PorygonCraft: Improving and Measuring the Scalability of Modifiable Virtual Environments using Dynamic Consistency Units

Bachelor Thesis Computer Science

Author:
J.J.C.J. Cuijpers

Supervisor:
J.J.R. Donkervliet, MSc
Co-reader:
Prof. dr. ir. A. Iosup

August 21, 2020

Abstract

The video game industry is a growing industry with revenues of over 150 billion a year as of 2019. Not only are video games used for entertainment purposes, but also for a variety of other uses such as, but not limited to, education. Modifiable virtual environments (MVE) are real-time, online, multi-user environments which allows its users to interact with every aspect of the world through programs by building, connecting. Minecraft-like games are Modifiable Virtual Environments and these are highly popular. However, these MVEs are poorly scalable, which raises the question: how can we improve the scalability of Minecraft-like games? We design, present, implement, and evaluate a system that utilizes *dynamic consistency units* to support larger numbers of players. Introducing bounded inconsistency in a Minecraft-like world will allow us to control the maximum inconsistency in a Minecraft-like game. Through the use of real-world experiments, we show a promising improvement in both the number of outgoing packets and the amount of outgoing bytes, as well as the utilized CPU cores.

Contents

1 Introduction **3**
1.1 Problem statement . . . 3
1.2 Research questions . . . 4
1.3 Methodology . . . 4
1.4 Structure of the thesis . . . 5

2 Background **5**
2.1 Workload partitioning . . . 5
2.2 Interest Management . . . 6
2.3 Minecraft-specific solutions . . . 7
2.4 Consistency in Distributed Systems . . . 8
2.5 Yardstick . . . 9
2.6 A Minecraft-like game: Glowstone . . . 9
2.7 Dynamic Consistency Unit . . . 9
2.8 Dyconits in PorygonCraft . . . 10

3 PorygonCraft: A Dyconit-based System for Minecraft-like Games **11**
3.1 Requirements . . . 11
3.2 High-level design of the PorygonCraft system . . . 11
3.3 Message preprocessor . . . 12
3.4 Dyconit control unit . . . 13
3.5 Dyconit collection . . . 13
3.6 Policy manager . . . 13
3.7 Message sender . . . 14

4 Consistency policies **14**
4.1 Donnybrook Uniform policy . . . 14
4.2 Vector-Field Consistency policy . . . 15

5 Experimental setup and results **16**
5.1 Environment . . . 16
5.2 Data collection and metrics . . . 16
5.3 Workload . . . 16
5.4 Experimental results . . . 17

6 Discussion **20**

7 Conclusion **20**

8 Future work **20**

1 Introduction

The video game industry is thriving, with more than 214 million video game players across the United States, video games have become an important tool to connect with others. 64% percent of U.S. adults regularly plays games, 65% of those adults play with others, and the large majority claims that video games provide them mental stimulation and relaxation [1]. The video game industry has generated almost $120 billion in revenue in 2018 [2] and reached 152 billion in revenue in 2019 [3, 4, 5], making video games the leading form of entertainment [1].

As video games are also used increasingly for educational and social purposes, in which they can increase empathy, co-operation, and the development of skills like math or problem-solving [6], we identify that video games are becoming increasingly important as they become more socially interactive. In contrast to board games, where players usually only play with family and close friends, it is now a lot easier to interact with strangers, blurring any form of cultural boundaries or socio-economic differences [7]. Additionally, more and more video games are made for educational purposes to increase the motivation and engagement of students and to be able to give them a hands-on approach.

A modifiable virtual environment is a real-time, online, multi-user environment that allows its users to modify the virtual world's objects and parts, create new content by connecting components, and interact with the world through programs [8]. We also refer to modifiable virtual environments as *Minecraft-like games.*

A prime example of a Minecraft-like game is Minecraft itself. Minecraft was created by Mojang in 2011 and purchased by Microsoft in 2014 for $2.5 billion. As of May 2020, it has sold over 200 million copies, and currently has around 126 million active players across all platforms each month [9].

The player emerges himself in a world consisting of blocks where everything is modifiable, allowing them to create and discover their world and having the possibility to share this world with others. Not only is Minecraft used as a game for entertainment, but it is also used for educational purposes, such as building computers [10]. This is made possible with the release of Minecraft Education Edition, a platform teachers can use to teach courses to their students [11].

Minecraft-like games do not scale as is also observed in Yardstick [12]. Despite the large number of active players, instances are limited to 200-300 players. Because Minecraft-like games are becoming more popular we identify this scalability problem as a problem that needs to be solved.

1.1 Problem statement

Scalability is one of the grand challenges in computer science. The Internet is a global-scale computer system that compromises over 100 million nodes, computer performance and price-performance have improved by 100% every year since 1985 [13]. However, there are still many of challenges when it comes to improving the performance and scalability of these systems. Improving the scalability of Minecraft-like games is especially challenging, because, they are real-time interactive distributed systems, which require low-latency, consistency, and real-time execution.

1.2 Research questions

In this thesis, we evaluate the scalability improvement on the communication aspect of Minecraft-like games by using dyconit consistency in modifiable virtual environments, which will be explained in-depth in Chapter 3. By introducing *dynamic consistency units* (dyconits), we aim to improve the scalability by reducing the number of outgoing messages. We do this by allowing increased inconsistency between players for data in which they are not interested, as proposed by Donkervliet et al. [8]. To address this scalability improvement, we address the following research questions.

- **RQ1:** How to design a dynamic consistency unit system for Minecraft-like games?

  Designing a dyconit system is challenging because:

  - They have stringent consistency and Quality of Service (QoS) requirements and;
  - They operate differently from databases, in which consistency models have been studied in much more depth, and for which conits were originally designed.

  Designing a system based on dyconits is important because it helps us to understand how we can utilize dyconits in solving this scalability problem and to deliver a general framework others can continue working on.

- **RQ2:** How to enable game operators to make trade-offs between consistency and scalability?

  There is no one-system-fits-all solution, therefore, we question how we can enable game operators to make trade-offs between consistency and scalability, as every application has, for example, its own requirements on how consistent it should be.

- **RQ3:** How to evaluate the performance benefits of such a system.

  We need to question how we evaluate such a system. An important part of answering the scalability problem is to use the right metrics so that we can quantify how well a solution is doing and to make it possible to compare solutions.

1.3 Methodology

We approach the first research question by identifying important system requirements. Then, we observe and understand the technical internals of the Minecraft-like game to further identify how our design should interact with the Minecraft-like game. For each requirement, we explore and deliver the component that will solve that particular requirement. Finally, these components form one coherent system.

We approach the second research question by making use of policies. These policies can be created and configured by the game developer to manage the dyconits. As a result, the game developers are able to make trade-offs between consistency and scalability.

We approach the third research question by designing and conducting real-world experiments. Based on the real-world experiments, we identify a set of metrics that we can use to determine the performance of our system.

