Vrije Universiteit Amsterdam

Bachelor Thesis

Design and Evaluation of a Cloud operated Storage System for Minecraft-like Games

---

Author: Yann Regev (2616577)

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

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

August 12, 2020

Abstract

The video game industry is immensely large, grossing over $150 billion dollars in revenues in 2019. Video game servers require high performance machines to operate, this poses financial issues for small game companies. Minecraft is a voxel-based game allowing players to modify a MVE by mining and placing blocks to build structures. Minecraft-like games and MVEs, offers players an endless virtual world to explore and modify. This, combined with the large number of player, provides motivation for scaling the games storage. To address these challenges, this thesis presents the design of a cloud-operated storage service for Minecraft-like games. To evaluate the system, we conduct real-world experiments. Our results show that using cloud operated storage reduces local storage usage, and can read and write game data from cloud operated storage while meeting the QoS required by MVEs. This system provides a first step to turn Minecraft-like games from monolithic to serverless architectures.

Contents

1 Introduction **4**
- 1.1 Problem statement . . . 4
- 1.2 Main contributions . . . 5
- 1.3 Thesis structure . . . 5

2 Background **6**
- 2.1 Modifiable virtual environments . . . 6
- 2.2 Serverless computing . . . 8

3 System design **9**
- 3.1 System requirements . . . 9
- 3.2 Design overview . . . 10
- 3.3 Layered storage design . . . 11
- 3.4 System parameters . . . 11
- 3.5 External tools . . . 12

4 Experimental setup **14**
- 4.1 Environment . . . 14
- 4.2 Yardstick . . . 14
- 4.3 Workload . . . 14
- 4.4 Data collection . . . 15

5 Experimental results **16**
- 5.1 World storage parameters results . . . 17
- 5.2 Latency hiding policies . . . 18
- 5.3 Related work . . . 18

6 Conclusion **18**

7 Future work **19**

1 Introduction

Online computer games are a billion dollar per year industry [10], with millions of players enjoying video games across the globe. The sizable community, consuming video games regularly, has brought with it a large and thriving market for video games.

With over a 176 millions copies sold, Minecraft is one of the most popular games of all times [13]. It offers players an effectively infinite and modifiable virtual world and defines no specific objectives, allowing players a large amount of freedom when choosing how to play the game. Minecraft is a voxel-based game, consisting of 3D blocks which represent different elements such as dirt, stone, ores, tree trunks, water, and lava. Players play the game by mining and placing blocks to build structures.

Minecraft offers online multiplayer sessions through LAN, local split-screen, game as a service, and servers self-hosted by private owners and businesses. Existing deployments of Minecraft servers do not scale well, requiring more resources to support more players [15].

Because of the poor scalablity of Minecraft-like games, only big companies can afford the activity of hosting Minecraft-like game servers. Therefore, alternative and cheaper solutions are sought to make this activity more affordable and available.

To scale the game, J. Donkervliet et al. [14], present a vision of scaling Modifiable Virtual Environments (MVEs) using serverless and cloud computing. Serverless computing is a type of cloud computing service which allows users to run event-driven applications where users pay for the resources utilized, and the operational logic is hidden from the user [7]. Furthermore, resource capacity is potentially much larger than privately owned hardware. Therefore, we conjecture that it can provide a solution to the scalability of data storage issues of Minecraft and Minecraft like games. Using serverless computing offers several advantages. Firstly, the resources are managed by the cloud provider. Secondly, the clients pay for the resources they use. Lastly, when the server is split to services it becomes more modular and therefore easier to add and maintain features.

Despite growing usage of serverless computing, the latency and performance of these services remains poorly documented. Low latency is a key requirement for playing online games. While the low latency requirements in games such as real-time strategy and RPG games is linear, in first-person games like Minecraft requiring high precision, the performance in precision and accuracy decreases in an exponential decay with a sharp drop in accuracy at around 100ms [2]. This shows that game-play experience degrades most noticeably in games with avatar game-model especially in the first-person perspective. Therefore, the possibility of running real-time systems using serverless technology remains unknown.

1.1 Problem statement

MVEs offer players an endless virtual world to explore and modify. Using procedural generation, the game creates world data as players explore the world [4]. In addition, Minecraft creates data such as location, items, experience level, and appearance of every player that has joined the server. Minecraft's monolithic architecture poses scalability challenges for data storage, due to the game data being stored locally, building up as more content is created.

In this thesis, we propose a new system that uses cloud operated storage instead of

traditional local storage for Minecraft-like games. We believe that using cloud operated storage will reduce the upfront storage required by self-hosted server owners, and will therefore make it more affordable to maintain privately owned server.

The challenge of using cloud operated storage is to meet the QoS required for MVEs. MVEs require data availability and consistency. Data stored in local storage, is quickly available to the game server. On the other hand, cloud operated storage latency remains poorly documented. However, because it relies on network latency and bandwidth, it is safe to assume that accessing files on cloud operated storage is much slower than locally stored files. In addition, cloud operated storage systems often exhibit eventual consistency [1]. This means the server may read stale data from the cloud, and data consistency is only eventually guarantied. There is no research on how this can affect QoS of MVEs. Therefore, the effects remain unknown.

