Vrije Universiteit Amsterdam

Bachelor Thesis

ObserverCraft: a large scale in-game spectating system for Minecraft-like games

Author: Milos Delgorge (2642474)

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

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

June 24, 2021

Abstract

The video game industry is the fastest growing entertainment industry with a revenue of over $179 billion dollars in 2020. Video games are becoming increasingly popular and are even starting to be used for purposes other than entertainment such as education. modifiable virtual environments (MVEs) are real-time, online, multi-user environments that allows users to modify the virtual world's objects and parts. Minecraft is the most pupular MVE with over 130 million active players as of 2020. Minecraft only scales up to around a couple hundred players, which prevents players from engaging with each other in the game world on a larger scale. Not only are games popular to play, they are also popular to watch. Platforms such as twitch and youtube allow users to watch gameplay streams that are broadcasted online, but do not allow users to spectate from within the game itself. A spectator of a video game is not held to the same quality of service constraints as a player, which raises the question: can we scale minecraft-like games by allowing lots of spectators to spectate the game world? We design, present and evaluate a system that makes use of these lower requirements in order to support a large number of in-game spectators. Through use of real-world experiments we show that indeed we can add a large amount of spectators to a minecraft like game without diminishing the number of actual players.

Contents

1 Introduction

Over the last few years, the video game industry has grown to be the largest entertainment industry with near $180 billion dollars made in revenue in 2020 [28]. Especially with many people being stuck at home during the Covid-19 pandemic, the gaming industry is thriving more than ever before [21]. People use videogames as a way to escape from reality and still feel a connection to others during lockdown [29]. Gaming is becoming more and more socially interactive, yet many popular games do not scale well.

One of the most popular online games is Minecraft. It is currently the most sold video game with over 200 million copies sold as of 2020, and has an active playerbase of 126 million players [10]. Minecraft is one of the most popular games, but as observed in Yardstick [27], Minecraft only scales up to around 200-300 players in a single instance. This lack of scalability puts a limit on the social aspect of games, and prevents players who might only want to spectate, or players who want to play in the game world but do not require many server resources to enjoy their experience, from participating. Instead, in order to participate and interact with the video games, people look to streaming platforms to watch others play games.

Video game spectating has become very popular over the last decade. The global live-streaming audience for video games is expected to hit 728.8 million viewers in 2021, with an expected growth to 920.3 million in 2024 .[23]. The most popular video game streaming platform is Twitch. Twitch has an average of 1.44 million concurrent viewers [20]. When watching a twitch stream, a user is only able to observe the actions of the streamer within a game world. This is a limiting factor as the user has no freedom to choose what to observe for themselves, they are bound by the streamer.

In recent times video games have been used as a platform for other forms of entertainment. During Covid-19, video games such as Minecraft and Fortnite have been used as platforms for artists to hold virtual concerts [11]. Due to scalability constraints in these video games, not many players can spectate these events in-game in a single instance, and many have to resort to watching these events on a live stream instead.

This thesis consist of developing and evaluating a prototype for a system that scales Minecraft like games by allowing spectators to observer the game, not via a live stream, but from within the game itself on a large scale.

1.1 Problem statement

Scaling computer systems is not an easy task, especially not in video games which are oftentimes under stringent Quality of Service (QoS) constraints. Spectating a video game is similar to playing one in the sense that you are experiencing the same game world, but for a spectator, certain QoS requirements that usually are very important for a player, such as ping, are not as much of a factor. Currently, due to scalability limitations, players that are not under these QoS constraints, or want to spectate in-game, are often unable to participate in Minecraft-like games. This raises the question of how to support these

players so they can also participate. In contrast to scaling the number of players, scaling the number of spectators might be easier due to the more flexible constraints. Video games are not only popular to play, they are also very popular to watch. Current state of the art game spectating platforms such as twitch, YouTube, and Facebook allow a user to spectate actions of a single player within a world. Due to the user not actually being in the game itself, the spectator's view is limited to only what the streamer does and shows. There has not yet been a good solution to combine spectating and playing on a large scale within a game itself for Minecraft-like games.

1.2 Research questions

RQ1 How to design a system that supports large scale in-game spectating in Minecraft-like games?

Designing such a system is challenging because oftentimes real-time online games supporting spectators requires disseminating the game state to large numbers of users while maintaining stringent QoS and QoE requirements.

RQ2 How to realize a system that supports large scale in-game spectating in Minecraft-like games?

Implementing such a system is challenging because it has to function within the existing Minecraft ecosystem.

RQ3 How to evaluate the performance of this system ?

There currently is no standard method of evaluating the performance of a distributed system. In order to evaluate our system properly, experiments with real world workloads are designed and conducted.

1.3 Methodology

To answer the first research question, we first need to establish system requirements. After that is done, we start iterating on a design that supports our requirements.

To answer the second research question, we will take our design and iteratively implement its components with the tools at hand. We will describe the process of realizing our prototype and what challenges it brings.

To answer the third research question. We will define experiments, define and create real world workloads and we will define a set of metrics by which we can evaluate our system and its impacts on the performance of the game server.

2 Background

To design our prototype, it is important to understand how Minecraft-like games function and what the current options for in-game and outside the game spectating are.

