DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Scaling the Simulation Core of Modifiable Virtual Environments with Serverless Computing

JAVIER RON ARTEAGA

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Scaling the Simulation Core of Modifiable Virtual Environments with Serverless Computing

JAVIER RON ARTEAGA

Master's Programme, Software Engineering of Distributed Systems, 120 credits
Date: September 20, 2021
Supervisor: Nicolas Harrand
Examiner: Martin Monperrus
School of Electrical Engineering and Computer Science
Host organization: Massivizing Computer Systems, VU Amsterdam
Host supervisor: Jesse Donkervliet / Alexandru Iosup
Swedish title: Skalning av simuleringskärnan för modifierbara virtuella miljöer.

Scaling the Simulation Core of Modifiable Virtual Environments with Serverless Computing / Skalning av simuleringskärnan för modifierbara virtuella miljöer.

Abstract

Modifiable Virtual Environments (MVEs) are widely popular, especially in the video game industry. An example application of MVEs is the game Minecraft, which has a player count reaching into the hundreds of millions. But despite the huge number of users, MVE applications have significant limitations regarding scalability. MVEs are generally deployed in client-server architectures, where servers can support, at a maximum, a few hundreds of clients simultaneously. This scalability issue severely hinders interaction between users and communities. Previous work in MVE scalability has been focused on scaling out servers using approaches that focus on user actions, or moving away from client-server architectures, specifically in the form of peer-to-peer or hybrid architectures. Yet, distribution through serverless computing has only recently been proposed, and no attempt at designing or gathering experimental data has been presented. Therefore, this thesis proposes a model for distributing the simulation process of MVEs and a prototype implementing said model over a serverless computing platform, with the main objective of increasing their scalability. The distribution model is specific to MVEs and exploits its design properties. We also present a system design to take advantage of said model, which consists of two additions to a traditional MVC client-server architecture: a client-side module, and a cloud module. Experimental results show that the proposed distribution model, applied in a serverless computing setting, is a viable approach for scaling MVE simulations. Furthermore, regarding performance and scalability, results show a meaningful increase in the volume of users and simulated constructs that the system can correctly handle.

Sammanfattning

Modifierbara virtuella miljöer (MVE) är mycket populära, särskilt i videospelsindustrin. En exempelapplikation av MVE är spelet Minecraft, som har ett antal spelare uppemot hundratals miljoner. Men trots det stora antalet användare, MVE applikationer har betydande begränsningar när det gäller skalbarhet. MVE används vanligtvis i klient-server arkitekturer, där servrar maximalt kan stödja några hundratals klienter samtidigt. Detta skalbarhetsproblem hindrar allvarligt interaktionen mellan användare och communityn. Tidigare arbete inom MVE-skalbarhet har fokuserat på att skala ut servrar med hjälp av tillvägagångssätt som fokuserar på användaråtgärder, eller att flytta ifrån klient-server arkitekturer, specifikt i formen av peer-to-peer eller hybridarkitekturer. Ändå har distribution genom serverlös datoranvändning bara nyligen föreslagits, och inget försök med design eller insamling av experimentella data har presenterats. Därför föreslår detta arbete en modell för distribution av simuleringsprocessen av MVE och en prototyp som implementerar modellen över en serverlös datorplattform, med huvudsyftet att öka deras skalbarhet. Distributionsmodellen är specifik för MVE och utnyttjar dess designegenskaper. Vi presenterar också en systemdesign för att dra nytta av modellen, som består av två tillägg till en traditionell MVC-klient-serverarkitektur: en modul på klientsidan och en molnmodul. Experimentella resultat visar att den föreslagna distributionsmodellen, som tillämpas med serverlös datoranvändning, är ett hållbart tillvägagångssätt för att skala MVE simuleringar. När det gäller prestanda och skalbarhet visar resultaten dessutom en meningsfull ökning av användarvolym och simulerade konstruktioner som systemet kan hantera korrekt.

Contents

Chapter 1

Introduction

Hundreds of millions of people play video games daily. In 2020, the gaming industry generated a revenue of over $175 billion [1], breaking records in a unique growth year, boosted by COVID-19 related lockdowns that propelled interest in video gaming worldwide.

With more than 200 million copies sold, and more than 130 million active monthly players, Minecraft is one of the most successful games of all time [2]. In Minecraft, players get together to explore, mine, and build in a virtual world. Players use Minecraft for a variety of generally beneficial purposes, such as entertainment, education [3], activism [4], and social events [5]. Minecraft, and other similar *sandbox* games are able to achieve this by providing users access to a *Modifiable Virtual Environment (MVE)*.

Virtual Environments are defined as organizations of sensory information that transfer perceptions from a synthetic to a non-synthetic environment. Specifically, Immersive Virtual Environments, refer to Virtual Environments that aim to create in the user a psychological state, in which they perceive themselves to *exist within* the Virtual Environment [6]. *Modifiable Virtual Environments* are a subset of Immersive Virtual Environments, where users are able to modify the environment's objects and parts, create new content and interact with it through programs [7].

Yet, despite having a gigantic player-base, sandbox games' scalability is limited. These games support their vast number of players by relying on the replication of isolated game-instances that do not communicate.

Particular in-game building blocks, allow players to create pseudo-digital-systems. By building and connecting logic gates and digital components, players can create a variety of devices such as digital computers and automated farms and factories. For example, Minecraft supports a feature called

Redstone, which gives access to players to a *Turing complete* language and the capabilities associated with it, but also a very broad design space that players can use to place a disproportionate workload on the system.

This work is part of a vision for large-scale MVEs presented in [7], and will contribute to the effort of realizing that vision, by addressing the research challenges regarding serverless environment simulation. The use of a serverless platform enables the transfer of MVE simulations from fixed hardware resources to on-demand, elastic environments, with potentially high scalability gains.

Enabling larger-scale MVEs has a direct impact in the way these are built and used. By increasing limits of MVEs, a greater space for applications is created (i.e., bigger worlds with vast numbers of users interacting simultaneously), which will most certainly be explored by the MVE community.

1.1 Problem Statement

Generally, MVE instances can support only 200–300 simultaneous players [8], and unfortunately, the use of resource-intensive features can compound this scalability challenge. MVEs typically update their state in *update steps*. The continuous update of the MVE state is called *MVE simulation* or simply *simulation*. The rate at which the world is updated is known as the *update rate*. A reduction of the update rate can cause time slowing down inside the virtual environment and other undesirable effects, resulting in a poor quality of experience, and ultimately in users quitting the environment. To prevent this, the MVE must complete each update step in time to start the next update step.

Maintaining a high and stable update rate is a challenging requirement to fulfill for MVEs. In contrast to traditional, non-modifiable virtual environments, where the means to shape the virtual world are not present, MVEs allow users to build and activate arrangements of arbitrary complexity, significantly increasing and changing the computational load on the environment.

Existing state-of-the-art scalability techniques for large-scale MVE such as online games, e.g., interest management and world partitioning, are designed to reduce the workload created by player actions. However, the workload created by MVEs is largely independent from user behavior, and instead, follows a deterministic set of rules. This derives in some degree of predictability that can be leveraged, with the use of well-known mechanisms (e.g. caching, ahead-of-time computation) to increase the maximum number of concurrent users in MVEs.

1.2 Research Questions

The aim of this work is to improve the scalability of MVEs through use serverless computing. This platform permits the design, adaptation, and evaluation of both novel and existing scalability techniques. Such techniques include computational offloading, speculative execution, and result caching.

Formally,

RQ1 How can MVE simulations be expressed in a model compatible with distributed computing?

Within MVEs, simulations are commonplace, and such a model can improve our ability to reason about their properties. The model also assists in determining if MVE simulations can be expressed as a workload that can be partitioned in sub-tasks and executed in a distributed environment.

Creating a model of MVE simulations is a challenging task, because despite the existence of a number of clear and well known categories (e.g., physics, user actions), MVEs provide players with the tools to build and modify the world, resulting in the space of possible simulations being extremely broad. For example, there might exist instances of simulations that are so inter-connected that partition would be impractical; and, on the other hand, simulations that are so simple that distribution would add unnecessary overhead. Thus, we benefit from a model that simplifies these complex range of simulations, and helps us to identify which improvements can be tried, and where.

RQ2 How can serverless computing be used in MVE simulations to improve scalability?

Improving the scalability of MVEs allows their large number of users to share the environment together. No systems for large-scale simulation of MVEs currently exist, and it is not known which design, algorithms, and mechanisms work well for this use-case.

It is important to note that MVEs are subject to performance constraints, and that the proposed solutions must contemplate the fact that known scalability techniques may introduce intolerable side-effects (e.g., eventual consistency, increased latency) in the context of MVEs' expected quality of experience. In MVEs and in games in general, the trade-off between consistency and performance is particularly difficult to manage, since these need to both have high performance and be largely consistent.

While MVE simulations by themselves have some degree of predictability, these are sometimes directed by human behavior which is unpredictable, thus affecting the degree of accuracy with which we can predict their behavior.

RQ3 How to evaluate the effectiveness of using serverless computing in MVE simulations?

By performing real-world experiments with a prototype, we produce new knowledge about the feasibility and effectiveness of our approach. Because no standardized evaluation method for distributed systems exists, this requires designing and conducting the experiments, and analyzing their results. Apart from comparisons focused on performance and scalability gains, these results also provide insights which are useful finding representative system configurations and workloads.

Enabling larger-scale MVEs has a direct impact in the way these are built and used. By increasing limits of MVEs, a greater space for applications is created (i.e. bigger worlds with a higher number of players interacting simultaneously), which will most certainly be explored by the MVE community.

1.3 Contributions

This thesis makes the following contributions to academic literature:

- A novel model of MVE simulations which is compatible with distributed computing. It can be used as a guideline to offload MVE simulations in a variety of distributed environments, as it singles out computational bottlenecks that can be partitioned and processed concurrently.
- A system design and prototype implementation built on top a client-server MVE architecture. It uses serverless computing as the concrete means of distributing MVE simulations. The design proposes, in

addition, the use of speculative execution and caching, to further enhance performance.

- An experiment design, results, and analysis comparing a base MVE implementation against our prototype, in terms of performance, resource consumption, and scalability differences. The prototype also aims to indirectly validate our model of MVE simulations, and the suitability of a serverless approach.

1.4 Outline

The remainder of this document is structured as follows: Chapter 2 presents relevant concepts necessary to the thesis development, Chapter 3 explores the current state of the art in regards of Virtual Environments' scalability. Chapter 4 describes in detail the MVE simulation process, and details how it can be modelled in a distributed-computing-consistent manner. Chapter 5 depicts a system design applying the proposed model, serverless computing, and other scalability enhancements. Chapter 6 and Chapter 7 present an experiment to evaluate the proposed system and its results, respectively. Chapter 8 discusses how the work presented throughout the thesis fits in the context of current literature, describe limitations, and ethical considerations. Finally, Chapter 9 summarizes the thesis and traces a path for future research work.

Chapter 2

Background

This chapter provides an overview of Modifiable Virtual Environments, with a focus on their architecture and main components. It also delves into serverless computing, and introduces the scalability techniques explored further ahead in the work.

2.1 Modifiable Virtual Environments

A Modifiable Virtual Environment (MVE) is a real-time, online, multi-user environment which allows its users to modify its objects and parts, create new content and interact with it through programs [7]. MVEs commonly use a client-server architecture where each user connects through a client to a remote server. The client processes are responsible for handling the input from the users and rendering their relevant subset of the environment. Correspondingly, the server process keeps the environment's state, and also performs all environment *simulations*: the continuous update of said state over time.

Figure 2.1 provides a general overview of how an MVE is deployed. The user interacts with a *client* ① by performing specific actions defined by the MVE's particular design. These actions are sent to the server through a *networking* ② component, and used to direct the virtual *world simulation* ③. The results of the simulation update the in-memory representation of the environment, which we call *world state* ④. Concurrently, the world state is continuously broadcasted through the network component towards the clients, which render the up-to-date world state. The server can also have a *persistent storage* ⑤ component which purpose is to save snapshots of the world state for backup or sharing purposes.

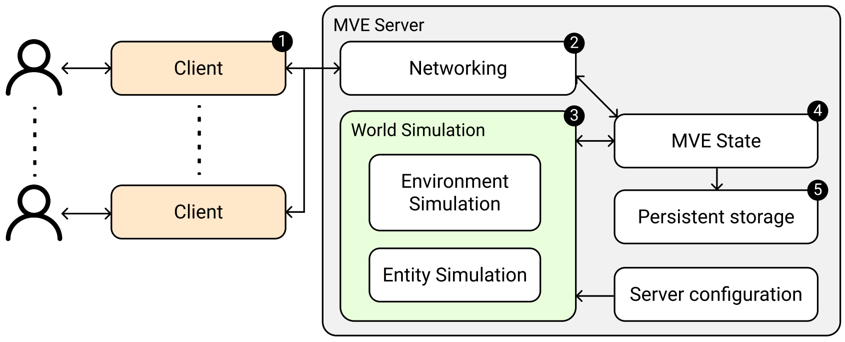


Figure 2.1 – MVE architecture general overview.