This thesis, aims to examine the performance of using the cloud operated storage to store the world data of a Minecraft server, while meeting the QoS required. This project offers to give insight on the possibility of running Minecraft as serverless systems. This thesis designs and evaluates a cloud-based storage system for Minecraft. The main research questions of this project are:

1. How to design a cloud operated storage service for serverless Minecraft-like games or MVEs, and how to meet the Quality of Service required for such games?
2. How to allow the user (game developer) to fine-tune the systems behavior, to trade-off local storage and network usage to meet their specific requirements?
3. How to evaluate the performance of such a system?

By addressing these points, we hope to give insight on the possibility of running Minecraft and Minecraft-like games on serverless platforms with relative ease, and at a fraction of the cost of conventional platforms.

1.2 Main contributions

The main contributions of this paper are:

MC1 The Design and implementation of a Minecraft server using cloud operated persistent storage, that allows developers to trade-off local storage and cloud storage.

MC2 The evaluation of this system, by conducting real-world experiments.

1.3 Thesis structure

The remainder of the thesis is structured as follows. Section 2 gives background information about Minecraft-like games and MVEs, and serverless computing. Section 3 presents a design of a cloud operated persistent storage system for Minecraft-like games, and describes its requirements. Sections 4 and 5 describe the setup of the experiments, the tools and environment that were used, and the analysis of the results.

2 Background

This section elaborates on the architecture of Modifiable Virtual Environments (MVEs), and on the persistent storage model of Minecraft and the scalability issues that exist in the current model. Section 2.1 explains what a Modifiable Virtual Environment is, the different elements that comprise the Minecraft universe, and the entities that exist in the game. In addition, the current storage model of the game, and the problem that arises from it. Section 2.2 discusses the background about serverless computing.

2.1 Modifiable virtual environments

Modifiable Virtual Environments (MVEs) are real-time, online, multiplayer environments which allow users to modify the world's parts, create new content by combining different components, and interact with the world through programs [5]. MVEs typically use a client server architecture, where the user runs a client which send and receive continuous updates between the user and the server. The server simulates the changes to the world and sends the new state to all the clients. The clients hold a cache of the surrounding of the avatar. The client can interact with the world by performing various different MVE actions (e.g. mining blocks, crafting items). Changes to the world are simulated locally while also sent to the server, this is done to give the user a hide latency from the user. The server is typically a multi-threaded monolithic application that has several important roles. Firstly, to keep a persistent copy of the different game states, this includes world, player locations, inventories, and world metadata (weather, time of day, etc.). Secondly, it simulates changes in the world such as Non Playable Characters, and world events (weather, time). Lastly, as players explore the world, the server dynamically generates newly explored areas of the world.

2.1.1 Minecraft worlds

Minecraft comprises three different worlds: the over-world, the nether, and the end. The over-world is the dimension where all players begin their journey. It is divided in biomes which determines the characteristic of the terrain inside it. The biomes type also determines the weather, mobs, and size existing inside of it. The nether is a dangerous hell-like dimension. To access this dimension players must construct a portal that will allow them to travel between the dimensions. The reason players would typically want to access this dimension is to gather resources which do not exist elsewhere. The nether is a cave-like location filled with lava, fire, and dangerous monsters. The end is a dark space-like dimension consisting of separate islands. To access this dimension players must find a special location in the world where they can construct a portal. The end consists of one large island surrounded by several smaller islands. In order for the player to travel back to the over-world the player must die or defeat a monster called the end dragon. The worlds are divided into smaller pieces. A world is comprised of innumerous amount of regions, regions are comprised of 32x32 chunks, a chunk is comprised of 512x512 blocks. Figure 1 depicts the structure of a region.

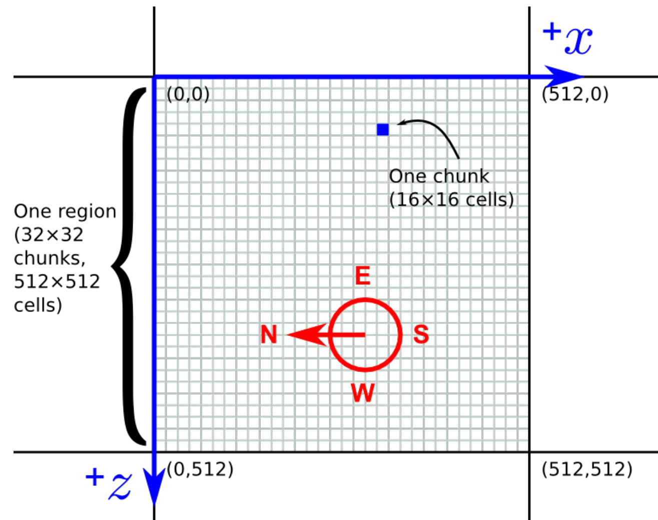


Figure 1: Representation of the Minecrafts world mapping.