2.1 Minecraft-like games

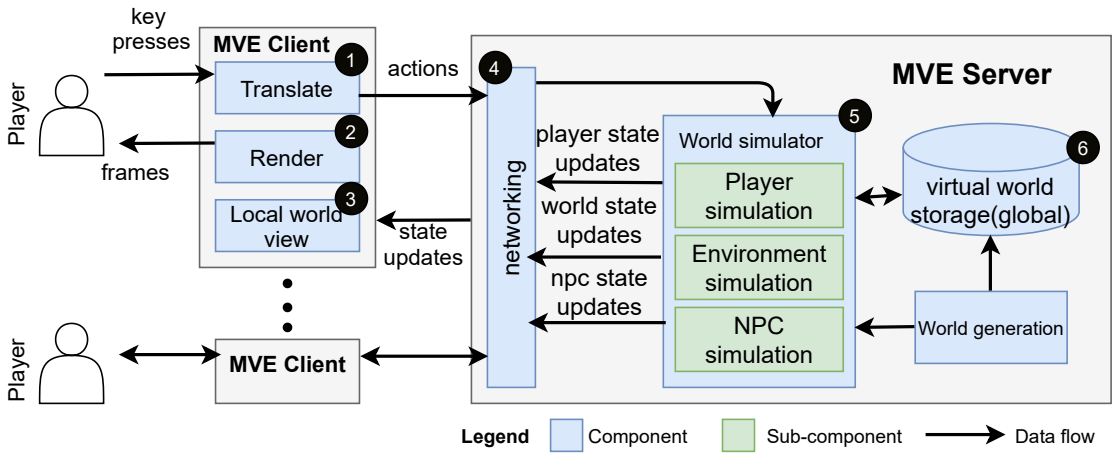

Figure 1: Operations of a Minecraft-like online game with multiple clients

Minecraft-like games are voxel based games that have a modifiable virtual environment (MVE). A MVE is a real-time, online, multi-user environment that allows its users to modify the virtual world's objects and parts [14]. Minecraft is the most popular Minecraft-like game. In Minecraft the world consists of chunks [18] of land, which each consist of blocks [17]. Players can place and destroy blocks to modify the terrain. A typical Minecraft-like game uses a client/server architecture, where a player runs a local client that connects to a remote game server. A single server services all players. Minecraft-like game servers have three main tasks. They keep track of the global game state, they simulate state changes and send them back to the clients, and they generate the game worlds terrain. The client translates user-input(❶) to Minecraft-like game specific player actions that are sent to the server over the internet and received by the network stack(❹). The server takes these actions and processes them in the world simulator (❺). The simulator is responsible for simulating player actions, NPC actions and environment changes. Changes are saved to persistent storage(❻) and game state updates are then sent back to the clients over the network. Clients have a local view of the world(❸). The state changes that the server sends to the clients are then applied to this local world view, synchronizing it with the world that the server has in storage. The client then renders(❷) this local world view back to the player.

2.2 Online game viewing

Traditionally gaming has always been targeted towards players, but in recent years interest in watching others play video games has risen considerably. The current state of the art video game live streaming platforms are Twitch, Youtube and Facebook. Streamers on these platforms can broadcast their gameplay footage for others to watch, as well as give audio commentary and show themselves on camera. Figure 2 depicts the typical model for a streamer streaming his gameplay to viewers. The streamer has a video game(❸) running on his/her device(❶), as well as capturing software(❷). The capturing software captures

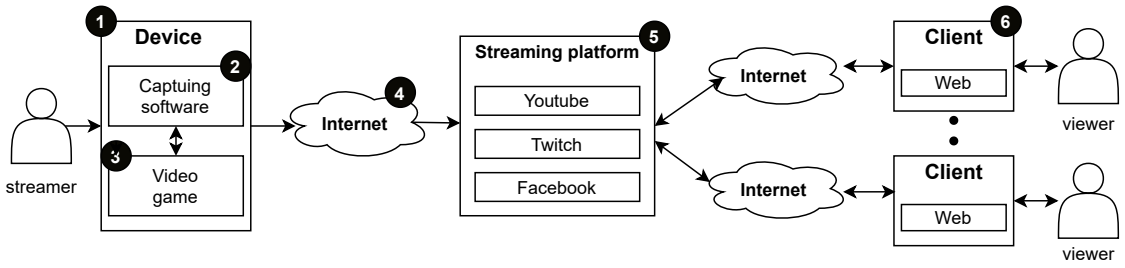

Figure 2: Videogame livestreaming process

the data from the video game, encodes it and sends it over a remote network(❹), such as the internet, to the servers of a live streaming platform(❺). The live streaming platform then uploads this data so other people can download it and watch the live stream via a oftentimes web-based client(❻). Depending on the streaming platform, there are other interactive features that let the spectators interact with the streamers such as a chatbox where viewers can talk to each other and/or the streamer, donations, cheering and polls.

There are many reasons for people to be interested in spectating video games, such as relaxation, distraction, escape, learning about game strategies, companionship and enjoyment [26]. Interest especially rose with the introduction of E-sports [16], meaning Electronic Sports which is the collective term for games that are being played on a competitive level.

There has been little research done on online game spectatorship. Most of the research that has been done has mainly been targeted at E-sports spectatorship and aims to identify who is interested in spectating E-sports and why [22] [19], and how the spectator experience for E-sports can be improved [9].

2.3 In-game spectating systems

In-game spectating is not a novel idea; many games do offer this feature in the form of an in-game spectating mode. This mode typically means that a spectator can follow all the events that happen inside the game, but can not influence the state of the game in any way. Many of the more competitive games like Counters-Srike and League of Legends do offer an in-game spectating mode that can be accessed from within the game client. [15][24]. Players can for example spectate their friends while they are in a queue for a new match, or spectate their friends just to learn something from them. There are usually two different types of views when spectating. First there is the view where the spectator is locked to the player. When in this mode, the spectator sees everything through the eyes of the player. This mode often comes in the form of a first person view mode, where you basically look through the eyes of player. Secondly there is the mode where the spectator is not bound to any player and can move around the world freely. This mode is often called the bird's-eye view mode.

Some research into designing in-game spectator system has been done, but most of it has been targeted towards improving the spectating interface to create a better quality of

experience for the viewer [8]. There has been little to no research done into the scalability of spectators in video games.

3 Design

In this section we discuss the high level design of our system. Our system allows users to spectate a Minecraft-like game world.

3.1 System requirements

- **R1:** Our system should have high scalability

  We want our system to scale Minecraft-like games by allowing a large number of spectators to observe the game world, in addition to all the players who are playing in it. The number of spectators we aim to support is anywhere in between the maximum number of players a Minecraft-like game supports, mentioned in Section 1, and the millions of viewers who are interested in watching Minecraft-like games online. We want our system to take up as little server resources as possible to not have to reduce the number of non spectators who can partake in the game.

- **R2:** Our system should be easy to adopt by users.

  We want our system to be easy to use. A client should for example not have to do any client side modifications in order to use our system. Modifying the client would be impractical and would add too much complexity.

- **R3:** Our System provides users different ways of spectating.

  Users might enjoy spectating in various different ways, whether that be an active or passive form of spectating, we want to make sure that our system can provide different ways of spectating to appeal to a broader audience.

- **R4:** A spectator should be able to spectate any part of the game-world in the fullest detail.

  We want spectators to be able to go anywhere in the world and see exactly what is happening there.

3.2 High level design of ObserverCraft

In this section we explain the design of ObserverCraft on the basis of a Figure 3, and explain the components of ObserverCraft and the role they play in making our prototype function.

