Design and Experimental Evaluation of a System based on Dynamic Conits for Scaling Minecraft-like Environments

Jesse John Robert Donkervliet

TU Delft

Design and Experimental Evaluation of a System based on Dynamic Conits for Scaling Minecraft-like Environments

by

Jesse John Robert Donkervliet

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday January 15, 2018 at 2:00 PM.

Project duration:	January 16, 2017 – January 15, 2018	
Thesis committee:	Dr. ir. A. Iosup,	Delft University of Technology, supervisor
	Dr. ir. E. Epema,	Delft University of Technology, committee chair
	Dr. ir. P. Pawełczak,	Delft University of Technology

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Minecraft is one of the most popular games of all time, and provides both entertainment and education to people around the globe. One of the main challenges in gaming is increasing the scalability of these games to support the large numbers of players they attract. Minecraft-like games make this challenge even more difficult by introducing the novel feature of a modifiable world. The world that players explore can be fully modified, allowing players to build anything from a recreation of Manhattan to a functioning digital computer. These modifications increase the need for communication between players, limiting scalability. We design and evaluate Meerkat, a prototype system that uses Dynamic Conits to reduce this communication and increase the scalability of Minecraft-like games.

Preface

The master thesis was by far the most difficult part of my education. I am very happy with completing this challenge and the things I have learned along the way. I think that I have become a better computer scientist, engineer, and person in the process. During the thesis, I spent hours frowning at my computer, shouting at my computer, or both at the same time, and I remember clearly wanting to quit on multiple occasions. Thanks to the help and support from many people, this did not happen. I am very grateful to them, and they deserve a special thanks.

First, I would like to thank my supervisor, Alexandru Iosup, for pushing me to deliver high-quality work and not letting me get away with a job half-done. Because of you I am better at *getting things done*, while also setting higher standards for myself and living up to these standards.

Second, I would like to thank Jerom van der Sar for sharing his expertise on Minecraft and his dedicated work on Yardstick. It is a pleasure working with you, and I hope I can keep doing so in the future. If you keep up the good work, I am sure you will *get a lot of things done*!

Third, I would like to thank my friends, Tim, Stefan, Otto, Pim, Other Pim, Vincent, and Laurens for both helping me with my thesis and distracting me from it. Tim, thank you being my rubber duck during programming, and for being an infinite source of helpful comments about both the technical and conceptual parts of this thesis. Stefan, thank you for your helpful feedback on my writing and for helping me get organized, plan tasks, and set deadlines; thank you for being so generous with your time. Otto, thank you for listening to, and asking questions about, my ramblings about consistency models and my complex solutions to (seemingly) simple problems.

Finally, I want to thank my family for their constant support, and inspiring me to become a teacher. I want to thank a number of family members specifically. Thank you, Grandma and Grandpa, for demonstrating a strong work ethic since before I was born. Thank you, Grandpa, for introducing me to the world of computers and technology at a young age. Thank you, Dad, for introducing me to the world of video games and encouraging my interest in computers. Thank you, Mom, for always being there for me and taking care of me, especially when I forget to take care of myself. Thank you, Elvan, for all your love and support. You get me back on my feet when I am down; you inspire me to work hard and never give up. I could not have done this without you. Thank you for everything. I love you.

Jesse John Robert Donkervliet
Delft, January 2018

Contents

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
1.1 What is Minecraft? 2
1.2 Problem statement and main research questions. 2
1.3 Research-oriented approach. 4
1.4 Main contributions 4
1.5 Structure of this thesis 5

2 Background **7**
2.1 Minecraft-related research 7
2.2 A brief introduction to consistency 8
2.3 The Conit consistency model 10
2.3.1 Important features 10
2.3.2 Conits and consistency dimensions 11
2.4 Similar consistency models 14
2.4.1 Consistency model research. 14
2.4.2 Games and distributed systems research 15

3 Dynamic Conit model for Minecraft-like games **19**
3.1 Dynamic Conit model requirements 19
3.2 Conit model extensions 20
3.2.1 Dynamically changing Conit bounds. 20
3.2.2 Optimistic consistency 21
3.2.3 Conit update messages 21
3.2.4 Multi-hop Conits 21
3.2.5 Speculation error 23
3.3 Application to Minecraft-like games 23
3.3.1 Guaranteed consistency between servers 23
3.3.2 Guaranteed consistency between clients and servers 24
3.4 Dynamic Conit performance model 25
3.4.1 Synchronization controlled by staleness bound. 25
3.4.2 Synchronization controlled by numerical error bound 26
3.4.3 Synchronization controlled by order error bound 26

4 Meerkat: design of a Dynamic Conit system **27**
4.1 Design overview 27
4.2 Consistency bounding 28
4.2.1 Bounding staleness 29
4.2.2 Bounding numerical error 30
4.2.3 Bounding order error 30
4.3 Dynamic Conit mechanisms 31
4.3.1 Dynamic bound policies 31
4.3.2 Pipeline for strict and optimistic consistency 32
4.3.3 Dynamic Conit information message 33
4.4 Meerkat performance model 34

5 Experimental setup **35**
5.1 Experiments overview . . . 35
5.2 Yardstick: design of a Minecraft-like game benchmarking tool . . . 35
5.2.1 System requirements . . . 36
5.2.2 Design overview . . . 37
5.2.3 Player behavior model . . . 38
5.2.4 Player emulation . . . 38
5.2.5 Yardstick collector . . . 39
5.2.6 Data publishing . . . 39
5.3 Experiment workloads . . . 39
5.3.1 Yardstick experiment workloads . . . 39
5.3.2 Meerkat experiment workloads . . . 40
5.4 Environment . . . 41
5.4.1 DAS-5 distributed supercomputer . . . 41
5.4.2 Akka framework . . . 42
5.5 Metrics and data collection . . . 42
5.5.1 Main metrics . . . 42
5.5.2 Relative utilization metric (Jerom van der Sar) . . . 43
5.5.3 Akka Logging . . . 44
5.5.4 Prometheus monitoring . . . 44

6 Experimental results **45**
6.1 Minecraft scalability experiments . . . 45
6.1.1 Main findings . . . 45
6.1.2 Analysis of finding 1 . . . 45
6.1.3 Analysis of finding 2 . . . 47
6.1.4 Analysis of finding 3 . . . 48
6.1.5 Discussion . . . 49
6.2 Meerkat experimental evaluation . . . 50
6.2.1 Main findings . . . 50
6.2.2 Analysis of finding 1 . . . 51
6.2.3 Analysis of finding 2 . . . 52
6.2.4 Analysis of finding 3 . . . 53
6.2.5 Analysis of finding 4 . . . 56
6.2.6 Analysis of finding 5 . . . 56
6.2.7 Discussion . . . 58

7 Conclusion and future work **59**
7.1 Main contributions . . . 60
7.2 Future work . . . 61

Bibliography **63**

Appendices **69**

A Minecraft-like game configurations **71**
A.1 Vanilla configurations . . . 71
A.1.1 server.properties . . . 71
A.2 Spigot configurations . . . 72
A.2.1 server.properties . . . 72
A.2.2 bukkit.yml . . . 72
A.2.3 spigot.yml . . . 73
A.3 Glowstone configurations . . . 75
A.3.1 glowstone.yml . . . 75
A.3.2 worlds.yml . . . 77

B Experimental results **81**
B.1 Minecraft scalability experiments . . . 81
B.2 Meerkat experiments . . . 83

List of Figures

1.1 Education in Minecraft . . . 3

2.1 Conit staleness bounding example . . . 12
2.2 Conit numerical error bounding example . . . 13
2.3 Conit order error bounding example . . . 14

3.1 Difference between Conits and Dynamic Conits . . . 22
3.2 Dynamic Conits applied to a game server cluster . . . 24
3.3 Dynamic Conits applied to a server and its clients . . . 24

4.1 Meerkat system design . . . 28
4.2 Meerkat staleness bounding mechanism . . . 29
4.3 Meerkat numerical error bounding mechanism . . . 30
4.4 Meerkat order error bounding mechanism. . . . 31
4.5 Meerkat processing pipeline . . . 32
4.6 Meerkat 2-way handshake for adding nodes or Dynamic Conits . . . 33

5.1 Yardstick system design . . . 37
5.2 Conceptual overview of a Minecraft server tick . . . 43

6.1 Minecraft tick frequency over number of players . . . 46
6.2 Minecraft relative utilization over number of players . . . 46
6.3 Minecraft CPU utilization over number of players . . . 47
6.4 Minecraft outgoing packet throughput over number of players . . . 48
6.5 Effect of consistency bounds on throughput . . . 51
6.6 Effect of consistency bounds on synchronization . . . 53
6.7 Meerkat throughput with fixed numerical error bound under **increasing** workload . . . 54
6.8 Meerkat throughput with dynamic numerical error bound under **increasing** workload . . 54
6.9 Meerkat throughput with fixed numerical error bound under **50-player trace** workload . 55
6.10 Meerkat throughput with dynamic numerical error bound under **50-player trace** workload 55
6.11 Effect of wait mechanism on throughput under **stress-test** workload on 4 nodes . . . 56
6.12 Effect of wait mechanism on throughput under **stress-test** workload on 16 nodes . . . 56
6.13 Meerkat throughput using **ADMI** policy under **50-player trace** workload . . . 57
6.14 Meerkat throughput using **PM-P** policy under **50-player trace** workload . . . 57

B.1 Additional experimental result: Minecraft incoming bytes throughput . . . 81
B.2 Additional experimental result: Minecraft outgoing bytes throughput . . . 81
B.3 Additional experimental result: Minecraft incoming packets throughput . . . 82
B.4 Additional experimental result: Minecraft memory usage . . . 82
B.5 Additional result from experiment 6.2.2. . . . 83
B.6 Additional result from experiment 6.2.3 . . . 83
B.7 Additional experimental result: using static bound of 0 on **increasing** workload . . . 84
B.8 Additional experimental result: using **ADMI** policy on **increasing** workload . . . 84
B.9 Additional experimental result: using **PM-P** policy on **increasing** workload . . . 84
B.10 Additional results from experiment 6.2.4. . . . 85
B.11 Additional results from experiment 6.2.5 . . . 85

List of Tables

1.1 Reader guide 5

2.1 Comparison of the Conit consistency model with similar models. 17

3.1 Summary of Dynamic Conit model– and system requirements 20

5.1 Summary of experiments 36
5.2 Summary of Yardstick system requirements. 36
5.3 Yardstick experiment workloads 39
5.4 Meerkat experiment workloads 40
5.5 DAS-5 node hardware configurations 41
5.6 Experiment metrics 42

6.1 Network packet distribution from Minecraft scalability experiment 49

1 Introduction

The gaming industry is relatively young but takes the world by storm, generating more than 20 billion dollars in revenue in 2016 in the United States alone [23]. To appreciate the scale of this industry consider that the total box office revenue[1] in 2016 in the United States was less than 12 billion dollars [46]. This large revenue is only possible because of the large number of people that buy and play video games. World of Warcraft, the largest *Massively Multiplayer Online Role Playing Game*, peaked at 12 million subscriptions [31], and Fantasy Westward Journey, an Massively Multiplayer Online Role Playing Game (MMORPG) popular in China, hit more than one million daily players [47].

A clear stereotype exists about the people who play games: young males that do little else besides playing games and barely see the light of day. However, this stereotype is nothing more than that. The average gamer is actually 35 years old, and the number of female gamers age 18 and older represent a greater portion of the gamer population than males age 18 and younger [23].

MMORPGs are a type of *Role Playing Game* in which millions of players explore the same virtual world. In a Role Playing Game players enter an immersive virtual world using a digital representation of themselves called an *avatar*. The player controls this avatar and uses it to interact with diverse content, varying from logic puzzles to epic battles. In an MMORPG, players can also interact with the avatars of other players from around the world and explore the virtual world together.