An interesting property of MVEs is that the world state is expressed as a set of spaces, where each space may hold an elemental object. We refer to these elemental objects as *simulated elements*. Furthermore, simulated elements can interact between each other creating altogether greater, composite objects with defined behavior. This is illustrated for simplicity in a two-dimensional space on Figure 2.2, however the same property holds for a three-dimensional space. The nature of how simulated elements can be arranged within the MVE state is combinatorial with respect to the number of different types of simulated elements.

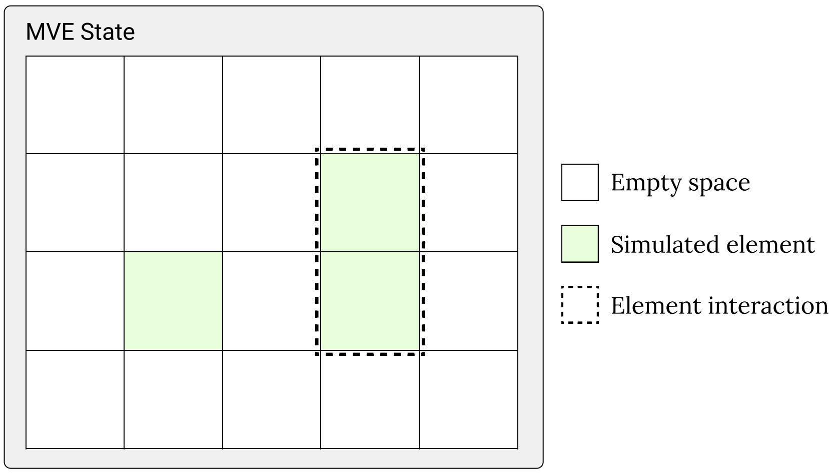


Figure 2.2 – A spatial representation of a two-dimensional MVE State.

Significant examples of applications of MVEs are *sandbox* video games, where simulated elements take the form of *blocks* which the players can destroy, create, combine, and interact with, to modify the environment. Figure 2.3 shows how different games render their state through their clients.

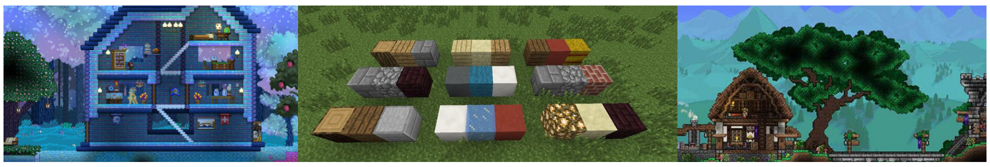


Figure 2.3 – Screen captures of Starbound, Minecraft, and Terraria games. All of which provide a modifiable virtual environment.

2.2 Serverless Computing

Serverless computing is a cloud computing execution model, where instead of thinking of applications as collections of servers, they are defined as a set of functions with access to a common data storage [9]. This model relieves developers of tasks related to capacity planning, configuration, management, maintenance, operating and scaling of infrastructure and runtimes; leaving these task entirely to the responsibility of the cloud provider.

The two main advantages of a serverless deployment are:

- Managed *fine-grained elasticity*, which means that applications scale dynamically by allocating and de-allocating resources as the workload changes, and in a completely transparent manner from the developers' perspective.
- A *pricing model* which charges developers only for the CPU time and memory used by their functions, as opposed to allocating servers or virtual machines that will likely incur in idle time. To contextualize, AWS EC2[1] virtual machine usage is metered and billed by the second, while AWS Lambda[2] does so by the millisecond, allowing a much higher resolution and thus a higher cost-effectiveness.

The popularity of serverless computing has risen significantly since the introduction of function-as-a-service (FaaS) offering AWS Lambda in 2014 [10], and currently all major cloud providers offer a similar option (e.g. Azure Functions[3], Google Cloud Functions[4], IBM Cloud Functions[5]). Since then serverless computing has become a viable alternative for the

[1]https://aws.amazon.com/ec2/
[2]https://aws.amazon.com/lambda/
[3]https://azure.microsoft.com/en-us/services/functions/
[4]https://cloud.google.com/functions/
[5]https://www.ibm.com/cloud/functions

implementation of a broad range of application domains including: real-time collaboration and analytics, urban and industrial management systems, scientific computing, machine learning, video processing, graph processing, and IoT [11].

2.3 Scalability Techniques in Distributed Systems

This section discusses three well-known scalability-improving techniques that are explored later in the work, as part of the proposed system design, albeit here, these are introduced in the general context of applications within computer science.

2.3.1 Computational Offloading

Computational offloading refers to the transfer of resource-intensive computation tasks to different execution environments [12]. Its main purpose is to overcome processing power limitations, as well as to leverage parallel computing for compatible workloads. This approach has the advantage that for any job, resources can be horizontally scaled as needed. While it is possible to offload tasks locally, (e.g. to other cores of the same processor, or a GPU), a single local system has a limited number of resources, a problem that is solved by outsourcing from a remote cluster or from the cloud.

In the context of distributed systems, offloading is often related to the use distributed computing systems such as MapReduce [13] or Apache Spark [14], which are designed to process high-volume datasets. These systems work by partitioning datasets into smaller, more manageable subsets that are processed in separate concurrent tasks. The partial results of said tasks are then aggregated into a complete result corresponding to the whole dataset. This approach makes possible to process petabyte-scale datasets in a time range of minutes [15].

From this method, several design and implementation challenges arise, related to optimally dividing and distributing the tasks. One pertinent challenge is managing *task granularity*, which measures the amount of computation involved in each task [16]. Task granularity is closely related to the completion time of the main process, as naturally, coarser-grained tasks take more time to complete than finer-grained tasks. Fine-grained tasks, however, may become inefficient as there may not be enough resources to process a high volume of them in parallel, and can cause higher overhead times due to synchronization.

2.3.2 Speculative Execution

Speculative execution is a performance improvement technique used in multiple areas of computer science. It describes the concept of executing tasks before being certain that they *need* to be executed [17]. This performance gain is possible because hardware is able to carry out several tasks at the same time, and therefore, tasks can be speculatively started earlier than the evaluation of their condition, given that there are resources available. If eventually, the results of the tasks are found to be not necessary, they are discarded. Speculative execution is often found in many layers of software and hardware and can take many different forms depending on the context. The following is a list of specific examples where speculative execution is used:

- In CPU pipelines, *branch prediction* is used to fetch and execute instructions before being certain of which branch of the code is the one to be executed. If correct prediction rates are high enough to outweigh misprediction penalties, the CPU will have a better overall performance [18]. Another speculative task performed by CPUs is *cache prefetching*, which moves data and instructions in advance from main memory to cache memory. This allows the CPU to take advantage of faster access times of cache memory versus main memory [19]. The mechanisms that decide the specific data or instructions to be prefetched can be implemented either in hardware or software.

- In transactional systems such as databases, *optimistic concurrency control* is a technique that allows multiple transactions to be executed concurrently, even though it is unknown if write conflicts will force them to rollback. Optimistic concurrency control is preferred in systems with low write-contention, where other methods such as locks substantially reduce the potential concurrency quota [20].

- In distributed computing, *task duplication* is commonly used as a mechanism to increase fault tolerance and suppress straggler node delays [21] Given a set of concurrent tasks and one of such tasks has been detected to be delayed, another instance of the same task will be spawned. The result of the task instance that returns first is used. This provides the main process with an efficient way to work around tasks executed on overloaded or crashed nodes.

2.3.3 Caching

Caching is a technique focused on improving systems' performance, namely latency and throughput metrics [22, 23]. It consists of storing data, with the intention of serving future requests for that same data faster. This is achieved by placing data caches both physically closer to the clients and in faster-access memory. A cache consist of set of entries, each entry contains an identifier and associated data. When a client performs a request, it will first trigger a query in the cache searching for the requested data. If the data is found in the cache, the request is immediately served; we call this event a *cache hit*. If the data is not found on the cache, we call this event a *cache miss*.

Performance of caches is usually measured by their *hit rate*, which refers to the proportion of requests that result in a cache hit. Cache performance is highly dependent on both cache configuration and target workload. Cache configuration entails eviction, admission and write policies which should fit the target workload to obtain a maximum of performance gains. In a distributed systems environment, caches are found at several layers, e.g. fast key-value stores to save shared computation results or proxy nodes and are intended to reduce network requests' latency. In distributed computing, frameworks like Apache Spark allow developers to explicitly cache intermediate computation results in order for them to be reused in further computation steps [14].

Chapter 3

Related Work in MVE Scalability

This section summarizes the work done found in literature about: (1) increasing scalability of virtual environments and games through novel deployments and architectures; and (2) the adoption of serverless computing in different application domains, where traditional architectures are predominant.

3.1 Distributed Architectures

Distributed architectures for virtual environments have been proposed in literature as an approach to scale the environment in both terms of size and user count. Diaconu et al. [24] present Kiwano, a scalable distributed infrastructure for virtual worlds is presented. The approach of this work is to divide the virtual world in non-overlapping zones, where each of these is managed by one server. Zone servers are organized in a peer-to-peer network, and are able to share their border content with neighboring zones. An important property of Kiwano is that zones are dynamic, this means that a load-balancing algorithm is executed to prevent overloading a single zone. This approach, however only considers movement of objects as the events happening inside the virtual world. The same authors also show Manycraft [25], which implements a server on top of Kiwano, and is compatible with Minecraft clients. Manycraft allows for a higher number of concurrent players compared to a Minecraft server, at the cost of modifiability, this means, Manycraft worlds are static and cannot be changed by the players.

Lake et al. [26] define virtual environment *simulator-centric* architectures, where all simulation and communication processes are executed on a single server. In the same work it is argued that this kind of architectures do not scale, regardless of the capacity of the server. Therefore, they propose the Distributed Graph Scene architecture, where the storage of state and simulation tasks are moved to distributed actors, allowing the virtual environment to scale with the addition of hardware.

Horn et al. [27] propose dividing virtual environments in three individually administered parts: (1) space servers, which store state, (2) object hosts, which execute simulated objects' code, and (3) content delivery networks, which offload communication of static content. Although the experiments do not present a comparison with other architectures in terms of scalability, they show that the system handles a high volume of interactions between simulated objects. Extending on this idea, Elfizar et al. [28] present a model where each object is managed by its own simulation process, allowing for even broader distribution.

Going beyond distribution of the server process, Vilardell et al. [29] state the need of gaming-aware distribution mechanisms and propose a hybrid architecture, where the game world is divided into a *main game* which is hosted in a client-server configuration; and other *auxiliary games* which are hosted by its participants in a peer-to-peer configuration. The results show that compared to a traditional client-server only architecture, lower latencies are achieved in average. This also comes at the cost of lower reliability and higher probabilities of failure, however, the authors argue that the scalability provided by the peer-to-peer addition amply compensates for this disadvantage.

In the realm of mobile games, Anand et al. [30] introduce the concept of *gamelets*. Gamelets are nodes within a micro-cloud infrastructure which provide support to otherwise resource limited mobile devices. Using a combination of techniques such as zone distribution, distributed rendering, adaptive streaming, and peer-to-peer configurations, gamelets are able to assist in the execution of resource demanding games and data transmission to mobile devices. Gamelets are evaluated on bandwidth consumption, processing efficiency and user perception parameters achieving mixed results.

3.2 Cloud Resource Efficiency

The cloud plays a central role in supporting a high volume of users through horizontal scaling, and proper management of cloud resources is key to achieve maximum cost-efficiency. This is not different in virtual environments such as Massively Multiplayer Online Games (MMOs). Nae et al. [31] propose a cloud-based ecosystem for MMOs, in which the interaction of business actors (resource providers, game operators, game providers and clients) is regulated through *Service Level Agreements*. This allows game providers to lease cloud resources dynamically and on-demand, while at the same time guaranteeing all Quality of Service requirements. Their results show a 60% decrease in operational costs compared to the use of a static infrastructure approach.

Gao et al. [32] take the same concept further and a similar approach to reduce over-provisioning is described, but with a focus on energy efficiency. They show how better provisioning policies can be found through the use of a genetic algorithm. The results show that a 54.5% energy saving can be achieved versus other state-of-the-art provisioning policies.

3.3 Serverless Deployments

Using serverless architectures to deploy virtual environments has been only recently proposed and has still many design and implementation challenges to be addressed, as described by Donkervliet et al. [7]. On a more hands-on approach, Liew [33] presents a proof of concept with a port of the videogame DOOM[1] to the Fastly Compute@Edge[2] platform. The original source of the game is split and modified so that the rendering and input handling is performed in a web browser, and the computation of each update step is done in a serverless function call. This work is presented as an experiment to showcase the capabilities of the Compute@Edge platform, and does not seek to achieve any performance improvement; however, it is successful in showing the feasibility of building games and virtual environments in a serverless architecture.