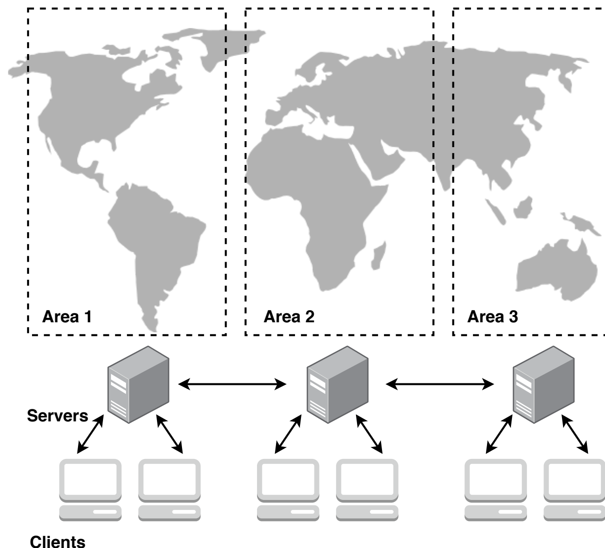

Figure 1: Workload partitioning, each server maintains a part of the world.

1.4 Structure of the thesis

The first Chapter provides a brief introduction to Minecraft and addresses the main challenges and research questions of this thesis. In Chapter 2, we introduce background information including scalability on games, and discuss background information on consistency models, the conit data structure, and Yardstick, a benchmarking tool for Minecraft-like games. Chapter 3 discusses the design of PorygonCraft, and Chapter 4 goes in-depth on the policies it uses. Chapter 5 will address the setup and results of the experiments. Finally, the discussion, conclusion, and future work of this thesis will be discussed in Chapter 6, Chapter 7, and Chapter 8 respectively.

2 Background

Current state-of-the-art large-scale multiplayer games like World of Warcraft, Guild Wars 2, or RuneScape are using client/server architectures. The servers are hosted by the game company, and each player can run a game client on their computer. Supporting large number of players is challenging, usually resulting in the need for more powerful hardware to deal with the amount of computation that needs to be done at the server-side. To reach a playable state in terms of quality of experience, these game companies use *world partitioning*.

2.1 Workload partitioning

A game server can only support a certain number of players. To build large-scale games, we need to divide the work over multiple machines. World partitioning is an intuitive and effective approach.

By partitioning the virtual world in geographical regions, as visualized in Figure 1, we fragment the workload over multiple machines. To understand why world partitioning does not solve the scalability challenge of large-scale online games, we have to introduce the concept of *consistency*. We consider a Minecraft-like game to be consistent if the

state of the world is in the same state as the world that a player sees. When two players see a different world or when the state of the world in the server is different from the state of the world that the player sees, we translate that as inconsistency.

Splitting up the world across multiple servers yields inconsistency between players. Although we divide the player base over multiple servers, we do not want to give up the possibility for players to interact with each other. Therefore, all servers —or at least the relevant ones— have to share data between them. If the inconsistency between players becomes too large, the player notices, leading to reduced gameplay experience due to out-of-order operations or reduced latency.

2.2 Interest Management

Much of the research done on the scalability of games focuses on peer-to-peer systems, which are decentralized systems with a network of interconnected nodes (in this case clients) that exchange data (in this case player updates). Three of such systems are Colyseus [14], Donnybrook [15], and Vector-Field Consistency [16]. These systems make heavy use of *interest management*. Interest management is determining what information is relevant to a certain player.

Donnybrook

Donnybrook's approach to reducing bandwidth—or reducing the amount of data that can be transferred from one end to another—is to make players only receive updates from other players in their interest set, the interest set which is the set of entities the player is paying attention to.

The interest set in Donnybrook is kept small by making use of the limitations of the human attention. Unlike *area of interest*, which grows with the density of players. The players outside of the interest set are shown as *doppelgängers*. A Doppelgänger is a computer-controlled player running on the client. Doppelgängers try to imitate the behavior of players outside the area of interest. Unlike characters in the interest set, the players do not get continuous updates on the *doppelgängers*.

Donnybrook's approach is boolean. Players are either in the interest set, or not, resulting in coarse-grained control over the inconsistency of the game state. In this thesis, we show that PorygonCraft can achieve fine-grained control of the inconsistency between players by using conits and by dynamically adjusting the consistency bounds and even quantify the inconsistency that is present.

Colyseus

Different from Donnybrook is Colyseus, which focuses on data replication rather than inconsistency through interest management. Colyseus takes advantage of two properties of games, namely that games tolerate inconsistency in the game state and that games have predictable read/writes of the shared states. Each participating player is considered a node and every node contains primaries and replicas of game objects. Every game object is assigned a primary node that has authoritative copies of it. The primary nodes are responsible for (re-)ordering the updates that are applied to those game objects and if any game object is updated at a different node than the primary node, then the primary node is free to reorder the update.

Colyseus does not guarantee consistency, but eventual consistency, among nodes or any form of bounded inconsistency but uses an optimistic approach to propagate the updates to peers after each game tick. The approach of Colyseus is not sufficient, because we want to limit the inconsistency in the system. PorygonCraft, however, optimistically bounds inconsistency between the player and other entities (which could also be another player).

Vector-Field Consistency

Santos, Veiga, and Ferreira proposed Vector-Field Consistency [16]. A new consistency model that takes time, sequence, and value criteria to limit inconsistency between data replicas with techniques based on locality-awareness.

Vector-Field Consistency considers the virtual world to be a 2-dimensional space, as an abstraction from an N-dimensional virtual world. Positioned in this virtual world are objects, and their consistency depends on the distance to a pivot, field-generated consistency zone (a region around the pivot), and consistency vectors (a vector consisting of values that indicate the bounds). A pivot could be anything, for example, a player. The consistency vectors associated with the objects within the consistency zone contain a numeric scalar for time, sequence and value that bounds the inconsistency between the pivot and the object.

The two main properties underlying the Vector-Field consistency model represents the general idea of PorygonCraft. The first property is that it has some rule regarding the importance of an entity. We make policies based on comparable rules and implement these as a policy that runs on PorygonCraft. The second property is that is has a way of keeping track of the inconsistency between entities in the game. We use dyconits to bound the inconsistency in the Minecraft-like game.

2.3 Minecraft-specific solutions

To improve the scalability of Minecraft, Diaconu and Keller introduced Kiwano [17], Kiwano is a distributed system that separates the virtual world into components. Those components are geographically allocated zones in the real world to reduce latency between players and the server. For example, every country has its own. One of the drawbacks of a peer-to-peer architecture is that the workload distribution across the servers is uneven. Kiwano solves this problem by forming zones into irregular shapes, these shapes are determined by the dynamic objects in the zone and undergo continuous reshaping to maintain an even distribution of the workload across the shapes. Consequently, it is fairly easy to allocate resources as the workload is balanced. A significant difference between a normal client/server architecture and Kiwano is that in the case of Kiwano the server sends the updates to Kiwano instead of directly to the clients. Kiwano then determines which clients need to be updated. A drawback of Kiwano is that it does not support world updates, which are an important element of Minecraft-like games because Kiwano only propagates movement updates. World partitioning introduces a decline in quality of experience when the players are in a densely populated area like a city, where the zones become so small that although players might be able to see each other, they are not sharing the same game state. Using Kiwano, Diaconu, Keller, and Valero have implemented for a system for Minecraft called Manycraft [18]. Manycraft allows an unlimited number of players to interact in a read-only environment by having a Manycraft node that communicates with Kiwano. Although Manycraft supports an unlimited amount of players in the same world, it is still not possible to

modify the world and play Minecraft to its full potential because, as indicated the world is read-only (i.e., it is not possible to modify the world).