There exist many other popular game genres besides (MMO)RPGs. The Role Playing Game (RPG) is one of the most popular game genres in 2016 in terms of sales. Other popular genres are *First Person Shooter*, *Sports*, and *Adventure* [23]. A First Person Shooter (or *FPS*) is a game in which players control an armed avatar with the goal to shoot enemies. In contrast to other shooters (for instance the popular arcade classic *Space Invaders*), First Person Shooters make the player look through the eyes of their avatar, making the game more immersive. Many popular sports games simulate real-world players, teams, and sports events. In these games players take control of an athlete or a team and try to win competitions against other players or computer controlled opponents. But not all sports games emphasize realism. For instance, *Kopanito All-Stars Soccer*[2] is a soccer game that focuses primarily on gameplay and features a cartoony art style instead of realistic looking real-world players and teams. Adventure games are similar to RPG games but have an emphasis on storytelling and world exploration instead of an emphasis on combat and improving the abilities of the avatar. An example of a popular adventure game series is *The Legend of Zelda*. In The Legend of Zelda, the player controls a character called Link, and is tasked with freeing princess Zelda from Ganon, who has taken her captive. The games usually feature a list of fixed missions that need to be completed to finish the game, but exploration is encouraged by rewarding the player with collectibles, rare items, and additional missions.

While games provide amusement for millions of people, games are also used for other important purposes. Games that are designed with a purpose other than entertainment are called *serious games*. Serious games can be used for societal goals such as training professionals [69] and treating trauma patients [29, 45] by simulating real-world scenarios, and educating people by presenting content in an immerse form [2, 24, 51, 53, 57].

[1] The revenue from selling tickets in movie theaters.
[2] `https://kopanitosoccer.com/`

The large population of gamers enables the growth of the game industry, but also poses challenges to game developers. Players need fresh content to stay engaged with a game, and do not want to replay content they have already seen, or content that has been seen by many other people [11]. Players also want to play together with friends, share experiences and meet new people. To meet these challenges, researchers look for novel techniques to create fresh content at a massive scale [32] and increase the performance of games and game platforms to scale games to massive numbers of players [33, 42, 62]. These challenges are both active topics of research. This thesis focuses on the challenge of game scalability, specifically for Minecraft-like games.

1.1. What is Minecraft?

Minecraft is a game that can be used both for amusement and as a serious game. It can do this because of a novel feature that allows players to modify the entire virtual world. In this thesis we refer to this feature as a *modifiable world*. Minecraft discretizes the virtual world into fixed-length columns called *chunks*. These chunks are themselves discretized into *blocks*. To create the virtual world, the game generates landscapes and towns using these blocks. In Minecraft, players can remove, place, and replace any block in the virtual world.[3] This creates a large sandbox environment in which players can be creative and create new entertaining or educational content. In this thesis we refer to games that feature a modifiable, discretized world as *Minecraft-like* games.

Because the world is modifiable, players are free to create *their own content* and share this content with others. This is not just a hypothetical: a large Minecraft modding-community has evolved online[4], modifying the game even further and sharing custom-built worlds and games built within Minecraft itself. Additionally, Microsoft acquired Mojang, the developer of Minecraft, for 2.5 billion dollars in 2014 [1] and has released an *education edition* of Minecraft that is used in primary schools to teach a variety of subjects such as history, anatomy, digital logic, and economics using custom Minecraft worlds [2]. A remake of *The Oregon Trail*, one of the most successful educational games of all time, is now available as a Minecraft world. Figure 1.1 shows examples of educational material in Minecraft.

The large community and the acquisition by Microsoft is evidence of the popularity and potential of Minecraft, as both a creative and educational tool. Providing education in a digital format is important because it can mean getting higher quality material to a larger audience for a lower cost. Furthermore, Minecraft can make educational material much more engaging for students than text-books and video, because it is an interactive game. As such, Minecraft facilitates an important role in society.

Unfortunately, the large scale of the industry does not match the scalability of Minecraft. It is difficult to scale games to millions of players (which are the number of players in popular MMORPGs), and Minecraft's modifiable world poses additional novel scalability challenges. This thesis focuses on increasing the scalability of Minecraft-like games using novel consistency techniques.

1.2. Problem statement and main research questions

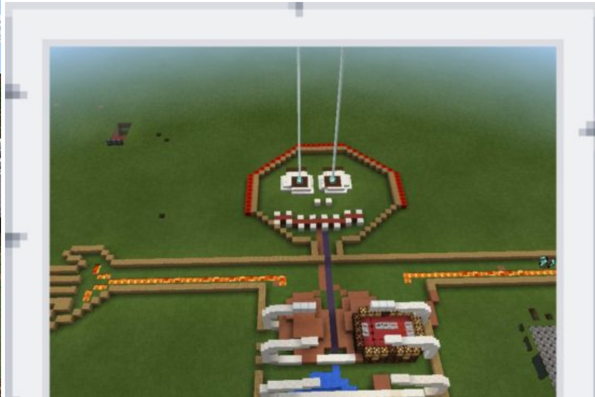
Our goal is to enable massive amounts of players to explore Minecraft's virtual environments to explore, create, and learn. In other words, we would like to transform Minecraft's virtual environment to a Massively Multiplayer Online Game (MMOG). An MMORPG, such as World of Warcraft or Fantasy Westward Journey, is a type of MMOG with a specific type of gameplay and content.

State-of-the-art commercial multi-player games use server-client architectures. Players run a game client on their local computer, while the servers are controlled by the game's developer. Because the game developer hosts the servers, it can improve player experience by placing servers geographically close to players (reducing latency) and running cheat-detection software on the servers. MMOGs can support millions of players, but a single node can support no more than a few thousand players; the virtual environment and its workload are too large for a single node. To solve this issue, MMOGs partition the virtual world and distribute these partitions over multiple servers, distributing the workload and partially parallelizing the system. Because the clients are distributed over the servers, the number of messages sent by each individual server to clients is reduced. To allow players connected to different servers to interact with each other, the game hides the partitioning from the users by sharing some relevant data between servers. Sharing this data introduces inconsistency between players that are

[3] Excluding the blocks that indicate the lower– and upper bound of the world.

[4] `https://bstats.org/global/bukkit`, `http://mcstats.org/`

(a) *The Oregon Trail* implemented in Minecraft.



(b) An anatomy lesson in Minecraft.

(c) A digital logic *AND gate* implemented in Minecraft.

(d) A lesson in Minecraft by the Council for Economic Education.

Figure 1.1: Screen-shots from multiple educational Minecraft worlds.

connected to different servers. This inconsistency can become visible to the players in the form of latency or out-of-order operations, which can reduce gameplay experience.

For a massivized version of a Minecraft-like game, these challenges cause more problems. By allowing players to modify the environment in addition to all the features of a regular MMOG, the same amount of players generates a heavier workload, which increases the need for partitioning and sharing data. In games, only dynamic data is communicated between players. Static content such as textures (e.g., the paint on a player's weapon or tool) and models (e.g., a building or vehicle) is stored locally on the player's device. Dynamic content, such as player behavior (e.g., movement, combat) and its consequences (e.g., finding an item, gaining experience points) has to be communicated between players. Minecraft features a virtual environment that is completely modifiable by players. For instance, players can decide to build a complete city, or dig a network of tunnels. This means that the complete environment is dynamic, and has to be communicated between players. We conjecture that this results in significantly more network traffic, forming a scalability bottleneck.

We formalize these problems in the form of four research questions.

RQ1 How to assess the scalability of Minecraft-like games?

It is important to quantify the scalability of the state-of-the-art before any claims of improvement can be made. However, currently no tools to measure the scalability of Minecraft exist.

RQ2 How to adapt the Conit consistency model to apply to Minecraft-like games?

The Conit consistency model is a model that allows bounded inconsistency between nodes in a distributed system. In this thesis we refer to this model as the Conit model, or simply as Conits. The Conit model is a general model that is aimed at general distributed systems and balances generality and practicality. Minecraft-like games are a very specific type of distributed system and have specific properties for which the Conit model was not designed. We conjecture that the Conit

model requires changes to be less general and more practical for its application to Minecraft-like games, but how the model should be changed is not obvious.

RQ3 How to design a system that improves the scalability of these games?

Designing a system that applies novel scalability techniques to Minecraft is challenging because no system currently exists that combines these elements.

RQ4 How to evaluate such a system experimentally?

To understand the proposed system, its performance must be quantified. However, no standardized tools exist to measure the scalability of computer systems.

1.3. Research-oriented approach

This thesis takes a four-step approach to answer the main research questions. This section describes the approach taken in this thesis to answer the research questions.

First, it is important to assess the current status of Minecraft's scalability to see where and how scalability improvements can be made. To this end we design and implement Yardstick, a distributed Minecraft benchmarking tool to assess the scalability and determine system bottlenecks. This is done in collaboration with Jerom van der Sar, who is a Minecraft expert and Bachelor Honors student. Yardstick is an ongoing research project and is currently used to perform a more in-depth study of Minecraft scalability.

Second, we propose the *Dynamic Conit* model: a consistency model based on the Conit consistency model, with additional mechanisms that makes it practical for Minecraft-like games. To show the flexibility of the Dynamic Conit model, we propose two applications of Dynamic Conits to Minecraft-like games.

Third, we design and implement *Meerkat*, a prototype distributed system that implements the Dynamic Conit model. The Dynamic Conit model specifies a number of mechanisms, but leaves selecting the right algorithms and system design up to the programmer. As such, Meerkat is one possible implementation of the Dynamic Conit model.

Fourth, we evaluate our approach by performing real-world experiments on Meerkat using a combination of synthetic and trace-based workloads. The experiments focus on the increase in scalability when allowing bounded inconsistency between nodes, and evaluating the effects of changing consistency bounds at runtime. We perform our experiments on the DAS-5, a distributed super computer for computer science research designed by the Advanced School for Computing and Imaging [10].

1.4. Main contributions

The main contributions of this thesis follow the main research questions presented above:

1. Evaluation of the scalability of Minecraft-like games using Yardstick, the first distributed large-scale benchmark of Minecraft-like games. (RQ1)
2. Design of a new consistency model called *Dynamic Conits*, which is based on the Conit consistency model and can be applied to Minecraft-like games. This is the first attempt modify the Conit consistency model such that it applies to games. (RQ2)
3. Design of a Meerkat, a prototype system that implements the Dynamic Conit model. This is the first system that implements Dynamic Conits, a consistency model that is designed to improve the scalability of Minecraft-like games. (RQ3)
4. Evaluation of Meerkat using real-world experiments and trace-based workloads. This is the first evaluation of scalability techniques for Minecraft-like games using the Conit or Dynamic Conit model. (RQ4)

We are currently writing an article based on the main findings of this thesis. This article will be submitted in Q1 of 2018. This thesis is part of the Opencraft research project[5], which is an ongoing research project from the @large research team, aimed at massivizing Minecraft-like games. Yardstick and Meerkat, systems introduced later in this thesis, are developed as part of this project.

[5] `https://atlarge-research.com/opencraft`

1.5. Structure of this thesis

This chapter introduces Minecraft, the challenges this thesis addresses, the formal research questions derived from these challenges, and the approach taken to answer these research questions. The next chapter gives background information on existing research in both game scalability and consistency. The main content chapters are Chapter 3 through 6. Chapter 3 discusses the Dynamic Conit model and how it can be applied to games. Chapter 4 discusses the design of *Meerkat*, a system that implements the Dynamic Conit model. Chapter 5 discusses the experimental setup and the design of *Yardstick*, the benchmarking application for Minecraft-like games. Chapter 6 discusses the results for both the experimental evaluation of Minecraft scalability as well as the experimental evaluation of Meerkat.

If you are interested in…,	you should read Chapter…									
	1	2		3	4	5		6		7
		2.1	≥2.2			5.1	5.2	6.1	6.2	
all content	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
using consistency models to increase scalability in distributed systems	✓	✗	✓	✓	✓	✗	✓	✗	✓	✓
scalability of Minecraft-like games	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓

Table 1.1: A reader guide for readers with a specific interest.

2 Background

Minecraft is one of the most popular games of all time and offers the unique feature of a modifiable world. This promotes creativity among players and makes Minecraft suitable for not only recreational but also educational purposes [7, 24, 43, 51, 57]. However, Minecraft was not built to scale with its popularity, limiting its usability such that the majority of servers only hosts tens of players simultaneously.