Moving traditional deployments to serverless deployments in other application domains has also been explored in the literature. While not directly linked to MVEs, the criteria underneath the deployment transition are similar, meaning that patterns and procedures may be reproduced.

Toader et al. [34] present Graphless, a serverless graph processing system. In contrast to existing graph processing systems that focus on performance, Graphless tries to be accessible to a broader user base. The serverless approach allows users to focus on small and stateless functions, and an easier architectural deployment. The experimental results show that in some specific cases Graphless can provide performance and operational costs similar to state-of-the-art graph processing systems.

Fouladi et al. [35] describe ExCamera, a system for low-latency video processing. Within the scope of the work, two contributions are made: (1)

[1]https://bethesda.net/en/store/product/DO1GNGPCBG01
[2]https://www.fastly.com/products/edge-compute/serverless

a framework to run general-purpose parallel computations on commercial serverless environments; and (2) a video encoder designed for fine-grained parallelism, and implemented using a functional programming style that splits computation into thousands of sub-tasks. The experiments show that ExCamera achieved comparable compression to other video processing systems, at the same quality level relative to the original uncompressed video, and was in some cases up to 300x faster.

Chapter 4

Modelling the MVE simulation process

At the core of virtual environments, we seek to imitate and manipulate real-world physical phenomena, these tasks are impossible to achieve without emulating the *passage of time*. This concept is what we refer to as *simulation*: the continuous update of the virtual environment performed in discrete steps at a regular frequency. Within an MVE that is deployed in a client-server architecture (Figure 2.1), the simulation process is performed by the MVE server, which computes each of the simulation steps (also referred to as *update steps* or *ticks*).

The following sections explore an approach to describe the MVE simulation process in a way that: (1) is compatible with distributed computing, and (2) enables scalability enhancements. It also introduces the concept of *simulated constructs*, and notation used to represent MVE simulation-related concepts. The ideas presented here are used throughout the work and specifically, play a key role in directing the system design proposed in Chapter 5

4.1 Environment Simulation

As noted in Chapter 2 the MVE server is responsible for the operation of, among others, three components: the *MVE state*, the *environment simulation* and the *networking* components. These three components continuously interact as part of the MVE simulation. Figure 4.1 shows, in a sequence of steps, how they interact in a *simulation loop*: ① The networking component receives input from clients and modifies the MVE state with external changes, if any. ② The MVE state is read by the simulation component, and ③ the simulation component applies the simulation rules on the MVE state and updates it. Finally, ④ the new MVE state is transmitted to the clients through the networking component. Each instance of the simulation loop is effectively an *update step* or *tick*. The *update rate* or *tick rate* is defined as the number of updates that are performed within a time unit, and ideally, the update rate should be stable and frequent enough to maintain a continuous-time illusion.

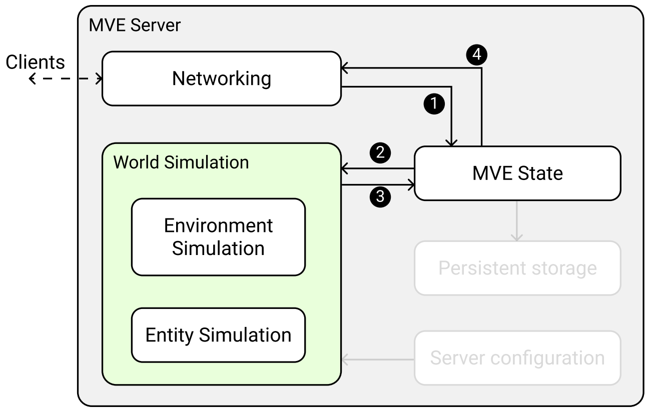


Figure 4.1 – MVE simulation loop.

Since the *world state* is a collection of simulated elements, updating the world state means updating the elements that constitute it. The amount of elements that are actively updated is determined by a configuration policy, often linked to spatial proximity to a user's avatar. To do so, the world simulation component traverses the set of elements and performs the corresponding computation of their next state. This continuous next-state computation for each element is what allows the server process to imitate natural phenomena such as physics, as well as to execute any arbitrary kind of time-dependant feature.

From an abstract perspective, the world state and the changes applied to it each tick, can be expressed as a collection of sets and functions:

Given the world state S at any point in time t, it can be defined as a set of simulated elements:

$$S^t = \{e_0^t, e_1^t, e_2^t, \dots, e_n^t\} \qquad \text{(i)}$$

The networking component applies external changes as a function N to the world state S

$$N(S^t) = S^{t'} \qquad \text{(Step 1) (ii)}$$

If there are no changes to the world state S introduced by users, N becomes effectively an identity function, and thus $S^{t'}$ will be equal to S^t

From there, the simulation component applies a function U to the world state, which computes the world state for the next time step:

$$U(S^{t'}) = S^{t+1} \qquad \text{(Step 3) (iii)}$$

Overall, we can express each simulation step as a function SIM applied to the world state S at time t, to compute the world state at time $t + 1$.

$$U(N(S^t)) = SIM(S^t) = S^{t+1} \qquad \text{(iv)}$$

Equivalently, the simulation step can be presented as applying the function to each simulated element contained in the world state:

$$SIM(S^t) = \{SIM(e_0^t), SIM(e_1^t), SIM(e_2^t), \ldots, SIM(e_n^t)\} \qquad \text{(v)}$$

Which is equivalent to computing the next state for each simulated element:

$$SIM(S^t) = \{e_0^{t+1}, e_1^{t+1}, e_2^{t+1}, \ldots, e_n^{t+1}\} \qquad \text{(vi)}$$

Expressing the MVE simulation as a collection of sets and functions is the first step towards providing a distributed computing-compatible model. A naive approach can be devised from (v): mapping the simulation of single elements to one distributed task. However, this produces one further complication: the required amount of tasks might be unreasonably high, as shown in Figure 4.2 which depicts a concrete MVE instance. This instance is constituted by a number of simulated elements that reaches the order of hundreds of thousands. The next section describes a solution to this problem.

4.2 Simulated Constructs' Model

A property of simulated elements is that they can interact with and between each other, as configured by the simulation rules. This allows for the existence of *simulated constructs*, this is, sets of simulated elements that interact with each other to provide —as a whole— a single, coherent and defined behavior. This behavior emerges naturally as the constituting simulated elements follow the simulation rules and respond to user input.

As a concrete example, a working instance of the LC3 computer [36] was

Figure 4.2 – A working quad-core computer built in Minecraft[1].

created by a member of the Minecraft community[2]. In this instance, modular components are built separately and connected. Such components include RAM, ALU, bus, and other basic components of computers. Figure 4.3 depicts a contraption designed to hold one bit of RAM. In this example each depicted block is an instance of a simulated element, and the RAM unit as a whole is an instance of a simulated construct.

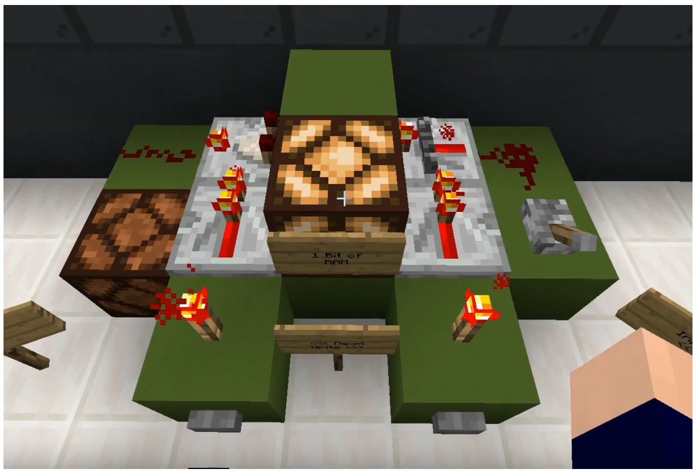
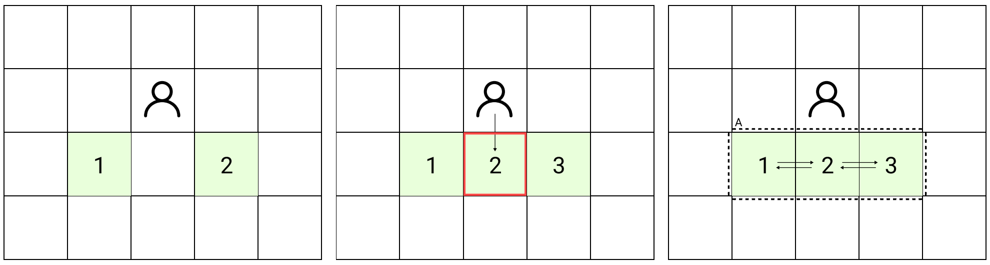


Figure 4.3 – A simulated construct example in Minecraft. A RAM module used in the construction of a working computer.

[2]LC3 in Minecraft: https://youtu.be/ecBFyjtjqvQ

[2]Quad-core copmuter: https://youtu.be/yobsIg3YL_U

From a more general standpoint, Figure 4.4 illustrates how a simulated construct can be created in a sequence of successive world states. Step 1 shows an arbitrary initial configuration, where the elements "1" and "3" do not interact with each other. Step 2 shows the user placing a simulated element in an available empty space. In Step 3, through a pre-configured rule of spatial proximity a simulated construct is formed. In that final state, the three simulated objects interact with each other, and from the user's perspective their behavior can be described as of that of a single entity.



1. A user avatar and two simulated objects, $\boxed{1}$ and $\boxed{3}$.

2. The user places the new simulated object $\boxed{2}$.

3. The simulated construct $\boxed{A}$ is created.

User avatar ☐ Empty space 🟩 Simulated object ⬚ Simulated construct

Figure 4.4 – Sequence of MVE world states where a simulated construct is created.

In relation the sets and functions definitions in Section 4.1, simulated construct can also be represented as such. Thus, the simulated construct C at time t becomes a set of simulated elements:

$$C^t = \{e_0^t, e_1^t, e_2^t, \dots, e_m^t\} \tag{i}$$

From there, the complete MVE state S can be expressed as the union of: all simulated constructs, and the set R of all remaining elements that do not constitute a simulated construct:

$$S^t = \{C_0^t, C_1^t, C_2^t, \dots, C_n^t, R^t\} \tag{ii}$$

Likewise, the simulation function SIM can be applied to simulated constructs to compute the next state S^{t+1}:

$$SIM(S^t) = \{SIM(C_0^t), SIM(C_1^t), \dots, SIM(C_n^t), SIM(R^t)\} \tag{iii}$$

Simulated Constructs may have other relevant properties, depending on simulation rules and the elements that make up a construct. For example, the interaction between them can result in a high number of computations in a single update step. An example of this is when elements subscribe to notifications to adjacent elements' changes, triggering —in the worst case— a cascade of computations that propagates exponentially as shown in Figure 4.5. Some interactions can also lead to inefficiency by repetition of computations, specially in constructs with looping behavior or when multiple instances of the same construct exist, as shown in Figure 4.6.

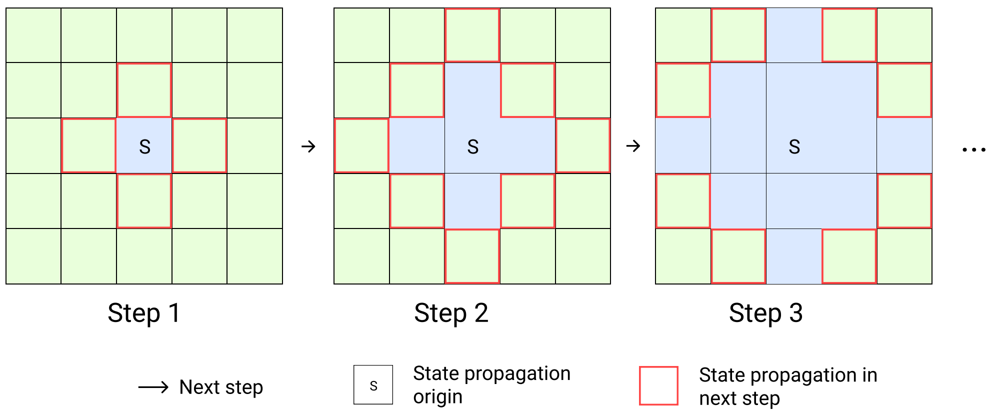


Figure 4.5 – Sequence of steps depicting element state change propagation within a simulated construct.

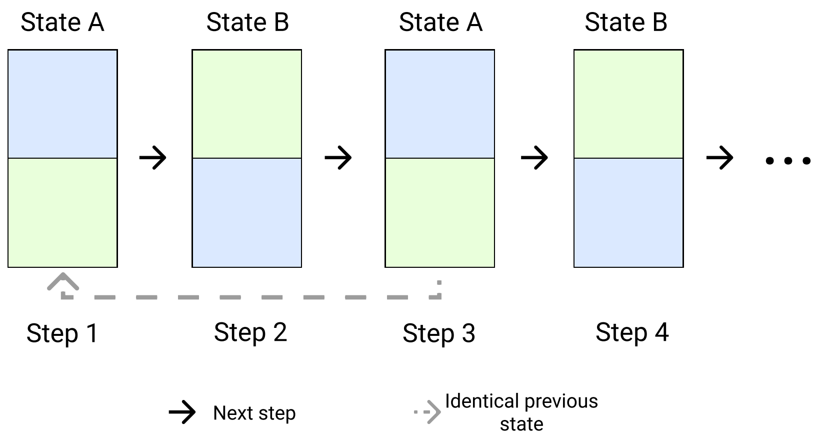


Figure 4.6 – Sequence of steps depicting a simulated construct with looping behavior. The colors represent any simulated element's non-specific internal state.

