.2 Terrain modification component

Two bots remain static in the same area, with one repeatedly placing and removing a block, while the other records the time when the mentioned action can be noticed. The purpose of this workload is to measure the time taken for the action of the first bot to be noticed by the observing bot, which represents **Metric 2** (§3.2.2). The number of roaming bots, as well as the number of repetitions of block placement/removal is configurable. For example, if we choose to do 100 repetitions, the bot will alternate placing and removing a block, having in total 50 placements and 50 removals.

We encountered various problems while testing and benchmarking. The most persistent issues are bots losing connection to the server, or getting stuck when moving, either due to impassible terrain or Mineflayer-dependent issues with finding a path to the next waypoint. To make our workloads more resilient to such interruptions, we use the following methods for recovering disconnected or immobile bots:

1. When a bot's connection to the server ends, it will always attempt to reconnect and resume its previous actions, until the server is shut down after the benchmark ends.

2. If a moving bot remains immobile for longer than 10 seconds, it will attempt to *respawn* - which resets its position to the world spawn point and its next movement waypoint. If the bot is still immobile, it will attempt to reconnect to the server after 10 respawn retries.

3.4 Integration

When integrating an application with Continuum, usually two software components must be implemented, a worker and an endpoint. When using Continuum with an MVE as the application to be tested, the worker is represented by the game server and the endpoint by the bot implementation.

For our server, we used two different options, the Paper and Opencraft servers. They are implemented as Docker images, modified to use the same world, and to support thee collection of our metrics (described in §3.2 and §3.3).

The bots are implemented as JavaScript files, mainly using the Mineflayer API. We provide two different implementations for bot behavior as mentioned in Section 3.3.

To integrate these with the Continuum Framework, we use Docker to create images for the bot and the server. The bot's image only installs dependencies of Mineflayer and executes the script.

The server's image is built using an Ansible Playbook, which aggregates the server itself, its configuration, and optionally the selected world into the same directory. An instruction that prints the contents of the server logs is then added, before creating the Dockerfile of the server image, so that metrics gathered by Opencraft can be collected by Continuum after server execution is finished.

To use our server as a Continuum cloud worker, we've decided to use Kubernetes as the resource manager for two main reasons: firstly, it simplified the integration of other server implementations, as the previous integration of Opencraft already had introduced manifest files for creating the needed resources for a Minecraft-like server, and secondly, since the purpose of Continuum is emulating cloud and edge environments, it would be appropriate to use a standard cloud method of resource management and container orchestration.

For starting the worker, we use an Ansible file to define the steps for deploying our server. For Opencraft, this file starts by creating a YAML Kubernetes manifest file, which describes two K8s resources. The first one is a Pod which holds a container with the server's image, having the CPU and memory resources allocated to the worker node, as stated in the Continuum configuration. The second resource is a NodePort service, which maps the port used by the endpoints for communicating with the worker to the Minecraft protocol port that is exposed by the server image. Continuum assigns consecutive ports to worker nodes when multiple are used, making the port mapping necessary. If multiple servers are used, our Ansible Playbook then duplicates this file and replaces placeholders

in the necessary fields to accommodate different unique servers. Finally, the Kubernetes resources are created via a kubectl create command.

So far in this section, we have discussed the typical elements Continuum requires for integrating a new application to be used in the framework. Additionally, our integration requires a few modifications to the core code flow of Continuum. These are needed mainly because we seek to test applications that are not very usual for cloud workloads and have different requirements. For example, a workload based on an online game does not have a clearly defined start and end compared to, for example, a machine learning workload. Therefore, the following changes are applied to the system:

- During startup, MVEs load assets and other resources from storage into memory. This process typically takes tens of seconds to complete. During this time, players cannot use the system, because it is unavailable. To reflect this in Continuum, we only indicate the service to be "Ready" when the game's initialization process is done. We do this by scanning the server log for specific keywords that indicate that loading has been completed. Only when these keywords are detected is the service marked as ready.

- Due to the real-time interactive nature of MVEs, clients and servers communicate through direct connections, instead of through common distributed systems approaches for communication such as pub/sub systems. Therefore, we modify Continuum to support direct connections between entities/nodes.

- In contrast to other common application types (e.g., data processing), MVEs are online persistent worlds that are always available. Therefore, workloads do not necessarily have a predefined start and end point. To support this, we allow workloads to define an arbitrary duration for which they are active. After this specified amount of time, the workload is considered completed, and the server is shut down.

- As this is the default and simplest method of obtaining results from Continuum Kubernetes benchmarks, we chose to retrieve our measured values from the logs of the Kubernetes pod our server lies in. To accommodate the size of the Opencraft server logs, which contain multiple lines of logs for each tick, we had to increase the default Kubernetes log size by adding an extra config file to the K8s cluster.

3.5 Technical limitations

We've encountered various difficulties with testing the Opencraft game server. The most problematic issue is the unexpected behavior of the bots when running on an Opencraft server. This behavior ranges from visual discrepancies, like bots walking through terrain, to bots suddenly disconnecting or dying. Furthermore, mitigation solutions like respawning or reconnecting the bots would not function, seemingly due to incompatibility between the Mineflayer API and Opencraft. Another issue for which no clear solution was found was that the containerized version of Opencraft would fail to load existing game worlds in most cases. Instead, on container restart or recreation the server would generate new world files to overwrite the existing ones. This would occur for both worlds generated by the server and for custom worlds that were packaged with the container. The only correlation found for successfully loaded worlds was that they were generated by the server and the server was running with them for a longer time period (at least 30 minutes). We ultimately decided to use PaperMC in place of Opencraft as the server for most of our experiments, as these issues were not encountered while using it.