Moll et al. proposed a concept for a Minecraft implementation with a distributed architecture based on Named Data Networking (NDN) [19]—a future internet architecture that lies outside of the scope of this thesis—replacing the current networking layer of games with one that is better suited to modern games like Minecraft. In state of the art MMOGs such as Minecraft, the virtual world is split into zones, each managed by its own server, to improve the scalability of those games. Instead of the server pushing information in the form of encoded chunk data to the clients, as is done in current implementations, Moll et al. propose a concept to make use of the advantages of NDN, such as inherent multicast functionality and security implemented at the packet level. While this concept focuses on managing the distributed game state in a multi-server architecture by using a different architecture for the network layer, the proposed concept in this thesis will be focused on the application layer and improving the scalability of a single server by better managing its workload.

2.4 Consistency in Distributed Systems

Unfortunately, both workload partitioning and interest management introduce inconsistency. Given a distributed system, it is impossible to simultaneously guarantee consistency, availability, and partition tolerance; this is known as the CAP theorem [**?**]. Generally, we can make a distinction between two types of systems that represent both extremes on a spectrum: BASE (Basically Available, Soft-state, Eventual consistency) [20], and ACID (Atomicity, Consistency, Isolation, and Durability) [21]. A BASE system focuses on keeping a system highly available, while an ACID system focuses on keeping a system highly consistent. An example of an ACID system is a traditional database. In this thesis, we introduce inconsistency with dyconits. Dyconits allow us to control the maximum inconsistency that we allow in our system. Therefore, taking a place on the consistency spectrum between the two extremes.

Because each client has its copy of the virtual world—or other shared data—whenever the server updates one of the clients, its data can differ from the data of the other clients. This is called inconsistency. The consistency model can influence how replicas, which is essentially copies of the same data, differ internally. However, not every consistency model specifies the difference of replicas. Therefore, a consistency model is a contract between processes and the data store [22]. Consistency models that tolerate inconsistency are for example sequential consistency and causal consistency.

To quantify inconsistency, Yu and Vahdat [23] introduced the consistency unit (conit). A conit specifies the data over which consistency is to be measured [22], using the following consistency dimensions:

1. **Numerical error**: the numerical difference between the local view and the global state, calculated as the sum of the weights, of the unseen writes.
2. **Order error**: the out-of-order writes that affect a conit, or in other words the level of guarantee in the ordering of updates.
3. **Staleness**: the maximum amount of time in which local data has not seen a write.

Any of these bounds can take any non-negative numerical value. Furthermore, a system can specify an arbitrary number of conits, enabling specified consistency architectures. When applied in virtual environments, we do not know what the consistency

requirements are, because entities are constantly moving. Therefore, we need conits that can change their bounds based on the current context of the data, a dynamic conit system.

2.5 Yardstick

To understand the performance and scalability of Minecraft-like games, Van der Sar, et al., introduced the benchmarking tool Yardstick [12], allowing the possibility of using relevant and realistic workloads and metrics in their benchmarks of Minecraft-like games in the benchmarking process. With Yardstick, the user can deploy workloads on a a Minecraft-like server that are determined by the virtual world and a set of emulated players, it is worth noting that users have the ability to modify the workload programmatically and by tuning predefined parameters. After the workload is deployed to the Minecraft-like server, Yardstick monitors the machine running the server, extracting system, application and derived performance metrics (i.e., network usage, or the number of messages per second).

2.6 A Minecraft-like game: Glowstone

Glowstone is an open-source Minecraft server written in Java. At the time of writing it has 119 contributors in their Github [24]. Glowstone has an active community on Discord, which is a free online instant messaging application, comprising almost 500 people. Glowstone supports Bukkit-, Spigot- and Paper-API plugins natively.

Just like Minecraft, the world in Glowstone consists of biomes, these biomes are by default compromised of 32x32 chunks, and every chunk is comprimised of 16x16 blocks. A biome is a region that represents an environment. This could be, for example, a forest or a desert. A chunk is a collection of blocks, and blocks get spawned per chunk. Blocks are the smallest unit of the world, the player can interact with.

We implement PorygonCraft—which will be specified in Chapter 3—on Glowstone. Although Glowstone has poorly implemented several complex features such as Redstone compared to the official Minecraft server, this is not a problem because Yardstick will only deploy bots that are walking around, and do not make use of these features.

2.7 Dynamic Consistency Unit

The dyconit is a consistency unit with bounds that can change over time. Dyconits are a data structure with a staleness bound, numerical error bound and an order error bound. In this thesis we will not focus on the order error bound because the game server imposes already a global order of all messages. The staleness bound indicates the maximum amount of time that the data associated with the dynamic conit does not need to be synchronized with the client, while the numerical error bound indicates the maximum difference in data between the actual state and the current perceived state of the client. The numerical error is quantified by giving each update a certain weight, based on its importance. The importance of an update, is determined by the current active policy.

Bounding by staleness

We define staleness as the age of a message. A message that has become stale is not seen by others for a certain number of time. When bounding by staleness $B_{staleness}$[1], we make sure that the server pushes the messages to the clients at least every $B_{staleness}$.

[1] $B_{staleness}$ in milliseconds.

Bounding by numerical error

When bounding by numerical error B_{num_error}, we make sure that the server pushes messages to the client when the client lags behind on a certain number of messages with a total weight of B_{num_error}. We define the weight of messages as the total number of messages. In case that we assign a value, indicating the importance of a message, the total weight becomes the sum of these values. For example, the weight of an update that changes the location of an entity outside of your rendering distance (the distance the game client can render for the user to see) can be 10, while a location update of an entity in front of you might have a weight of 50, because it is more important.

It is worth noting that the weights on their own do not hold any value. The value of the weights is solely determined relatively to the numerical error bound. For example, if the numerical error bound is set to 100, then an update that is assigned a value 100 is considered very important, because it will trigger the server to immediately push the update to the client. However, if the numerical error bound is set to 1000, the update is seen as less important, because the server will not immediately push it to the client.

For both the staleness- and numerical error bound, we cannot determine the bound during initialization. It is impossible to determine a fixed value during initialization, because different amount of incoming updates per time frame can significantly impact the number of updates the client can be behind without the player noticing.

2.8 Dyconits in PorygonCraft

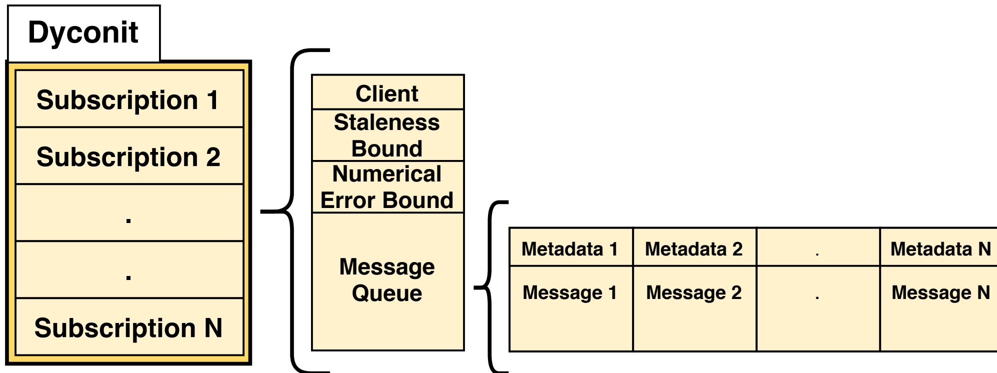

Figure 2: Dyconit design.