4.3 Simulated Constructs' Properties

To objectively describe and quantify simulated constructs within an MVE, we characterize them by using a comprehensive set of properties. These are selected from a broad spectrum, comprising from well-known systems-level properties to top-level design properties. These properties are classified in three main categories: (1) *System-level properties* which relate to performance and resource consumption from a system's perspective; (2) *Application-level properties* which relate to MVE implementation and technical specifics; and (3) *High-level properties* related to behavior that emerges from the MVE's rules and design.

By observing and measuring the values of these properties, it is possible to correlate distinct configurations to the overall MVE performance. Furthermore, high-level properties may be used to recognize patterns and direct what changes may be appropriate given the goal of improving the system's scalability.

System-level properties

We determine the consumption of hardware resources to be a relevant property of simulated constructs. Measuring this allows us to establish a relation from different configurations of simulated constructs to a well-understood domain using metrics that are common to evaluate all software. Within these metrics we specifically name: *CPU load*, *memory consumption*, both in the process of simulation and holding state; and *network bandwidth usage*.

Application-level properties

Application-level properties of simulated constructs are directly related and constrained by the specific implementation of the MVE. Consequently, some application-level properties are directly inherited from the MVE, such as the *geometric properties* which describe how the shapes and positions of elements are represented and computed within the virtual environment. Another inherited property is the *update rate*, that equivalently describes how many times is the construct updated per unit of time. We have described simulated constructs as collections of simulated elements, therefore a relevant property is the number of elements that constitute them; we refer to this property as the construct's *size*.

High-level properties

Simulated constructs may have some properties that arise from the design space provided by the MVE, i.e. interactions between different simulated elements, taking into account all their different configurations. This properties are based on the concept of emergence [37], this is behavior that is not directly described by preset rules, but rather by higher-level interaction of elements and components of a system. For simulated constructs we identify two relevant high-level properties. Simulated constructs may be *deterministic*, this is, any future state of the construct can be computed from the simulation rules, and does not depend on any random events or state-altering input from any external entity. Simulated constructs can have *periodic* behavior, which means that a set of its states will recur at intervals over time. High-level properties of simulated constructs are of particular interest since this kind of behavior can be used to direct specific enhancements on the MVE implementation. For example, determinism means that the simulation of a construct and the update rate can be decoupled, supporting ahead-of-time simulation of constructs. Likewise, periodicity implies that the same state transitions have to be applied periodically to some simulated constructs, providing a suitable use case of a cache to reduce repeated computations of state transitions.

4.4 Summary

MVE simulation is a process that takes place within the MVE server process. It is tasked with periodically updating the state of simulated elements according to predefined simulation rules, and transmitting the new overall state of the environment to the users. Inside the environment, a set of simulated elements can be arranged into a simulated construct, meaning that for a user it may appear as if this set of elements is a single coherent entity. The design space of simulated constructs can be very broad and lead to configurations that require a high number of computations to be completed in each update step.

Simulated constructs can be described by: properties directly related to their execution: system-level properties; properties intrinsic to the MVE where these are instanced: application-level properties; and properties that arise from the MVE design and element interaction rules: high-level properties. Table 4.1 summarizes the described properties of simulated constructs.

System-level	**Unit of measurement**
CPU load	percentage
Memory (simulation)	bytes
Memory (state)	bytes
Network usage	packets per second

Application-level	**Value**
Spatial dimensions	2D/3D
Geometry	voxel/polygon -based
Size	# of components
Update rate	updates per second

High-level properties	**Value**
Periodicity	Yes/No
Determinism	Yes/No

Table 4.1 – Summary of identified properties.

We argue that a model of the MVE simulation based on simulated constructs is well suited for serverless computing and enables overall scalability enhancement of the MVE. The reasoning for this claim is that the simulation of each construct can be offloaded in a serverless function invocation, therefore, freeing resources to handle a higher volume of simulated elements or users. This topic is further explored in the Experimental Design (Chapter 6) and Results (Chapter 7) chapters.

Chapter 5

System Design and Implementation

This chapter presents a system design whose main goal is to leverage serverless computing to increase scalability of MVEs. Therefore, it builds on top of two previously detailed notions: *simulated constructs* defined in Section 4.2, and a set of *scalability techniques* presented in Section 2.3. Simulated constructs are used as a model to break down the MVE simulation process into smaller tasks compatible with serverless computing. Computational offloading and speculative execution are achieved by performing the update of simulated constructs in serverless functions; and caching is used on the server to store the serverless functions results for reuse. The following sections describe in detail the system's requirements, and the design proposed to meet them. Finally, it delves into the specific platform and tools used to implement the system.

5.1 Requirements

Consistent with the research questions, the system's requirements are directly related to the use of serverless computing to increase scalability of MVEs. Concretely, this means building on top of a client-server MVE implementation to increase the volume of clients served concurrently by the MVE server.

Regarding performance and quality of service, MVEs have tight constraints on update rate and latency as perceived by clients, with the update rate being of special significance for the proposed system.

As stated in Section 4.1, the *update step time* (or *tick time*) is the time it takes the system to perform a complete simulation loop. In the same section, we defined the *update rate* as the number of simulation steps completed per time unit. Given that the system has a target update rate, the update step time becomes directly constrained. From the target update rate, we derive a maximum update step time that acts as a limit: if update steps take more time to complete than this limit, then the target update rate is not met, and the performance constraint violated. The maximum update step time is defined by the following equivalence:

$$\text{UpdateStepTime}_{\text{max}} = \frac{1}{\text{UpdateRate}_{\text{target}}}$$

As an example, given an MVE server with a target update rate of 20 *updates per second*, the update step time cannot be any higher than 50ms. Consequently, If the update step time is higher than 50ms, we consider the MVE server to be server overloaded.

Regarding serverless computing, it must be noted that the latency of serverless function invocations cannot be predicted and may have a degree of variability[38]. As stated previously, MVE servers must fulfill a target update rate in order to provide the expected quality of experience. Therefore, if the actual update rate of an MVE depends on the uncertain return time of a serverless function, then the update rate also becomes uncertain. In other words, if the serverless functions take more than to complete the maximum update step time, then the target update time constraint would not be met. The proposed system design must take care of this consideration, and provide an alternative to complete the update steps, even when the serverless functions have not yet returned.

With respect to the system's features, in addition to the MVE simulation process described in Section 4.1, the system should implement a mechanism to interface with a serverless computing provider and make use of it to enhance performance and increase scalability. Such mechanism can be described as a sequence of steps:

1. Partition the MVE simulation workload into smaller tasks.
2. Offload the simulation of the generated tasks to serverless functions.
3. Store the returned results of the serverless functions for reuse.
4. Synchronize the results into the main MVE simulation.

Each of these steps can be performed by a different component of the system which are detailed in the following section.

5.2 System Design

Broadly, the proposed system can be depicted as an MVE platform with an interface to a serverless computing provider, where parts of the MVE simulation process can be offloaded. The design builds on top of a traditional MVE client-server architecture (Figure 2.1), and consists of the addition of two modules, as shown in Figure 5.1: A module embedded in the MVE server process Ⓐ and a module hosted in a cloud services provider Ⓑ.

The server-side module is responsible for: accessing and partitioning the MVE state, interfacing with the cloud services, and transmitting the data to the external simulation function. It also listens for incoming computation results to be stored and applied into the world state. The cloud module is composed by the simulation function, which contains the MVE simulation rules.

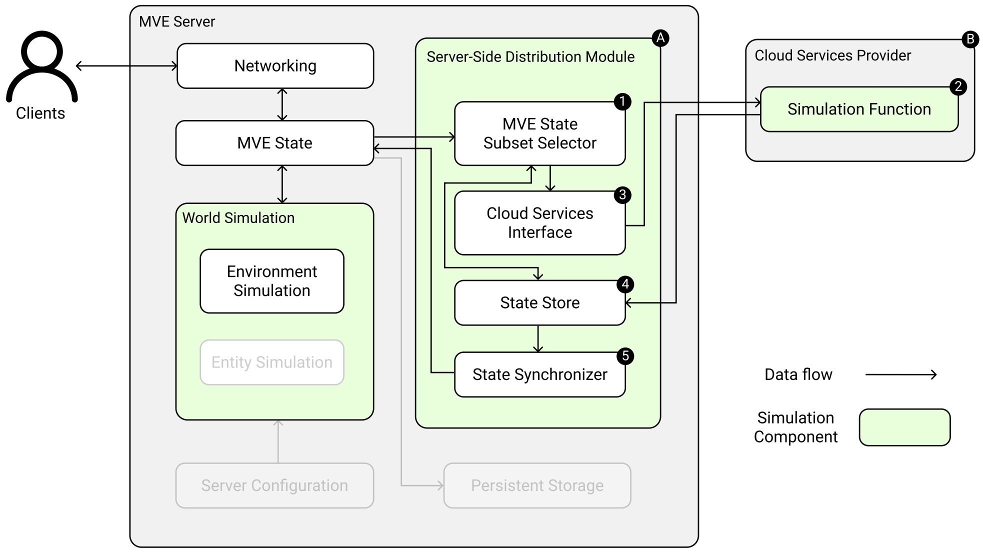


Figure 5.1 – Design of an MVE with serverless simulation capabilities.

The server side-module has by four components. These components handle MVE state access and modification, storage of the pre-computed state subsets, and communication with the cloud module. Figure 5.1 also shows the interactions between its components as well as interaction with the cloud module and MVE state component. The following sections detail each of the system's components and their role in fulfilling the system's requirements.

5.2.1 Workload Partitioning

To enable the use of serverless functions we rely on dividing the MVE state into self-contained subsets. This is done by the MVE state subset selector component ①, and the division follows a model relying in the simulated construct concept, implying that each partition contains one simulated construct. Ideally, the resulting partitioning results in one partition for each simulated construct, and a remainder partition where there are not any simulated constructs. Doing this, the MVE state is effectively divided as noted on Section 4.2:

$$S^t = \{C_0^t, C_1^t, \ldots, C_n^t, R^t\} \qquad \text{(Section 4.2 ii)}$$

5.2.2 Simulation Offloading and Ahead-of-time Simulation

This step relies on two components being in place: ② a serverless function where the simulation will be performed, and ③ a method to invoke the serverless function from the MVE server. The serverless function receives a payload that consists of a serialized MVE state subset, and applies the simulation rules to it. The result of the invocation is the next state of said subset according to the simulation rules:

$$SIM(C_i^t) = C_i^{t+1}$$

From there, if we assume that no external changes will be applied to the MVE state subset, the simulation rules can be applied an indefinite number of times within the same serverless function's environment. If we define this number as n, then we can compute a set of states for the next n updates. The consecutive application of the simulation rules can be denoted as S^n where n is the number of updates to be computed, and its result is as follows:

$$SIM^n(C_i^t) = \{C_i^{t+1}, C_i^{t+2}, C_i^{t+3}, \ldots, C_i^{t+n}\}$$

It is relevant to note that when computing several steps at once, the obtained result is effectively speculative and obtained ahead-of-time, since it is unknown if C_i will be modified by an external entity in between any of the computed states.

5.2.3 Simulation Caching

To speed up the simulation loop and reuse previously computed states, we set up a cache $\textcircled{4}$ to store the results generated by the serverless function. It is built as a key-value store, where the keys are unique identifiers for each state subset C_i and the values are the results $S^n(C_i^t)$, returned by the serverless function.

$$\text{Entry} = \langle C_i, \{C_i^{t+1}, C_i^{t+2}, C_i^{t+3}, \dots, C_i^{t+n}\}\rangle$$

There is one main use case where caching is useful in this context. Given a simulated construct C_i^t that is placed in the MVE state such that it is similar to C_j^u, and therefore:

$$SIM^n(C_i^t) = SIM^n(C_j^u)$$

If the simulation for the first construct has already been computed, it can be reused for the second construct.

The same reasoning applies a simulated construct that behaves periodically, such that C_i^t is equal to C_i^{t+n}, and therefore the computed states can be reused for the same construct.

$$SIM^n(C_i^t) = SIM^n(C_i^{t+n})$$

The cache entries are invalidated if a modification on the simulated constructs occurs, such that it causes divergence from the pre-computed states. To detect modifications, a hash function $H(C_i^t)$ is used. To illustrate, given that the current state of C_i is C_i^{current} and the synchronization mechanism is set to apply C_i^{t+1}, it first has to check if $H(C_i^{\text{current}})$ is equal $H(C_i^t)$. If these values are not equal, the system determines that the behavior of the construct has diverged and $S^n(C_i^t)$ is discarded from the cache. A new computation $S^n(C_i^{\text{current}})$ is then triggered for C_i. The use of a hash function is preferred for state comparisons given that: (1) hashes for the pre-computed states can also be pre-computed in the serverless function environment, and (2) the comparison of hash values is inherently faster than the comparison of whole states of a simulated construct.