4

Experiments

In this chapter, we present our experiments discuss the main findings, and analyze the results and data plots. We begin with an overview of our experimental environment and describe the 3 experiments we ran (Section 4.1). Then, in Section 4.2 we present the main conclusions of our experimental results. Finally, Sections 4.3 - 4.5 show our plotted results and provide a more in-depth analysis for each experiment.

4.1 Overview of experiments

To evaluate the use of the Continuum Framework as a testing platform for online MVE games, we ran 3 experiments, to examine the effects of 3 parameters commonly known to impact an MVE game server's performance.

We run our experiments on a node of the AtLarge computing cluster, a machine that uses two 10-core Intel Xeon Silver 4210R CPUs and 256GiB of memory. We use the Continuum Framework to deploy our game server and clients on individual VMs.

Table 4.1 provides an overview of our experiments, their changing parameters, and workloads. The system has a mainly static configuration throughout each experiment, except for the "Parameter", as the purpose of each experiment is to determine their influence on the server's performance and the players' Quality of Experience.

We define the experiments' workload to be composed of the following elements:

1. The server, which represents the implementation of the game server used in the experiment.

2. The bot's behavior, which represents which Mineflayer bot implementation is used in the experiment. The possible options are "R" and "W+R", which refer to Response

Table 4.1: Overview of experiments

Section	Parameter	Workload			
		Server	Behavior	Join Strategy	Bot number
§4.3	Network latency	Opencraft, PaperMC	R	Default	2
§4.4	Number of players	PaperMC	W+R	LinearJoin	20-80
§4.5	CPU Cores	PaperMC	W+R	LinearJoin	40

time measuring, and Walk + Response time measuring respectively. These behaviors are explained in Section 3.3.

3. The "Join Strategy" - this refers to how the bots will be joining the server (3.3), and how the end of the benchmark is determined (3.4).

4. The Bot number, which represents the number of players used in a particular experiment.

All of our experiments use a single server to which Mineflayer bot clients connect. Our Opencraft and Minecraft server configurations, used in our server images, can be found in Appendix ?, they are unchanged for all of the experiments. Further details will follow for specific experiment setups.

Continuum allows the configuration of the core count for each VM tier (cloud, edge and endpoint). Internally, these represent the count of KVM vCPUs assigned to each of these VMs. These should correspond to a physical CPU core in computational resources, but the virtualization scheduler decides which core is used at a particular time per VM, to maximize efficiency.

For our experiment on response time, we used a purpose-built script, described in Section 3.3.2, to deploy 2 bots for measuring the response time, a metric described in Section 3.2.2. For this experiment, we use one Continuum endpoint node and a cloud node. We test both with an Opencraft server and a PaperMC server. The server's Kubernetes pod was assigned 8 CPU cores and 8.5GB of RAM. Since the response time measuring bots have minimal movement, and mainly just place and break a block repeatedly, we host the two bots on a single VM. This VM is assigned 1GB of memory and one CPU core. Furthermore, the relative CPU quota of this core is set to 0.5. This means Continuum sets

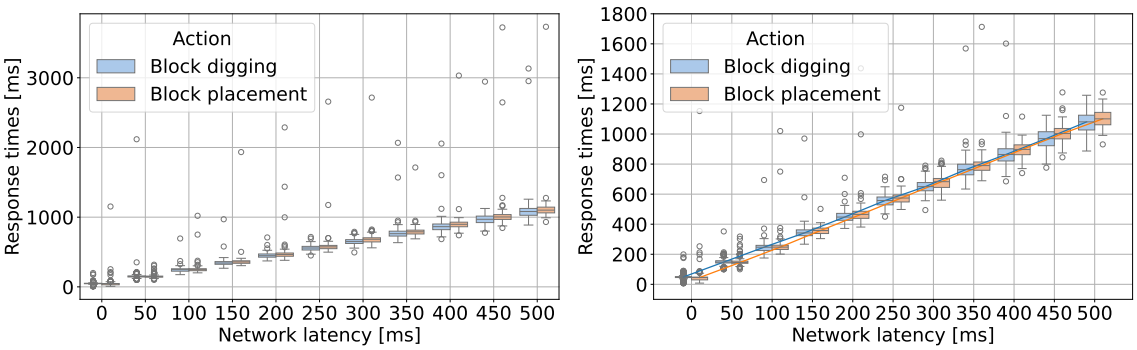
the Linux KVM to only use the CPU half the time. The experiment is comprised of 11 tests, with each one incrementing the network latency and latency variation between the VMs of the bots and the server. We use 50ms increments for the network latency and also apply a 10% latency variation, i.e., for a latency of 100ms, a transmitted network packet could take $100 \pm 10ms$ to be received. Except for the first test, which emulates no extra network between VMs, we use a network throughput of 7.2 Mbits per second for all tests. The bots take a total of 400 measurements in each test, which means 200 blocks placed and 200 blocks dug.

For our player impact experiment, we have multiple bots with a movement-based workload, described in Section 3.3.1. We use as many Continuum endpoint VMs as we have bots, that individually connect to a server on a cloud node. We use PaperMC as the server under test for this experiment. Each bot's VM was assigned one CPU core with 1GB of memory. The server uses 6 CPU cores and 7.5GB of memory. These values were decided so that bots would get minimal CPU allocation while still performing normally, without getting disconnected by the server. We perform 4 tests, in which we gradually increase the total number of players from 20 to 80, in intervals of 20. Throughout a test, the bots connect to the server using the *Linear Join* strategy, which works as follows: we begin with 10 bots at the start and have 10 more bots join every 2 minutes until the total number of players is reached. After two more minutes, the test is finished.