2.1.2 Players

Players are controlled by client software. They can modify the world by “mining” blocks and placing them elsewhere to build things, players can also craft items by combining different blocks together.

In order to store blocks and other items each player has an inventory consisting of 27 storage slots, in addition there are eight slots called quick-bar for quick access. When a player logs out of the server information about the players such as location, inventory, avatar, and equipped items are stored in persistent memory in a unique player data file.

2.1.3 Non playable characters

Non playable characters (NPCs), also known in Minecraft as mobs, are “living” entities which are affected by the world in a similar way that a player is. Mobs can be divided into three different categories: passive, neutral, and hostile. Passive mobs are harmless to players and will not attack back if provoked. This type consists of livestock, villagers, and fish. Neutral mobs will only attack when they are provoked. Hostile mobs will attack players when he is in a certain range. Mobs existing on chunks that have been offloaded from the game-loop are stored in persistent memory and remain unchanged until they are loaded again.

2.1.4 Storage model

Minecraft generate the content of the world dynamically and stores the data that is created for current and future use on the servers local storage, this includes both player data, and world data. The server includes a cache where locations recently visited by players are loaded. This means that the more players join and explore the world the more content is created, and the more capacity is needed to store the data.

2.2 Serverless computing

Serverless computing is a form of cloud computing which allows users to run event driven operations and pay for the execution time and resources used [8]. This allows developers to focus on high level abstractions, leaving the lower end such as hardware and supporting services to be mapped by the cloud operators. Serverless computing adheres to three principles

1. Operational logic is hidden from the user.
2. Users only pay for the resources they utilize.
3. The computational model is event driven [7].

Due to high cost, software used to be developed as monolithic architecture to fit on one server. However, this meant that every change to the code could bring the entire server down. Serverless computing, brought a change due to the ability to provision resources, lowering prices. Developers now had the possibility to build applications as microservices where each service is made into Functions-as-a-Service (FaaS) [6]. A function is an executable code that runs on demand with the arrival of an event [11]

2.2.1 Cloud operated persistent storage

Cloud operated persistent storage is a type of serverless computing which has three important properties. It should not provision its users, storage should scale to fit without user intervention, and the user should pay for the amount of data stored and amount of requests made. When using traditional storage, over-provisioning is the only method available to prepare for traffic spikes. The elasticity of Cloud operated storage provides a solution for over-provisioning. In addition, by paying only for the resources utilized, the cost scales with storage used, preventing the users paying for unused storage, unlike traditional storage.

Amazon S3, is a cloud based persistent storage. S3 is expected to offer low data-access latency, 99.999999999% durability per object, 99.99% availability, and eventual consistency. Since its launch, S3 has acquired a large range of costumers for private users to small and large businesses [?]. Charging for S3 is based on storage volume, and per requests and data retrieval.

S3 organizes user data in buckets. Each bucket can store an unlimited amount of objects. Each object has a name, data, and metadata.

Evaluation of Amazon S3 by M. Palankar et al. [12], show that S3 has data durability and high availability. However, D Bermach et al. [1] show that S3 eventual consistency can lead to inconsistencies in data. To the best of our knowledge, limited research has been done on incorporating Cloud operated storage in Minecraft-like games.