5.2.4 State Synchronization

A mechanism to synchronize the computation offloaded to serverless function is needed $\textcircled{5}$, this is, the results returning from the serverless function invocations are consolidated back into the main MVE state at the appropriate time.

The selected solution consists of 2 steps: First, since the simulation of the remainder set R is not offloaded, its elements are updated according to the usual MVE simulation rules. Then, the system iterates over all the simulated constructs defined at the partitioning, and either: (1) changes its state to the corresponding next state stored in the cache, or (2) if not present in the cache, invoke the serverless function to compute its next states. If we differentiate the SIM function in SIM_{local} and $SIM_{\text{serverless}}$ depending on the execution environment, the complete simulation step can be expressed as:

$$SIM(S^t) = \{SIM_{\text{serverless}}(C_0^t), \ldots, SIM_{\text{serverless}}(C_n^t), SIM_{\text{local}}(R^t)\} \quad (5.1)$$

Where the results of $SIM_{\text{serverless}}$ may be available on the cache.

5.3 Implementation

This section describes the system's implementation. As directed by the system design, A server-side module, as part of Opencraft[1]; and a cloud module which uses Amazon Web Servies (AWS)[2].

5.3.1 Server-side Module

We implement the server-side module as part of the Opencraft project[3] which is a platform for research in massivising Minecraft-like networked virtual environments. The Opencraft project develops and maintains a homonymous MVE server: Opencraft. Opencraft is an open source implementation of a Minecraft server, written in Java and fully compatible with Minecraft clients. Opencraft is mainly used as a research sandbox to implement novel features and conduct related experiments[8, 39]. The following is a list of noteworthy problems solved while implementing the designed components into Opencraft.

Latency hiding

According to our requirements, as described in Section 5.1, an update step cannot take more than 50ms, however latencies from the serverless function

[1]https://github.com/atlarge-research/opencraft
[2]https://aws.amazon.com/
[3]https://atlarge-research.com/opencraft/

can be higher than that limit. To overcome this problem all offloaded constructs must be locally simulated while waiting for the serverless functions to return. Figure 5.2 shows a timeline, where offloaded simulation is invoked for state subset C_i at time t, however local simulation is still needed while waiting for the offloaded result to arrive. Once it arrives, local simulation is stopped for C_i and the pre-computed values are used instead to update the next states for C_i.

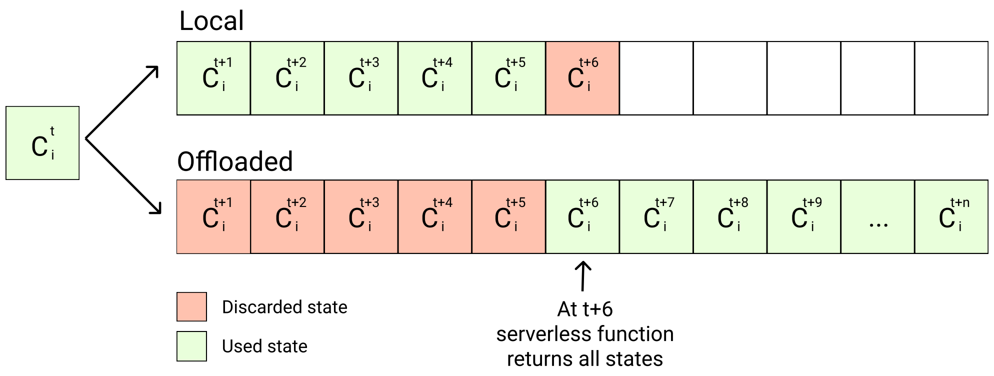

Local
Offloaded
C_i^t
C_i^{t+1} C_i^{t+2} C_i^{t+3} C_i^{t+4} C_i^{t+5} C_i^{t+6}
C_i^{t+1} C_i^{t+2} C_i^{t+3} C_i^{t+4} C_i^{t+5} C_i^{t+6} C_i^{t+7} C_i^{t+8} C_i^{t+9} ... C_i^{t+n}
Discarded state
Used state
At t+6 serverless function returns all states

Figure 5.2 – Simulation asynchronous wait.

MVE state partition with cuboidal bounds

To select a subset from the MVE state we make use of the geometry properties of the Opencraft MVE. In Opencraft simulated elements are arranged as a set of *voxels*, this is, the elements' positions are represented as discrete values in a regular, rectangular three-dimensional grid. Figure 5.3 illustrates in a two-dimensional space, how two anchor points, "A" and "B", can describe a rectangular bounding area. Likewise, to select a rectangular bounding volume in Opencraft we extend the same method to a three-dimensional space.

MVE state serialization

Serializing the MVE state subsets prior to transmission to and from the serverless function has a noticeable impact in the systems' performance. While the AWS SDK provides the means to easily serialize Java objects as JSON objects, this proves to be quite inefficient and does not adhere to low latency constraints. To overcome this limitation, we encode elements as an array of 6 integers that describe the constituting elements' positions, types, and internal states. Encoding elements in this manner reduces the payload sent to the serverless function by orders of magnitude as shown in Table 5.1.

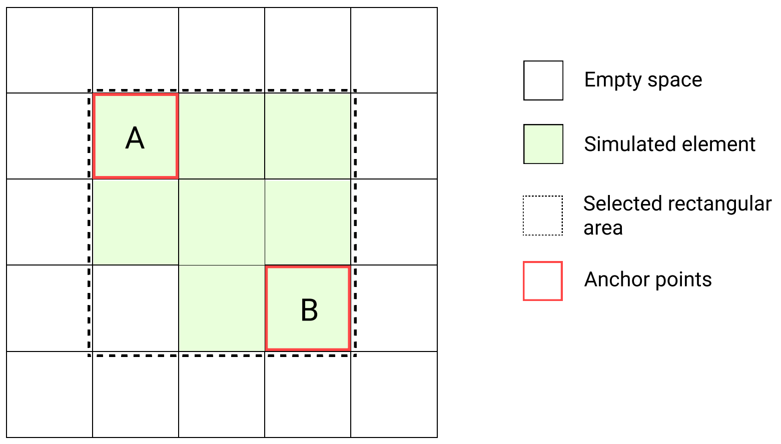


Figure 5.3 – Two-dimensional representation of MVE state subset selection strategy.

JSON encoding	Custom encoding
`{`	
`"state":{`	
`"x":119,`	
`"y":61,`	
`"z":-48,`	
`"typeId":3,`	
`"data":{`	`119,61,-48,3,0,0`
`"type":3,`	
`"data":0`	
`}`	
`},`	
`"attachedToFace":"DOWN"`	
`}`	

Table 5.1 – Encoding a simulated element, JSON vs. custom encoding.

5.3.2 Cloud Module

For the implementation of the cloud module we rely on Amazon Web Services, specifically on AWS Lambda. As discussed in Section 2.2, AWS Lambda is a FaaS offering with support for multiple programming languages, including Java. A serverless function that follows the system design is uploaded to the platform. The MVE simulation rules are available to the function through a Maven dependency.

Loop detection

To exploit the periodicity property of simulated constructs a loop detection algorithm is executed after computing the configured number of state in the serverless function, if periodical behavior is detected then the resulting set of pre-computed states is truncated, and the result flagged as periodical. Loop detection allows the pre-computed states to be valid for an indefinite time, and also reduces the size of the serverless function response.

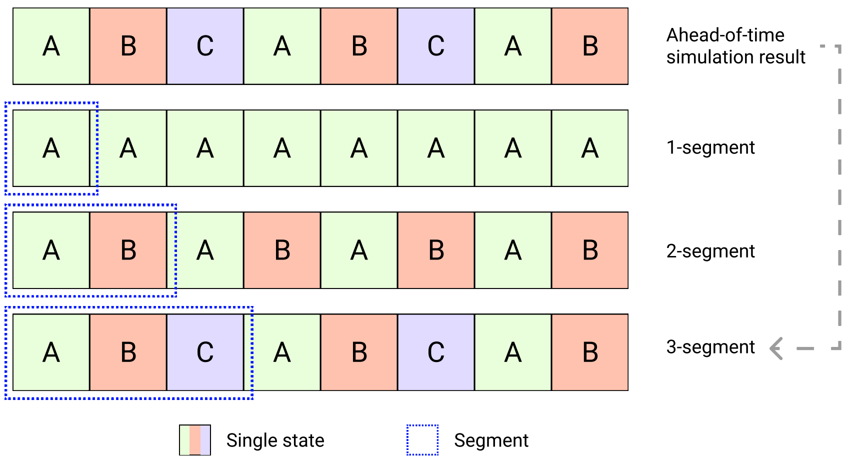


Figure 5.4 – Loop detection procedure.

Loop detection is implemented as follows: After computing all requested states, these are assigned a value by using a hash function, we then operate over this list of hash values. Figure 5.4 Shows the procedure: we select a segment of the list, ranging from positions 0 to n (n-segment), and then create a new list containing only successive instances of this segment, we then compare the new list to the original list. If they match, then only the segment is returned as the result. The algorithm is executed from $n = 0$ to $n = length/2$, or until a looped segment is found.

Chapter 6

Experimental Protocol

This chapter shows an experimental approach which purpose is to test the capabilities of the system presented in Chapter 5. To accomplish this, the system is compared against an MVE environment deployed in a traditional client-server architecture, in this case, an unmodified version of Opencraft.

Broadly, the approach consists on two sets of experiment configurations. These are respectively intended to provide data on the suitability of the system, and the actual effects on the system's performance and scalability. Table 6.1 shows an overview of the relevant metrics and parameters used. Set 1 is aimed at validating the use of serverless computing, provided that the remote functions execute within reasonable latency. Set 2 is aimed at measuring specific performance and scalability differences between the serverless-enhanced and base MVE implementations.

Set	**Parameters**	**Metrics**	**# of configurations**
1	SC size	Serverless func. latency	3
2	Simultan. MVE clients Number of SCs	Update step time CPU usage RAM usage	6

Table 6.1 – Overview of experiment configurations. (SC) stands for *simulated construct*

The following sections describe the details on how the experiments are carried out, which metrics are used to compare both systems, and the environment where the experiments are executed.

6.1 Experiment Design

The experiments presented in this section aim to provide insight on the research questions **RQ2** and **RQ3**, therefore, are related to measuring the suitability and performance variations of a serverless-enhanced MVE. The results produced by the experiments are also used to indirectly validate the simulated constructs model, presented in Chapter 4 and directly related to **RQ3**.

6.1.1 Parameters

To produce relevant experiment configurations, the workload size and partitioning is modified through the following parameters: *Number of simulated constructs, Simulated construct size, Number of MVE users, Offloading*. These will be adjusted between every experiment configuration to produce relevant and comparable metrics. Other workload-related parameters and properties from simulated constructs are kept constant through the experiment, e.g., all simulated constructs used in the experiment have the determinism and periodic behavior properties.

The following is a description of each selected parameter:

Number of simulated constructs The amount of simulated constructs present in the MVE state. Assuming that each construct is different, then this parameter also indicates the number of serverless functions that that are invoked in each experiment configuration.

Simulated construct size This parameter is taken directly from the *size* property of simulated constructs. This is, the size of simulated constructs is given by the number of simulated elements that constitute it. While the exact unit in which the size of simulated constructs is measured is the amount of constituent simulated elements, an alternate scale (S,M,L) is provided to emphasize the difference in scale, rather than specific numeric differences.

Number of MVE users The amount of MVE clients connected to the server and receiving the updates caused by MVE simulation. This parameter is dynamic, this means that it increments from 0 to n over time within the experiment execution.

Serverless-enhanced Indicates if the experiment is performed in the serverless-enhanced or base MVE implementations.

Table 6.2 describes the values of the parameters used for the experiment; each experiment configuration is a combination of these.

Parameter	**Values**
Number of simulated constructs	**50** 100 200
Simulated construct size	**252 (S)** 484 (M) 2015 (L)
Number of MVE users	**0 - 100**
Serverless-enhanced	**Yes** No

Table 6.2 – Parameter values.

While there are other properties and configurable values that can be used as experimental parameters, these will remain static for all experiment configurations. Table 6.3 shows these values.

Parameter	**Value**
Target update rate	20 updates per second
Maximum update step time	50 ms
Ahead-of-time updates	50

Table 6.3 – Static values used throughout all experiment configurations.

6.1.2 Configurations

Experiment configurations are divided into two subsets: One subset is used to assess how suitable the serverless-enhanced system is and under what circumstances; while the other subset relates to performance and scalability variations. To measure the suitability of our solution, we determine if the results of the offloaded simulations fulfill the latency and update step time constraints. Since the latency of the offloaded computation is dependent on the size of the simulated constructs, we variate this size to determine if and under what configurations are the constraints met. Table 6.4 shows the selected experiment configurations for suitability.