Regarding our CPU impact experiment, we use multiple Continuum endpoint VMs with a single bot on each, using the movement-based behavior described in Section 3.3.1, and a single VM on a cloud node that contains the server. We only test the PaperMC server implementation for this experiment. For this experiment, we used 40 bots for each test. We assign each bot with one CPU core and 1GB of memory. The server's container is assigned 7.5GB of memory. We perform 8 tests, incrementing the number of CPU cores available to the server, ranging from 1 CPU core with 0.5 core quota to 7 cores. The bots connect the server using the same Linear Join strategy - every 2 minutes 10 players join the server until they reach 40 total players.

4.2 Main findings

MF1: Response time scales linearly with network latency. We observed in our experiment that the median response time can be calculated as such: $ResponseTime = 2 * NetworkLatency + SD$, where SD is the server delay for sending a response back



(a) Full plot with outliers.

(b) Zoomed-in plot.

Figure 4.1: Opencraft Server Response Times.

after an action from the player is received (see Section 3.2.2 for more details). For our case, the average SD is 49ms (Section 4.3).

MF2: Our system supports Our system supports fewer players than what previous studies show under similar conditions. While we can't conclusively assert the maximum number of players the game provided a good Quality of Experience for, we can affirm that under our experimental conditions, the game will become *overloaded* when more than 40 players are used in the test. This is very unexpected and does not match similar studies' results. We speculate this to be caused by CPU saturation of the system executing the test, due to too many players being simulated (Section 4.4).

MF3: Scaling server performance by increasing CPU core count offers limited benefits. While acceptable server performance heavily depends on the available CPU resources, in our experiments we reach a point where we see minimal improvement from adding cores, despite still experiencing outlying results (Section 4.5).

4.3 Impact of latency

Figures 4.1a, 4.1b and 4.2 show the results of our experiment which measured the time taken for a player's action to be observed from the other players' perspective. The first two refer to the tests run with the Opencraft server, the former being the plot of the full results, while the latter is a close-up version without all the extreme outliers. The third is our complete plot of the results of the tests run with the PaperMC server. This measurement can be seen on the vertical axis, in milliseconds. The horizontal axis indicates the latency between the game client's machine and the server's, as configured via Continuum. Not

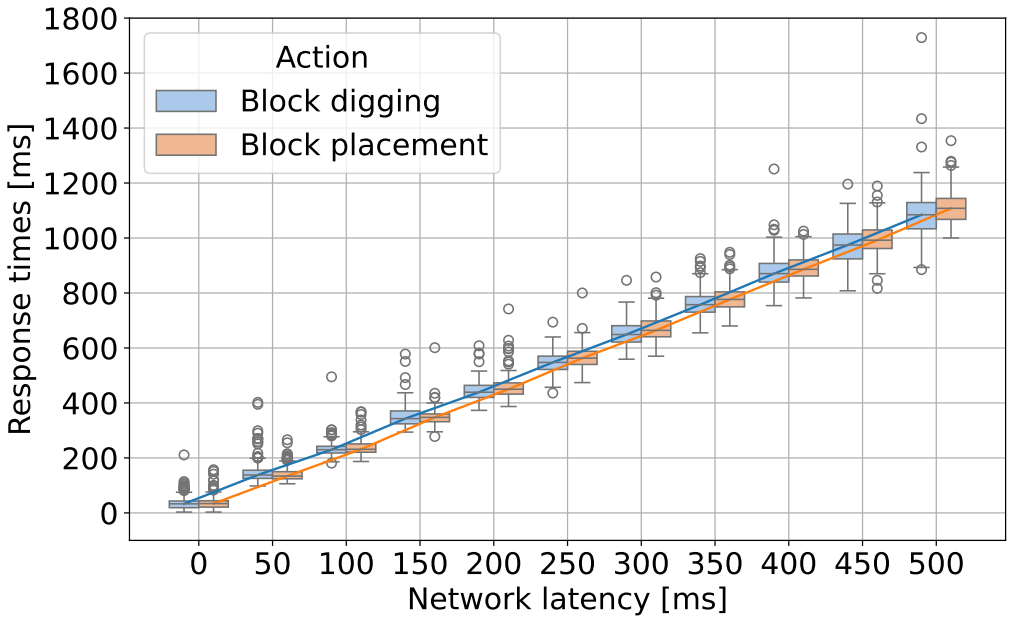
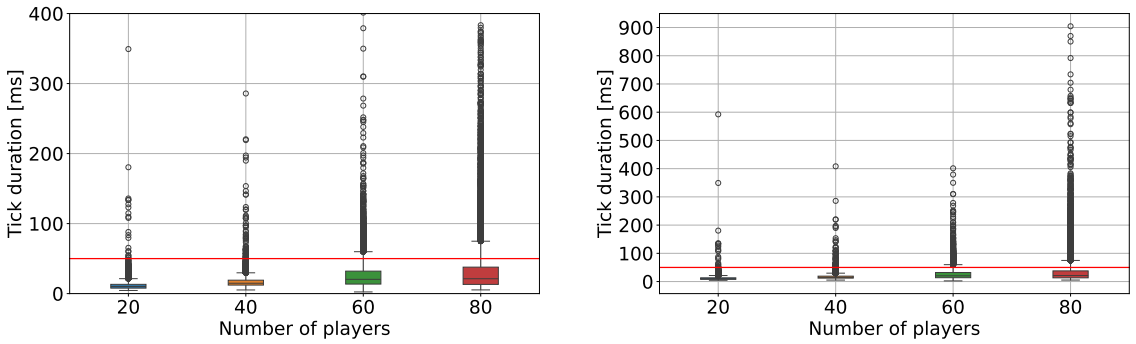


Figure 4.2: Minecraft Server Response Times.

mentioned in the figure, but still present in the experiment, is the 10% latency variation, as explained in Section 4.1.