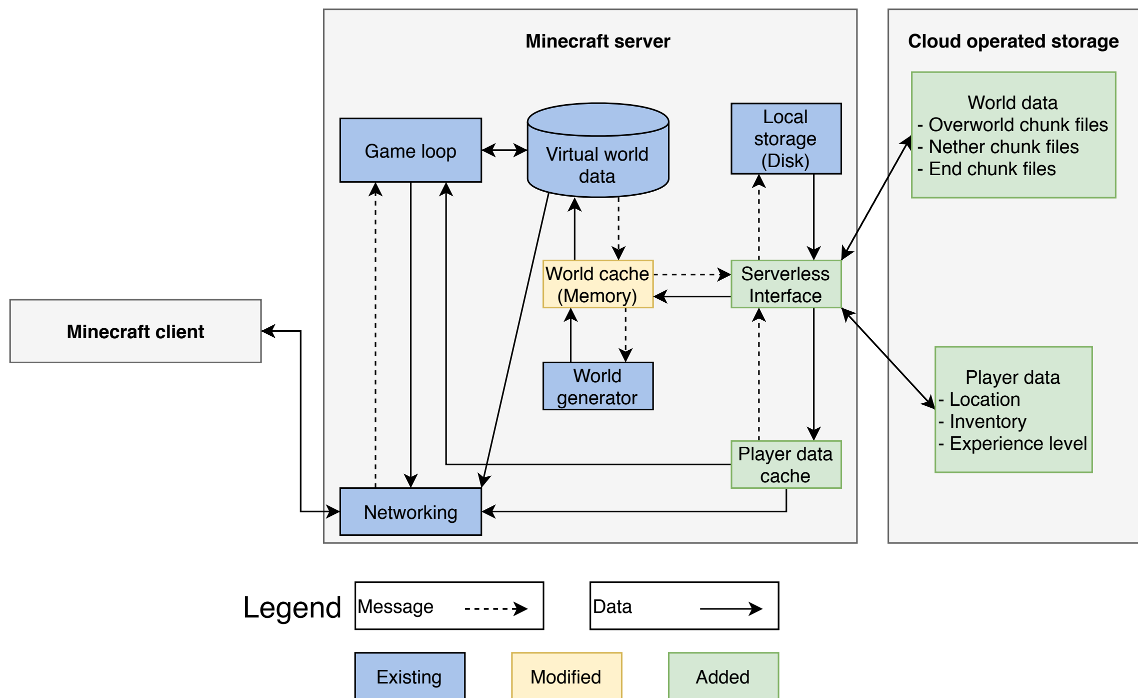


Figure 2: Cloud operated storage model

3 System design

This section presents a design of a serverless/cloud-based persistent storage system for Minecraft-like games. Section 3.1 describes the requirements of the system, and of the cloud operated storage. Section 3.2 describes the design of the Minecraft server with Cloud operated storage, and presents the layered storage design of the system. Section 3.3 Describes the layered storage design used in the system. Section 3.4 Describes the system parameters adjustable by the user, how they affect storage and network use, and how they can be used by developers. Section 3.5 Describes the external tools used in the system.

3.1 System requirements

This section, gives the requirements that need to be met for this system to be a viable solution to the scalabily issues of Minecraft-like games.

R1 Enable running MVEs on devices with limited storage.

R2 Hide read/write latency from the user.

R3 Must read chunks from cloud operated storage before they are in a players view port.

For Cloud operated storage to provide a viable solution for Minecraft-like game storage it must provide the following:

- Durability: Loss of data is costly or even unacceptable as players might lose progress they have made.

- Availability: Player must have access to all the world and their individual player data. Therefore, world data and player data must be available at any given time.
- Performance: If data retrieval is too slow, players might experience missing chunks in the world leading to lose of immersion. Therefore, fast data access is an important feature.
- Consistency: Players sharing the same world must see the same world and experience the same events. Therefore, consistency in data between players is important.

3.2 Design overview

This subsection describes the changes made to the server to support storage on c.loud operated storage. One of the biggest changes to the server is turning the storage system into a layered design. Figure 2 illustrates the different layers, the new components are shown in green. The first layer is the cloud operated storage, this layer hold data that is currently unused by the server. The serverless interface is the gate to request files, the interface determines if the file exists locally or on the cloud and retrieves the file to the server if it exists. The second layer is the local cache, this layer holds data for a specified integer time after it has been closed by the server. When the time has elapsed, the file is written to the online storage. The third layer consists of the player and world cache, this layer hold files currently read or written to by the server. The server cache holds the files for a specified integer amount of time. When the timer has expired the file is closed and kept in the local storage cache.

3.2.1 World data storage

The world data is currently stored in region files which hold 32x32 chunks, the server opens the entire region file for editing while it loads the individual chunks when in the proximity of players, this causes more data to stay locally while it could be stored on cloud storage. By splitting up region files to their individual chunk it will therefore be possible to decrease the amount of local storage required. Figure 2 shows the caching layer between the storage and the server is in place in order to reduce loading time for areas that are currently or have been recently explored by players. When a player reaches a chunk that is not currently loaded, the cache layer is searched for the chunk. When a cache miss occurs the cloud storage is searched for the existence of the file. If the file exists its data is downloaded to the cache and loaded into the game

3.2.2 Player data storage

Player data consist of the current location of the player, the contents of his inventory, and the players current experience points. The file should be modulated into different components, some components are always displayed to the player and cannot be unloaded, but other less frequently accessed data should be stored in cloud storage and moved to the cache and loaded to the server once the player accesses it.

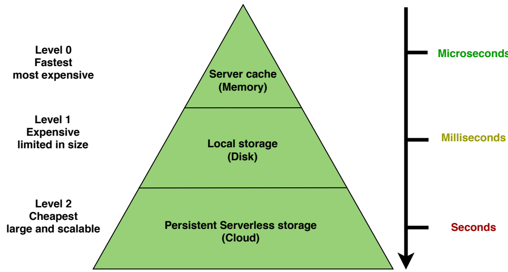


Figure 3: Layered storage design.

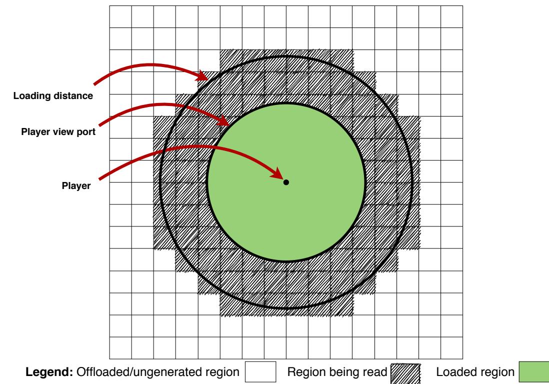


Figure 4: Depiction of the adjustable parameter loading distance.