Dyconits in PorygonCraft are used to limit the outflow of messages and inconsistency in the game. It does so by buffering the messages in the dyconit and having consistency bounds. We define buffering as temporarily storing messages. However, dyconits in PorygonCraft have messages of multiple clients stored and the dyconit might need to have different consistency bounds for each client. We solve this by introducing multiple queues in the dyconit with each their own consistency bounds. A queue with consistency bounds is called a subscription. Figure 2 shows that each subscription has information about the client, the bounds, and the message queue. The message queue holds all the messages that needs to be send to the respective client. Each message is appended with metadata, that contains information needed to successfully send it. The process of appending metadata is further elaborate in Chapter 3.

3 PorygonCraft: A Dyconit-based System for Minecraft-like Games

In this Section, we discuss the main components of the dynamic consistency system PorygonCraft. PorygonCraft is a system that reduces the game's bandwidth usage while limiting inconsistency. PorygonCraft captures outgoing messages before it determines, with the help of dyconits, if and when it should send the message to the player. To design the dynamic consistency unit system, we set up requirements.

3.1 Requirements

- **R1:** Efficiently maintaining the state of a large number of dyconits.

  We need to be able to efficiently maintain the state of a large number of dyconits. Two of the important follow-up operations are the possibility of creating and removing dyconits from our system, and modifying the consistency bounds of the dyconits at runtime. However, the computational time it takes to maintain the dyconits should not be small, because it will affect the Quality of Service for the player.

- **R2:** Create and remove dynamic conits at runtime.

  For the system to reduce the time needed for the game to process outgoing messages, its overhead needs to be small. PorygonCraft has to iterate across all dyconits every time it gets the signal to assess the consistency bounds. Therefore, we should never have dyconits in the system that are not in use.

- **R3:** The ability to modify the consistency bounds of the dyconits at runtime.

  What separates a dyconit from a conit (consistency unit), is that the consistency bounds of the dyconit are subject to change depending on the context of the game and the active policy—which will be discussed in the next chapter. Because the state of a virtual world is constantly changing (i.e., day/night cycles, in-game events, etc.), we need to change these bounds while the game is running, or in other words, we need to be able to modify these bounds at runtime.

- **R4:** Keep track of the consistency bounds of the dyconits and prevent exceeding bounds.

  Our last requirement is a result of the three other requirements. Now that we have full control and transparency over our dyconits we can use them to bound inconsistency and monitor their inconsistency.

3.2 High-level design of the PorygonCraft system

As a result of the requirements set up in the previous section, we present PorygonCraft and discuss each component. PorygonCraft is a system that enables Minecraft-like games to use dyconits, because of the ability to control what we do with the messages. We define a message as an event that happens in the Minecraft-like game that needs to be propagated to the client.

When PorygonCraft captures a message it attaches metadata to it. The metadata ensures that we keep information that we need to send the message to the client. The

message, together with the metadata, is buffered to a dyconit after. A message gets buffered to a dyconit based on the location of the entity (i.e., another player or a computer controlled character) the message is coming from. Every time the Minecraft-like game sends a signal to PorygonCraft to asses the bounds, message queues are forwarded to the client, if and only if the consistency bounds of the dyconit are exceeded.

A policy enforcer will continuously assess the consistency bounds of the dyconit and adjusts them if necessary. The active policy is determined during the instantiation of the game by the policy manager.

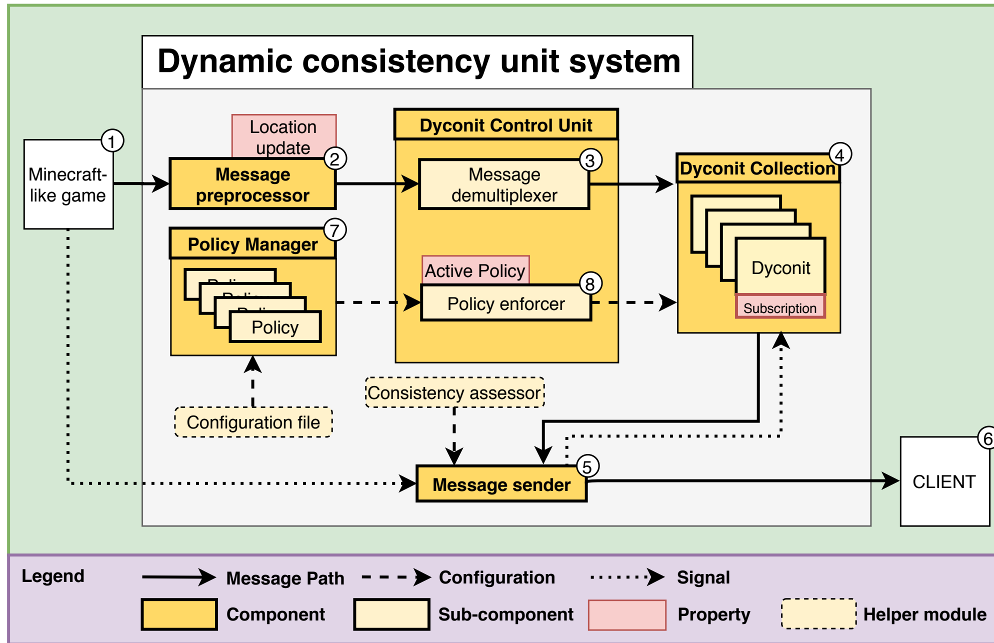

Figure 3: PorygonCraft system design. The orange boxes indicate the main components, the yellow boxes indicate subcomponents, the red boxes are properties of the (sub)component it is attached to.

3.3 Message preprocessor

The message preprocessor (component 2) is shown in Figure 3. The messages need to be appended with metadata to determine information such as type, content, and recipient of the message. We will need this information later to forward the message to the correct client. PorygonCraft does this by preprocessing the messages with metadata as soon as it captures the message. It also enables PorygonCraft to be used in a large collection of Minecraft-like games, instead of being specifically tailored to Glowstone. In the PorygonCraft implementation on Glowstone, the message preprocessor does the following:

1. Compute the location of the sender of the message.
2. Store the client recipient and its session, enabling us to send the message at a later point in time.
3. Pass the message with metadata to the Dyconit Control Unit.

3.4 Dyconit control unit

The dyconit control unit (DCU) is where we create, adjust, and remove dyconits at runtime. It enables PorygonCraft to satisfy requirements **R1** and **R2**. Additionally, the DCU is the place where PorygonCraft monitors the consistency bounds on the dyconits, and adjusts them when needed, satisfying requirement **R4**. This is done with the two subcomponents located in the DCU: the message demultiplexer (component 3), and the policy enforcer (component 8). Starting with the message demultiplexer, which takes the message that was tagged with metadata by the message preprocessor, and determines to which dyconit the message belongs. In the occurrence that the dyconit does not exist yet, the DCU will create a new dyconit. Each message is assigned to a single dyconit.