# of constructs	Construct size	# of users	Serverless-enhanced
50	{S, M, L}	100	yes

Table 6.4 – Experiment: Suitability of serverless computing.

When measuring the performance and scalability of MVEs, two dimensions need to be taken into account. The MVE server is said to have better scalability if it can handle either: a *higher volume of users, or a higher volume of simulated elements*. To achieve this we variate the number of constructs in our system per configuration, and execute said configurations both in the serverless-enhanced and base MVE implementations. Table 6.5 shows the selected experiment configurations for performance and scalability.

# of constructs	Construct size	# of users	Serverless-enhanced
{50, 100, 200}	S	100	yes
{50, 100, 200}	S	100	no

Table 6.5 – Experiment: Performance and scalability variation.

6.1.3 Metrics

From the proposed experiment configurations, three relevant metrics are kept to track the *latency of serverless function calls, update step time, and CPU consumption*. Latency is measured to determine if offloaded and ahead-of-time computations are performed quickly enough, so that latency and step time constrains are met. A difference between configurations with smaller and larger constructs is expected, as larger constructs take longer to be simulated. Regarding scalability variance, our system is measured in 2 dimensions: (1) the number of users that the server can correctly handle given a fixed number of simulations, and (2) the number of simulated elements that the server can correctly handle given a fixed number of users. We consider any number of simulations or users to be *correctly handled* if the update rate of the server does not drop from the predefined threshold of 20 updates per second, or equivalently it does not take more than 50ms to execute an update step. It is also relevant to measure CPU and RAM consumption values, as it provides insight on hardware resources usage. Table 6.6 lists the selected metrics and their unit of measurement.

Metric	**Unit of measurement**
Serverless function latency	milliseconds (ms)
Update step time	milliseconds (ms)
CPU consumption	percentage (%)
RAM consumption	percentage (%)

Table 6.6 – Experiment: metrics.

6.2 Experiment Setup

The experiments are performed on an environment that is representative of a production deployment. To achieve this we use the DAS-5[1] computer, which allows us to spawn powerful nodes suitable for executing an MVE server, or multiple instances of MVE clients. The following table describes the hardware specifications of the avalible nodes on the DAS-5 computer.

Parameter	**Value**
# of nodes	5
CPU	2x Intel® Xeon® Processor E5-2630 v3
Memory	64GB
Network connection	Gigabit Ethernet + InfiniBand

Table 6.7 – DAS-5 node specifications.

To operate our system we use a scriptable Minecraft CLI client[2], the scripted behavior is set to create the configured amount of simulated constructs and to trigger their offloaded execution. To simulate user workloads we use Yardstick [8], a tool that allows us to emulate the connection and basic interaction of multiple MVE clients. We use in the experiment several Yardstick instances, with a maximum of 50 clients per node. The following is the disposition of nodes on DAS-5 and their usage.

Regarding the cloud environment, AWS Lambda also provides configurable values, from which the following were selected, as to not constrain any computation within the serverless function.

[1]https://www.cs.vu.nl/das5/

[2]https://github.com/ORelio/Minecraft-Console-Client

Node index	**Usage**
0	Opencraft server
1	Minecraft CLI scriptable client
2,3,4	Yardstick instance / 50 users per node

Table 6.8 – DAS-5 node usage.

Parameter	**Value**
Runtime	Java 8
Memory	1GB
Timeout	300 seconds

Table 6.9 – AWS lambda configuration values.

Chapter 7

Experimental Results

This chapter presents the results of the experiments described in Chapter 6. The selected metrics are used to focus the findings around the applicability of serverless computing in MVE simulations, and performance variations between the serverless-enhanced and base implementations. The following list describes the experiments' findings.

F1 Offloading of MVE simulations in serverless functions is achieved with reasonable latency.

F2 The serverless-enabled system can support a higher volume of clients and MVE simulations without overloading.

F3 The serverless enabled system prevents CPU bottlenecks otherwise caused by high volumes of simulated constructs.

F4 RAM used in the MVE server to handle simulated constructs is lower in the serverless-enabled system.

In the following sections we describe in detail how each finding is interpreted from the experimental data.

7.1 Serverless Simulation Latency

The results of this experiment show that the latency of simulations computed through our serverless approach can be reasonable (Finding F1). Figure 7.1 shows the response time of the simulation serverless function calls. The horizontal axis is a logarithmic measure of time, and the vertical axis displays the three size categories defined in the previous chapter. There is a latency threshold set by the combination of update time and ahead-of-time updates requested to the function. For any of the states contained in the function response to be timely, it must return before the threshold has expired. This threshold is represented by the vertical dashed line at 1500ms. As expected, the response time increases with the simulated construct size. Regarding the latency constraints, *small* and *medium* simulated constructs return in a reasonable amount of time, this is, before the latency threshold. For *large* constructs all responses arrive much later, making the pre-computed states usable only if the determinism and periodicity properties hold permanently. Furthermore, the simulation of *large* constructs is affected by higher variability, which is against the goal of achieving stable update times.

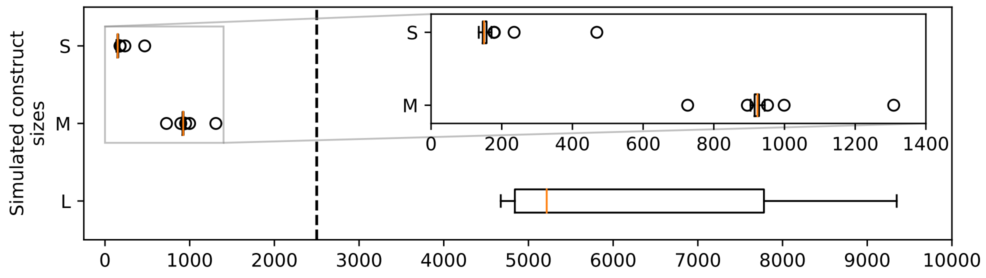


Figure 7.1 – Latency for varying simulated construct size. The dashed line at 2500ms indicates the maximum acceptable latency threshold.

In Table 7.1 request and response payload sizes of the serverless functions are compared. From these values it can be observed that, although the simulations take more time to be performed, the difference in network usage is not drastic.

Simulated construct size	Request payload size	Response payload size
S	4.2 KB	33 KB
M	8.5 KB	34.1 KB
L	33.7 KB	134.9 KB

Table 7.1 – Serverless function request and response payload sizes.

7.2 Update Step Times

The update step times achieved by moving simulations off the MVE server are significantly lower (Finding F2), even with a higher count of simultaneous clients connected. Figure 7.2 shows comparisons of update step times between the proposed system and the normal implementation. The vertical axis represents the time it takes to complete an update step, measured in milliseconds. The bottom horizontal axis, is the runtime of the experiment, measured in seconds. At $t = 0ms$, the MVE state already contains the configured number of simulated constructs. The top horizontal axis describes the number of connected MVE clients there are at that point in time.

The trend described by the lines follows intuition: with higher numbers of simulated constructs and MVE clients, it takes longer to complete an update step. The horizontal dotted line at 50 ms is the configured maximum update step time. This means that both the 100- and 200-simulated constructs base configurations overload the server, it is also observed that these two configurations cause overload or near-overload, even when there are no MVE clients connected. Furthermore, the performance of the 200-simulated construct base configuration is degraded to the point of crashing the simulation thread. It can also be observed that the three serverless-enhanced configurations are able to handle the workload without overloading the server. Comparing the base and serverless-enhanced configurations with the same number of simulated constructs (which share the same color in the figure), it can be seen that the serverless-enhanced solution performs better, achieving lower update step times in all three instances. Comparing specifically the 50-base and 200-serverless configurations, both with update step times below the threshold, an improvement of at least 4x can be observed in reference to the number of supported simulated constructs.

Regarding specific features of the curves of Figure 7.2, we can observe that serverless-enhanced configurations have lower variability, this is, the update step times are more stable throughout the experimental runs. There are also spikes in each curve, which are caused by a periodic process of saving the current MVE state to persistent storage. During the this save period, the simulation and saving processes go through synchronization actions to ensure consistency. Finally, at the beginning of the best-performing curves, there is a dip where update step times drop significantly. This drop is caused by the disconnection of the MVE client which was responsible of configuring all the simulated constructs. After the disconnection of this client, the MVE server is not serving any clients, and thus the simulation loop does not involve any networking, resulting in lower update step times.

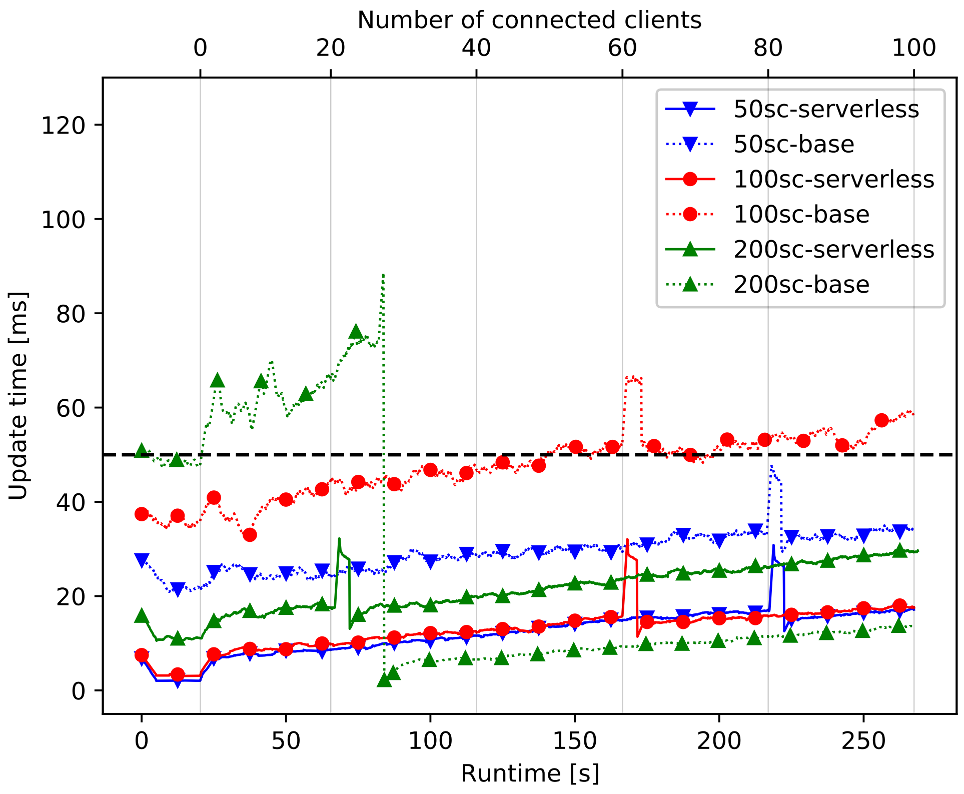


Figure 7.2 – Update step time vs Runtime/Client count.

7.3 MVE Server CPU Consumption

Figure 7.3 shows comparisons of CPU consumption on the same configurations used in Figure 7.2. The vertical axis measures CPU as a percentage value normalized by the maximum amount of CPU usage registered. The bottom horizontal axis, is the runtime of the experiment, measured in seconds. At $t = 0$[ms], the MVE state already contains the configured number of simulated constructs. The top horizontal axis describes the number of connected MVE clients there are at the point in time. In the figure, the lines individually follow a correlation between CPU usage and number of connected MVE clients. Both serverless and base implementation follow a trend with respect to the amount of simulated constructs. In base configurations, CPU usage drops as the amount of simulated constructs increases to 200, this can be attributed to the execution of the MVE server process being throttled due to the high volume of constructs that need simulation, and not meeting the target update rate. Conversely, in serverless configurations CPU usage increases with the amount of simulated constructs, behavior consistent with offloading the simulation of constructs which cause a processing bottleneck (Finding F3). Furthermore, the relation between each base-serverless configuration pair is not similar. For the 50- and 100-simulated construct configurations pairs, it is observed that the serverless-enhanced configurations register a lower usage of CPU. However, in the 200-simulated constructs configuration pair, the CPU consumption of the base configuration is considerably lower.