Our results show a linear growth in response time: as the network latency increases, the median values increase in similar amounts for each step. We can see this in both Figure 4.1b for Opencraft and 4.2 for Paper, where we have plotted a line between the median values of the response time. We can also notice the induced latency variability - the boxes and whiskers increase in size along with the network latency. Both actions, block digging and block placement, have similar response times. Data from both server implementations appears to have the same range of values and has similar variability, except for their outliers.

Regarding our Opencraft results, we can see some extreme outliers for most boxes, with some maximum values over one second higher than the rest. This does not happen nearly at all when running the same experiment on a Paper server, as the highest outlier there is around 650ms higher than the median. We therefore conclude that the cause of these high outliers lies in the implementation of Opencraft or Glowstone, the underlying server Opencraft is based on. However, for both servers, we have minor outliers that are more numerous and dense, positioned around 0-100ms higher than the tops of the boxes. This is especially noticeable for the lower latencies when the latency variation is low. As latency variation increases along with the IQR of our data, these outliers appear to be less frequent,



(a) Zoomed-in plot.

(b) Full plot with outliers

Figure 4.3: Tick duration in Endpoint Scaling Experiment.

and we speculate that they become more uniformly distributed with the rest of the data. These outliers could have a noticeable impact on player experience if many of them occur in a short span of time.

We cannot accurately discern the cause of these outliers. However, since these outliers also seem to occur outside of Continuum experiments in tests using the same Mineflayer bot behavior, we speculate the source of these is either the internals of the Mineflayer API, our implementation of the response time measurement, or the virtualization process of Continuum.

We also compared the response times obtained from our experiment with the Continuum Framework on the AtLarge node with the ones obtained when testing the workload on a local setup. When running this scenario, where the bots connect to a server running on a Docker container, all on the same machine, we observed lower median values for response times (by approximately 10-15ms) compared to the median results obtained from the test with zero network latency (49ms) (Figure 4.1b). This test ran on Continuum and was configured to not use any network emulation like the rest in the figure. Therefore, we believe this overhead comes from the inner workings of the Continuum Framework and consequently, all experiments would be affected by it.

4.4 Number of supported players

For our experiment on the impact of the number of players on server performance, Figure 4.3 shows the tick duration over 4 tests with an increasing number of total players (from 20 to 80 players). The players join in waves of 10 every two minutes throughout these tests, starting at 10 players and ending at the specified number of each test. To make

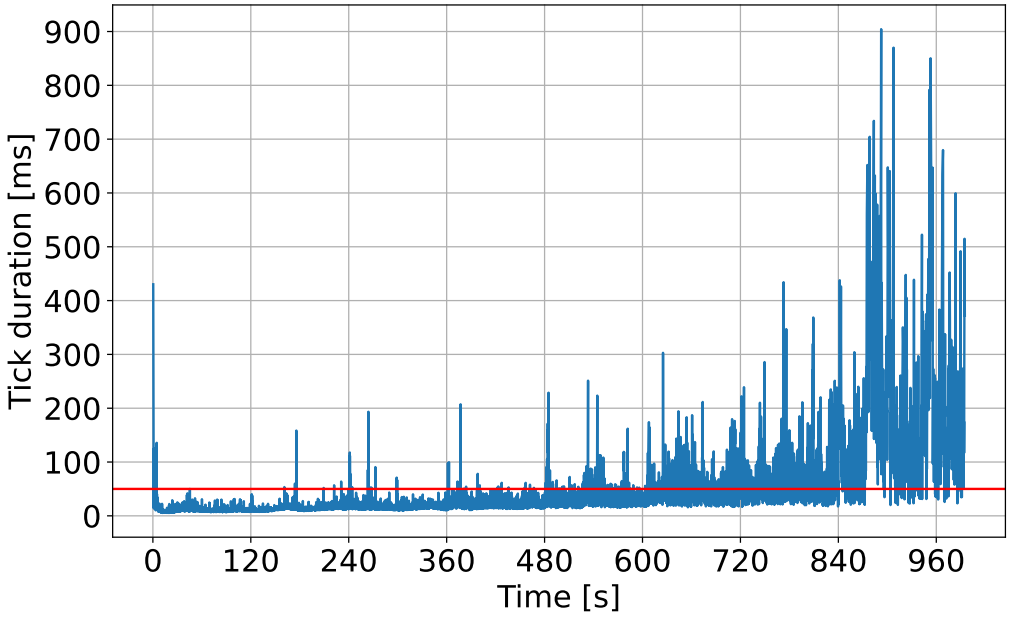


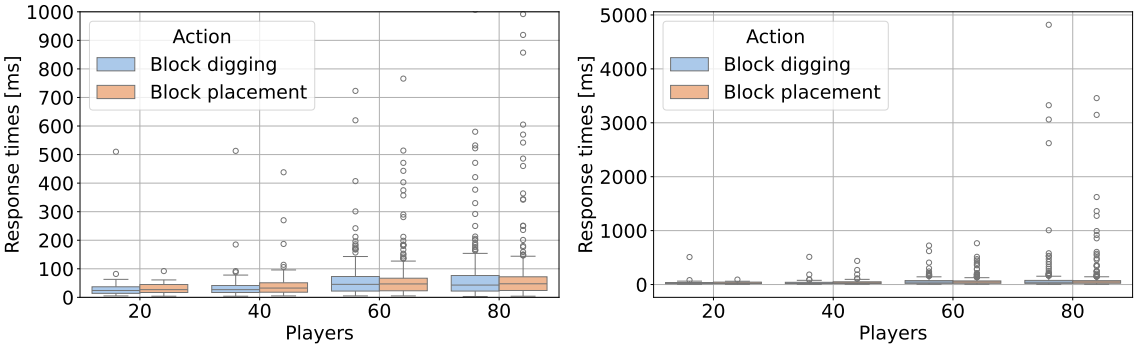
Figure 4.4: Tick duration over time in the test run from 10 to 80 players.