Our system acts as a proxy between a Minecraft client(❶) and a Minecraft server(❻). To satisfy requirement **R2**, we decided to use a Minecraft server(❷) as the connection point to our proxy for users. Because of this, a user can connect to our system in the exact same way as he/she would connect to any other Minecraft server. This makes our system very accessible and easy to interact with for users, making it very easy to adopt. For our connection point to the Minecraft server we decided to use the standard Minecraft protocol. We create a client(❺) and connect this client to the Minecraft-like game server

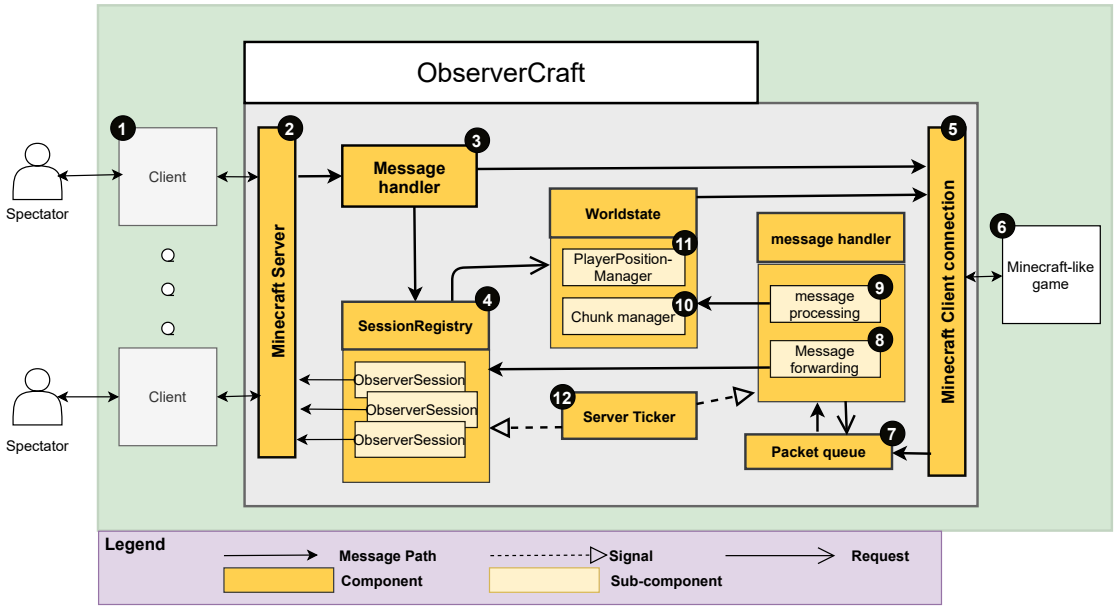

Figure 3: ObserverCraft design overview

as if this client were a normal player. To satisfy requirement **R4**, we have added some functionality to the server that allows it to send all game-state updates to our system.

When a spectator connects to our system, a new Observer session is created and stored in the SessionRegistry(❹). The Observer session contains important information about the spectator that we need in order to properly forward game state messages. The components of the session registry are highlighted in Figure 4. It contains the session that we have with the client, a spectator object we associate with each connection that holds the spectators id and position, and the message queue that contains the messages that need to be sent to this client.

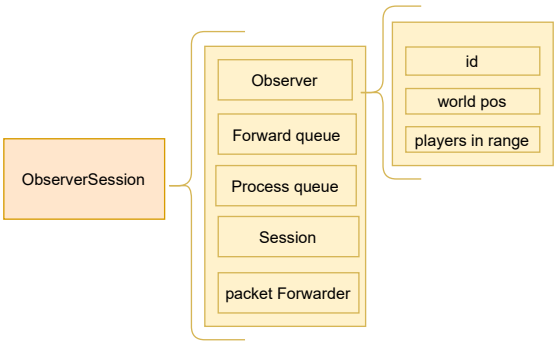

Figure 4: Components of an observerSession

After a connection is established with a spectator, their messages will be processed by the message handler(❸). The message handler will check the incoming messages and filter out the important ones that our system needs to process such as player position messages and chat messages. The messages that need to be processed get put in the process queue for the given observerSession as depicted in Figure 4. Chat packets are used to create commands for our system and position packets are used by our system to keep track of where our spectators are in the world, which is information we need to know for interest

management which is explained in section 4.3.

We also have a Message handler for the messages we receive from the Minecraft-like game server. Our system receives all world-state updates from the server and puts them in a queue(❼). The messaging handling component then can request packets from the queue to process. The message handler for the server packets consists of two main components. First there is the message processing component(❾). Each packet we receive goes through the message processing component. It filters out the important messages that we need to handle such as player information and chunk data and forwards these packets to the WorldState component for them to be handled. Second there is the message forwarding component(❽). This component checks for each spectator connected to our system if player related messages should be forwarded to this spectator or not. There are two criteria that decide whether a packet should be forwarded to a spectator or not. Firstly we have defined an area of interest for each spectator. We do not want spectators to receive player movement information from players that are nowhere near them. The message forwarder checks for each player movement related message if it belongs to a player that is in the area of interest of the spectator or not and forwards it based on this. We do this to help satisfy **R1**. By reducing the number of update messages we sent to our spectators we preserve some bandwidth allowing us to service more spectators in total. The message forwarder also checks whether or not the spectator is in follow mode. Follow mode is a feature of our system that we will describe in more detail in Section 4.4 that we implemented in order to satisfy requirement **R3**. If the spectator is in follow mode, certain player messages need to be forwarded a little differently than usual, and the message forwarder handles that logic too.

To be able to accurately forward the game state to our spectators when they connect, we have to keep track of it in our system. The world state component is responsible for doing this and consists of two main components. The PlayerPositionManager(⓫) is responsible for keeping track of the positions of all the players that are connected to the Minecraft-like game server. As explained in Section 2.1, Minecraft's game world is composed of individual chunks. Our system needs to keep track of all the chunks in order for us to be able to provide our spectators with an accurate view of the game world when the connect to our system. The chunk manager component(⓾) is there to keep track of all the chunks that we receive from the game server. Because the world is comprised of all these individual chunks, we can send or request only small parts of the world instead of the whole world at once. This means we do not have to request the entire game world from the game server every time a new spectator connects to our system. We only have to re-request the chunks that have been modified since we last received them. The chunk manager is responsible for handling this logic.

The World State component also stores other relevant state information such as mob entities, the server time, and the weather.