This thesis uses bounded inconsistency between nodes as a novel scalability technique for Minecraft-like games. To this end, this chapter discusses existing research on both Minecraft and consistency models in the following two sections. The third section of this chapter discusses the Conit consistency model, the model used and extended in this thesis. The last section discusses consistency models that are similar to the Conit model and discusses their positive and negative properties when applying the model to a Minecraft-like game.

2.1. Minecraft-related research

While Minecraft is one of the most popular games of all time, not much research exists on the scalability of the system and the behavior of its players. This section summarizes Minecraft-related research on these topics.

Alstad et al. analyze the performance of the Minecraft server using custom built player emulation using ProtocolLib[1], a library for the Minecraft network protocol [6]. The experiments confirm that Minecraft does not scale well with the number of players. However, the experiments do not represent a real-world scenario since a completely flat map is used, entities and weather effects are turned off, and the behavior of the bots is not based on any movement model.

HeapCraft is a project aimed at analyzing the behavior of players in Minecraft and improving collaboration [49]. Part of the HeapCraft project is a collection of plug-ins that collect data on player behavior and asks players which activity they are involved in, which is used to build a classifier to automatically detect player activities [48]. Because Minecraft features a modifiable world, player behavior in Minecraft likely differs from player behavior in regular games. Insight into player behavior in Minecraft is necessary to create movement models that can be used for realistic performance testing of Minecraft server implementations.

Kiwano is a distributed system for scaling virtual worlds [20]. Kiwano improves scalability by reducing network traffic using spatial partitioning. With spatial partitioning, geographically contiguous areas of the virtual world are partitioned into zones. Servers are then allocated to simulate each of these zones individually. Static spatial partitioning of the virtual world into zones is used in state-of-the-art commercial MMORPGs. Partitioning based on other properties, such as social connections between players, is an active research topic [67]. Kiwano partitions the virtual world into zones *dynamically*, based on the current location of players. This balances the number of players in each zone, balancing the total workload. This makes allocating resources easier, because the workload caused by the players in each zone is roughly equal. With Kiwano, instead of a server distributing updates to all its clients, players run a server locally. This server then forwards the updates from its only client to Kiwano. Kiwano decides which other clients should receive the updates based on the position of their

[1] `https://github.com/aadnk/ProtocolLib`

avatars in the virtual world; players only receive updates from other players that are geographically close. This approach has three drawbacks. First, Kiwano makes a binary decision about forwarding client updates to other clients. Network traffic could be reduced further by decreasing the frequency of updates received for clients that are in vision of, but not of interest to, the player. Second, Kiwano only works for updates related to moving entities, and does not support other events, such as the placement or removal of blocks in a virtual world. It is likely that such updates occur frequently in Minecraft, because a modifiable world is its distinguishing feature. Third, dynamically partitioning zones based on player locations can reduce the gameplay experience in areas that are densely populated, such as the main city areas in World of Warcraft. Because the avatar density of these areas is high, the dynamic partitioning makes the zones small to balance the workload. This can lead to unrealistically small view ranges for players, because they do not receive updates from players in non-neighboring zones.

Manycraft [21] is an architecture to scale Minecraft through the use of Kiwano. In Manycraft, clients locally run a *Manycraft node*, which contains a Minecraft server and a proxy to communicate with the underlying Kiwano network. Kiwano communicates updates from the client to other Manycraft nodes, making updates visible for other players. A drawback of this approach is that, because Kiwano only supports communicating entity locations, Manycraft does not support modifiable Minecraft environments. Additionally, the scalability improvements of Manycraft are only evaluated using coarse-grained system-level metrics such as memory usage. Although these metrics are related to scalability, experiments using application-level metrics are needed to evaluate the scalability as perceived by the players.

Koekepan [22] aims to provide a research platform for novel scalability techniques for virtual worlds. It is built on top of Minecraft using the default Minecraft client and a modified version of the Minecraft server. Koekepan allows Minecraft to use a cluster of servers by providing a proxy between the Minecraft client and the servers. Because the Minecraft client only communicates with the proxy, it is unaware of the distributed server setup, and the default Minecraft client can be used to connect to the system. Koekepan is designed to use dynamic spatial partitioning to balance the load over the server cluster. Koekepan is implemented as a proof-of-concept, and does not present any experimental results. It is unclear how much Koekepan's architecture has increased scalability compared to the vanilla implementation of Minecraft.

Spatial partitioning, used in Kiwano, Manycraft, and Koekepan, aims to improve scalability by reducing the number of messages sent from the servers to the clients. This allows each server to spend more time simulating the virtual world and player actions. This thesis takes a similar approach, but takes a more formal route by using *bounded inconsistency* offered by the Conit model to reduce network traffic. To understand how the Conit model works, it is important to understand the concept of consistency in distributed systems. The next section gives a brief introduction to this topic.

2.2. A brief introduction to consistency

Consistency is a large field which has been researched for many years. Where early research focused on shared-memory architectures and multi-core processors, today's consistency research focuses on distributed systems that span the entire globe. This section aims to give some context to the material presented in this thesis by discussing some of the most well-known consistency models in the field of distributed systems.

To understand why there are so many existing consistency models, it is worth to briefly discuss the CAP theorem [26]. The CAP theorem describes an inherent trade-off in distributed systems between Consistency and Availability caused by the possibility of network Partitions. In a distributed system, multiple nodes work together towards a common goal. To do this, data often needs to be shared between these nodes. The World Wide Web is an example of a distributed network. In this example, the data shared between the nodes are web pages. Consistency is defined as the degree of agreement between the nodes on the value of the shared data. For instance, if a web browser caches a web page from a server, the data is consistent across these machines. However, when the server updates the web page, inconsistency is introduced in the system: the server and the web browser have *different versions* of the same data. Availability is defined as the ability of users or clients to communicate with the distributed system and use its features. The World Wide Web is an example of a *highly available* system: a system designed to be always available to users, even if accepting user input means that inconsistencies are introduced in the system. An alternative type of system, such as ACID databases,

guarantee *strict consistency* to users: these systems are always consistent, even if this means temporarily preventing users from making changes to the data. In the event of a network partition, which means that a subset of the nodes in the distributed system cannot connect to the remaining nodes in the system, the CAP theorem states that all distributed systems have to choose between remaining available to users (allowing inconsistency to be introduced in the system), or to remain consistent (refusing users to make changes to the data). In this thesis, refusing users from making changes to the system is referred to as either *limiting availability* or *blocking*. Research on consistency in distributed systems has resulted in many different consistency models, which can be used to define, quantify and/or guarantee consistency in a distributed system. This section proceeds by describing a number of well-known consistency models.

Early consistency models focused on multi-processor and shared-memory systems, and aimed to be strict but computationally efficient [39]. An advantage of these models is that they can give strong guarantees about the consistency of the system. The rise of the World Wide Web, distributed computing, and later cloud computing, changed this trend to consistency models that are less strict, but tolerate node failures and message loss, and provide high availability and low latency to users. An advantage of these models is that they hide the performance penalty incurred when synchronizing data across large distributed systems. The most extreme version of relaxed consistency is *eventual consistency*, which gives *no* consistency guarantees except that if no more accesses are issued, the system will eventually converge to a consistent state. One of the most well-known distributed algorithms that guarantees eventual consistency in a distributed system and tolerates node failures and message loss is Paxos [37]. Although these are desirable properties in a distributed database, implementing Paxos in practice and integrating it in a distributed system is non-trivial [16]. Over the last two decades, in a move back to stricter consistency models, researchers have suggested multiple consistency models that still provide high availability and low latency, but also give *some* (probabilistic) consistency guarantees. Although some work had already been done on the topic of staying available in the case of network partitions [19, 25, 34, 63], these models were mostly built on top of existing consistency models that did not formulate their consistency guarantees from the perspective of the user. This section briefly describes a number of well-known consistency models from the distributed systems field, ordered from the strictest to the weakest model (unless readability is impaired, in which case the reversed order is explicitly indicated). This provides the reader with a context to understand the features of the Conit consistency model, which is discussed in the next section.

Linearizability is one of the strictest models that can be used in practice. Linearizability guarantees that the result of a set of accesses on a particular data item is equivalent to a sequential ordering of these accesses that satisfies two additional conditions [30]:

1. Accesses from one node appear in the same order as they where issued.
2. For any access a that is completed before an access b is started according to wall-clock time, a is ordered before b.

This matches with an intuitive meaning of a consistent system: any access to a data item is aware of all earlier accesses to that data item that are completed, no matter where these access occur in the system. This model does not offer high availability: a write must be communicated across the entire system before new accesses can be started.

Sequential consistency is similar to linearizability, but it does not require the ordering of the accesses to match the wall-clock time order, dropping the second ordering requirement from linearizability [36]. Dropping this requirement makes sequential consistency less strict, but it can also yield results that may be confusing for users. For instance, if data item x has an original value of 5, $w(x, y)$ writes value y to data item x, and $r(x) \rightarrow y$ reads data item x, which yields result y. Let A and B be nodes in a distributed system. Then the following execution is sequentially consistent:

$$\begin{aligned} A, w(x, 7) \\ A, r(x) \rightarrow 7 \\ B, r(x) \rightarrow 5 \end{aligned}$$

Furthermore, enforcing sequential consistency in distributed systems is costly and cannot be used in highly-available systems.

Causal consistency is a more relaxed consistency model that can be supported in highly available distributed applications [3]. Instead of a total order, only a partial order is guaranteed under causal consistency. The ordering used in causal consistency is an acyclic directed graph called the *happened-before* graph. For every two accesses a and b in this order where there is a path from a to b, one of the following rules must hold:

1. a is placed before b in the original program order (and come from the same node).
2. a reads a value that is written by b.
3. There is an access c in between a and b such that a causally precedes c, and c causally precedes b.

If there is no causal relation between two accesses, the causal consistency model sees these accesses as *concurrent*. No ordering is specified for concurrent accesses. Furthermore, if nodes do not (indirectly) communicate with each other, their accesses are concurrent to each other. Because no ordering is created for concurrent accesses, nodes that do not do not communicate with each other are allowed to diverge from each other, becoming increasingly inconsistent. The causal consistency model does not include mechanisms to prevent nodes from diverging because they do not communicate.

Real-time causal consistency (RTC) is a modification of causal consistency that adds an additional requirement to the ordering of accesses. It is therefore a stricter model than causal consistency. In RTC, an access a that happens after an access b in real time may not be ordered before b in the happens-before graph [44]. However, a and b can be ordered as *concurrent* accesses. In case a and b are both writes to the same data item, it is up to the application to resolve the conflict. Many systems that implement causal consistency actually implement RTC, because ordering a before b only happens in artificial circumstances (such as lowest-id writer wins). RTC is stricter than causal consistency, but not as strict as sequential consistency. However, RTC can still be guaranteed in highly available systems, in contrast to sequential consistency which cannot.

COPS is a distributed database that formally defines and uses the *causal+* consistency model. This consistency model is also an adaptation of causal consistency. In causal+ consistency, concurrent operations are resolved to prevent conflicts and diverging nodes [41]. causal+ consistency is a stricter model than causal consistency, but not as strict as RTC because it lacks the real-time requirement.

Eventual consistency is a *catch-all* term for consistency models that do not give any consistency guarantees but one: if no more accesses are issued, the system eventually converges to be fully consistent. Reducing the consistency guarantees also reduces the complexity of the design and implementation of large-scale distributed systems, which is one of the reasons why these systems have gained popularity. Eventually consistent systems are highly available, and can be consistent *most* of the time [9], but no guarantees can be given.

This section only gives a brief overview of some of the most well-known consistency models to create a context for the remainder of this thesis. For a more in-depth exploration of consistency models, the reader may consult one of the many surveys on consistency models and mechanisms [12, 27, 68].

2.3. The Conit consistency model

This section discusses the Conit consistency model [70]. This thesis uses the Dynamic Conit consistency model, an adapted version of the Conit model, to improve the scalability of Minecraft-like games. To understand the Dynamic Conit model, and how it is different from the original Conit model, it is important to first look into the Conit model.

This section is divided into three parts. First, it discusses five important features of the Conit model and motivates why these features are desirable in the context of Minecraft-like games. Second, it introduces the notion of a *Conit* and its *consistency dimensions*. The Conit consistency model uses these Conits to achieve the five important features. Finally, it discusses a number of limitations of the Conit model that limit its applicability to Minecraft-like games.