the majority of the data more visible, we cut off some of the higher outliers in Figure 4.3b. The tick duration distribution over time for our test with 80 players is shown in Figure 4.4. We also capture the response time metric for this experiment and show it in Figure 4.5a. A closeup of the data near the quartiles is presented in 4.5b.

Since the server's normal rate of tick transmission is 20 ticks per second, the maximum tick duration for normal operation is 50 ms. Values over this threshold delay the regular process. In our tick duration plots (Figures 4.3 and 4.4), we included a red horizontal line to indicate this 50ms mark. A few spread-out outliers over this threshold would not be problematic, as explained in Section 3.2.1. However, consistent values over 50ms in a short time will lead to inconsistent states across clients and a loss of Quality of Experience for the player.

While the mean and median values stay under the 50ms line, the whiskers of the boxes from the 60 and 80 players tests still cross our line, as we can see from our box plots on tick duration. A significant part of the measurements of the 60 and 80-player tests lie outside the 50ms boundary. Outliers are present in all tests, but they are much denser and reach extreme values for these tests.

From figure 4.4 with tick duration over time for the test with up to 80 players, we can see high values right at the start of the test, when player load is minimal at a count of 10 players. We assume these outliers are due to measurements starting before the cold start-up



(a) Zoomed-in plot.

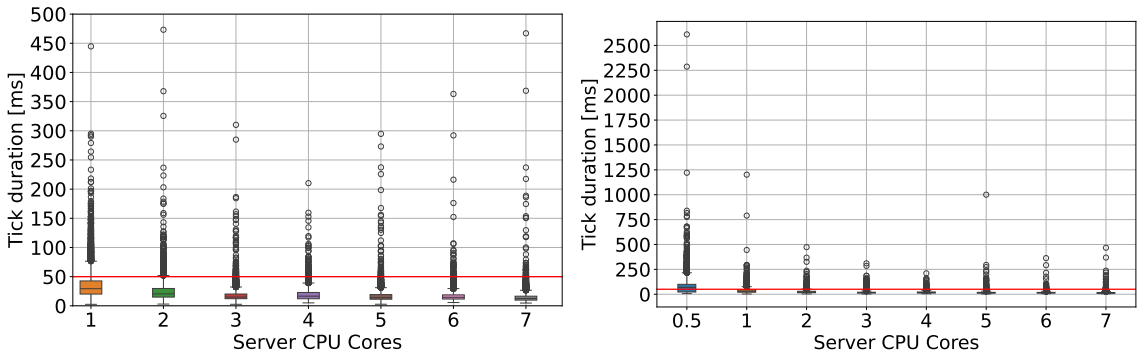
(b) Full plot with outliers

Figure 4.5: Response time in Endpoint Scaling Experiment.

of the server is finished, as metric collection starts as soon as the server is ready to receive connections. Since every test begins this way, we can deduce that for our experimental setup, all tests would suffer from a few outlying values related to the server's cold start-up. This is why we see some high outliers for all test runs, no matter the player count.

The Linear Join strategy used in this experiment makes 10 bots connect to the server every 120 seconds until the target number of bots is reached. The collection of the tick duration begins as soon as the bot containers are started. Still, after the bots' containers start-up and the connection time, we estimate an average delay of 5 seconds between the start of metric collection and the bots joining the server. With this information, we can observe some patterns that arise from bots joining the server from Figure 4.4. On the horizontal axis, we have ticks every 120 seconds to roughly estimate when more bots join the server. We notice a few spikes in tick duration around these moments, as well as an increase in the average value after a certain amount of bots is reached, such as the 480 and 600-second marks. Tick duration seems to consistently reach over 50ms after 500-600 seconds into the test, leading to a noticeable degradation in player experience. After the last wave of bots is added after 840 seconds, it appears that only a minor amount of measurements were under the acceptable 50ms threshold.

It is important to note studies have shown that CPU utilization is the main bottleneck of Minecraft-like services, such as the server under test in this experiment (27). Our server and bot clients run on the same physical machine, orchestrated by the Continuum Framework. This limits the number of players that can be simulated while still obtaining realistic results from the server, as when the number of players increases past a certain threshold, the availability of the CPU cores the server uses is significantly lowered. This is part of our reasoning for the dramatic change in performance when the player count surpasses 60 - the



(a) Zoomed-in plot.

(b) Full plot with outliers

Figure 4.6: Tick duration in CPU Impact Experiment.

bot processes demand more CPU time to be able to perform their workload, leaving less computational power for the server to handle the extra players. Additionally, this CPU insufficiency was observed to also affect the bots. During testing, Mineflayer bots have been seen having issues maintaining a connection to the server if they weren't assigned enough CPU resources. As bot numbers reached 60 and above, we started to see similar errors. Given that bots have to reconnect every time they lose connection, and during our 80-player test there were 49 disconnections all in the last 5 minutes of the test, we expect this to also have contributed to the poor server performance in that part of the test.