We also have the server ticker component(⓬) which is responsible for a couple of things:

1. Signalling to the session-registry to process the received player packets for each session.

2. Signal to the session-registry to empty the message queues.
3. Signal to the session-registry to update the players within proximity of each observer
4. Signaling to the message handler on the server side to process all the messages in the packet queue.

3.3 Design Decisions

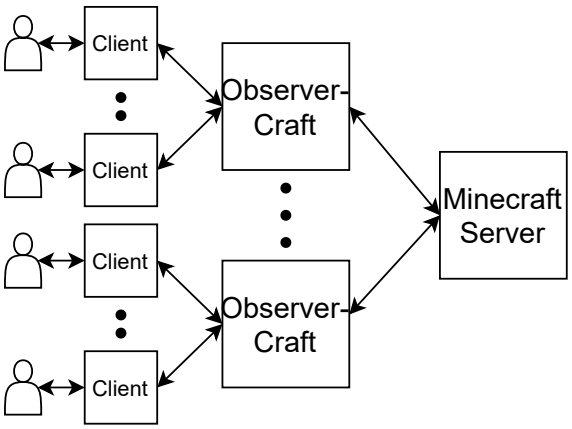

Figure 5: Vision for Observercraft

When thinking about how to satisfy **R1**, we considered to implement a server side system. However, this would have required the server to handle the forwarding of state changes to spectators, which would be too costly if the number of spectators was scaled up high. To get around this problem we had to come up with a stand alone system that sits between the server and the client. We take this approach for two reasons.

Firstly, this way, the server essentially has to only service a single extra connection, which will in turn service all other spectators. The connection with our observer will require more bandwidth than a connection to a regular player, as we have to send every world-state change message to this connection, but this still requires way less bandwidth compared to if the server had to also handle forwarding messages to spectators and handling their individual states.

Secondly, designing our system as a stand alone system means that we can connect multiple instances of ObserverCraft to the server (Figure 5), each of which will ideally service a great number of spectators. If the overhead of our system is small, we will be able to massively scale up the number of spectators.

Initially we did not have a ticker and let the messages get processed continuously, but we needed a metric to measure the performance of our system. For this reason we decided to implement a ticker which allows us to specifically measure how long certain processes in our system take.

4 System realization

In this section we discuss the realization of ObserverCraft. We build our prototype on top of the open-source Glowstone based Minecraft-like game Opencraft. This section will explain in detail how we realize our system.

A lot of the initial work that went into creating the prototype was related to understanding the Minecraft networking protocol [3]. We used a small program that set up a connection to the Opencraft server that logged every packet it received. We then also connected another client we could control via the standard Minecraft client to the Opencraft

server and observed which packets our test connection received as we were performing various actions with this client. By analyzing these packets we figured out what all the packets were exactly responsible for, and we could then start processing them accordingly in our own system.

Our system connects to the Opencraft server via the standard Minecraft protocol. We have created a special command for Opencraft that when issued will flag our connection as 'observer' to the server, so we can treat it differently from the other connections. We can call this command by sending a chat message to Opencraft via our connection with the server. For both the client and the server that our system implements, we have used a Minecraft protocol library [1].

4.1 Retrieving the game state

For our spectators to be able to spectate the entire game world, our system needs to receive all game state update messages that the server sends out. Opencraft allows operators to specify how game state update messages are sent to its clients by modifying the messaging system. For our prototype, we decided to use the dyconit messaging system [13] as it proved to scale Opencraft's playercount the highest compared to other available messaging systems. The Dyconit messaging system allows operators to break the game world up into dyconits, to which players can subscribe. We want our system to receive every game-state update that the server puts out. We accomplished this is by creating our own policy for the dyconit system. Operators can specify their own policies for the dyconnit system. The policy dictates which dyconits each player should be subscribed to, and specifies all the different dyconits. In our policy, every chunk is a dyocnit. Our policy subscribes players flagged as observers to all dyconits. This way our observer will receive all game state updates that the game sends out via the messaging system.

Not all the game state updates are managed through the messaging system, there are also instances where the server directly sends messages to clients. In order to receive these messages we created some new functions in the Opencraft server that send these extra messages to the observer.

4.2 Maintaining the game state

For our spectators to receive the up-to-date game state when they connect to our system, we have to keep track of important game state information. As explained in Section 3, the world state component is responsible for doing this.

To keep track of all the player positions of players connected to the game server we have to keep a list of all player entities together with their current position. When our system connects to the game server, it starts receiving all game-state update packets that the server sends out. There are a variety of packets that are used for the movement of entities. What all of them have in common is that they have a field for the ID of the entity that is associated with the movement packet. Each entity in Opencraft has their own unique entity ID. When ObserverCraft first gets connected to the Opencraft server, it receives player spawn messages for all the connected players. These messages contain the location and IDs of the players. We store this information and use it to create player

objects. After having received the spawn packets we then check for every incoming entity movement related message whether it belongs to a player object that we have stored. If this is the case we take the position data from that packet and update our players location accordingly. When a spectator then connects to our system, we can reference this data to generate the appropriate player spawn packets.

We also keep track of all the chunk data we have received from the server. We need to keep track of this information in order to send our spectators the up-to-date game world when they connect. As explained in Section 2.1, the Opencraft server simulates the game world and keeps track of the global game state. Whenever a new spectator connects to Opencraft, Opencraft can reference the global game state and send out the up-to-date chunks. Our system however is not able to simulate block changes in chunks, as we do not have the parameters and information that would be required to do this. Even if we were able to do the simulating within our system, we would not want to do this as this is computationally expensive and would come at the cost of our system being able to support a lot less spectators. What this means is that we can not update the chunk data after we have received it, and have to re-request chunks from the server if we want to get the up-to-date ones. In an earlier version of the prototype, we simply re-requested every chunk when a new spectator connected to our system, but this overloaded our system when many spectators joined in a small time span. In order for us to not be forced to re-request all chunks at once, the chunk manager keeps track of which chunks have been modified after having been received by our system. When the Opencraft server generates and populates a new chunk, a chunk data packet is created and sent out to clients and our observer. This packet contains the terrain data as well as an x and z coordinate for that chunk. We store all the chunk data packets we receive. We now also want to store whether a received chunk has been modified or not. The way we keep track of this is by analyzing incoming block change packets. Whenever a block is broken or placed in Minecraft a block change packet gets sent out. This packet, among other things, contains the x and z coordinates of where this block change occurred. By doing some simple math we can take the coordinates from the block change packet and figure out which chunk they occurred in. When we receive a block change packet for a chunk, the chunk manager then flags this chunk as modified. The way we flag chunks is by storing the chunk data packets in a hashmap together with a boolean. When we want to flag a chunk as modified we simply set the boolean to true.