The second subcomponent in the DCU is the *policy enforcer*. The policy enforcer is responsible for adjusting the consistency bounds of a dyconit. In the event that a dyconit has subscriptions, then every subscription has its own set of bounds. Every time a player receives a message, all their subscriptions will be assessed and the corresponding consistency bounds are adjusted based on the current policy. The policies will be discussed in more detail in Section 4.

3.5 Dyconit collection

The dyconit collection (component 4), is closely connected to the DCU. Its only function is to hold all the dyconits and its subscriptions currently present in the system, and to track which dyconit is associated with each chunk.

In the PorygonCraft implementation on Glowstone, we associate every chunk with a dyconit. However, multiple clients can be interested in a single chunk. In order to solve this problem every dyconit has multiple subscriptions (each subscription is associated with a single client) and every subscription has its own bounds.

3.6 Policy manager

The policy manager (component 7), holds a set of different policies that can be assigned to the policy enforcer. The server moderator can decide which policy he wants to initialize the game with. If no specific policy is chosen, it will use a null policy, which causes, every message to be send immediately to its recipient. The null policy acts like a Minecraft-like game without PorygonCraft. The policy manager enables the possibility to change the active policy at runtime. This is not evaluated in this thesis, but lends itself for future research.

The policy manager can have any number of policies that are implemented through an interface, and its only task is to tell the DCU which policy is the current active policy. Initially, the policy is chosen by a configuration, however, the current design makes it possible to expand on it by adding a feature that would allow for changing policies at runtime, which again, would be done by a set of rules (a policy).

Policies change the staleness bounds and numerical error bounds of the dyconits. Therefore, it is possible to implement any policy that would affect these bounds, but one could be constrained by the variety of information that could be extracted from the Minecraft-like game. For example, if you want to implement a policy based on the area that the player can see, but the Minecraft-like game does not offer a way to extract this information, then it would not be possible to make a policy based on this information,

however, if the Minecraft-like game does offer this information, then it is possible to implement this policy. Another concern about implementing policies is that they need to be time-efficient. If enforcing a policy on a dyconit takes too long, due to computations, then we might have more inconsistency in our system than we allow. The system does not guarantee that the bounds are met, but it optimistically bounds the inconsistency.

PorygonCraft assumes that it can extract information about the game to enforce the policies. However, this is not a part of the design. For PorygonCraft to be able to enforce the active policy, it needs to be able to access the same information that is used in the active policy.

3.7 Message sender

Depending on the architecture of the Minecraft-like game, The message sender (component 5), can be called from outside of PorygonCraft. As soon as the message sender is invoked, it will fetch the dyconit collection, iterate over every dyconit and/or subscription, and determine if it should send the messages that are collected to the recipient. Upon exceeding the consistency bounds the message queue will be emptied, and the dyconit will be reset (the staleness and the numerical error of the dyconit will be set to zero).

4 Consistency policies

To change a consistency unit into a dynamic consistency unit we need a guideline on how to change the consistency bounds. Policies are introduced as a main component in PorygonCraft. In this thesis, we introduce two different policies, each showing a different set of rules, and the ability to mimic existing solutions, such as Donnybrook and Vector-Field Consistency.

4.1 Donnybrook Uniform policy

The first policy evaluates the behavior of Donnybrook, reducing the frequency of sending less important messages, thus lowering the amount of bandwidth needed. The notion of doppelgängers is not treated, because doppelgängers are a client-side optimization and are not part of the dyconit policy.

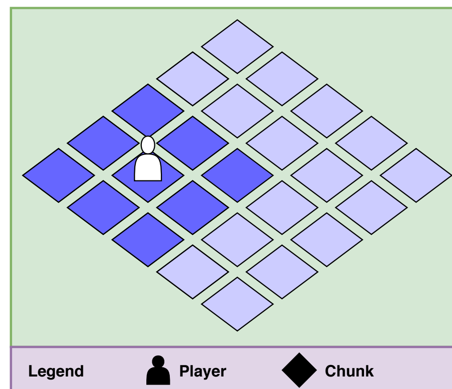

Figure 4: A visual representation of the Donnybrook policy. The dark squares correspond with a lower amount of inconsistency and the light blue squares correspond with higher amounts of inconsistency.

Bounding the inconsistency is a binary event when the Donnybrook policy is applied to PorygonCraft. Having bounded inconsistency is realized by making sure that a player gets updated at least every 100 milliseconds when they are in an arbitrarily large region around the player.

Similar to Donnybrook, the area of interest is where bounded inconsistency is enforced. In Glowstone we take the area of interest as

the chunk the player is standing on, and the
chunks around the player, as seen in Figure 4. When the player moves, the area of interest changes according to our movement, as seen in Figure 5, and PorygonCraft will change the consistency bounds of the dyconits accordingly.

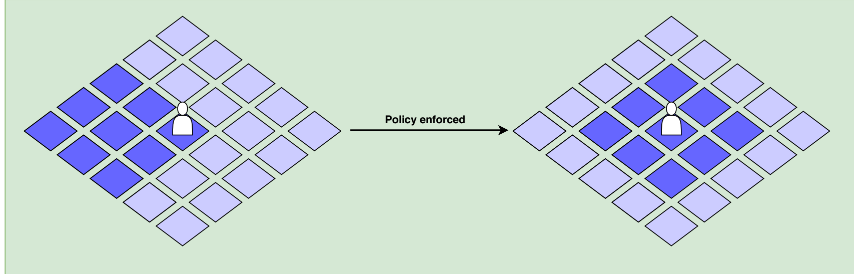

Figure 5: Changing the consistency level of the chunks based on movement under the Donnybrook policy.

4.2 Vector-Field Consistency policy

The second policy is more fine-grained than the previous policy. Instead of having a binary event, we introduce a gradient of consistency based on distance. This policy is the implementation of the Vector-Field Consistency model.

The level of inconsistency of the world is determined by the distance of an entity and the player, where a shorter distance results in a lower inconsistency, due to the bounds being lower. We define a ring of chunks, of a player, as the chunks around the player of which the associated dyconits have the same consistency bounds. This means that there is less inconsistency closer to the player, as opposed to a ring of chunks that is farther away from the player, as seen in Figure 6.

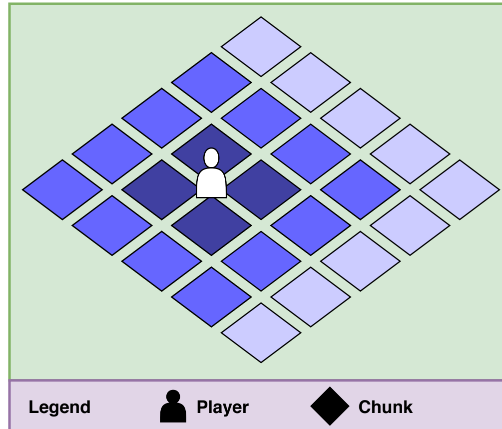

Figure 6: A visual representation of the epicenter policy.