Our response time measurements (Figures 4.5a and 4.5b) show a similar performance picture. Our median response times start at around 25ms and end at 50ms on the highest amount of players. The variation of our results doesn't change very much either, the whiskers of our box plots staying under 100ms until we reach 60+ players, and at a maximum of 150ms. However, while we see few outliers for our 20 and 40-player tests, we have a significant increase in outlying values when adding more players, with some extreme values over 1000ms for the 80-player test. We speculate that lack of CPU availability is the cause of these extreme outliers, as we see similar results for our low CPU core number scenario in Section 4.5.

4.5 Impact of CPU resources

This section presents the results obtained from our experiment on the impact of CPU resources on the server's performance. Similarly to the experiment described in 4.4, we show our plotted tick duration and response time data in two figures each, for showing both the complete data, and for showcasing the greatest concentration of measurements

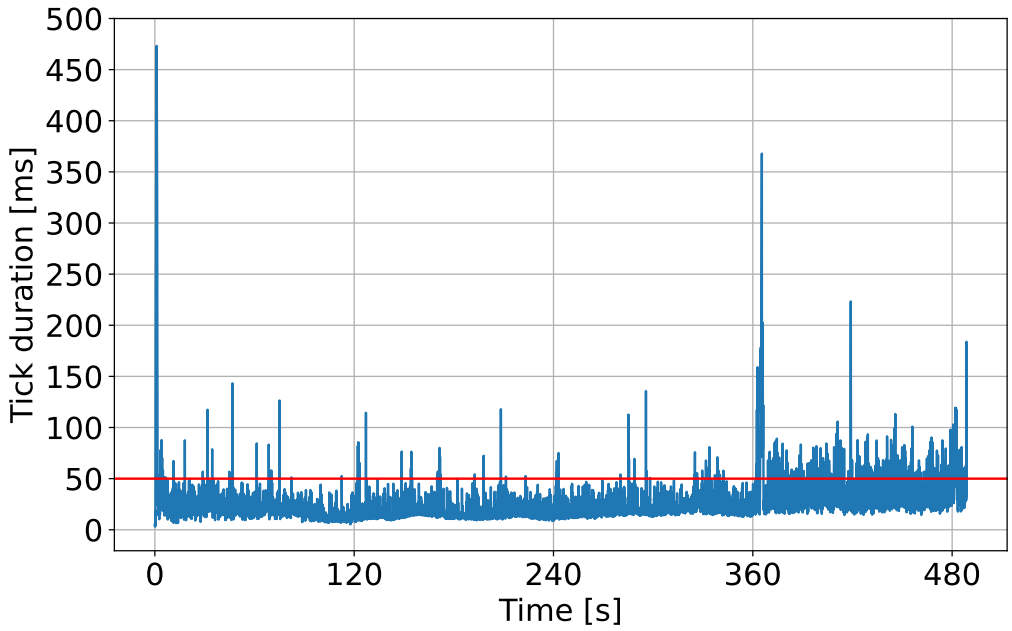
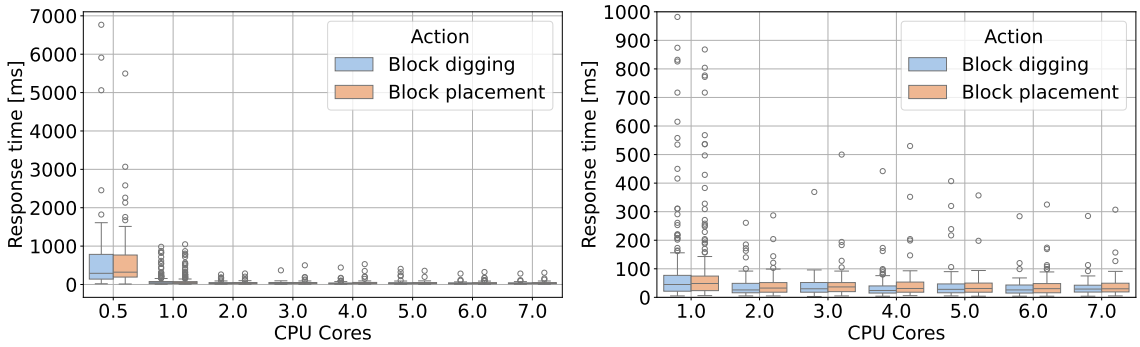


Figure 4.7: Tick duration over time in the test run with 2 CPU Cores.

separately. Figures 4.6a and 4.6b show the tick duration, and Figure 4.7 also shows a plot of the measurements from our 2-core test over time. Figures 4.8a and 4.8b show the response time. Tick duration plots also show a red horizontal line to indicate the 50ms mark.

As we can see from our tick duration plots, most test runs, except those with 0.5, 1, and 2 cores for the server, have their non-outlier values under 50ms. Our run, labeled as 0.5, with a single core being used only half of the time, behaved very differently than any other tested setup, with over half of the ticks taking significantly longer than 50ms to process. This would be inadequate for respecting the standard tick rate of 20 per second.

We observe a downward trend in the mean tick duration and the variability of the measurements as we increase the core count. However, we do not see further significant improvement in tick duration from 3 cores onwards. We see some variation in the size of the whiskers for our results between 3-7 cores, but it doesn't seem that consistent with the increase in core count. Furthermore, outliers are present for all our test runs. Extreme outliers for our higher core test runs can be attributed to cold starts, as explained in Section 4.4. Nevertheless, we still see dense concentrations of outliers for all our test runs. These large concentrations scale down from up to 200ms for our 1-core test to about 100ms for our 6-core test, after which the differences become negligible. Still, portions of them consistently appear over the 50ms line for all our tests.