We also keep track of other important world information such as the server time and weather. Whenever our system receives packets associated with these events, we update our time and weather variables stored in the world state with the newly received ones.

4.3 Forwarding the game state

When a spectator joins, we request the up-to-date world data and player data from the world state component. The chunk manager loops through all the chunks it has stored and sends the unmodified chunks directly to our spectator. It then takes the modified chunks and re-requests them from the server after which they also get forwarded to our spectator. After the chunks have been sent, entity spawn messages get created and sent out for all the entities that our system has stored. After the spectator has received the up to date chunks and entity spawn messages, our system can then start forwarding all

incoming update messages from the server to the spectator, which will apply these changes to its local copy of the game world.

We do not want to forward the entire game state to every spectator at every tick. In order to preserve bandwidth we have chosen to take inspiration from already existing interest management techniques [6] to reduce the number of game state messages sent to our spectators.

Interest management is a technique often used in video games to improve scalability and reduce bandwidth usage. When playing a game, a user is usually not interested in the information from the whole game world, only the part of the world where they are currently in. There are many different ways of doing interest management, we have chosen a relatively simple one based on distance.

Most updates sent by the game server are entity position related. As a spectator you are mainly interested in the events that happen in the area directly around you. You are for example not interested in the state of a player that is not in render distance. Our system defines a distance based area of interest for each spectator. The distance we take is the absolute distance between two sets of x and z coordinates within the Minecraft world. Currently the distance is set to 200. When we receive a player entity related packet, our system finds the player's position by requesting the position from the player position manager using the entity ID from the message. If the player that the message belongs to is within the specified distance from our spectator, the message gets forwarded, if not, the message gets ignored.

4.4 System features

This section will talk about the features of our system. Our goal is to publish our system as open-source software for others to use for their Minecraft-like games. Our system is not merely a one way message forwarding system. Out of usability considerations we decided to implement a couple of features we think will make community adaptation of our software more likely. We utilized the player chat as a way of receiving player commands in order to implement these features.

Following a player

Spectating in the traditional sense is a complete passive activity. When one thinks of spectating, one would usually think of sitting down comfortably and watching a screen to observe an event. Our system, in its purest form, connects a player to a Minecraft server and treats it like a normal player. This means that for the spectator to start spectating, he/she would still have to actively be inputting on the keyboard to walk or fly around the world. This on one hand is great, as it allows the player to, on their own, explore every little part of the world without being bound by another entity. Chen et all [7] show that spectators experienced increased arousal when they have access to information that players do not have access to. When a spectator joins our system, he/she initially will have a bird's-eye view, giving them more information than a player has. On the other hand, roaming the world is an active experience, something a traditional spectator may not be looking for. To give the user more options we have created the "/follow [name]" command

When a spectator is connected to our system, he/she can see all players who are connected to the Minecraft server via the scoreboard. A spectator can pick a player and issue the follow command with any of the players names. When this command is issued, the spectator goes from having a bird's-eye view of the world to having a first person view of the specified player. This means that the spectator will experience the game world as if they were in the shoes of the player. When in follow mode, the ObserverCraft intercepts all the selected player's entity related update messages and replaces the entity id field of that message with the entity id that we have given to our spectator. The Minecraft client will then apply these movement messages to the spectators character when received. This way, the spectator does not have to actively give our system any inputs, and will be able to observe the worldview of the given player completely passively. A spectator can exit the follow mode by simply typing "/unfollow".

Teleporting to player

Our system allows a spectator to teleport to any player in the game world. Oftentimes when one wants to spectate a game, one wants to spectate the actions of a specific player, such as one's friend. Allowing our spectators to teleport to anyone makes finding the people you want to spectate very easy. A spectator can teleport to a player by typing "/teleport [name]". ObserverCraft then gets the player location of the requested player and sends a teleport packet to the spectator with that location.

Chatting to players

Chen et al. [7] show that the more interactive a spectating experience is, the higher the perceived arousal as well as the engagement of the spectator is. In order to make the experience more interactive, we thought it would be nice if our spectators could communicate with the players on the actual game server. Even though spectators are not directly connected to the server, we can still forward their chat messages via our proxy connection with the server. When a player types /chat [msg], the server will display the message as a global server message on the live game server, together with the name of the spectator that it came from. Allowing communication between players and spectators opens up the possibility for our system to be used for more interactive means such as education or mini games.

5 Experimental Setup and Results

This section discusses the experimental setup of the experiments we have designed in order to evaluate our system. First, in Section 5.1 we discuss the environment in which our real-world experiments are performed. Then, in Section 5.2 we discuss the different types of metrics and method of data collection for our experiments. In Section 5.3 we explain the different types of workloads our experiments will use. Finally, Section 5.4 presents the results.

We will perform a variety of different types of tests; we will perform an overhead test, a scalability test and a latency test. The overhead experiment will help us understand

what strain our system puts on the game server. The scalability experiment will help us understand to what extent our system is able to scale a Minecraft-like game. Finally, the latency experiment will show us what impact not being directly connected to the game server has on the latency of a user.

5.1 Environment

We want our experiments to be representative of a real world setting. To accomplish this we want to run our experiments on representative hardware and with workloads that are as representative as possible of a real-world workload. We will run our experiments on the DAS5, and we will use Yardstick for our workloads.

5.1.1 DAS5

Game servers are typically run on powerful hardware, more powerful than we would have access to when doing testing on a consumer grade computer. The DAS5 was the most representative hardware available to what we believe modern day game servers are run on. The DAS5 is a distributed supercomputer designed by ASCI and funded by NWO/NCF. The DAS, among other things, provides researchers with an infrastructure for distributed and parallel computing. It allows users to reserve compute nodes on which they can run experiments. Each of these nodes run a dual 8-core cpu, 64 gigabytes of memory and all nodes are interconnected via an FDR InfiniBand network [5].

5.1.2 Yardstick

Yardstick is designed by Van der Sar, et al., as a tool to help benchmark performance scalability in Minecraf-like games [27]. Yardstick allows operators to deploy specific workloads on Minecraft-like games. These workloads consist of a virtual environment with a specified number of emulated players in it. Yardstick allows users to deploy any environment they want, and also supports multiple different behavioral patterns for the emulated players. After the workload is deployed, Yardstick will monitor the server on which the Minecraft-like game is running and collect system, application and performance metrics that can be used to evaluate the scalability of the Minecraft-like game.