2.3.1. Important features

The Conit consistency model has five important features that differ from the models discussed in the previous section:

F1 The Conit model supports a spectrum of consistency bounds.

F2 The Conit model supports indicating the importance of individual updates.

F3 The Conit model supports defining real-time bounds on writes.

F4 The Conit model supports arbitrary consistency bounds on arbitrary data.

F5 The Conit model supports defining consistency bounds per node.

This section discusses the Conit consistency model using the five features listed above.

There exists a trade-off between consistency and synchronization. In general, if a stricter consistency model is enforced on a system, the nodes in that system require more synchronization. Synchronization between nodes takes time. Because this time could also be spent accepting new accesses from clients, the trade-off between consistency and synchronization can also be formulated as a trade-off between consistency and performance, where performance is defined as system throughput. For MMOGs throughput is an important metric: it is proportional to the number of entities that can be updated in a single tick. A consistency model that offers a spectrum of consistency settings could offer high throughput when needed while keeping consistency as high as possible.

Assigning importance to individual writes is useful because it can be used in combination with a player's *area of interest* [40]. For a single player, some updates may be more important than others. A trapdoor opening a hundred meters away is much less interesting (and allows lower consistency) than a trapdoor opening directly under the player's feet!

A notion of time in the consistency model is important because games are real-time systems. The time of an event is often just as important as the event itself. For instance, if a player triggers a trap or shoots an arrow, not only is it important that the trigger (pressing a button, releasing the arrow) is visible before the event (a trapdoor opens, the arrow flies off), it is also important that the event is visible *soon* after trigger.

The Conit consistency model can be implemented as part of a distributed system, located between the data and the network. The Conit model does not know the semantics of the data that is exchanged [70]. This allows flexibility in its application to Minecraft-like games: Conits can be defined for the area in which the player is currently located, for other players with whom the player interacts, for items in which the player in interested, etc.

The ability to set consistency bounds per node allows the system to tune consistency and throughput where needed. For instance, if a player opens an in-game menu which covers most of the screen, that player may tolerate much lower consistency than a player that is building a structure together with (and in close proximity of) friends.

2.3.2. Conits and consistency dimensions

In the Conit consistency model, consistency bounds are specified individually by each node by creating a so-called *Conit*. A Conit is a data structure that consist of a list of nodes that share the Conit, and three numerical values: an allowed inconsistency boundary for three distinct *consistency dimensions*. These dimensions are:

1. Staleness, which controls the staleness of data.
2. Numerical error, which controls the amount of unseen change.
3. Order error, which controls ordering guarantees.

These bounds can take any integer value in the domain $[0, \infty)$. The Conit consistency model allows a system to define an arbitrary number of such Conits.

Using Conits, systems are able to quantify and bound inconsistency between nodes. The Conit model allows consistency to vary between the extremes of linearizability and eventual consistency. Every access (read or write) indicates if it affects or depends on one or multiple Conits. Inconsistency is quantified over updates; every write changes a value by a certain amount. For this reason, writes are also called *updates* in the remainder of this thesis. The Conit consistency model guarantees that the inconsistency of the data shown to the user never exceeds the configured value. If Conit consistency bounds are exceeded (determined by the updates that affect the Conit), synchronization between nodes is required before the access can be completed. Synchronization is a blocking mechanism that guarantees the system consistency stays within bounds from the perspective of the user. Updates are

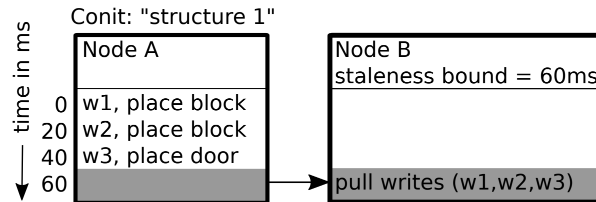


Figure 2.1: An example of a node B bounding staleness to 60ms by sharing a Conit called "structure 1" with a node A. All writes shown in the figure affect this Conit.

never discarded, but are communicated between nodes during synchronization. The system is fully consistent when all nodes have seen all updates.

This section proceeds by explaining each of the three consistency dimensions, and how they are used to bound the inconsistency. These sections make use of an example Conit called "structure 1". This name refers to a player-built structure in a modifiable world. We assume that the game is able to detect such a structure, and ensures both reads and writes that affect the structure are indicated as such. In the example, this Conit is shared between two nodes, A and B. The values of the consistency bounds that are used by the Conit depend on the consistency dimension that is discussed.

Staleness

The amount of latency between a game server and its client has a significant effect on the gameplay experience of the player [8, 17, 54–56, 59, 64]. First Person Shooter (FPS) games are most sensitive to latency. For these types of games, the gameplay experience of players decreases when the latency reaches values in the hundreds of milliseconds [8, 54]. For Minecraft-like games, which focus on player-to-player and player-to-environment interaction in a large open world, a latency in the order of hundreds of milliseconds is the the acceptable limit for the games to be playable [17, 56, 64], but some players might still be able to detect latencies as small as 26-40ms [55]. In general, high latency can cause player experience to decrease. Because of this it is not only important for the latency to be bounded, it is also important that the latency remains within a range that is tolerable for the players.

Games are typically tick-based, updating the game state at a fixed frequency. The tick frequency affects latency: if a server operates at a tick frequency of 20Hz, a new tick is started every $\frac{1}{20} = 0.05$ seconds, or 50ms. This means that without taking into account network latency, updates from the server can take 50ms to reach a client. Assuming stable network conditions, games use the tick frequency to effectively bound the latency on updates. For instance, Minecraft-like game servers send updates such as entity locations to all connected clients every tick.

The Conit model bounds latency using a consistency dimension called *staleness*. Staleness is the amount of time an update may go unseen by one of the nodes in the Conit. The Conit model lets the implementation up to the programmer, but TACT, a prototype system that implements the Conit model [70], uses a pull-based approach combined with limiting availability to guarantee the staleness bound. As discussed in Section 2.2, limiting availability means that the system blocks: the system is unavailable to users while synchronization is in progress. Using this approach, a node frequently checks the last time it has pulled updates from other nodes. If the elapsed time since the pull of updates exceeds the staleness bound, the node blocks and pulls updates from the other nodes before it becomes available again. Although the pull-based approach provides a strong guarantee, is has the downside that it can be inefficient: other nodes can be contacted at times that no updates are available.

As an example we use a scenario where two clients participate in a Minecraft-like game and share a Conit called "structure 1". To make the real-time aspect more explicit, the number of milliseconds is displayed on the left-hand side. Node B specified a staleness bound of 60ms. Node A places a block every 20ms. After 60ms, node B has not received writes from its neighbors for 60ms, which is equal to its staleness bound. This means node B must now pull the writes from its neighbor, node A. This is indicated by the black arrow from A to B.

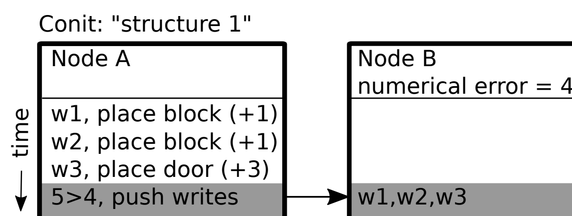


Figure 2.2: An example of a node B bounding numerical error to 4 by sharing a Conit called “structure 1” with a node A. All writes shown in the figure affect this Conit.

Numerical error

When pulling updates at a fixed frequency, no guarantee is given about the *importance* of the received updates. Some updates might not be noticeable to players or are simply irrelevant. Two examples of such updates in a Minecraft-like game are modifications that are made to the virtual world that are far from the player’s avatar, and minuscule movements of other avatars in the game. The Conit consistency model uses a numerical error dimension that can be used to communicate updates when they are relevant.

The numerical error of a node is defined by first assigning a global weight to each write, and then calculating the sum of the weights of locally unseen writes. Nodes first share their numerical error bounds with other nodes. Then each node propagates writes to other nodes to prevent them from exceeding their numerical error bound. The decision to propagate writes is taken conservatively based on local information. Two observations are important here.

1. Because the error is defined between a node r and a *global* state, there may at an arbitrary point in time be *no* nodes that have a numerical error of 0. That is: none of the nodes have seen all writes.

2. The numerical error is defined as the sum of the weights of the writes. This means that clients may still perceive different information when consulting different nodes when the numerical error is 0, because of differences in write order between the nodes.

An example of the effect of setting a numerical error bound can be seen in Figure 2.2. In this figure, we see two nodes that are participating in a Minecraft-like game. The accesses shown in the example affect a Conit called “structure 1”, which the nodes use to define special consistency rules for a particular building in the game. The nodes are both clients, and client A is placing blocks. Client B has specified a numerical error bound of 4, which means that the total weight of unseen writes may not exceed 4. Client A first places two regular blocks, which have a global weight of $+1$. This brings the total weight to 2, so no communication between the nodes is required (but is allowed). However, client A places a door, a more significant type of block, which has a weight of 3. This brings the total weight of writes not seen by B to 5, which is more than the numerical bound set by B. Therefore, before A can accept new accesses, its writes need to be communicated to B.

Order error

Even when all updates are synchronized between all nodes, different nodes can still have a different view of the resulting data. The reason for this inconsistency is that synchronizing updates between nodes does not specify in which order to *apply* the updates. Current Minecraft-like games are all designed as single-server systems, in which the server determines the order of all updates. However, all MMOG games use multiple servers to enable scaling to large numbers of players. These servers exchange updates and need to agree on the order of these updates to give players a consistent view of the data. Creating a fixed order of updates can be time consuming, and might not be necessary after every update. The *order error* dimension of Conits allows nodes to specify the number of updates that can be accepted without agreeing on their order with other nodes. This reduces the number of times such an order needs to be established. More formally, the order error is defined as the number of writes on a node that may still be reordered. In this thesis, we refer to such writes as *uncommitted*

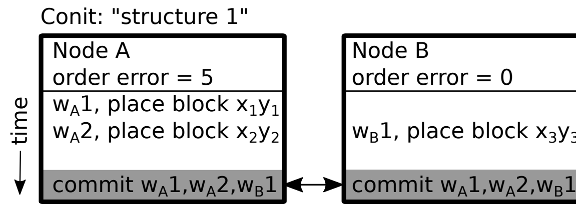


Figure 2.3: An example of a node B bounding order error to 0 by sharing a Conit called "structure 1" with a node A. All writes shown in the figure affect this Conit.

writes. The algorithm used to commit writes may be chosen freely by the system that implements the Conit model.

An example of the effect of setting an order bound is shown in Figure 2.3. Two clients play a Minecraft-like game and issue accesses over time that affect a particular Conit called "structure 1". In this example, only regular blocks are placed. Detecting conflicting writes is up to the application and is therefore not discussed here. To make this explicit, the blocks in this example are each placed on a unique location. In this example, both nodes specify an order bound. Node A specifies an order bound of 5, while node B specifies an order bound of 0. First node A issues a write, which raises its error bound to 1 and is accepted. Next, both node A and node B issue writes. At this time, node A's order error is raised to 2 and node B's order error is raised to 1. Because node B specified an order error bound of 0, its messages must be committed immediately. Committing messages involves creating a total order for the messages that are committed, which requires communication. Which commit algorithm is used is not important, only that after the commit phase there is no node in the system that exceeds its order error bound.

2.4. Similar consistency models

This section discusses a number of consistency models that are similar to the Conit consistency model. This section differentiates between two types of models. The first type of model originates from consistency model research. Usually these models are designed to be applicable to a specific type of system and are accompanied by a prototype implementation that is used to evaluate the model. The second type of model originates from game– or distributed-systems research. These models are usually embedded in a real-world system. This allows a more in-depth experimental evaluation, but can restrict the generality of the consistency model. These consistency models have a number of features that are desirable for the domain of Minecraft-like games. The important features from the Conit model discussed in Section 2.3 and the features from the related consistency models discussed here are summarized in Table 2.1.