3.3 Layered storage design

The system is designed to have a layered storage shown in Figure 3. Level 0 is the server cache, it exists in the servers memory and therefore the fastest to access, this level of storage contains files currently being observed and modified by users. It is the most costly level as it requires memory and storage, and it increases searching time in the cache. Level 1 is local storage, files in this storage have a caching time after being offloaded from the server cache. This level offers quick access to files, however high caching times will increase reliance on local storage. Level 3 is the Cloud operated storage. This level is the slowest to access but has a large cheap storage space. Files on Cloud operated storage await unmodified until required by the server. Level 0 and level 1 have a caches and therefore have adjustable caching time. The trade-off of adjusting the caching time is between reliance on local storage and latency.

3.4 System parameters

There are three parameters modifiable by the host. They are adjustable to set the trade-off between local storage reliance and the cloud storage latency, to compensate for high latency or small storage space. The first parameter, is the integer time of the world cache. This parameter lets the developer adjust the amount of time region files exist in the server cache. When a region file is loaded to the server the cache timer starts, when it expires the file is closed from the game and moved to local storage. Changing this parameter allows the host to adapt the behavior for the amount of memory and local storage available, which corresponds to **R1**, longer times will require more memory and storage but might be preferable when the network has high latency. The second parameter, is the integer time of the local storage cache. This parameter lets the developer adjust the time a file exists on the servers local storage, when the timer expires the file is written to cloud storage and removed from local storage. Longer times will increase the local storage use but in the case the network has high latency or the cloud operated storage has high reading times, this corresponds to **R1**. The third parameter, is the distance from the player the world files are read from cloud operated storage. Figure 4 depicts the distance port, and

the different stages of the regions before being displayed to the user. This parameter lets the developer decide on the optimal distance from a player for a chunk to be read from cloud storage to local storage. Based on the reading speed from cloud storage, this parameter should be modified in order for the players not to experience missing chunks or corrupt data. These parameters address the requirements by giving flexibility to the game developer to allowing the trade-off between local storage or network reliance, this corresponds to **R2**, and **R3**.

3.5 External tools

This subsection gives details about the external tools used in the system. Section 3.5.1 gives information about the Glowstone server and the modification required for it to work with cloud operated storage. Section 3.5.2 gives information on Amazon S3 cloud operated storage, and gives some example code used in the system.

3.5.1 Glowstone adaptation

Glowstone is an open source lightweight minecraft server written from scratch. Its main goal is to provide a lightweight implementation of the bukkit API[1] , its simplicity affords it a performance improvement over the original software Minecraft provides. Glowstone creates the first four regions of each world when the server is first started. The different worlds are generated with the use of a seed, it is an integer which represents a starting point for the world generation formula. It provides a thread per world model and provides synchronization only when required by the bukkit API.

The Minecraft server region file class is adapted to work with cloud operated storage. There are two main changes to enable the use of cloud operated storage while still maintaining QoS. The Minecraft server is modified to work with cloud operated storage. One of the changes was adding the ability to load and unload region files from local storage to the cloud, this is done in the serverless interface. to determine whether a file can be safely uploaded to cloud storage, the server determines the distance of the players from the region and whether further changes should be made to the file. To determine if a file should be read from the cloud, every pulse the players position was evaluated and when sufficiently close, the file is retrieved to local cache where it could be used by the server. Another change is the addition of a player chunk distance measurement (Loading-port) shown in Figure 4. This was necessary to be able to hide the reading time from the cloud. This measurement was used to load a chunk at distance greater than the players field of view to avoid reading missing chunks or corrupt data.

```
1 if (!path.exists() && s3.doesObjectExist(bucketName, fileName)) {
2     S3Object object = s3.getObject(new GetObjectRequest(fileName));
3     FileUtils.copyInputStreamToFile(object.getObjectContent(), path);
4 }
```

Listing 1: Querying and reading a file from S3

[1] https://bukkit.gamepedia.com/Main_Page

```
for (int x = centralX - radius * distance; x <= centralX + radius *
    distance; x++) {
    for (int z = centralZ - radius * distance; z <= centralZ + radius *
        distance; z++) {
        File f = new File(filename + (x >> 5) + "." + (z >> 5) + ".mca");
        if ((!f.exists()) && (s3.doesObjectExist(bucketName, filename))) {
            try {
            f.createNewFile();
            final int x1 = x;
            final int z1 = z;
            new Thread(() -> {
            try {
                S3Object object = s3.getObject(new
                    GetObjectRequest(bucketName, filename));
                FileUtils.copyInputStreamToFile(object.getObjectContent(),
                    f);
            } catch (IOException e) {
                e.printStackTrace();
            }
            }).start();
        } catch (IOException e) {
            e.printStackTrace();
        }

        }
    }
}
```

Listing 2: Distance-port, and reading from S3

3.5.2 Amazon S3

Amazon S3 uses the AWS-SDK for program integration. The SDK uses a simple key-value store objects, these object are stored in one or more buckets. An object consists of a key, value, and metadata. AWS has worldwide servers, therefore the region wished to be used, needs to be set on the s3 client. For this project the region chosen is EU-central. Then, a bucket is created for the server to hold the world data, and a folder is created for each of the three worlds. Once the setup is complete, the region files can be written to S3. Listing 1 shows how a region file is read from S3 if it does not exists locally. Listing 2 shows how the distance-port cycles through the chunks near the player and determines if a region file needs to be requested from S3, and starts a downloading thread if needed.