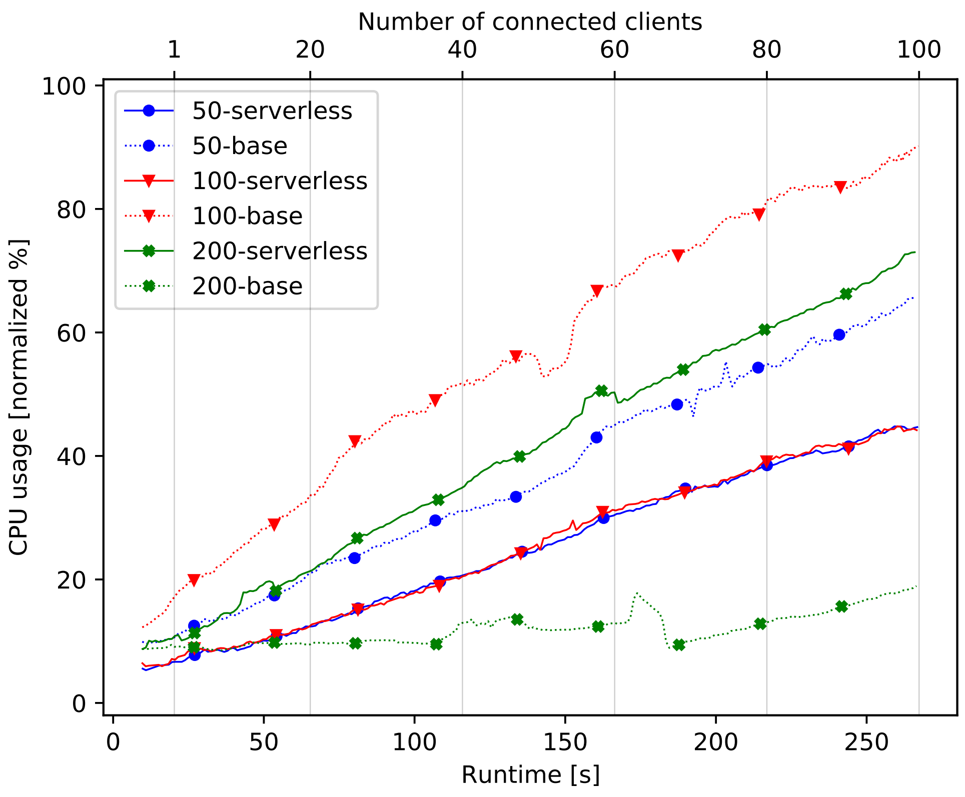


Figure 7.3 – CPU usage vs. Runtime/Client count.

Correlating each measurement on CPU consumption data with its analogous in Figure 7.2, we can observe that the use of CPU is more efficient, at least from the viewpoint of the MVE server. In this context efficiency means using less CPU resources to maintain the update step time below the threshold. It is important to reiterate that the efficiency measured is from the viewpoint of the MVE server, given that, in serverless-enhanced configurations, the total CPU consumption of the MVE server plus the serverless functions is unknown.

7.4 MVE Server RAM Consumption

Figure 7.4 shows comparisons of RAM consumption at the time in which each experiment has completed building all simulated constructs, and before all MVE clients start to connect. The vertical axis measures RAM usage in GBs. The horizontal axis shows the labels each experiment configuration, denoting the number of simulated constructs, and if the base or serverless-enabled implementations are used. Each serverless-base pair shows consistently that the serverless-enabled implementation uses a smaller amount of RAM to hold and process simulated constructs. It can also be seen that for the chosen configurations, the RAM usage is well below the 32 GBs of total RAM available on the server. This shows that the system's demand for RAM is not causing a bottleneck, and is unlikely to cause update step time increments.

However, following the connection on MVE clients on each experiment, RAM usage varies without a discernible pattern. This is shown on Figure A.1, and suggests that RAM usage when clients are present does not correlate to any of the chosen experimental parameters.

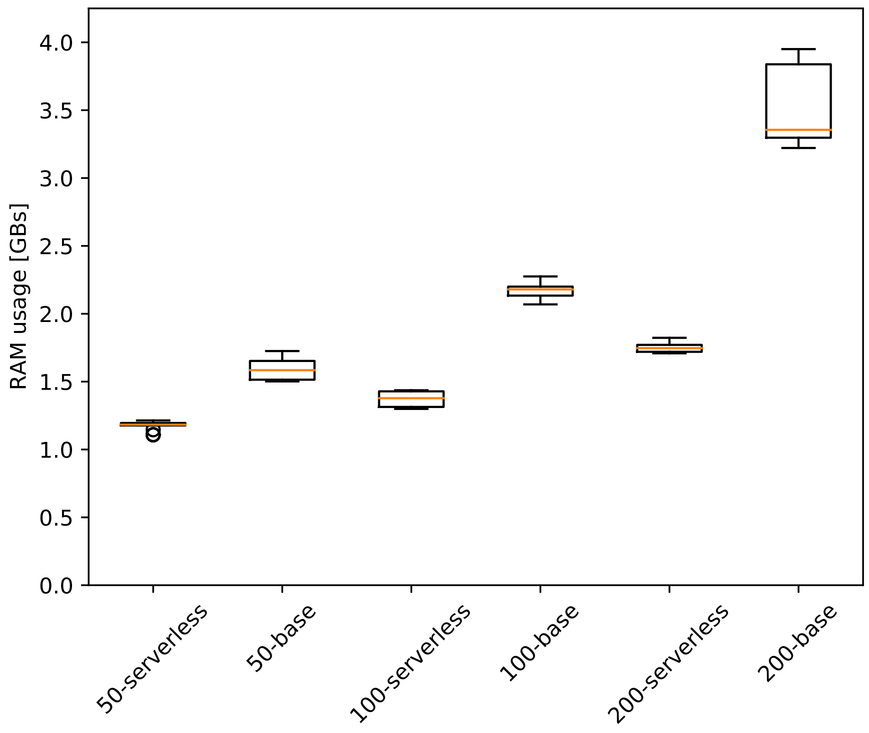


Figure 7.4 – RAM usage on each experiment configuration.

Chapter 8

Discussion

This chapter presents a analysis relating the main experimental findings up to each proposed research question. Also, the work's limitations are discussed, and finally, concerns related to ethics and sustainability are presented.

8.1 Distributing MVE Simulations

In the context of distributed computation and computational offloading, among many other design decisions, managing workload partitions and task granularity are important challenges. In the case of MVEs and their simulation process, and given the nature of their design, a way of partitioning the workload is self-evident. Since the MVE state, is already divided in elemental components, finding a way to arrange and group these elements into self-contained subsets stands out as a reasonable approach. The model of simulated constructs provides a useful method to distribute the workload of MVE simulations. Results show that it is possible to offload only the computationally expensive simulation operations from the server, while freeing the server's fixed resources to either support more of these operations, or support more client connections.

The simulated constructs model presents an alternative to distributed architectures found in the literature, which rely on replicating MVE servers, and use techniques that focus on user activity. In contrast the simulated constructs model is directed dynamically by the content of the MVE state, and is able to leverage the elasticity provided by serverless computing.

8.2 Applicability of Serverless Computing to MVEs

According to the analyzed literature, scalability is a major challenge in commercial MVEs, where restrictions to the number of clients or environment size are in place. An approach based on serverless computing can help loosen those restrictions, while both freeing otherwise busy resources in the MVE server, and allowing the scaling of computational resources in the form of serverless functions. A system based on the simulated constructs model and related experiments, shows that offloading the MVE simulation using serverless functions is a viable approach and can provide performance and scalability enhancements. This system also benefits from synergies with caching and ahead-of-time simulation. Results show that under the correct configuration the number of simulated constructs supported by a single MVE server can be increased by up to 4x by offloading their computations. The same experiment shows that a serverless-enhanced design achieves a higher client capacity given the same amount of simulated constructs, compared to the base implementation.

However, it is clear that only certain configurations are able to achieve this. Dividing the workload into small subsets with periodic behaviour is the most effective configuration. Workload partition into larger tasks, namely large simulated constructs, is not practical due to high latency, unless pre-computed and cached values are valid for very long times. In terms of systems metrics, from the server's perspective, CPU usage is also more efficient. With the use of serverless computing it is able to handle a higher volume of constructs with a lower or equal CPU consumption, when compared to the base implementation. In the best case scenario, server CPU consumption is halved, at the same time that update step times are drastically lowered; in the worst case, server CPU consumption is relatively equal, but at the same time update step times are notoriously lower. Regarding RAM usage both serverless and base configurations perform in a similar capacity. From a high level point of view, the previously mentioned results open the possibility of applying the same concepts to other Virtual Environments, as all benefit from higher counts of users and more efficient resource usage. Furthermore, the simulated constructs model is an arbitrary, but sensible task partitioning solution, which can be used as well in other Virtual Environment applications, to direct system designs capable of achieving similar results.

8.3 Limitations

There are several limitations that this work has encountered. These range from the system's technical limitations, to experiment limitations which threaten the validity of the results and related conclusions.

Most importantly, the serverless-enhanced system is implemented on top of an open-source research platform with known differences with its production counterpart. These differences can be appreciated as divergence of behavior between the implementations. However, since the source and implementation details of a commercial MVE are not available, the specific causes of the divergence are unknown. It is altogether possible that the model of simulated constructs is not as practical given other design and implementation details of MVEs, and therefore, the results derived from the experiments may be incompatible.

MVEs provide an immense design space, covering a high proportion of this design space is a difficult problem. Since simulated constructs are subsets of MVEs they inherit this complexity, which make the task of evaluating them equally difficult. The simulated construct instances chosen for the experiments can be considered a representative sample in the reference frame provided by the simulated constructs model. These instances prove to be useful in validating the simulated constructs model as an instrument for computational distribution. However, real-world, extensive workloads, which explore a broader subset of the design space are not publicly available, which poses a limitation difficult to overcome.

Regarding technical limitations, data transmission rates to serverless functions may not be good enough. Payloads encoded in binary format have the potential to provide more efficient communication between the MVE server and the serverless functions. Furthermore, the response of the functions could be a stream instead of a single response, making the pre-computed states available as soon as they are created, instead of having to wait until the function has finished completely.

8.4 Ethics and Sustainability

The economic and cultural relevancy of virtual environments is rapidly expanding, as applications widen in entertainment and other diverse domains. As society moves towards an extensive use of information technologies in regards of services, work, and entertainment, it is relevant to optimize the means and infrastructure that these technologies use to function. Virtual environments have proven to be one of these technologies. For example, in certain spaces through the COVID-19 pandemic, its use has enabled people to participate in remote work, scientific conferences, and events such as sporting meets, in ways that do not expose users to any danger.

Regarding energy consumption and sustainability, it is known that data center energy consumption accounts for a significant amount of the global energy consumption [40], and also given the existing high volume of MVE users, it becomes of high relevance to make efficient use of the available computational resources. This work however, acknowledges its limitations in terms of measuring the total energy consumption of the proposed system, since energy consumption details of the cloud module and serverless functions are unknown.

Chapter 9

Conclusions

This chapter provides a summary of the work presented in previous sections and possible directions for future work.

9.1 Recapitulation

Modifiable Virtual Environments are widely popular, specially in the video game industry. An example of this is the game Minecraft. Minecraft has a player count reaching into the hundreds of millions. Despite this, current implementations have known limitations regarding scalability. MVEs are generally deployed in client-server architectures, where servers can support, at a maximum, a few hundreds of clients simultaneously. This scalability issue severely hinders interaction between users and communities, and potential use cases (e.g., massive collaboration) are beyond reach.

Related work in MVE scalability has been focused in horizontally scaling out MVE servers using approaches that focus on user actions. However, moving away from client-server architectures has also been proposed, specifically in the form of peer-to-peer or hybrid architectures. Yet, distribution through serverless computing has only recently been proposed [7], but no attempt at designing or gathering experimental data has been presented.

Therefore, this thesis explores the feasibility of distributing the simulation process of Modifiable Virtual Environments over a serverless computing platform, with the main objective of increasing their scalability. To achieve said objective, we formally define research questions regarding: an appropriate distribution model, a system design harnessing said model, and observing performance and scalability benefits. We develop the research questions as separate chapters, each making specific, but connected contributions which are the core of this work.

RQ1 How can MVE simulations be expressed in a model compatible with distributed computing?

The presented distribution model exploits design properties of MVEs, and relies on two main components of an MVE server: the MVE state and MVE simulation. Since the MVE state is expressed in elemental units, defined as simulated elements, the model proposes arranging and grouping these elements into self-contained subsets: simulated constructs. Simulated constructs are arbitrary arrangements of simulated elements, but with the characteristic of being perceived by users as single entities. We are then able to simulate each construct as independent and parallel tasks.

RQ2 How can serverless computing be used in MVE simulations to improve scalability?

We present as system design to take advantage of the simulated constructs model. The system consists on two additional modules to a traditional MVE client-server architecture: a client-side module, and a cloud module. The client-side module is responsible of selecting the simulated constructs from the MVE state, and triggering their simulation in serverless functions, as well as synchronizing the results back into the MVE state. The cloud module consists of a serverless function that receives a serialized MVE state subset and simulates it a certain number of steps ahead. The server-side module also implements a cache where results from the serverless functions are stored for re-use, a feature possible because of periodical and deterministic properties of some simulated constructs.

RQ3 How to evaluate the effectiveness of using serverless computing in MVE simulations?

We present experiments to test the suitability and measure performance and scalability variations compared to a base implementation of an MVE. The results show that the simulated construct model, applied in a serverless computing setting, and under the correct configuration, is in fact a viable approach for distributing the MVE simulation. Regarding performance and scalability, the results show a substantial increase in the volume of users and simulated constructs that the system can correctly handle, as well as a very clear differences in CPU and RAM usage patterns.