(a) Plot of the complete data.

(b) Zoomed-in plot.

Figure 4.8: Response time in CPU Impact Experiment.

From our figure of tick duration over time for our run with 2 CPU cores, we can see many spikes of values over 50ms, as well as more constant values above this threshold after the 360 second mark. We associate the initial spike with the cold start of the system, similarly to the one seen in Figure 4.4 from the player impact experiment. Also like in the aforementioned figure, our horizontal axis ticks are every 120 seconds, which is roughly the moment when a player wave joins the server. We see spikes of tick duration on these intervals where players join, but we also see the average value of the metric increase as the bottom of the blue line gets slightly higher with time. On the last wave of players joining we have a bigger spike in comparison with the previous and we start seeing values over 50ms consistently. While this is partly due to the server only having two cores assigned to it for this test, we expect there is another factor contributing to the drastic difference in performance after the 360 second mark. One reason we speculate is the same as for our player impact experiment with a high number of players - that we observe the combined CPU demand of the bots and the server saturate the usage of our testing platform's CPUs, leading to a somewhat unexpected increase in our metric.

For our latency results, we observe that most of our tests except for our 0.5 and 1 core tests provide similar results. In the test with a single core used half the time the median response time is above 200ms, having box whiskers above 1500ms and outliers up to 6000-7000ms. Our 1 core test outliers can be seen frequently in the 200-1000ms range, with whiskers above 150ms. We consider these results inadequate for a good player experience and deduce that for our workload the CPU resources provided are insufficient.

For the tests with 2 or more cores, we see a very slight downward trend for median values and whiskers, but nonetheless a minimal overall difference in response times when scaling core count. Additionally, the outliers for these tests seem somewhat inconsistent with CPU

resources, as we have lower outlier values for our 2-core test compared with higher core count tests.

To summarize, through our experimentation we found a linear relationship between response time and network latency. Also, we notice that server performance scales poorly when increasing the computational capacity of the machine it's running on. Finally, we evidenced a non-linear association between the player count and the server's performance (represented by the tick rate).

5

Conclusion

To this day, online MVE games maintain their appeal and continue to grow in complexity. Emerging services such as edge computing could be useful tools in supporting this growth and helping scale these online games to more players. There is, therefore, a need to deduce which of these deployment methods are generally beneficial. This work sought to help towards this goal by showcasing the design of a benchmark for a popular MVE online game.

5.1 Answering Research Questions

In Section 1.2, we established 2 research questions to answer in this project.

RQ1: How to design and implement a benchmarking platform for analyzing MVE games' performance on the computing continuum?

Since our benchmark is heavily reliant on the Continuum framework, our design was inspired by existing benchmarks created for the framework, and we attempted to keep a similar structure for our benchmark where possible. Nonetheless, the core challenge in our implementation was successfully interconnecting Continuum with our application and our metric collection mechanisms. Section 3 addresses this question by presenting a complete overview of our benchmark's design and providing implementation details.

RQ2: How can we evaluate an MVE game's performance in various environments in the Computing Continuum?

The key aspects of the experimentation process are determining what metrics to collect for analysis, creating realistic workloads, and visualizing and analyzing our

results. Our metrics were chosen to easily compare our system's performance to results from similar studies, and with our experiments, we attempted to reproduce a variety of environments to observe their effects on the game. Our main findings as as follows:

MF1: Response time scales linearly with network latency.

MF2: Our system supports fewer players than what previous studies show under similar conditions.

MF3: Scaling server performance by increasing CPU core count offers limited benefits.

Details about our experiments and the analysis of their results can be found in Section 4.

5.2 Limitations and Future Work

The first limitation of our benchmark, which is noticeable in our experimentation, is our unexpectedly low number of supported players. Further investigation is needed to confirm the cause and whether there is a noticeable performance overhead to using Continuum with applications like MVEs. A first suggestion for this issue would be to run the same experiment on a system with more CPU resources, as this would rule out the speculated cause of this discrepancy, as mentioned in Section 4.2.

Additionally, even in the current form of our system, there is still much more information that can be obtained by using different Continuum configurations. For example, an experiment that targets the variability of network bandwidth or latency.

Another direction for research could be the comparison of different commonly used solutions and services for deploying online games, such as comparing different environments part of the compute continuum, or popular game server hosting providers. Continuum can also deploy benchmarks using cloud-based VMs, instead of using locally created ones, which could allow for another type of comparison.

Finally, using the existing system, more Minecraft-like MVEs could be integrated for testing with Continuum. Hypothetically, any containerized game server that uses the same Minecraft communication protocol can be used, with limited code changes needed for the system.

References

[1] Ben Gilbert. **VideoGame Industry Revenues Exceed Sports Film Combined in 2020 | Business Insider**. https://www.businessinsider.com/video-game-industry-revenues-exceed-sports-and-film-combined-idc-2020-12?international=true&r=US&IR=T, December 2020. 1

[2] **Streamlabs & Stream Hatchet Q2 2020 Live Streaming Industry Report**, May 2024. [Online; accessed 4. May 2024]. 1

[3] **Europe gaming attitudes during COVID lockdowns 2020 | Statista**. https://www.statista.com/statistics/1222697/gaming-attitudes-lockdown-covid-europe/, April 2020. (Accessed on 12/31/2022). 1