Property	Value
OS	Windows 10 Home
CPU	Intel core i7-4510U 2.0GHz
Disk	Toshiba mq01abd100
Network	40Mb/s
RAM	16GB DDR3
Glowstone	1.12
aws-java-sdk	1.11.465

Table 1: Specification of the machine

4 Experimental setup

One of the fundamental ideas of this project is to give the ability to run Minecraft servers on affordable machines. Therefore, for the experiment a local affordable machine was used. Section 4.1 describes the specification of the machine used for the experiment, and the tools and their versions. Section 4.3 describes the workload used in the experiments in order to evaluate the performance of the system. Section 4.4 describes how data was collected from the experiments to visualize and analyse the results.

4.1 Environment

Table 1 shows the specification of the machine used to run the experiment, in addition the table shows the Glowstone, and Amazon AWS software development kit that were used for the experiment. The modified Glowstone and yardstick were run on the machine to prevent network issues between the client and user to affect the experimental work.

4.2 Yardstick

Yardstick is a benchmarking tool for Minecraft-like games [15]. It provides a framework that subjects a Minecraft server to workloads determined by the virtual world and a set of bots that simulate real player behavior. Yardstick monitors both the machine and the application running the emulated players. It is comprised of three main components, the server and API's, the player emulation, and the monitoring and logging tool. For this project, a customized player behavior was written for yardstick to add the ability for players to fly in a straight line in the map.

4.3 Workload

The first experiment was conducted to test the difference in local storage use, when cloud storage was used to store files not currently used by the server, and when only local storage was used. The experiment used a newly initialized server with no generated region files. Yardstick was used to simulate two players joining the server and flying straight in two opposite directions to generate as much terrain as possible. This experiment was repeated thirty time, in thirty minutes intervals to see the variance and consistency of the storage usage. The Glowstone cache was set to one minute to minimize the amount of unused

files stored, the local cache was set to thirty seconds before writing to cloud storage and removing the local copy. The second experiment was conducted to test different latency hiding policies. This involved using the distance measurement added to Glowstone. Glowstone has a default view-port of eight chunks from the player which makes it a critical distance, if a chunk is fully read less then eight chunks away from a player, the player will experience missing chunks and possibly corrupt data. The experiment involved testing three different distances from the players, the values tested were 24, 32, and 40 chunks on the x and z axis in a square area. When a chunk was in this range from a player it was read from cloud storage. For this experiment, the game server was initialized with region files where the bot will visit generated and stored on the cloud, yardstick was used to create one bot to fly in the area of already generated regions. A server cache time of one minute was used, and a local cache of thirty seconds.

4.4 Data collection

To collect the data from the experiments, a script written in Python, and a logger written in Java and incorporated in the server to log certain events such as request made to the cloud, and reading and writing time to the cloud.

For the first experiment, the folder containing the world's region files was sampled every minute. The first measurement was a cumulative measurement, the second was periodic measurement. The cumulative measurement logged the total sum of data generated by the server. The periodic measurement logged the amount of data currently in the folder. The data was plotted in a graph to show the difference in storage usage between the two approaches.

Data collection for the second experiment used the log generated by the server which included the players location, and the chunk location to determine the distance between them when the region file containing the chunk was fully read from cloud storage. The local storage size was measurement using the periodic measurement script from first experiment. The data was plotted on two separate plots. The first is a CDF showing the distance in chunks between the player and chunk read from the cloud. The second is a graph showing the amount of local storage used.

5 Experimental results

The section discusses the experiment results obtain from both experiments. Our main findings are:

MF1 Local storage use is reduced when using cloud operated storage to store currently unused files.

MF2 Adjusting the parameters allows to make a trade-off between local storage and network reliance.

MF3 Using latency hiding policies it is possible to hide the reading time from cloud storage from the users.