2.4.1. Consistency model research

The Timed consistency model specifies a variable Δ that indicates the maximum amount of time a write may go unseen by other nodes [65, 66]. Δ takes a numerical value and is analogous to the staleness dimension that are part of Conits: it creates a consistency spectrum which bounds the real-time delay between an update being accepted and it being visible on all nodes. A difference with the Conit model is that Timed consistency does not support assigning an importance or weight to individual writes. Additionally, Δ is a global parameter, which means that all data that is synchronized between nodes adheres to the same consistency bound. In the Conit model, individual bounds can be assigned to each Conit for each node.

Beehive introduces *Delta consistency*. Similar to the Conit staleness bound and Timed consistency, Delta consistency guarantees that a read returns the last value that was written at most *delta* time units before that operation [61]. Similar to Timed consistency, Delta consistency also has *delta* as a global parameter of the application. The value delta can be specified as a time unit relative to a node's wall clock or a form of logical clock called *virtual time*. Using the wall clock makes delta consistency applicable for real-time systems such as games. Similar to TACT, Beehive limits availability (by blocking) if consistency bounds are exceeded, thereby guaranteeing the consistency bound.

The consistency model suggested by Krishnamurthy et al. [35] describes a model similar to Conits.

It defines both staleness and order error as dimensions to measure inconsistency. However, it does not support bounding inconsistency on numerical error, which means that it does not support assigning importance to individual updates. The model specifies order error as a global service-level parameter, and staleness as a parameter set by the client. The Conit model does not use the notion of a client, instead it considers all nodes as being equal and sharing the same data. The model also defines the staleness of data using a logical clock that progresses with the number of updates. Staleness is then the number of updates that have been submitted but not seen by the contacted node. This behavior can be modeled using the numerical error bound of Conits if each write is set to have a weight of 1. It does not correspond to the Conit definition of staleness, which uses wall-clock time and is defined as the amount of time a data item can be accessed on a node before that node needs to pull updates from the other nodes in the system. The model lets clients specify a staleness bound and a probability with which this bound should be met for each access. In the Conit model, consistency is enforced by blocking if bounds are exceeded, whereas the probabilistic guarantees of this model allow it to be always available. Allowing bounds to be set per access the model supports dynamically changing consistency bounds at runtime. This is an important feature because the workloads on games are also dynamic, which means that the amount of inconsistency that is tolerable for the players can also be dynamic. Another important feature from this model is that client-server communication is used, which maps directly to Minecraft-like games. In contrast, the Conit model uses all-to-all communication between nodes, a setup that is feasible for a small network of nodes, but not for globally distributed systems such as MMOGs.

2.4.2. Games and distributed systems research

Vector-Field consistency [58] uses a consistency model based on the Conit consistency model. It guarantees arbitrary levels of consistency measured across time, order, and value differences.[2] Their algorithms are simple because their system uses a lock-step approach. This is feasible since their use-case is an ad-hoc network with a single node in control of all traffic. The consistency model introduces a notion of *consistency zones*, areas around a *pivot* such as a player that determine the guaranteed levels of consistency of data in the consistency zones around the pivot. The consistency bound can be defined for each consistency zone using a *consistency degree*. These areas define monotonically decreasing levels of strictness in consistency. Consistency zones, consistency degrees, and pivots can be defined for any set of game objects. This model creates flexibility in assigning different consistency degrees to different game objects, but these differences are defined based on spatial properties (the zones around the pivot). The Conit consistency model offers greater flexibility by allowing any update to affect any Conit, allowing arbitrary consistency bounds to be assigned to arbitrary sets of data. Similar to the consistency model proposed by Krishnamurthy et al., the Vector-Field consistency model uses client-server communication, which directly maps to Minecraft-like games. This is an important feature that is missing in the Conit consistency model.

There has been extensive research published on the topic of scaling virtual environments. Donnybrook [15] has been proposed by Bharambe et al. as an architecture for large-scale high-speed peer-to-peer games. Donnybrook utilizes area-of-interest (AoE) management, defining an *interest set* for each player. Players receive frequent updates for other players that are in their interest set, but infrequent updates for players that are not. This binary choice means that Donnybrook does not offer a consistency spectrum. It also does not allow indicating the importance of individual writes: writes are either important to a player, or they are not. Reducing the number of updates for some entities in the virtual world corresponds to the approach of this thesis. However, using Dynamic Conits allows fine-grained control over the consistency between players because Conit bounds can be set to arbitrary values. Donnybrook does not give strong consistency guarantees, but dynamically configures for each players which updates are important and which are not.

Colyseus is an architecture for large-scale online games that uses relaxed consistency to improve system performance [14]. In the Colyseus architecture, game objects are assigned a *primary node* that is in charge of ordering the updates that are applied to that object. If an object is updated at a different node, these updates are applied *tentatively*, and may be reordered later by the primary node. Colyseus relates the types of inconsistency that can occur to the Conit consistency model, but it does not enforce consistency guarantees among the nodes. Instead, an optimistic approach

[2]The value difference corresponds to the numerical error bound used by the Conit model.

is used that propagates updates to nodes after each tick. Colyseus implements almost none of the important features from Table 2.1. However, Colyseus is the only system that applies weak consistency to increase the scalability of games running on multiple servers. Both the Opencraft project and this thesis want to use bounded consistency for the same goal.

PNUTS is a distributed database developed at Yahoo! that uses *per-record timeline consistency* [18]. For each record, or data item, writes are ordered using sequence numbers. These sequence numbers are available to clients, allowing the clients to specify their required level of consistency through specific API calls. For example, if clients require high availability the `read-any` call can be used which may return a stale value of the data item. If clients require the most recent data and can afford waiting, the `read-latest` call can be used. PNUTS also supports a `read-critical(version)` call which returns a version of the data item equal or greater than the one specified in the call, which allows dynamically bounded inconsistency based on data item versions. This API effectively creates a consistency spectrum for reads issued by clients. Because this spectrum is defined for *versions* of the data, it does not consider the real-time delay on the visibility of updates. Because the required consistency is defined per read, arbitrary bounds can be set for arbitrary data which can be different for each client.

Model features	Conits	Timed/Delta	Krishnamurthy et al.	Vector-Field	Donnybrook	Colyseus	PNUTS
Consistency is	guaranteed	guaranteed	probabilistic	optimistic	optimistic	optimistic	guaranteed
Consistency spectrum (F1)	LIN-EC	✓	✓	✓	✗	✗	✓
Write importance (F2)	✓	✗	✗	✓	✗	✗	✗
Real-time bound (F3)	✓	✓	✗	✓	✗	✗	✗
Arbitrary bounds on arbitrary data (F4)	✓	✗	✓	~	✗	✗	✓
Consistency granularity (F5)	per node	global	per read	per client	per client	global	per read
Non-blocking bounding mechanisms	✗	✗	✓	✗	✓	✓	✗
Dynamic reconfiguration	✗	✗	~	✗	✓	✗	✓
Interaction	all-to-all	all-to-all	client-server	client-server	client-server	server-server	client-server

Table 2.1: Comparison of the Conit consistency model with similar models.

3

Dynamic Conit model for Minecraft-like games

The previous chapter discusses the Conit consistency model and five of its important features that can improve the scalability of Minecraft-like games. This chapter discusses the Dynamic Conit consistency model. The Dynamic Conit model is an extension of the Conit model with additional features that make the model applicable to Minecraft-like games. This chapter is divided into four sections. First, nine requirements are formulated based on the features of the Conit model and the other consistency models discussed in Chapter 2. Second, the additional features from the Dynamic Conit model, and how these features satisfy the requirements, are discussed. Third, two possible applications of the Dynamic Conit model to Minecraft-like games are presented. Last, a performance model is introduced that quantifies the performance improvement for Minecraft-like games when using the Dynamic Conit model.

3.1. Dynamic Conit model requirements

Based on the discussion of Minecraft-related research and the comparison of the Conit consistency model with other consistency models, this section formulates nine requirements for a consistency model to be applicable to Minecraft-like games. These requirements are listed in Table 3.1. The table indicates for each requirement which model or system satisfies the requirement. This section identifies four drawbacks from the Conit consistency model that make its application difficult in today's large scale distributed systems. These drawbacks are translated into the bottom four requirements shown in Table 3.1.

First, the Conit model is configured to use a fixed consistency bound at runtime, but in a distributed system that deals with varying workloads, this means that the system is occasionally too consistent, or not consistent enough. If the system is too consistent, resources are spent synchronizing updates between nodes unnecessarily. If the system is not consistent enough, the players might encounter unexpected inconsistencies in the game. Changing the consistency bounds for existing Conits allows keeping data only as consistent as it needs to be, further reducing the communication required between nodes.

Second, the Conit model provides consistency guarantees by picking consistency over availability, blocking on user accesses when consistency cannot be guaranteed. While this might work well in some cases, this is not desirable behavior for a real-time system such as a game. If the system blocks when consistency bounds are not met, the game can become unresponsive to the input of the player, decreasing the gameplay experience.

Third, the Conit model does not include any bootstrapping mechanisms for changing the nodes or Conits in the system at runtime. The number of players in an MMOG is a main component of the system workload. However, the number of players experiences large variations, both on short (day/night cycle) and long (game popularity) time scales. This variability means that MMOGs experience a *dynamic* workload [50]. To scale with the workload, new server nodes may be started, and work may need to

#	Requirement	Conits Chapter 2	Dynamic Conits Chapter 3	Meerkat Chapter 4
1	A spectrum of consistency bounds	✓	✓	✓
2	Indicating the importance of individual updates	✓	✓	✓
3	Real-time bounds on updates	✓	✓	✓
4	Arbitrary consistency bounds on arbitrary data.	✓	✓	✓
5	Consistency bounds per node.	✓	✓	✓
6	Consistency bounds can be changed at runtime.	✗	✓	✓
7	Non-blocking consistency bounding mechanisms.	✗	✓	✓
8	Nodes can join and leave the system at runtime.	✗	✓	~
9	Consistency bound mechanisms work without all-to-all communication.	✗	~	✗

Table 3.1: Summary of Dynamic Conit model– and system requirements

be redistributed over these servers. The consistency model should support this by explicitly allowing nodes to join and leave the system, and let these nodes create new Conits.

Fourth, the Conit model requires all-to-all communication between nodes. While this may be possible in some setups (for instance within a rack in a data-center), this is prohibitive in globally distributed systems without accepting latencies in the order of seconds. Moreover, when partitioning the virtual world to balance the load over multiple servers, not all servers are interested in the same data. Players should only receive updates for data that is relevant to them, and servers should only exchange updates with other servers that are relevant to their players. To prevent nodes communicating data that is not relevant to them, the consistency model should work without all-to-all communication.

3.2. Conit model extensions

This section discusses five extensions to the original Conit model. The first four extensions are to fulfill the requirements specified in the previous section. How these features are implemented is later discussed in Chapter 4. The fifth extension does not satisfy a particular requirement specified in this thesis. Instead, the extension shows the expressiveness of the Conit consistency model by incorporating dead reckoning, a technique used in state-of-the-art games to hide network jitter, into the model. As a result, the inconsistency introduced by dead reckoning is quantified and bounded.

3.2.1. Dynamically changing Conit bounds (Requirement 6)

Which data is important for a player depends on that player's interest set. Because a player's interest set changes frequently, specific content can change from being very important on day x to being neglected on day $x+1$. Because of these changing interest sets, the level of consistency of specific data also changes. If a particular entity is in the interest set of many players, its location should not allow much inconsistency. As the interest in this entity declines over time, however, its location can allow increased inconsistency without negatively affecting the gameplay experience of players.

The error bounds on a Conit determine when writes are synchronized between nodes. Because the bounds are expressed as numerical values, they create a consistency spectrum. A bound of 0 guarantees strict consistency between nodes. A bound of ∞ guarantees eventual consistency between nodes (that is: no guarantees are given). In games, timing is important. This is no different for Minecraft-like games. Generally speaking, this means that higher consistency is better. However, higher consistency also means a higher exchange of messages between nodes to stay synchronized. Considering the dynamic workload of Minecraft-like games, committing a large amount of resources to stay consistent may not always be desirable or feasible. Increasing the consistency bound reduces the number of messages that need to be sent between nodes to synchronize. The throughput saved by reducing these messages may be used to support additional players.