9.2 Future work

The system presented on this thesis can be extended in several dimensions. A limitation of the implemented system is that it is not able to perform workload partitioning autonomously, this means that simulated constructs have to be selected by a purposeful user. A mechanism to automatically detect when a combination of simulated elements constitutes a simulated construct and automatically trigger an offloaded computation is a relevant topic.

While a caching component is used for re-usability of computations, other caching-related approaches remain to be explored. For example, pre-computed simulations may be shared between MVE servers by using a server-agnostic encoding, given that the simulation rules are the same for a set of servers. This approach allows for a shared cache layer that could achieve better performance results.

A complete evaluation on the effectiveness of the specific caching and speculative execution components remains to be done. From a computer systems' perspective, a thorough analysis of said components is very relevant to due to very specific characteristics of the workload. Furthermore, different configurations and policies can be compared to find more efficient, and better performing solutions.

Finally, the contributions from this thesis: the simulated constructs model, the serverless-enhanced system, and experimental results can be analyzed in a more general context, given that access to commercial MVE implementations and real-world workloads is obtained.

Appendix A

Additional MVE Server RAM Consumption Figure

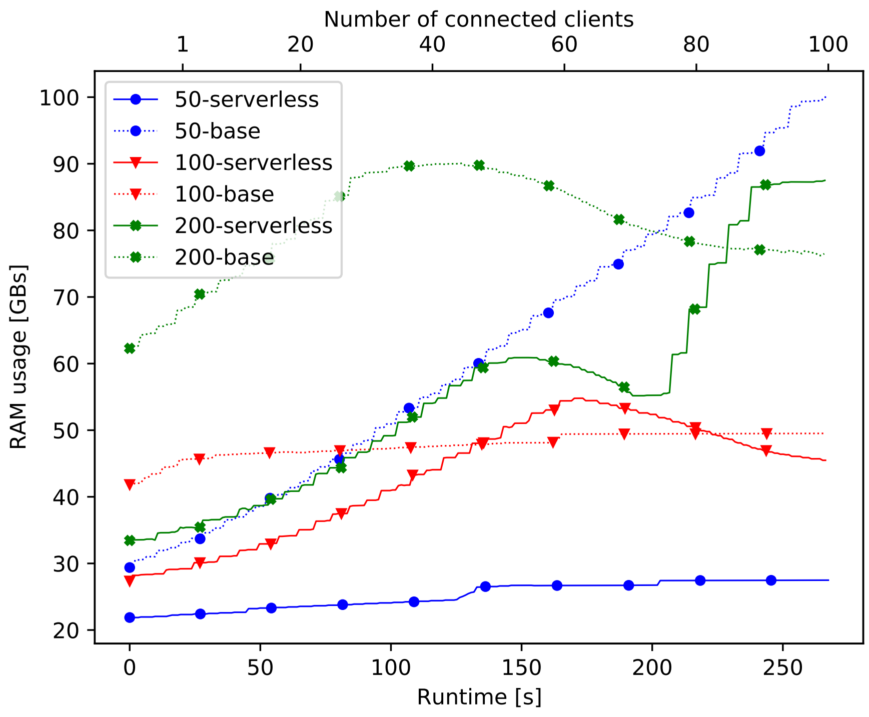


Figure A.1 – RAM usage vs. Runtime/Client count.

References

[1] Newzoo, "Newzoo Global Games Market Report 2021," https://newzoo.com/insights/trend-reports/newzoo-global-games-market-report-2021-free-version/, 2019, accessed: 2021-07-26.

[2] D. Curry, "Minecraft Revenue and Usage Statistics (2021)," https://www.businessofapps.com/data/minecraft-statistics/, June 2021, accessed: 2021-07-26.

[3] "Minecraft: Education Edition," https://education.minecraft.net, accessed: 2020-02-08.

[4] A. Natividad, "How Greenpeace Used Minecraft to Stop Illegal Logging in Europe's Last Lowland Primeval Forest," https://bit.ly/MinecraftGreenpeace, Jan 2018, accessed: 2020-02-08.

[5] "Block By Blockwest: the closest we've come to an authentic online festival," https://bit.ly/3lg9KFS, accessed: 2020-07-20.

[6] J. Blascovich, *Social Influence within Immersive Virtual Environments*. London: Springer London, 2002, pp. 127–145. ISBN 978-1-4471-0277-9. [Online]. Available: https://doi.org/10.1007/978-1-4471-0277-9_8

[7] J. Donkervliet, A. Trivedi, and A. Iosup, "Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, jul 2020. [Online]. Available: https://www.usenix.org/conference/hotcloud20/presentation/donkervliet

[8] J. van der Sar, J. Donkervliet, and A. Iosup, "Yardstick: A Benchmark for Minecraft-like Services," in *ICPE*, 2019.

[9] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*, A. Clements and T. Condie, Eds. USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[10] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, and R. Buyya, Eds. Springer, 2017, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-981-10-5026-8_1

[11] H. Shafiei and A. Khonsari, "Serverless computing: Opportunities and challenges," *CoRR*, vol. abs/1911.01296, 2019. [Online]. Available: http://arxiv.org/abs/1911.01296

[12] A. Bhattacharya and P. De, "A survey of adaptation techniques in computation offloading," *J. Netw. Comput. Appl.*, vol. 78, pp. 97–115, 2017. doi: 10.1016/j.jnca.2016.10.023. [Online]. Available: https://doi.org/10.1016/j.jnca.2016.10.023

[13] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. doi: 10.1145/1327452.1327492. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, 2008.

[15] G. Czajkowski, M. Dvorský, J. Zhao, and M. Conley, "Sorting Petabytes with MapReduce - The Next Episode," 2018, accessed: 2020-04-28.

[16] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, 1st ed. McGraw-Hill Higher Education, 1992. ISBN 0070316228

[17] B. W. Lampson, "Lazy and speculative execution in computer systems," in *Proceeding of the 13th ACM SIGPLAN international conference*

on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008, J. Hook and P. Thiemann, Eds. ACM, 2008. doi: 10.1145/1411204.1411205 pp. 1–2. [Online]. Available: https://doi.org/10.1145/1411204.1411205

[18] J. E. Smith, "Retrospective: A study of branch prediction strategies," in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, G. S. Sohi, Ed. ACM, 1998. doi: 10.1145/285930.285940 pp. 22–23. [Online]. Available: https://doi.org/10.1145/285930.285940

[19] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982. doi: 10.1145/356887.356892. [Online]. Available: https://doi.org/10.1145/356887.356892

[20] A. Dan, D. F. Towsley, and W. H. Kohler, "Modeling the effects of data and resource contention on the performance of optimistic concurrency control protocols," in *Proceedings of the Fourth International Conference on Data Engineering, February 1-5, 1988, Los Angeles, California, USA*. IEEE Computer Society, 1988. doi: 10.1109/ICDE.1988.105486 pp. 418–425. [Online]. Available: https://doi.org/10.1109/ICDE.1988.105486

[21] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," *Comput. Commun. Rev.*, vol. 45, no. 5, pp. 379–392, 2015. doi: 10.1145/2829988.2787481. [Online]. Available: https://doi.org/10.1145/2829988.2787481

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012. ISBN 978-0-12-383872-8

[23] J. Wang, "A survey of web caching schemes for the internet," *Comput. Commun. Rev.*, vol. 29, no. 5, pp. 36–46, 1999. doi: 10.1145/505696.505701. [Online]. Available: https://doi.org/10.1145/505696.505701

[24] R. Diaconu and J. Keller, "Kiwano: A scalable distributed infrastructure for virtual worlds," in *2013 International Conference on High Performance Computing Simulation (HPCS)*, 2013. doi: 10.1109/HPCSim.2013.6641489 pp. 664–667.

[25] R. Diaconu, J. J. Keller, and M. Valero, “Manycraft: Scaling Minecraft to Millions,” in *Annual Workshop on Network and Systems Support for Games, NetGames ’13, Denver, CO, USA, December 9-10, 2013*. IEEE/ACM, dec 2013. doi: 10.1109/NetGames.2013.6820617. ISBN 978-1-4799-2961-0. ISSN 21568146 pp. 1:1—-1:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2664635http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6820617

[26] D. Lake, M. Bowman, and H. Liu, “Distributed scene graph to enable thousands of interacting users in a virtual environment,” in *9th Annual Workshop on Network and Systems Support for Games, NetGames 2010, Taipei, Taiwan, 16-17 November, 2010*. IEEE, 2010. doi: 10.1109/NETGAMES.2010.5679669 pp. 1–6. [Online]. Available: https://doi.org/10.1109/NETGAMES.2010.5679669

[27] D. Horn, E. Cheslack-Postava, B. F. Mistree, T. Azim, J. Terrace, M. J. Freedman, and P. Levis, “To infinity and not beyond: Scaling communication in virtual worlds with meru,” *tech. report CSTR 2010-01 5/11/09*, 2010.

[28] Elfizar, M. S. Baba, and T. Herawan, “Object-based simulators for large scale distributed virtual environment,” in *Proceedings of the Second International Conference on Advanced Data and Information Engineering, DaEng 2015, Bali, Indonesia, April 25-26, 2015*, ser. Lecture Notes in Electrical Engineering, vol. 520. Springer, 2015. doi: 10.1007/978-981-13-1799-6_2 pp. 11–19. [Online]. Available: https://doi.org/10.1007/978-981-13-1799-6_2

[29] I. B. Vilardell, C. Roig, and F. Giné, “Distributing game instances in a hybrid client-server/p2p system to support MMORPG playability,” *Multim. Tools Appl.*, vol. 75, no. 4, pp. 2005–2029, 2016. doi: 10.1007/s11042-014-2389-0. [Online]. Available: https://doi.org/10.1007/s11042-014-2389-0

[30] B. Anand and A. J. H. Edwin, “Gamelets - multiplayer mobile games with distributed micro-clouds,” in *Seventh International Conference on Mobile Computing and Ubiquitous Networking, ICMU 2014, Singapore, January 6-8, 2014*. IEEE Computer Society, 2014. doi: 10.1109/ICMU.2014.6799051 pp. 14–20. [Online]. Available: https://doi.org/10.1109/ICMU.2014.6799051

[31] V. Nae, R. Prodan, and A. Iosup, "Autonomic operation of massively multiplayer online games in clouds," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC '13. New York, NY, USA: Association for Computing Machinery, 2013. doi: 10.1145/2494621.2494629. ISBN 9781450321723. [Online]. Available: https://doi.org/10.1145/2494621.2494629

[32] Y. Gao, L. Wang, Z. Xie, W. Guo, and J. Zhou, "Energy-efficient and quality of experience-aware resource provisioning for massively multiplayer online games in the cloud," in *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11236. Springer, 2018. doi: 10.1007/978-3-030-03596-9_61 pp. 854–869. [Online]. Available: https://doi.org/10.1007/978-3-030-03596-9_61

[33] "Compute@Edge: porting the iconic video game DOOM," https://www.fastly.com/blog/compute-edge-porting-the-iconic-video-game-doom, accessed: 2021-05-31.

[34] L. Toader, A. Uta, A. Musaafir, and A. Iosup, "Graphless: Toward serverless graph processing," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2019. doi: 10.1109/ISPDC.2019.00012 pp. 66–73.

[35] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017. ISBN 978-1-931971-37-9 pp. 363–376. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[36] M. K. Stojcev, Y. N. Patt, and S. J. Patel, "Introduction to computing systems: From bits and gates to C and beyond second edition, mcgraw-hill higher education, boston (2004) ISBN 0-07-121503-4 softcover, pp 632, plus XXIV," *Microelectron. Reliab.*, vol. 45, no. 2, pp. 405–406, 2005. doi: 10.1016/j.microrel.2004.08.010. [Online]. Available: https://doi.org/10.1016/j.microrel.2004.08.010

[37] J. Schell, *The Art of Game Design: A book of lenses*. CRC press, 2008.

[38] S. Ginzburg and M. J. Freedman, "Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms," in *WoSC@Middleware 2020: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Virtual Event / Delft, The Netherlands, December 7-11, 2020*. ACM, 2020. doi: 10.1145/3429880.3430099 pp. 43–48. [Online]. Available: https://doi.org/10.1145/3429880.3430099

[39] J. Donkervliet, J. Cuijpers, and A. Iosup, "Dyconits: Scaling minecraft-like services through dynamically managed inconsistency," in *ICDCS*, 2021.

[40] J. Mao, T. Bhattacharya, X. Peng, T. Cao, and X. Qin, "Modeling energy consumption of virtual machines in dvfs-enabled cloud data centers," in *39th IEEE International Performance Computing and Communications Conference, IPCCC 2020, Austin, TX, USA, November 6-8, 2020*. IEEE, 2020. doi: 10.1109/IPCCC50635.2020.9391552 pp. 1–6. [Online]. Available: https://doi.org/10.1109/IPCCC50635.2020.9391552

TRITA -EECS-EX-2021:716