5.2 Data collection and metrics

To measure the scalability of our system we need to measure when the game server, as well as our own system gets overloaded. The most commonly used metric to measure the performance of Minecraft servers is the tickrate. At each "tick", the Minecraft server processes a single iteration of the game loop. The maximum time this can take before the server is considered overloaded is usually said to be 50ms. In order to measure the performance of the server, as well as our own system we will use the time it takes for a single tick to process. We will also capture network metrics such as the amount of bytes sent over the network. We will use the psutil [25] library to capture these metrics. For our latency experiment we will use Minecraft chat packets to measure the round trip time from a client to the server and back.

5.3 Workload

We will use Yardstick to connect a number of emulated players to the Opencraft game server. The behavior used for the bots is a movement model where the bots walk around in a confined space within the world. The world we will use is a flat Minecraft world. When using a non flat world we were seeing too much performance variability, making the results not very usable.

An overview of the experiments with their given workloads is given in Table 1.

Experiment	Focus	Bots Serv	Bots Proxy	Duration [min]
1	Measuring overhead	40	0	3
2	Measuring overhead	40	0	3
3	Measure scalability	40	200	3
4	Measure latency	1	1	1

Table 1: Experiments overview table

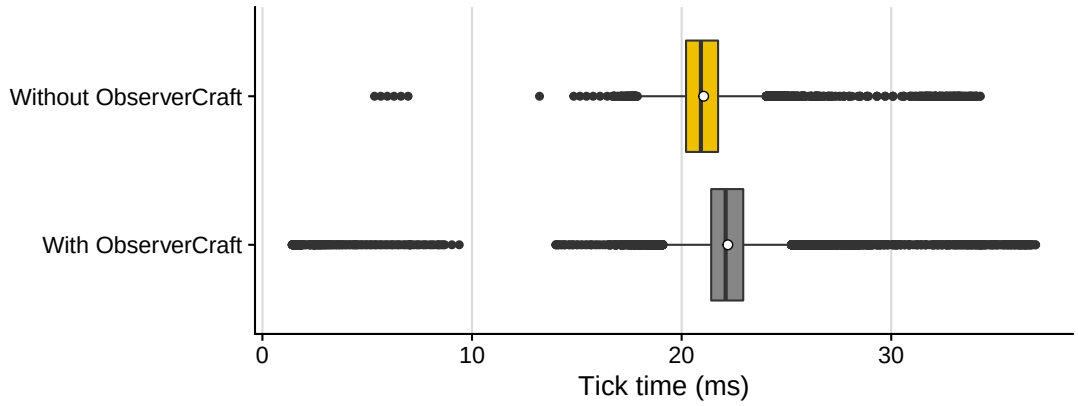

Figure 6: Tick time for Opencraft with and without observer

5.4 Experimental results

From our experiments we have come to three main findings:

MF1 ObserverCraft has a low overhead on the game server. We find that the tick duration of the game server only increases by 1.1 millisecond, and we measure a 1.3% increase in bytes per second sent over the network.

MF2 ObserverCraft supports a large number of in-game spectators. We find that our system can support up to 1000 players before becoming overloaded.

MF3 ObserverCraft introduces a significant amount of latency to the users. We find that the round trip time for packets routed via ObserverCraft exhibits variance up to 50ms

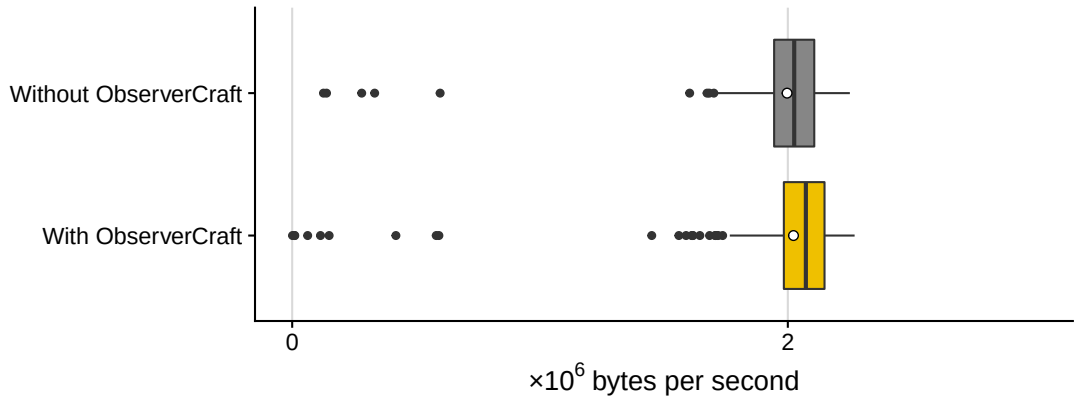

Figure 7: Average number of bytes per second sent out by Opencraft with and without observer

We now explain each main finding in more detail on the basis of the results of our experiments.

MF1: Observercraft has a low overhead on the game server

In our overhead experiment we measured the tick time of the server with and without our system connected, as well as the average number of bytes sent out over the network.

Figure 6 shows the tick time of Opencraft with and without ObserverCraft connected. We ran 5 iterations of the experiment with a duration of 3 minutes each. The horizontal axis depicts the tick duration in milliseconds, and the vertical axis depicts the configuration. The mean tick duration of Opencraft without ObserverCraft connected was 21.1 milliseconds. The mean tick duration of Opencraft with ObserverCraft connected was 22.2 milliseconds. This means that there is an increase of 1.1 milliseconds in tick duration.

This result is expected. To make our observer receive all the necessary game state updates, we had to introduce additional functions to the server that get called every tick. This will naturally increase the tick duration. An increase of 1.1 millisecond is very small however, which bodes well for ObserverCraft.

Figure 7 shows the average number of bytes per second sent over the network. These results come from the same experiment run as the previous plot. We ran 5 iterations, each with a duration of 3 minutes and measured the network usage. The mean for the average number of bytes per second sent over the network for Opencraft without Observercraft connected was 1996942 bytes/s , and with ObserverCraft connected was 2022756 bytes/s. This is an increase of 1.3%

This result was again expected. The server has to forward all the game state updates to our observer at each tick, which will result in a little more overall data sent over the network.