In the Dynamic Conit model, nodes are allowed to change their consistency bounds used for any Dynamic Conit in which they participate. Both staleness and order error are enforced using only local information. This means a node can change both these bounds without any communication to the other nodes. Changing the numerical bound of a Conit involves the node informing its neighbors about

the changed value. For all three bounds, if the bound is decreased, the node needs to check if the new bound is exceeded. If this is the case, the node needs to synchronized writes with other nodes. Increasing the consistency bound can never cause a consistency bound to be exceeded, because increasing the bound increases the allowed inconsistency in the system.

3.2.2. Optimistic consistency (Requirement 7)

The Conit consistency model can enforce strict consistency, limiting availability to users. In a real-time system such as a game, blocking is undesirable. The system may become unresponsive to the input from the player while they are in the middle of an action sequence or battle. A simple solution to this problem is to synchronize asynchronously, and not wait for confirmation. Unfortunately, this does mean that the consistency bounds can no longer be guaranteed, because reads or writes may be executed on a node that has not yet completed synchronization. This thesis proposes a weaker consistency guarantee: optimistic consistency. Under optimistic consistency, if consistency bounds are exceeded, a synchronization mechanism is triggered. If the underlying network behaves correctly, and the latency between nodes is bounded, consistency between nodes is still guaranteed. This type of consistency is also used by Colyseus [14].

3.2.3. Conit update messages (Requirement 8)

In a large scale game, the workload varies over time; players join and leave the game, and server nodes are started or shutdown depending on the amount of players in the game. Furthermore, in a Minecraft-like game players enjoy the additional feature of being able to modify the virtual world. This means that players can create their own content. This changing set of nodes participating in the system, and the creation and removal of player-created content means that the data that should be kept consistent also changes over time. To support this, Conits should be able to be created or removed while the system is running, and new nodes should be able to synchronize with existing Conits.

For example, if Conits are defined for the area of interest of each player, Conits should be created or destroyed when players join or leave the virtual world respectively. Another example is an important structure in the virtual world that has its own Conit, which guarantees how inconsistent a player's view of this structure can be. If this structure is destroyed in the virtual world, the Conit is no longer affected by any data and can be removed.

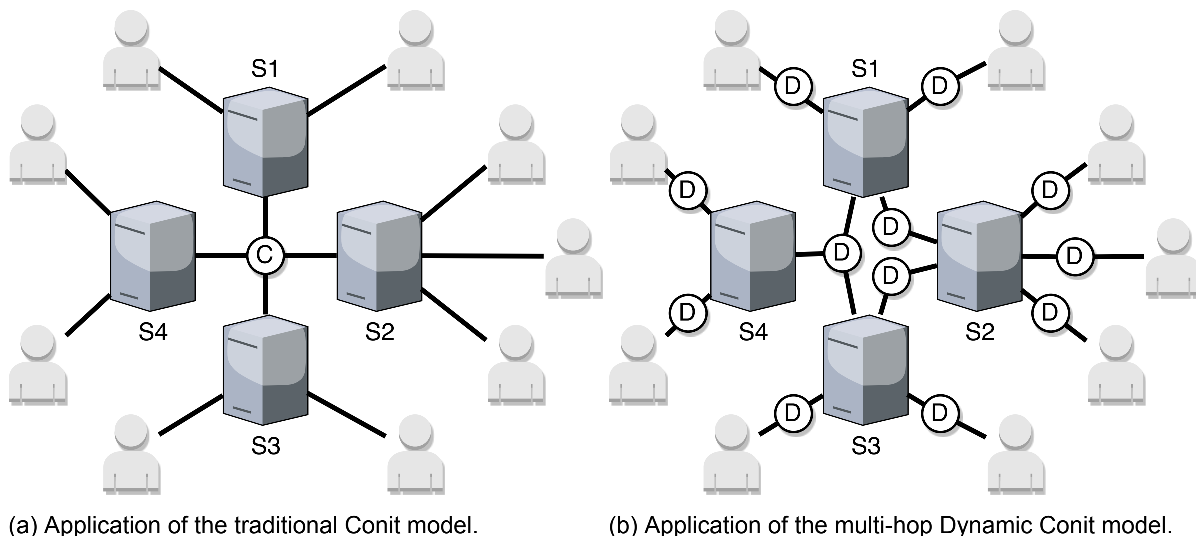
The Dynamic Conit model explicitly supports nodes from creating new Dynamic Conits and sharing these Dynamic Conits with other nodes at runtime. To this end, each Dynamic Conit keeps a member list, which tracks the nodes that participate in the Dynamic Conit. Nodes in the system can contact each other to participate in a new Dynamic Conit or remove themselves from it. Allowing new nodes and Conits to join the system at runtime is different from allowing Conits to change their consistency bounds at runtime. The former supports distributed system in which the set of participating nodes changes over time. The latter supports distributed systems in which the workload of existing nodes changes over time.

3.2.4. Multi-hop Conits (Requirement 9)

In the original Conit model, updates are exchanged between nodes that all share the same data set. Data exchanged in a Minecraft-like game is about players and the virtual world. If one node receives an update on a data item, the node forwards the update to all other nodes.

Figure 3.1a shows a typical application of Conits to a distributed system. Here the computer tower icons are servers, the person icons are users, the © indicates a collection of Conits, and the connections indicate which nodes share this collection of Conits. This setup can be mapped to the servers of a Minecraft-like game in a data-center which host instances of a virtual world. If a player is connected to one of the server nodes and builds a house or digs a tunnel, this change must become visible to players that are connected to other nodes.

The Conit model does not consider the inconsistency between a game server and its clients. The amount of inconsistency between a server and a client does not only depend on the consistency between the servers, it also depends on the synchronization between the client and the server. For the player, inconsistency becomes noticeable in the form of out-of-order updates and latency. As discussed in Section 2.3.2, the amount of latency between the server and the client has a significant effect on the gameplay experience of the player. However, guaranteeing low latency and high consistency requires high synchronization between the server and the client, increasing the workload at both ends.



(a) Application of the traditional Conit model.

(b) Application of the multi-hop Dynamic Conit model.

Figure 3.1: Comparison of the application of Conits and Dynamic Conits in a distributed system. The traditional Conit model does not support multiple hops of Conits in the network, and requires all-to-all communication between the nodes sharing the Conits.

To make Minecraft-like games more scalable, the workload on the server should be reduced. If Conits could be used between the server and the client, bounded consistency can be used to keep the gameplay experience high and reduce the amount of synchronization required between servers and their clients. Sharing Conits not only with servers but also with clients means that millions of nodes are added to the network, which all need to communicate with one another. This is neither feasible nor desirable. Furthermore, most clients may not be interested in most Conits. Conits that enforce consistency on the area of interest of player a might not be of interest to player b, whose avatar can be on the other side of the virtual world. Even if the bounds specified by player b are ∞, Conits require each node to allocate a certain amount of memory per node to correctly handle the consistency guarantees of other nodes, which can cause significant memory overhead in large networks.

To apply Conits in this context, we propose *multi-hop* Conits. Figure 3.1b shows a possible application of these Conits. In this model, a network of nodes is modeled as a set, in which special multi-hop Conits can be created for arbitrary *subsets*. A subset for which one or multiple Conits are defined is referred to as a *subnet*. When these subnets are viewed in isolation, the Conits function exactly as in the traditional model, using the same consistency bounds and mechanisms to enforce consistency. However, a new mechanism is introduced to support nodes that are part of multiple subnets. If such a node receives a write from one subnet via synchronization, it checks if the write affects any Conits shared with other subnets. If this is the case, the node bounds consistency according to these Conits. If the subnets form a connected graph, any write will be propagated throughout the entire network. [1]

Multi-hop Dynamic Conits makes it difficult to reason about the consistency guarantees over multiple hops in the network; nodes may be separated by multiple Dynamic Conit hops, each with their own consistency bounds. Figure 3.1b shows two clients that are connected to server 3 (S3). Both of these clients use Dynamic Conits to reduce communication with the server. Reasoning about the consistency between one of the two players and the server can be done using the traditional Conit model. However, consistency between both clients is more difficult to quantify. A mathematical investigation of the consistency bounds over multiple hops is out of the scope of this thesis. Instead, consider this toy example: assume that the staleness of location updates of the left player to the server is bounded with a numerical error bound of 1. Also assume that the staleness of location updates of the server to the right player is bounded by 50ms. In this case, the consistency bound between both players can be seen as the *sum* of both consistency bounds[2]: the location update arriving at the left player cannot have *both* a numerical error greater than 1 *and* a staleness value greater than 50ms.

[1] Using multi-hop Dynamic Conits requires using optimistic consistency: when the system waits for synchronization to complete, deadlocks can occur when subnets synchronize with each other.

[2] Using a liberal definition of the word ‘sum’.

3.2.5. Speculation error

The previous sections describe additions made to the Conit consistency model to satisfy the requirements shown in Table 3.1. This section describes another extension of the Conit model that does not satisfy a particular requirement. Instead, the extension discussed in this section shows the expressiveness of the Conit consistency model by incorporating dead reckoning into the model. As a result, the inconsistency introduced by dead reckoning is quantified and bounded.

The Conit model offers three application-agnostic dimensions to calculate and bound inconsistency between nodes. While these dimensions are expressive, they are created from the perspective of a distributed database. However, a Minecraft-like game is not only a distributed database, it is also a distributed *simulation*. This section defines *speculation error* as a new consistency dimension from the perspective of a distributed simulation. While still being application-agnostic, this new consistency dimension could allow more expressive bounded consistency in the domain of MMOGs, Minecraft-like games, and other domains that rely on distributed simulation.

This consistency dimension builds on the observation that simulations may benefit from relaxing the frequency with which updates are checked for correctness. A game server that hosts a Minecraft-like game performs many simulations. In some cases, these simulations can be very structured. For instance, a player might start a computation in a digital circuit, or travel by train over a rail. In these scenarios, clients can locally simulate these events simultaneously with the server without exchanging information. Another use-case for speculation error is the propagation of updates concerning player locations. In a virtual environment with many moving entities, servers can reduce communication load by synchronizing entity locations less frequently. Clients can use speculation error to locally estimate the locations of entities before the true location is received from the server.

This thesis defines *speculation error* as the maximum number of speculative updates on one node. A speculation error of x allows a node to generate x updates locally without communicating with other nodes. When a node reaches x updates, it agrees with the other nodes whose updates to accept and drops its own speculative updates. Speculation error makes *dead reckoning* part of the consistency model by quantifying how many speculated updates are allowed before synchronization is required. An implementation of this consistency dimension requires a consensus algorithm to determine which updates to keep. In the scenario of a MMOG with multiple server nodes, a trivial consensus algorithm would be to appoint a primary node for each data item and always accept its updates as the final updates.

3.3. Application to Minecraft-like games

Dynamic Conits are a flexible consistency model which can be applied to Minecraft-like games in multiple ways, increasing the scalability of these games. This section discusses two examples of the application of Dynamic Conits to Minecraft-like games and how these applications can increase the scalability of these games. The use of Dynamic Conits can reduce the communication between nodes, increasing system scalability. The two examples discussed in this section are chosen to target the two main channels of communication in MMOGs: the communication between the different server nodes, and the communication between the server and its clients.

The Conit consistency model distinguishes accesses from users and synchronization between nodes. In a Minecraft-like game, updates are exchanged between clients and servers. To map the Conit model to Minecraft-like games, accesses are defined as changes made to the virtual world by a player or server. Examples of modifications are moving the player's avatar or placing a block in the virtual world. Synchronization between nodes is defined as the messages exchanged between servers and clients that update the state of the virtual world on the receiving node. Examples of these messages are avatar displacement messages and block placement messages.