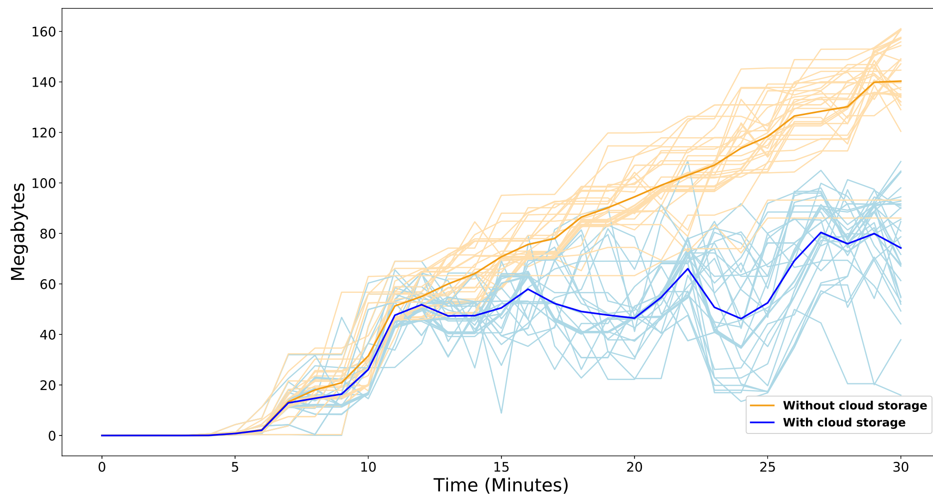
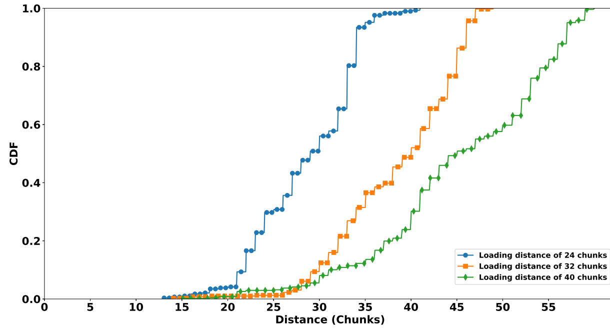


Figure 5: The amount of data in Megabytes in local storage when using only local storage and when using cloud storage. The faded curves indicate runs of the experiment, and the solid curve indicates the mean result.

Figure 5 shows the result of the first experiment. Because of variance in results, the experiment was conducted with thirty iterations of thirty minutes and shows the difference in local storage usage with and without cloud storage. The faded lines show the results of the iterations, and the thicker lines show the mean of the results. The vertical axis shows the amount of world data stored in local storage in Megabytes. The horizontal axis shows the progression over time. The results show that while there is a similar increase within the first twelve minutes of the experiment, the graph shows a downwards trend in local storage use as the experiment progresses. The graph shows local storage use grows linearly when more parts of the world are generated. When using cloud storage the Figure shows a wave pattern where local data usage increases when new content is generated before offloading unused regions to the cloud, which creates the pattern seen.



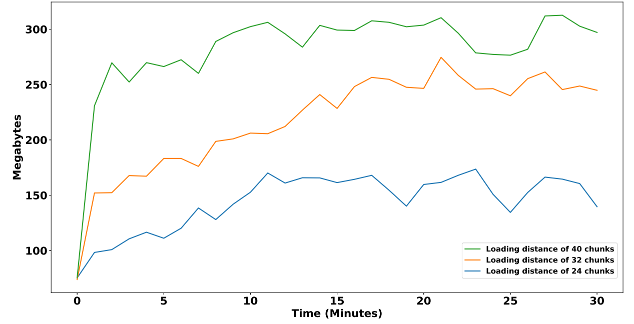


Figure 6: CDF plot showing the distance of the player from a chunk when the read from cloud storage is completed. The blue color represents a loading distance of 24 chunks. The orange color represents a loading distance of 32 chunks. The green color represents a loading distance of 40 chunks.

Figure 7: Shows the amount of local storage used in experiment two. The blue color represents a loading distance of 24 chunks. The orange color represents a loading distance of 32 chunks. The green color represents a loading distance of 40 chunks.

The results of experiment two are shown in Figure 6 and Figure 7. Figure 7 shows the distance in chunks between a player and a chunk when its corresponding region file has been fully read from cloud storage. The horizontal axis, shows the distance in chunks between the player and the chunk, when the region file containing the requested chunk was fully read from S3. The vertical axis shows the fraction of data. The Figure shows that while there are a few exceptions, especially very close to the player, the larger the loading distance from the player, the further away the chuck are fully read. The minimal loading distance was 13, 14, and 15, respectively, and the maximal 41, 49, and 60, respectively. The view-port was at a distance of 8 chunks away. Therefore, the results show that all distances were successful in hiding the reading time from cloud storage. Figure 6 is linked to Figure 7. It shows the amount of local storage that was required for the different values of the distance-port. The vertical axis shows the amount of data stored in local storage in Megabytes. The horizontal axis shows the progression in time. The results show that the larger the loading distance from the player the larger the amount of data required in local storage. However, this also gives more time for chunks to be fully read from cloud storage. This is favorable when network latency is high, but will require more available storage.

5.1 World storage parameters results

The results from experiment 2 shown in Figure 6 and Figure 7, show that all the policies have successfully managed to fully read the regions required by players before they are within viewing distance. This is good results, the implication of this, is players do not experience missing chunks caused by the cloud operated storage latency. Figure 7 shows, that changing the loading port range changes the distance files are fully read from cloud

operated storage. Figure 6 shows the change in local storage reliance by using different loading port distance. The amount of reads from cloud storage has increased with the loading distance, for a distance of 24 an average of 63 reads were made, for a distance 32 an average of 109 reads were made and for a distance of 40 an average of 130 requests were made. The increase is due to more region files required for the same duration as more distant chunks are required. However, the fact that more files are loaded further away allows more time for these files to be fully read, this is useful when having high network latency. These results show that changing the loading port distance grant developers the choice between local storage and cloud operated storage reliance. This is useful when developers are working with limited storage space, or high latency networks.