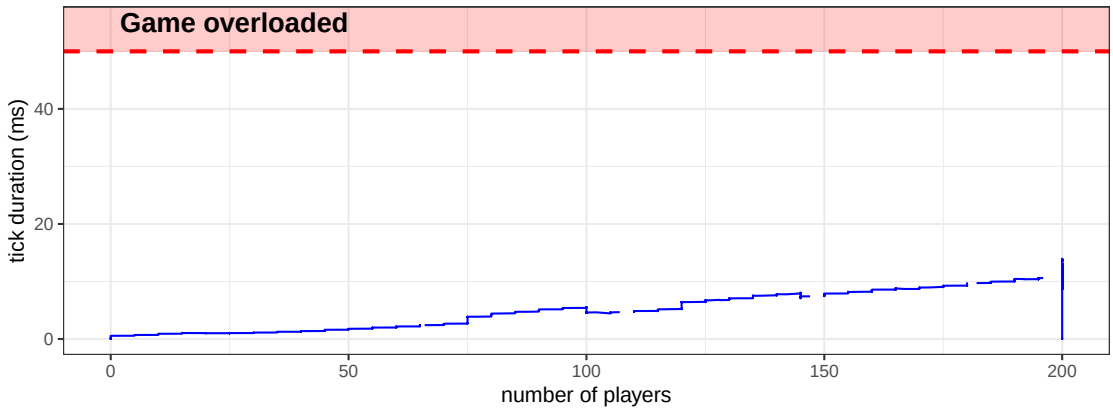

Figure 8: Tick duration for ObserverCraft

MF2: Observercraft can support a large amount of in-game spectators

In our scalability experiment we connected emulated players to ObserverCraft and checked when the system became overloaded. For this experiment we had 40 players connected to the Minecraft server and then connected an increasing number of spectators to our system.

As observable in Figure 8, our system reached a tick time of around 10ms at 200 spectators. If we make the assumption that it would keep increasing linearly, and we assume that our system becomes overloaded at a tick duration of 50ms, we come to the conclusion that our system can support up to 1000 spectators. The average concurrent viewership of Minecraft on twitch is currently around 83.000 concurrent viewers, with peaks up to close to a million viewers [2]. When comparing our result of 1000 spectators to those numbers, we can clearly see that, even if multiple instances of our system are connected to a Minecraft server, we are nowhere near supporting all those players in-game. If we however compare our result to the maximum number of players a Minecraft-like game can support in a single instance, namely 200-300, or the Mojang recommended number of players for a Minecraft Realm world [4], which is only 10, our result looks quite good.

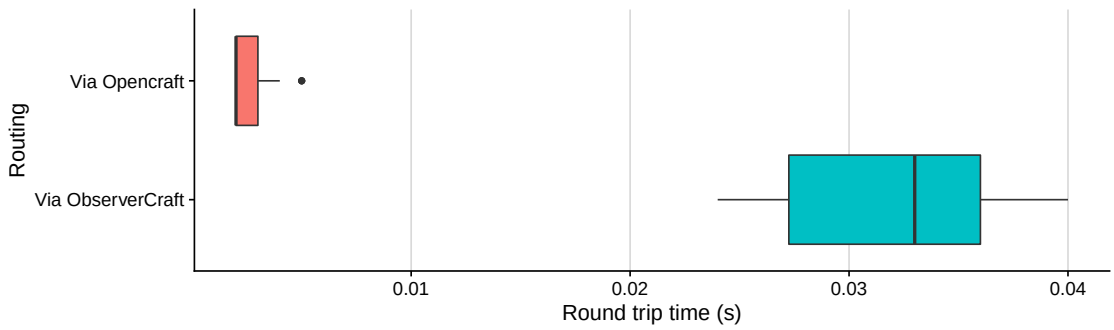

Figure 9: Round trip time for a chat packet

MF3: ObserverCraft introduces latency to its users

We measured the round trip time of a chat message. The obtained results are depicted in Figure 9. To measure the round trip time, we sent out a chat packet every second and measured the time it took to get back to the sender. The graph depicts the round trip time in seconds for routing via Opencraft(the server) and via ObserverCraft (our proxy). As we can see, the round trip time via ObserverCraft is significantly higher compared to that of the server. This can be explained by one of the design decisions we took. In order to measure our systems performance we decided to implement a ticker that ticks every 50ms. As explained in Section 3, this ticker, among other things, is responsible for signaling to the message handler to empty the queue that contains the messages we have received from the server and process them. When a new packet enters our system, it will have to wait for the tick that was running at that time to finish before it gets processed. This means that it can take anywhere from 1 ms, if the packet arrived just as a tick ended, to 50 ms if the packet arrives just as a new tick started, before the packet can start getting processed. This explains the larger spread in round trip time visible in the plot in Figure 9.

For a player of a Minecraft-like game, an increase of 50ms in latency would likely lead to a break of QoS requirements. ClayPool. et al. [12] show that for first person perspective video games, of which Minecraft is one, at a latency of 100 milliseconds there is a measurable drop in player performance as the game becomes less responsive. An increase of 50ms would likely get a player very close to or over this 100ms threshold. For a spectator however the latency is of much less concern as they are only passively observing the game world and not interacting with it. If a spectator still receives all the game state updates, the only effect that an increase in latency would give is that they might not see the changes and actions of players in the game world at the same time as they are actually happening. This is not detrimental to the Quality of Experience for a spectator, as functionally nothing changes. So even though traditionally an increase in latency of 50 ms would be considered a bad result, for our system it is acceptable.

6 Discussion

Our results are promising and show that we can indeed support large numbers of spectators for Minecraft-like games without diminishing the total number of actual players. It is important to discuss the limitations of our system and experiments.

We only managed to complete one iteration of design, implementation and experimentation. This means that our prototype does not have all the features that we wanted it to have and the testing we did is not as extensive as we wanted. We were only able to do 5 iterations for our overhead experiment. In order to properly capture performance variability, 5 iterations is not enough.

Another issue came up during the scalability experiment. Due to a certain configuration file for Yardstick being hard-coded, we were not able to connect Yardstick to two different systems with a different configuration for each. We could have connected more players to our system, but in doing so we would be forced to also connect more players to the Opencraft server as well. We wanted our experiments to be done with the Opencraft server in a non overloaded state. We found however that connecting more than 40 players

overloaded the server. This resulted in us only being able to connect 200 players to ObserverCraft during our scalability experiment, as we could only configure each instance for yardstick to connect 40 players, and we only had a limited amount of nodes on the DAS5 available. Due to time constraints we decided to not try to fix this issue and use the results we got with 200 connections. We were able to extrapolate the result of 1000 spectators based on the measured results of the experiment done with 200 players, but it would be better if we could connect 1000 emulated players.