3.3.1. Guaranteed consistency between servers

As discussed in Section 1.2, to support large numbers of players, state-of-the-art MMOGs partition the virtual world and distribute these partitions over multiple servers. Traditionally, a geographic partitioning scheme is used, but this is not required. For instance, a partitioning based on social connections is also possible [67]. These parts are then assigned to different servers by a scheduler. The game servers share some of their data to hide the partition from the players. For instance, players that are at the border between two partitions should be able to see players at the other side of the border. Sharing

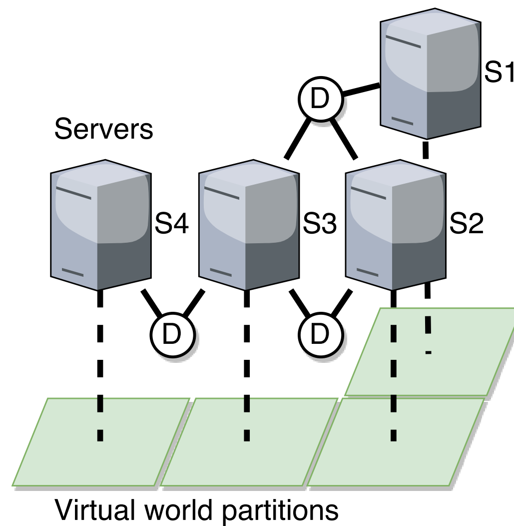


Figure 3.2: An application of Dynamic Conits to bound inconsistency between the servers that simulate a partitioned virtual world.

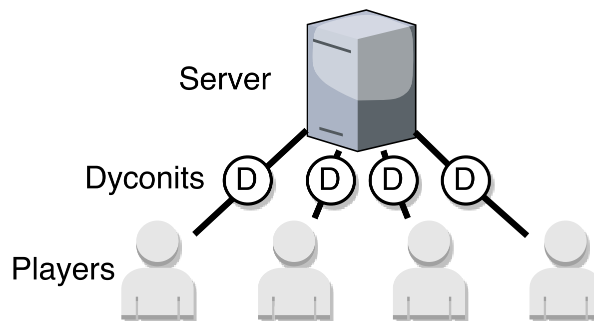


Figure 3.3: An application of Dynamic Conits to bound inconsistency between a game server and its clients.

this data introduces inconsistency between players that are connected to different servers.

To keep gameplay experience high, the inconsistency between the players should be bounded to a value that is acceptable for players. While the Conit model can bound inconsistency between nodes, the game servers are unlikely to share all their data with all other servers. To keep the system scalable, both the number of servers with which to synchronize and the amount of data that needs to be synchronized should be small.

Figure 3.2 shows the application of Dynamic Conits between servers. In the figure, the circled 'D' indicates a collection of Dynamic Conits, and the connections indicate which nodes share this collection of Dynamic Conits. The bottom part of the figure shows a bird-eye view of the virtual world and its partitioning. The green areas indicate zones in the virtual world. The dotted lines indicate which servers host which parts of the virtual world. The servers communicate with a subset of the remaining servers (indicated by the connections from the circled D's) using the multi-hop feature from the Dynamic Conit model. Multiple Dynamic Conits are shared between servers which host adjacent parts of the virtual world. These Dynamic Conits are affected by updates that take place close to the border, which causes these updates to be synchronized to the 'adjacent' server according to the configured Conit bounds.

3.3.2. Guaranteed consistency between clients and servers

In the traditional model, the Conits are shared between servers. Consistency between clients and servers is not considered. However, in Minecraft-like games, clients share a large amount of data with the servers and continuously synchronize updates. This synchronization requires resources from both the client and the server. If this synchronization can be reduced, the server may use these resources instead to support a larger number of players.

Currently, servers synchronize with their clients every game-tick. To reduce synchronization without

reducing gameplay experience, Dynamic Conits can be used. By sharing Dynamic Conits between the client and the server, bounded inconsistency is introduced that reduces communication between the two nodes. By defining strict consistency bounds for data in a player's interest set [13] (for instance the actions of nearby entities with which the player can interact), and defining loose consistency bounds for data that is less important to the player (for instance block placements in a structure that is out of reach of the player), communication can be reduced without reducing gameplay experience.

Figure 3.3 shows an example of this application of Dynamic Conits to games. Here a single server is connected to four clients. The circled 'D' indicates a collection of Dynamic Conits, and the connections indicate which nodes share this collection of Dynamic Conits. In this application, each player shares a collection of Dynamic Conits with the server. This collection can contain Dynamic Conits with strict consistency bounds for entities that are close to the player, or for the state of a digital circuit with which the player can interact. The collection can also contain Dynamic Conits with loose consistency bounds for entities that are far away from the player, or are not in the player's interest set.

3.4. Dynamic Conit performance model

This section defines and discusses a performance model for Dynamic Conits to quantify scalability improvements when using bounded inconsistency. The model rests on the assumption that every node has a maximum number of messages that can be processed per second: t_{max}. We define $T_{max} = n \times t_{max}$ as the maximum number of messages that can be processed per second across the entire system. Here, n is defined as the number of nodes in the system. Let T be the actual number of messages processed by the system per second, this leads to the following invariant:

$$\begin{aligned} T_{max} &\geq T \\ &\geq W + S \end{aligned} \tag{3.1}$$

Here W is the number of updates per second caused by the system *workload*, the actual writes and reads performed by clients. S is the number of messages per second caused by the Dynamic Conit model synchronizing between nodes. The total throughput is comprised of these two variables. In a real-world system, the update throughput, W, is an uncontrolled a variable, whose value is determined by the number of users in the system and their behavior. The synchronization throughput, S, is a controlled variable, whose value is determined by the workload W and the configuration of the Dynamic Conits. This means that scalability improvements must be made by modifying S. The value of S is determined by the workload, W, the number of nodes, n, and the consistency bound that is used, b_i:

$$S = f_i(W, n, b_i) \tag{3.2}$$

To simplify the model, we assume that only one consistency bound is used at once. This means that $i \in \{s, n, o\}$. Separate functions are used to calculate S, depending on if a staleness bound, numerical error bound, or order error bound is used. These three functions are defined and explained below.

3.4.1. Synchronization controlled by staleness bound

When using a staleness bound b_s, each node pulls writes from all other nodes at most every b_s milliseconds. In theory, pulling writes takes two messages.[3] This means that when using n nodes, each node exchanges two messages with $n-1$ other nodes $\frac{1000\text{ms}}{b_s}$ times per second. This can be written as:

$$\begin{aligned} S &= f_s(W, n, b_s) \\ f_s(W, n, b_s) &= \frac{2n(n-1) \times 1000\text{ms}}{b_s} \end{aligned} \tag{3.3}$$

Note that when using a staleness bound, S does depend on n (the number of nodes in the system), but not on W (the update throughput).

[3] If the staleness bound is very large, the number of writes that are synchronized may also be very large. In practice, these writes may not all fit in a single, large, synchronization packet.

3.4.2. Synchronization controlled by numerical error bound

When using a numerical bound, each node pushes writes to the other nodes if the bound is exceeded. Nodes communicate their numerical error bound to each other. Other nodes then push their writes conservatively based on local information. The numerical error bound specifies the weight of the writes a node may not have seen, compared to the global state (all writes). This means that for a constant numerical error bound, *the synchronization throughput per node increases quadratically with the number of nodes.*

As an example, let $b_n = 4$ and $n = 2$. For simplicity. assume that all writes have a weight of 1. For clarity, let's call these nodes Alice and Bob.[4] If Alice receives 4 writes from clients, she keeps them to herself because she knows that Bob allows a numerical error of 4. However, when Alice receives a fifth write, she concludes that the numerical error bound might be exceeded: Bob might not have seen any of the 5 writes received by Alice. Alice now bounds consistency by synchronizing all 5 writes with Bob.

Now consider the same example with $n = 3$: Charlie also joins the game. Alice now receives 2 writes from clients. Again, she keeps these writes to herself without communicating to either Bob or Charlie. When Alice receives a third write, she cannot proceed as in the previous example. Instead, she has to communicate her writes to both Bob and Charlie. Because the numerical error is 4, Bob allows 4 writes with weight 1 to be unseen by him. Alice does not know how many writes have already been received by Charlie. If Charlie has also already received 2 writes, the current number of writes unseen for Bob is $3 + 2 = 5$ (the writes from Alice and Charlie together), which exceeds Bob's numerical error bound. Because Alice does not know how many writes Charlie has received, she makes a conservative decision by synchronizing writes with Bob. Without loss of generality, Bob and Charlie can be swapped in this example, which is why Alice should also synchronize writes with Charlie. This guarantees that the consistency bound is never exceeded. This behavior can be written as:

$$\begin{aligned} S &= f_n(W, n, b_n) \\ f_n(W, n, b_n) &= \frac{2(n-1)W}{\frac{b_n}{n-1}} \\ &= \frac{2(n-1)^2 W}{b_n} \end{aligned} \tag{3.4}$$

3.4.3. Synchronization controlled by order error bound

When using an order bound, b_o, each node may have b_o writes that are *uncommitted*, which means they could still be reordered. Bounding this error means committing at least every b_o writes. How committing these messages works is unspecified in the original Conit model, and is also left unspecified in the Dynamic Conit model. Because the implementation is undefined, this section makes a simplifying assumption: a single node is appointed as the primary node, and orders all writes in the system. For a node to commit messages, it sends a list of uncommitted writes to the primary node and receives a list of committed writes back in the appropriate order. Every node manages their uncommitted writes individually. This can be written as:

$$\begin{aligned} S &= f_o(W, n, b_o) \\ f_o(W, n, b_o) &= \frac{2W}{b_o} \end{aligned} \tag{3.5}$$

[4]The author hopes the field of cryptography does not have a copyright on using these names in examples.

4
Meerkat: design of a Dynamic Conit system

To answer research question 3, specified in Section 1, we design and evaluate Meerkat. Meerkat is a prototype implementation of the Dynamic Conit model: it provides a set of mechanisms and protocols to measure and bound consistency according to the Dynamic Conit model. To focus the implementation of the prototype on the novel Dynamic Conit features, we design Meerkat according to the *Keep it simple, stupid (KISS)* principle. To this end, Meerkat uses multiple simple algorithms and single-purpose components to implement the Dynamic Conit model. This chapter discusses the design of Meerkat, while Chapter 6 evaluates Meerkat experimentally. In contrast to Minecraft-like games, Meerkat does not provide a graphical interface to a player, nor does it simulate a virtual environment. Instead, it accepts updates and reads on data, and synchronizes this data with other Meerkat nodes according to the set Dynamic Conit consistency bounds. This chapter discusses both the components and mechanisms that allow Meerkat to support Dynamic Conits and how Meerkat satisfies the requirements listed in Table 3.1.

4.1. Design overview

To understand how the different components of Meerkat interact, it is important to first understand what the components mean. This section discusses the main components in Meerkat's design, and how these components interact. Figure 4.1 shows Meerkat's system design. Some components in Figure 4.1 are marked with the letter D in the top right hand corner. Every component with this mark is required to support Dynamic Conits. The remaining orange components are required to support regular Conits. Forwarding accesses is indicated using solid connections. Forwarding commands or meta-data is indicated using dashed connections.

Meerkat is a distributed system that can run with a variable amount of nodes, and is located between the virtual world simulation and the data store. The virtual world simulator simulates the virtual world based on data obtained from both the player and Meerkat. The resulting data is sent back to Meerkat, which stores it locally in the data store and synchronizes the updates with remote servers and clients based on the configuration of the Dynamic Conits. Each access passes through the Dynamic Conit manager, which checks the inconsistency in each of the three inconsistency dimensions, based on the Dynamic Conits that are configured. Because the virtual world simulator knows which data is needed and how consistent it should be, it can configure the local Dynamic Conits through the Dynamic Conit manager. If synchronization is required for one of the consistency dimensions, the access is forwarded to the corresponding component. For instance, if the numerical error bound is exceeded, the access is forwarded to the Numerical actor. These actors synchronize with other nodes by exchanging updates with the corresponding actors on remote nodes. The figure only shows one remote node, but there can be arbitrarily many remote nodes to synchronize with.