5.2 Latency hiding policies

The results in Figure 6 and Figure 7, show that all the policies used in the experiment were successful at hiding the reading time from cloud storage. The players view port was set to eight chunks away, while the closest chunk was loaded at a distance of 13 chunks away. However, network latency can change with factors such as distance from the cloud storage servers and network activity. Therefore, this numbers might not always be true for the same workload, it is therefore important to measure the network speed and latency based on the server location and the cloud operated storage being used and adjust the parameters accordingly.

5.3 Related work

This thesis proposes one way of scaling Minecraft-like games using serverless computing. Other methods to increase scalability are being researched. Peer-to-peer or hybrid architectures offer to support a large amount of players however they suffer draw backs of cheating and limited bandwidth [16]. P. Kabus et al [9] suggest in their paper a spectrum of options that might solve the cheating issues that arise in peer-to-peer systems. Manycraft [3] increases the scalability of a single Minecraft instance to 1,000 players in a static world.

6 Conclusion

Minecraft-like games do not scale well and require large high-performance machines to maintain operations. This poses a challenge for small game studios or private server owners as the upfront and operational cost is high. Serverless computing, offers a solutions for the high cost required, this is due to users only paying for the resources utilized. In addition, serverless computing resource capacity is larger then privately owned hardware, therefore it can provide a solutions to Minecraft-like games scalability issues.

The results of the experiments show, that when using cloud operated storage the amount of local storage required stays stable while the amount of data without serveless grows linearly. The results also show that by employing policies that read chunks further than the view port of the player, it is possible to hide the latency of the cloud operated storage reading time from the user. However, the latency of the cloud operated storage depends on several factors such as distance from the server, and network speed. To ensure optimal

work of the cloud operated storage with Minecraft-like games it is necessary to adjust the parameters to fit the host's situation.

7 Future work

This work tested several latency hiding policies. However, many such policies can still be tested to reduce network activity while still keeping the local storage use at a minimum. Some of the policies that can be tested are, reconsigning player hot-spots, portal locations, and locations hidden from player view such as by mountains. In addition, this paper only reviewed and benchmarked the Amazon S3 cloud operated storage. Many different cloud operated storage solutions are available and benchmarking them might yield better result, and will give more flexibility to developers. Another important work, is performing experiments and gathering feedback from real players, this can add more depth to the results and add data that cannot be collect by using bots.

This system is just the first stepping stone in to what will hopefully become a fully serverless system, where terrain is generated by serverless event-driven platforms, and NPCs are controlled by serverless containers.

References

[1] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior. 2011.

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

[3] Raluca Diaconu, Joaquín Keller, and Mathieu Valero. Manycraft: Scaling Minecraft to Millions. In *NetGames*, pages 1 – 6, 2013.

[4] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems. In *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*, 2020.

[5] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[6] Erwin Van Eyk, Alexandru Iosup, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, and Cristina L. Abad. The SPEC-RG reference architecture for faas: From microservices and containers to serverless platforms. *IEEE Internet Comput.*, 23(6):7–18, 2019.

[7] Erwin Van Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, and Alexandru Iosup. Beyond microbenchmarks: The SPEC-RG vision for a comprehensive serverless benchmark. In *Companion of the 2020 ACM/SPEC International Conference on Performance Engineering, ICPE 2020, Edmonton, AB, Canada, April 20-24, 2020*, pages 26–31, 2020.

[8] Erwin Van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uta, and Alexandru Iosup. Serverless is more: From paas to present cloud computing. *IEEE Internet Comput.*, 22(5):8–17, 2018.

[9] Patric Kabus, Wesley W. Terpstra, Mariano Cilia, and Alejandro P. Buchmann. Addressing cheating in distributed mmogs. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, page 1–6, 2005.

[10] Newzoo. Newzoo. 2016 global games market report. annual report of trends, insights and projections for the global games market. 2016.

[11] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 57–70, 2018.

[12] Mayur Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? 2008.

[13] Minecraft statistics. https://minecraft-statistic.net/en/global_statistic.html. 2020.

[14] Alexandru Uta, Dmitry Duplyakin, Cristina Abad, Nikolas Herbst, and Alexandru Iosup. 3rd workshop on hot topics in cloud computing performance (hotcloudperf'20): Performance variability. In *ICPE '20: ACM/SPEC International Conference on Performance Engineering, Edmonton, AB, Canada, April 20-24, 2020*, pages 301–302, 2020.

[15] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. Yardstick: A benchmark for minecraft-like services. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*, pages 243–253, 2019.

[16] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):9:1–9:51, 2013.