7 Conclusion

Gaming is more popular than ever. Minecraft is especially popular as it is the most sold and one of the most played video games of all time. With this growth in popularity comes the need for more scalability when it comes to playercount, yet research has shown that Minecraft-like games only scale to around 200-300 players in a single instance. Minecraft-like games are especially hard to scale because they are real-time interactive distributed systems that are under stringent quality of service constraints. We identify that a spectator does not require the same level of QoS as a player. With this in mind, we designed ObserverCraft, and we implemented a prototype in Opencraft, a Minecraft-like game. We offer different spectator modes to allow users to spectate in active as well as passive ways. With real world experiments we show that our system has a low overhead on the game server, thus having almost no effect on the number of players that can play on the server while our system is active. We also show that our system supports up to 1000 spectators in a Minecraft-like game. ObserverCraft allows users to easily and effortlessly connect to it via the standard Minecraft client. Operators can set up multiple instances of ObserverCraft for their servers to allow a large number of spectators to observe the game world.

8 Future work

One thing that this thesis has not touched on is more efficiency when it comes to forwarding the game state to our observer system. Currently the Minecraft-like game server forwards the entire game-state to ObserverCraft, but spectators might only be active in a small region of the world, thus not actually needing the game-state information from the other parts. If we could, with perhaps a custom networking protocol, find a way to only send the game state updates that are part of the areas relevant to our spectators , we might be able to reduce the overhead our system puts on the game server. A lower overhead would allow more actual players to play, or more instances of our system to be hooked up to the game server. Future work could focus on creating a custom networking protocol for Minecraft-like games that allows an observer to only request information relevant to its spectators instead of the entire game state. Similarly, future work could also focus on making the way our system forwards the game state to spectators more efficient. The more efficiently we can forward the relevant game state information to our spectators, the more spectators our system will be able to support.

A second item that we have not addressed in this paper is improving the spectator experience. We do offer different ways of spectating, but have not identified how we could specifically enhance the spectating experience for Minecraft-Like games. Future work could try to identify what metrics and information would be of interest to a spectator of a Minecraft-like game, and try to incorporate this information into the spectating system.

Lastly, our research has not done any case studies that evaluate how spectators like using our spectator system. In the future it would be great to do a survey to find out which features of our system users like and which they do not. This information is very valuable for designing future iterations of spectating systems for Minecraft-like games.

References

[1] Mcprotocollib. https://github.com/Steveice10/MCProtocolLib/tree/1.12.2-2.

[2] Minecraft. https://twitchtracker.com/games/27471.

[3] Minecraft protocol. https://wiki.vg/Protocol.

[4] Minecraft realms. https://www.minecraft.net/en-us/realms-plus.

[5] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[6] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, page 6–es, New York, NY, USA, 2006. Association for Computing Machinery.

[7] Raphaëlle Brissette-Gendron, Pierre-Majorique Léger, François Courtemanche, Shang Lin Chen, Marouane Ouhnana, and Sylvain Sénécal. The response to impactful interactivity on spectators' engagement in a digital game. *Multimodal Technologies and Interaction*, 4(4), 2020.

[8] Christian Carlsson and A. Pelling. Designing spectator interfaces for competitive video games. 2015.

[9] Sven Charleer, Kathrin Gerling, Francisco Gutiérrez, Hans Cauwenbergh, Bram Luycx, and Katrien Verbert. Real-time dashboards to support esports spectating. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play*, CHI PLAY '18, page 59–71, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Helen Chiang. Minecraft: Connecting more players than ever before. https://bit.ly/2F4cpkF, 2020.

[11] Myko Chiong. It's not just fortnite: Video games may be the future of live music, 2020.

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

[13] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. Dyconits: Scaling minecraft-like services throughdynamically managed inconsistency. In *41th IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Virtual Event, July 7 - July 10, 2021*. IEEE, 2021.

[14] 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)*. USENIX Association, July 2020.

[15] Counterstrike Fandom. Spectator. https://counterstrike.fandom.com/wiki/Spectator.

[16] James Fong and Brian Trench. The rise of a new entertainment category: Esports. *https://upcea.edu/*, March 2017.

[17] Gamepedia. Block. https://minecraft.fandom.com/wiki/Block, 2020.

[18] Gamepedia. Chunk. https://minecraft.gamepedia.com/Chunk, 2020.

[19] Juho Hamari and Max Sjöblom. What is esports and why do people watch it? *Internet Research*, 27, 04 2017.

[20] Mansoor Iqbal. Twitch revenue and usage statistics (2021), 2021.

[21] Bryan Lufkin. How online gaming has become a social lifeline. https://www.bbc.com/worklife/article/20201215-how-online-gaming-has-become-a-social-lifeline, 2020.

[22] Shang Ma, Kevin Byon, Wooyoung Jang, Shang-Min Ma, and Tsung-Nan Huang. Esports spectating motives and streaming consumption: Moderating effect of game genres and live-streaming types. *Sustainability*, 13:41–64, 04 2021.

[23] Newzoo. Newzoo's global esports live streaming market report 2021. https://bit.ly/3w04Tx3, 2021.

[24] League of Legends Fandom. Spectator mode. https://leagueoflegends.fandom.com/wiki/Spectator_Mode.

[25] G. Rodola. psutil 5.8.0. https://pypi.org/project/psutil/, 2020.

[26] Max Sjöblom and Juho Hamari. Why do people watch others play video games? an empirical study on the motivations of twitch users. *Computers in Human Behavior*, 75:985–996, 10 2017.

[27] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. Yardstick: A benchmark for minecraft-like services. In Varsha Apte, Antinisca Di Marco, Marin Litoiu, and José Merseguer, editors, *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*, pages 243–253. ACM, 2019.

[28] Wallace Witkowski. Videogames are a bigger industry than movies and north american sports combined, thanks to the pandemic, 2020.

[29] Zheng Yan, Rui Gaspar, and Tingshao Zhu. How humans behave with emerging technologies during the covid-19 pandemic? *Human Behavior and Emerging Technologies*, 3(1):5–7, 2021.