In the Vector-Field Consistency policy the amount of inconsistency increases exponentially as the distance to the player becomes larger. For example, the chunk the player is standing on has, for example, a staleness bound value of 10 milliseconds, the ring of chunks around the center chunk has a staleness bound of 100 milliseconds, and so on untill we reach the ring of chunks that is at the specified distance away from the player. If we only rely on the staleness bound, we risk collecting a number of messages in a dyconit, that will create a spike in bandwidth usage when sent to the player. To limit the amount of messages, the policy specifies an numerical error bound that triggers the dyconit to send the messages to the player when it reaches a certain number of messages in its message queue. Just as with the Donnybrook policy, PorygonCraft updates the consistency bounds of the dyconits based on the location of the player, which means

that if you move one chunk in an arbitrary direction, the consistency bounds for the chunks around the player are recomputed.

5 Experimental setup and results

In this Section, we discuss the setup for the experiments we use to evaluate the effects of PorygonCraft on a Minecraft-like game. In Section 5.1 we will discuss the environment that we used to conduct real-world experiments in. In Section 5.2 we will discuss the data collection and the metrics that we use. Finally, in Section 5.3 we will present the results.

This thesis evaluates PorygonCraft. To evaluate the performance of PorygonCraft we will run several experiments that measure the network and CPU usage. Those experiments are performed with different sized workloads operated by Yardstick. Every experiment will be repeated twenty times, and each experiment will measure the network and CPU usage for one minute.

5.1 Environment

The performance of a Minecraft-like game server is tested by performing experiments in a real-world setting. We use the DAS-5 because it is the most representative hardware we have access to, because we expect games to run on similar systems.

The DAS-5 is a system designed by the Advanced School for Computing and Imaging and funded by the NWO/NCF. The DAS-5 offers compute nodes to students and researchers to conduct controlled experiments in fields such as, but not limited to, parallel and distributed computing. Each of these compute nodes are equipped with a dual 8-core CPU and is connected to both an Ethernet and InfiniBand network. Additionally, some of the nodes are equipped with special hardware such as HPC accelerators [25].

5.2 Data collection and metrics

The data is collected by tracking the system metrics of the DAS-5 by making use of the system monitoring library psutil. Psutil allows retrieving information on running processes and system utilization such as CPU statistics and network counters.

The number of CPU cores utilized is a metric to quantify the CPU usage. The numerical value that we measure represents the number of cores that are currently in use. The number of packets sent over the network and the number of bytes sent over the network are metrics used to quantify the throughput of the server. Throughput is a valuable way to measure the bandwidth usage and we will measure the number of packets sent per second, as well the number of bytes that are sent per second to determine the performance improvement of Glowstone when PorygonCraft is implemented. Lastly, we look at the average size of a packet in bytes, because we expect them to be larger. We expect them to be larger, because we send messages less frequently compared to not using PorygonCraft.

5.3 Workload

The workload used in the experiments are created by Yardstick. For each size workload Yardstick deploys bots in the same area, all with identical behavior. The behavior of the bots consists of walking around the world. Yardstick is not capable of deploying

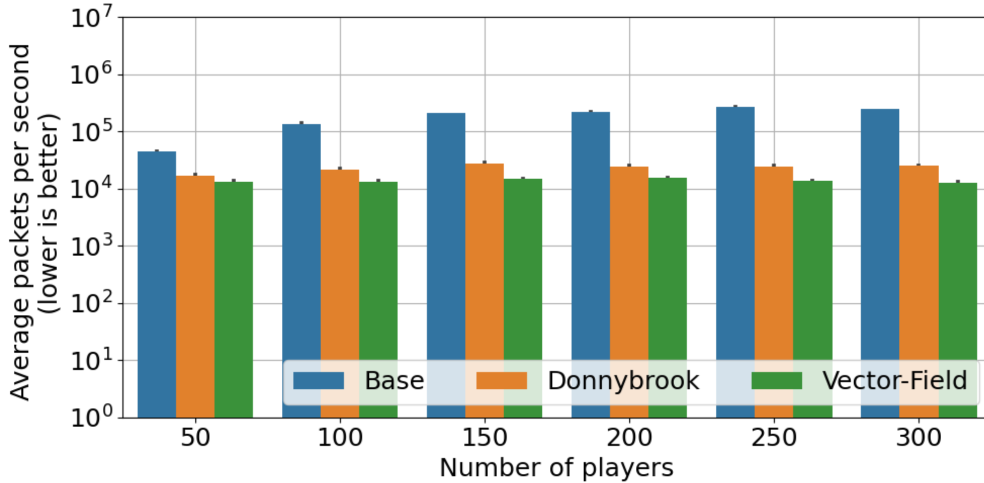

Figure 7: Average outgoing packets per second with a fixed players workload.

all the bots at once, therefore, the bots have had some time to randomly walk around before the experiment starts. At the end of the experiment, the bots that form the workload are disconnected from the server at once.

5.4 Experimental results

This Section presents the results of the network throughput, bandwidth measurements and CPU utilization. We first present the number of packets and amount of bytes that are sent over the network per second and then look at the CPU utilization and how the different policies affect this. In all the experiments, we observe that the implementation of PorygonCraft has a significantly positive impact on the bandwidth usage and CPU usage.

Each bar in the following figures is determined by repeating the corresponding experiment twenty times. One iteration of an experiment consists of deploying a fixed amount of players in Glowstone with the help of Yardstick. We sample the system metrics for 60 seconds with a one-second interval. Finally, we average the data, by taking the arithmetic mean, to obtain the average amount of the measured metric per second.

Network throughput and bandwidth

We present both the results of the experiments that measure the packet forwarding rate and the experiments that measure the bandwidth.

Figure 7 and Figure 8 consists of the measurements of the average packets per second and average data transfer in bytes per second for two policies and a Minecraft-like game without PorygonCraft respectively. The vertical axis shows the average packets/bytes per second exponentially in powers of ten. The horizontal axis shows the number of players, starting from 50 players to 300 players in steps of 50. Every bar is associated with Base (implementation without PorygonCraft), Donnybrook (implementation with PorygonCraft and Donnybrook policy), or Vector-Field (implementation with PorygonCraft and Vector-Field Consistency policy). The error bars indicate that there is little variability between the executed experiments.