[4] Jeremy Winslow. **Minecraft Reached 140 Million Monthly Users And Generated Over $350 Million To Date - GameSpot**. https://www.gamespot.com/articles/minecraft-reached-140-million-monthly-users-and-generated-over-350-million-to-date/1100-6490962/, May 2021. (Accessed on 10/27/2022). 1

[5] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. **Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency**. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 126–137. IEEE, 2021. 1, 9

[6] **Minecraft inspires crafty way around government censorship**, May 2024. [Online; accessed 4. May 2024]. 2

[7] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. **Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems**. In Amar Phanishayee and

Ryan Stutsman, editors, *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020. 2

[8] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum**. In *Proceedings of the International Conference on Performance Engineering, Coimbra, Portugal, April, 2023*, 2023. 3, 7, 11

[9] Matthijs Jansen, Auday Al-Dulaimy, Alessandro V. Papadopoulos, Animesh Trivedi, and Alexandru Iosup. **The SPEC-RG Reference Architecture for The Compute Continuum**. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 469–484, 2023. 5, 7

[10] Yi Wei and M Brian Blake. **Service-oriented computing and cloud computing: Challenges and opportunities**. *IEEE Internet Computing*, **14**(6):72–75, 2010. 5

[11] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. **Improving Cloud Gaming Experience through Mobile Edge Computing**. *IEEE Wireless Communications*, **26**(4):178–183, 2019. 5

[12] **Azure IoT – Internet of Things Platform | Microsoft Azure**. `https://azure.microsoft.com/en-us/solutions/iot/`. (Accessed on 08/10/2023). 5

[13] Jinke Ren, Yinghui He, Guan Huang, Guanding Yu, Yunlong Cai, and Zhaoyang Zhang. **An edge-computing based architecture for mobile augmented reality**. *IEEE Network*, **33**(4):162–169, 2019. 5

[14] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. **The architectural implications of autonomous driving: Constraints and acceleration**. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018. 5

[15] Mahadev Satyanarayanan. **The Emergence of Edge Computing**. *Computer*, **50**(1):30–39, 2017. 6

[16] Eustace M. Dogo, Abdulazeez Femi Salami, Clinton O. Aigbavboa, and Thembinkosi Nkonyana. *Taking Cloud Computing to the Extreme Edge: A Review of Mist Computing for Smart Cities and Industry 4.0 in Africa*, pages 107–132. Springer International Publishing, Cham, 2019. 7

[17] Jürgo S. Preden, Kalle Tammemäe, Axel Jantsch, Mairo Leier, Andri Riid, and Emine Calis. **The Benefits of Self-Awareness and Attention in Fog and Mist Computing**. *Computer*, **48**(7):37–45, 2015. 7

[18] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking performance variability in cloud and self-hosted minecraft-like games extended technical report**. *arXiv preprint arXiv:2112.06963*, 2021. 7

[19] Yi Zhang, Ling Chen, and Gencai Chen. **Globally synchronized dead-reckoning with local lag for continuous distributed multiplayer games**. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, pages 7–es, 2006. 8

[20] Lothar Pantel and Lars C. Wolf. **On the suitability of dead reckoning schemes for games**. In *Proceedings of the 1st Workshop on Network and System Support for Games*, NetGames '02, page 79–84, New York, NY, USA, 2002. Association for Computing Machinery. 8

[21] Ashwin Bharambe, John R Douceur, Jacob R Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. **Donnybrook: Enabling large-scale, high-speed, peer-to-peer games**. *ACM SIGCOMM Computer Communication Review*, **38**(4):389–400, 2008. 8

[22] Siqi Shen, Shun-Yun Hu, Alexandru Iosup, and Dick Epema. **Area of simulation: Mechanism and architecture for multi-avatar virtual environments**. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, **12**(1):1–24, 2015. 8

[23] PaperMC. **Paper**, April 2024. [Online; accessed 9. Apr. 2024]. 8

[24] **jmxClient**, May 2024. [Online; accessed 4. May 2024]. 14

[25] Jose Saldana and Mirko Suznjevic. **QoE and latency issues in networked games**. *Handbook of digital games and entertainment technologies*, pages 1–36, 2015. 15

[26] **mineflayer**, February 2024. [Online; accessed 27. Feb. 2024]. 15

[27] Jerom Van Der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A benchmark for minecraft-like services**. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 243–253, 2019. 16, 28

Appendix A

Reproducibility

A.1 Abstract

Obligatory

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

Obligatory. Use just a few informal keywords in all fields applicable to your artifacts and remove the rest. This information is needed to find appropriate reviewers and gradually unify artifact meta information in Digital Libraries.

- **Algorithm:**
- **Program:**
- **Compilation:**
- **Transformations:**
- **Binary:**
- **Model:**
- **Data set:**
- **Run-time environment:**
- **Hardware:**
- **Run-time state:**
- **Execution:**
- **Metrics:**
- **Output:**
- **Experiments:**

- **How much disk space required (approximately)?:**
- **How much time is needed to prepare workflow (approximately)?:**
- **How much time is needed to complete experiments (approximately)?:**
- **Publicly available?:**
- **Code licenses (if publicly available)?:**
- **Data licenses (if publicly available)?:**
- **Workflow framework used?:**
- **Archived (provide DOI)?:**

A.3 Description

A.3.1 How to access

Obligatory

A.3.2 Hardware dependencies

A.3.3 Software dependencies

A.3.4 Data sets

A.3.5 Models

A.4 Installation

Obligatory

A.5 Experiment workflow

A.6 Evaluation and expected results

Obligatory

A.7 Experiment customization

A.8 Notes

A.9 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

...

A. REPRODUCIBILITY

Appendix B

Self Reflection

...

Appendix C

Additional Experiments

...