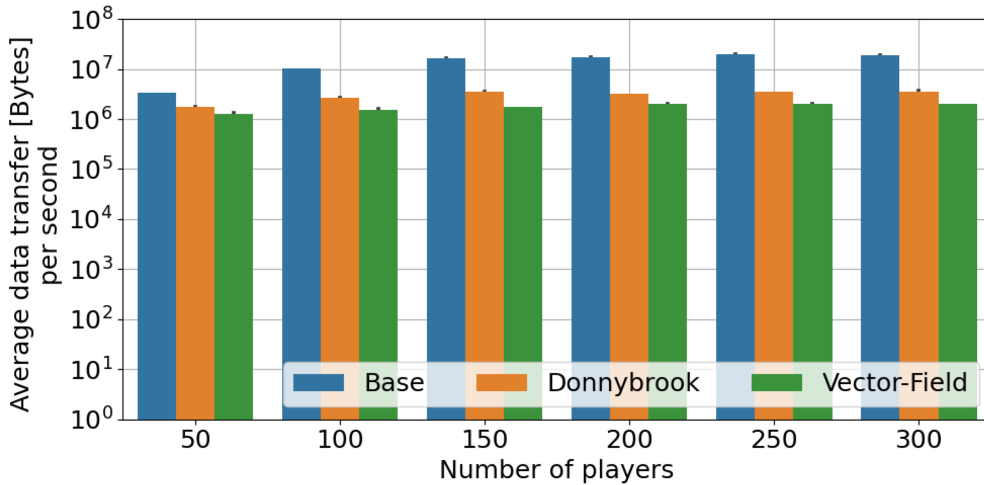

Figure 8: Average outgoing bytes per second with a fixed players workload.

Figure 7 shows the number of outgoing packets of the Minecraft-like game Glowstone per second for a fixed player workload. In Figure 7, we observe a difference between the throughput with fixed workloads of Glowstone without PorygonCraft (Base) and Glowstone with PorygonCraft (DonnyBrook and Vector-Field), especially for the larger workloads where we can identify around an order of magnitude improvement in bandwidth usage (lower is better). Additionally, we identify that PorygonCraft with the Vector-Field policy performs better than PorygonCraft with the Donnybrook policy. The Vector-Field policy performs better because dyconits associated with chunks far away from players have generally higher bounds, than that would be the case in the Donnybrook policy, because we send messages less often, we also send less packets or data.

In Figure 8, we measure the average amount of outgoing bytes per second for a fixed player workload. We notice that we encounter the same behavior as seen in Figure 7. Again, when the workload increases, we identify that we are getting approximately an order of magnitude improvement, where we deem fewer bytes sent per second as better. This improvement is explained by the behavior of the bots, because the bots are randomly moving, more bots will then automatically mean a larger spread of bots. When there is a larger spread, then there is also a larger distance between certain bots and they may not be interested in each other anymore, getting their messages very inconsistent (although still bounded).

Both the results of the experiments that measure package forwarding rate and bandwidth show that Glowstone with PorygonCraft delivers a considerable improvement over Glowstone without PorygonCraft. Additionally, we can conclude that on average an implementation with PorygonCraft results in less traffic over the network. However, the average size of a packet is larger in comparison to a Minecraft-like game without PorygonCraft, as seen in Figure 9.

Figure 9 shows the average size of a packet for the different implementations on the vertical axes in steps of 25 bytes. It is expected that for Donnybrook and Vector-Field the packets are larger than the packets for Base, because we send messages to the clients

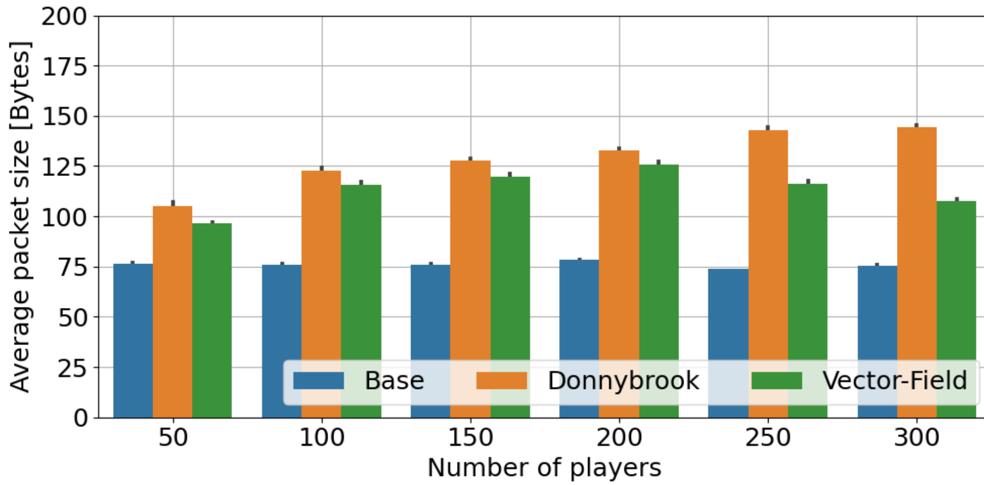

Figure 9: Average bytes per packet per second for fixed player workloads

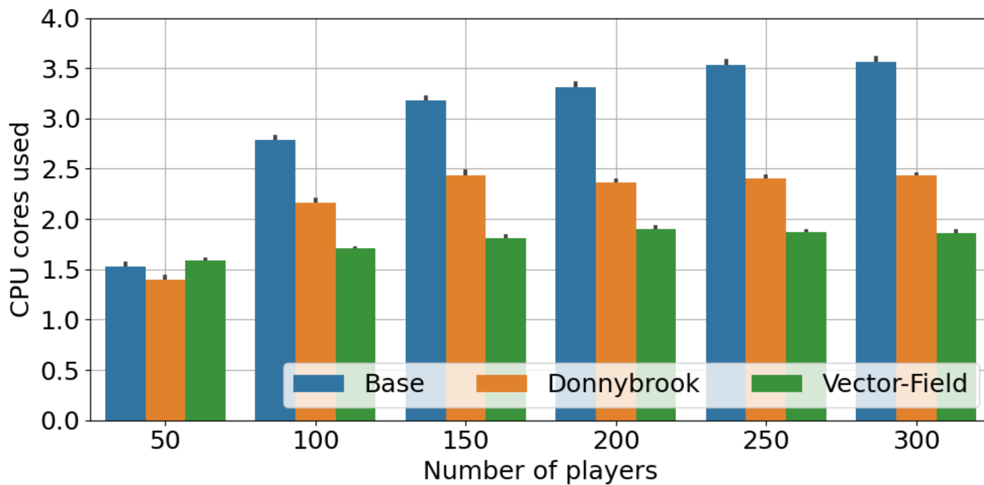

Figure 10: Average amount of CPU cores used per second.

less often, thus we send more messages at the same time. Again, we observe that Vector-Field performs better than Donnybrook, which is explained by having lower bounds in chunks around the player than that would be the case for Donnybrook. Although, the packets that are associated with an implementation with PorygonCraft are larger. The average amount of bytes send over the network is still lower, as we observed in Figure 8.

CPU utilization

In Figure 10, we present the average number of CPU cores used. The vertical axis shows the number of CPU cores used in steps of 0.5 cores. Here we see that the Minecraft-like game without PorygonCraft uses almost double the number of CPU cores in comparison with the Minecraft-like game with PorygonCraft, especially if we look at the 'Vector-Field' policy. The difference in CPU core is explained by the amount of I/O operations we have to perform. We have seen that implementations with PorygonCraft

send less packets on average. Consequently, the CPU also performs less I/O operations and has less interrupts. Because there are less interrupts the CPU is utilized more efficiently.

6 Discussion

Our results are promising and give a positive outlook on the capabilities of PorygonCraft. However, it is important that we discuss the limitations of our experiments and discuss the potential threats they introduce. First, we are not measuring the inconsistency in the game. Therefore, it is not possible to make a an assertion about the playability of the game by any form of quantitative metric, we can only make an assertion on our own experience when playing the game, while using PorygonCraft.

Secondly, Glowstone appears to become unstable if too many players are simultaneously on the server. To fully explore the performance improvements of PorygonCraft, we would need to test it on larger workloads, but in the current situation, we are not able to confidently perform experiments with workloads larger than 300 players.

Lastly, the bots that are provided by Yardstick do not act like real players. This could potentially affect our experiments in such a way that we are performing real-world experiments, but they may not be representative of real workloads. The current workload creates favorable conditions for the game. In spite of this limitation, we find that the current behavior represents a real-world workload good enough to show PorygonCrafts efficacy.

7 Conclusion

Minecraft-like games are growing and need to become more scalable to support their large numbers of players. Improving the scalability of Minecraft-like games is especially challenging, because they are real-time interactive distributed systems, which require low-latency, consistency, and real-time execution. We have designed PorygonCraft by carefully identifying the requirements that PorygonCraft needs and implemented them in Glowstone. Policies are implemented to enable game operators to make trade-offs between consistency and scalability; these policies were based on related work. We identified that a good first step is to have policies that are based on solutions of which we already have seen positive results. Based on real-world experiments we identify that PorygonCraft results in an order of magnitude improvement on network throughput and bandwidth, for more than 100 players. However, we also identify that the average size of a packet becomes larger when using PorygonCraft in comparison to not using PorygonCraft. Lastly, PorygonCraft reduces the CPU usage between 40% and 50%. A Minecraft-like game using PorygonCraft lowers the data send over the network and needs fewer CPU resources in comparison to a Minecraft-like game without PorygonCraft for growing player workloads.

8 Future work

Several items have not been touched upon in this thesis. One of them is the usage of policies based on the numerical error bound instead of the staleness bound. Currently, the policies affect the staleness bound of the dyconits, however, we would like to see the

effects on performance when we introduce policies that affect the numerical error bound or both the staleness -and numerical bound. Additionally, we want to observe the effect of more complex policies on the performance because the current policies are only based on the distance between players and entities.

The second item that we want to address is associated with the experiments. The current player workloads are bots that walk around in the Minecraft-like world. However, these bots do not portray real-world player behavior. Because Minecraft-like games invoke so many other activities, we aim to perform experiments with near representative workloads.

The third and last point addresses PorygonCraft because we would like to be able to perform experiments with the following two metrics: We want to measure the overhead it creates and the size of the overhead as PorygonCraft is utilized for very large player workloads. Additionally, we want PorygonCraft to be able to measure the amount of inconsistency that is currently in the Minecraft-like game. This could for example be done by taking the sum of the numerical error and/or staleness of all the dyconits at runtime.

References

[1] ESA, "ESA Essential Facts 2020," 2020. [Online]. Available: https://bit.ly/32pg3PM

[2] Superdata, "Games and Interactive media earned a record 120.1B dollar in 2019," 2019. [Online]. Available: https://bit.ly/2WkzbdM

[3] Newzoo, "Newzoo Global Games Market Report," 2019. [Online]. Available: https://bit.ly/GamesMarketReport2019

[4] K. Webb, "Microsoft, Sony, and Google are investing in subscriptions and streaming to give gamers more choice, but the real challenge is changing the way people play," 2019. [Online]. Available: https://bit.ly/301PahS

[5] BusinessInsider, "The $120 billion gaming industry is going through more change than it ever has before, and everyone is trying to cash in," 2019. [Online]. Available: https://bit.ly/3j1DTqJ

[6] C. Bailey, E. Pearson, V. Gkatzidou, and S. Green, "Using video games to develop social, collaborative and communication skills," 2006.

[7] A. Lobel, I. Granic, and R. C. M. E. Engels, "Stressful gaming, interoceptive awareness, and emotion regulation tendencies: A novel approach," *Cyberpsychology Behav. Soc. Netw.*, vol. 17, no. 4, pp. 222–227, 2014.

[8] J. Donkervliet, A. Trivedi, and A. Iosup, "Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems." HotCloud, 2020.

[9] H. Chiang, "Minecraft: Connecting more players than ever before." 2020. [Online]. Available: https://bit.ly/2F4cpkF

[10] Gamepedia, "Tutorial: Redstone Computers," 2020. [Online]. Available: https://bit.ly/2CBuCol

[11] Mojang, "Math subject kit," 2020. [Online]. Available: https://bit.ly/2WhJmQw

[12] J. van der Sar, J. Donkervliet, and A. Iosup, "Yardstick: A benchmark for minecraft-like services," in *ICPE*, 2019, pp. 243–253.

[13] J. Gray, "What next?: A dozen information-technology research goals," *ACM*, vol. 50, no. 1, pp. 41–57, 2003.

[14] A. R. Bharambe, J. Pang, and S. Seshan, "Colyseus: A distributed architecture for online multiplayer games," in *NSDI*, 2006.

[15] A. R. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang, "Donnybrook: enabling large-scale, high-speed, peer-to-peer games," in *Proceedings of the 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*, 2008, pp. 389–400.

[16] N. Santos, L. Veiga, and P. Ferreira, "Vector-field consistency for ad-hoc gaming," in *Middleware 2007, ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, USA, November 26-30, 2007, Proceedings*, vol. 4834, 2007, pp. 80–100.

[17] R. Diaconu and J. Keller, “Kiwano: Scaling virtual worlds,” in *Winter Simulation Conference, WSC 2016, Washington, DC, USA, December 11-14, 2016*, 2016, pp. 1836–1847.

[18] M. Valero, R. Diaconu, and J. Keller, “Manycraft: Massively distributed minecraft,” in *Annual Workshop on Network and Systems Support for Games, NetGames '13, Denver, CO, USA, December 9-10, 2013*, 2013, pp. 17:1–17:3.

[19] P. Moll, S. Theuermann, H. Hellwagner, and J. Burke, “Distributing the game state of online games: Towards an NDN version of minecraft,” in *17th IEEE International Conference on Communications Workshops, ICC Workshops 2019, Shanghai, China, May 20-24, 2019*, 2019, pp. 1–6.

[20] D. Pritchett, “BASE: an acid alternative,” *ACM Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[21] B. Medjahed, M. Ouzzani, and A. K. Elmagarmid, “Generalization of ACID properties,” in *Encyclopedia of Database Systems*, 2009, pp. 1221–1222.

[22] M. van Steen and A. S. Tanenbaum, *Distributed Systems, third edition.*

[23] H. Yu and A. Vahdat, “Design and evaluation of a conit-based continuous consistency model for replicated services,” *Trans. Comput. Syst.*, vol. 20, no. 3, pp. 239–282, 2002.

[24] Glowstone, “A fast, customizable and compatible open source Minecraft server.” [Online]. Available: https://bit.ly/2B5d2sB

[25] H. E. Bal, D. H. J. Epema, C. de Laat, R. van Nieuwpoort, J. W. Romein, F. J. Seinstra, C. Snoek, and H. A. G. Wijshoff, “A medium-scale distributed system for computer science research: Infrastructure for the long term,” *IEEE Computer*, vol. 49, no. 5, pp. 54–63, 2016.