To meet Requirement 6 of the Dynamic Conit model, the Workload monitor and Dynamic bound scheduler are included in Meerkat's design. The Workload monitor runs on all Meerkat nodes and keeps track of the message throughput. The Workload monitors across all nodes connect to a single

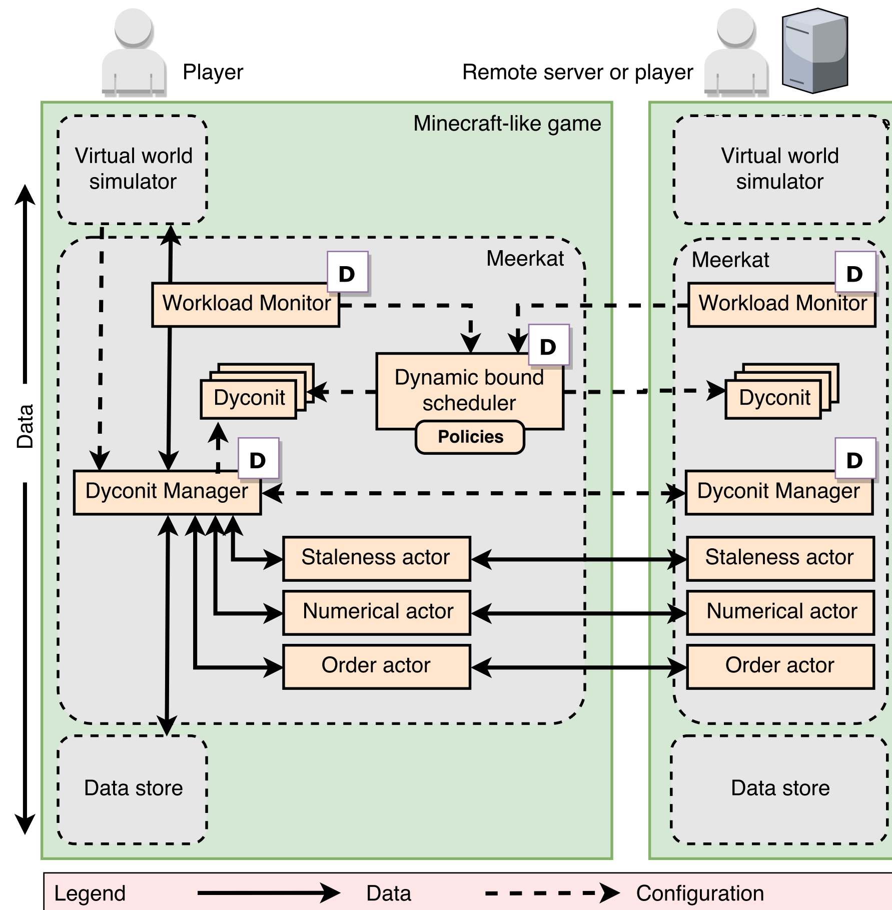


Figure 4.1: Meerkat system design. The orange boxes indicate Meerkat system components. The boxes marked with a 'D' indicate Meerkat system components required to support Dynamic Conits.

primary node, which runs the Dynamic bound scheduler. Based on the system throughput and the dynamic bound policy that is used, the consistency bounds of the Dynamic Conits are modified.

To meet Requirement 8 of the Dynamic Conit model, the Dynamic Conit manager takes care of updating Dynamic Conit information such as adding or removing nodes from the Dynamic Conit, adding and removing Dynamic Conits completely when no longer needed, and updating the numerical error bounds from remote nodes when their consistency bounds are changed.

4.2. Consistency bounding (Requirement 1, 4, 5)

Meerkat satisfies Requirements 1 and 4 by bounding consistency in the system using Dynamic Conits. Dynamic Conits use the same three dimensions as regular Conits: staleness, numerical error, and order error. Each node in the system can configure its own consistency bounds by selecting a numerical value for each of the three dimensions, for each Dynamic Conit. Because the consistency bounds can take any integer value greater or equal to zero, Meerkat implements a consistency spectrum (Requirement 1). In the Dynamic Conit model, every read and write specifies which Dynamic Conit is affected. Writes also have a weight associated with them to enable numerical error bounding. This allows clients to specify arbitrary bounds for arbitrary data (Requirement 4).

Although the Dynamic Conit model specifies how to quantify inconsistency, it does not specify which mechanisms and algorithms to use to bound the inconsistency. Meerkat uses a reactive approach, that bounds inconsistency on each of the three dimensions if necessary, after a new access is received from a client. An access means a read or write operation. While the reactive approach does not hide synchronization latency, it does make the system more predictable, making it easier to study its behavior. Meerkat's design is simplified by having three single-purpose components that match the three consistency dimensions. The components are executed sequentially, and each component is responsible

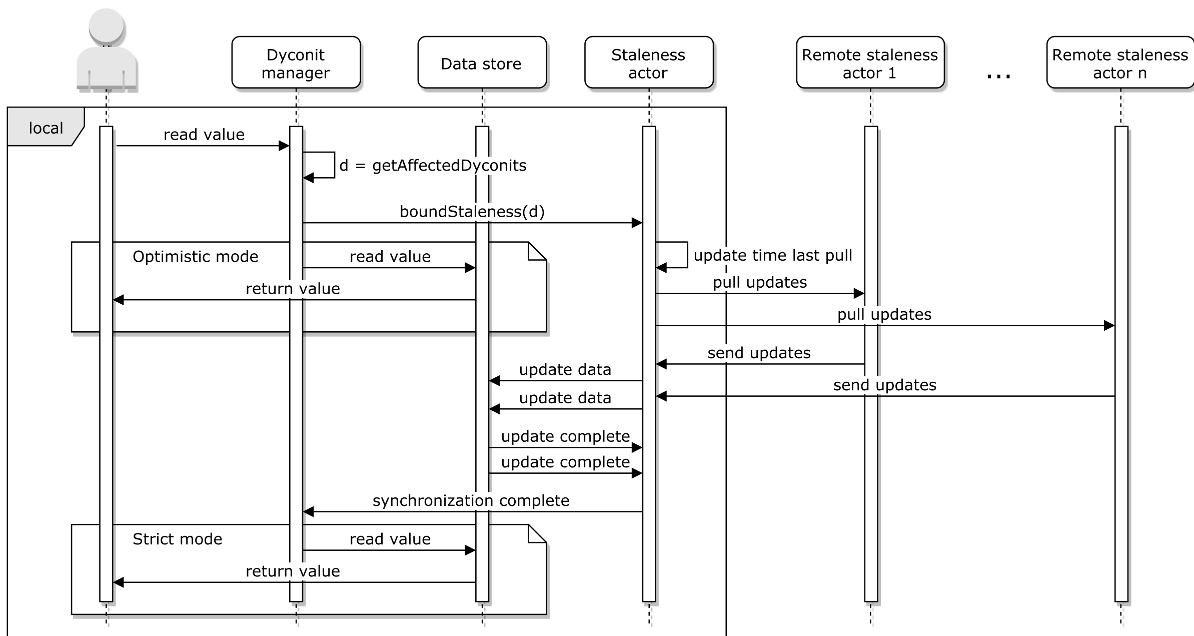


Figure 4.2: Meerkat staleness bounding mechanism

for bounding the corresponding consistency dimension. Figure 4.5 shows the sequence in which the individual components are activated when an access is processed by Meerkat. The actions within these components are discussed in the sections below. For both reads and writes, first the staleness bounding mechanism is triggered before the data store is accessed. For writes, after accessing the data store, the order error bounding mechanism and the numerical error bounding mechanism are triggered. Each of the actor components synchronize with remote nodes as soon as a consistency bound is exceeded.

4.2.1. Bounding staleness (Requirement 3)

Bounding staleness is done for both reads and writes, and is performed before the access is processed by the data store, such that the view of the data is within bounds during the execution of the read or write. To bound staleness error, each node keeps track of the last time writes where pulled from each of its neighbors. On each new access the staleness bounding mechanism executes the following steps:

1. Select all Dynamic Conits affected by access.
2. Get the union of all nodes that share at least one of these Dynamic Conits.
3. Ask each of these nodes for writes they have seen after their last contact with this node.

Figure 4.2 shows this behavior in the form of a sequence diagram. Here the Remote Staleness actors 1 through n are the actors corresponding to the collection of remote nodes that share one or more Dynamic Conits with the local node. The figure shows a user (top left) that wants to read a specific value. In a Minecraft-like game, this could be the game client that requests the contents of a chest of items after the player opens it. The Dynamic Conit manager (second from the left) selects the Dynamic Conits affected by the read, and forwards these to the Staleness actor (middle). The Staleness actor checks for each remote node that shares one of the affected Dynamic Conits if it has synchronized its updates within the set staleness bound. If this is not the case, the Staleness actor synchronizes with the remote node by requesting its updates. The local data in the data store (left of Staleness actor) is updated according to the synchronized updates. For simplicity, the actions taken on the remote nodes are not shown in the figure. After all synchronization is complete, the relevant data item is read and its value is returned to the user. If Meerkat is running using optimistic consistency instead of strict consistency, it returns the value to the user without waiting for synchronization to complete. Optimistic consistency is a Dynamic Conit feature discussed in Section 3.2.2. Its implementation in Meerkat is discussed in Section 4.3.2

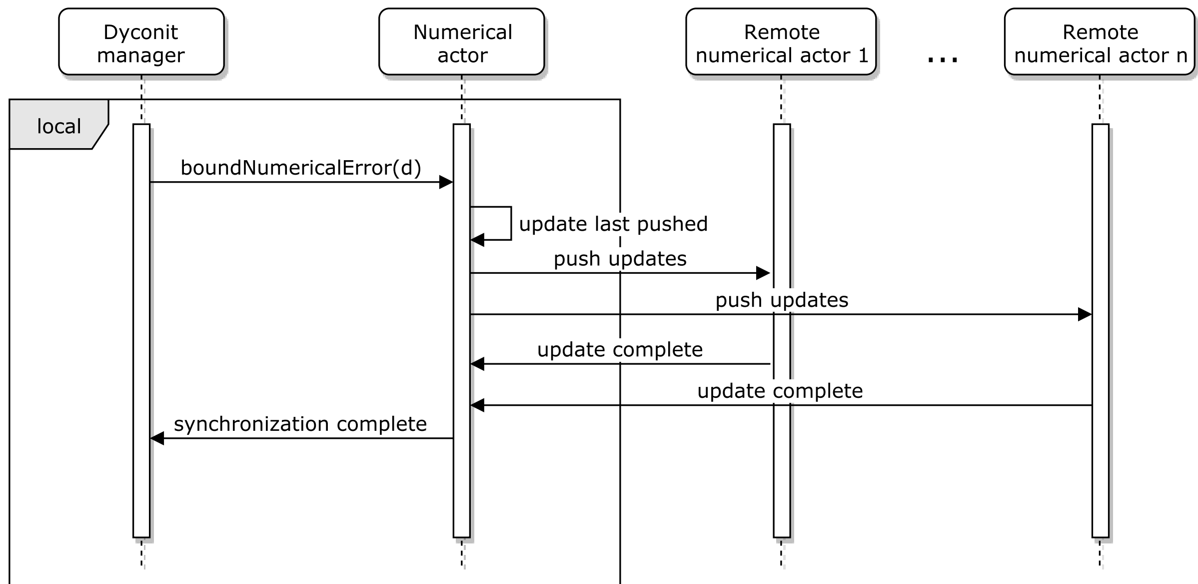


Figure 4.3: Meerkat numerical error bounding mechanism

The figure shows the importance of following the KISS design principle: although the actions taken by the local node to bound staleness are conceptually simple, its implementation involves multiple layers of message passing.

4.2.2. Bounding numerical error (Requirement 2)

The numerical error of a node is defined by first assigning a global weight to each write, and then calculating the sum of the weights of locally unseen writes. Nodes share their numerical error bounds with remote nodes when first establishing a connection, or when changing their numerical error bound. This is done using a mechanism discussed in Section 4.3.3. When a node receives a write, it calculates the numerical error conservatively using the method described in Section 3.4.2. If the numerical bound is exceeded, the node propagates writes to the other remote nodes. To bound numerical error, Meerkat executes the following steps:

1. Select all Conits affected by the new write.
2. For each Conit, for each node, determine if their numerical error bound could be exceeded.
3. If so, forward writes and wait for the remote nodes to complete processing.

Figure 4.3 shows this behavior in the form of a sequence diagram. After bounding staleness, Meerkat bounds the numerical error by forwarding the affected Dynamic Conits to the Numerical actor. The Numerical actor computes for which nodes the numerical error bound could be exceeded based on the numerical error bounds received from the remote nodes. For each remote node for which the numerical error bound is exceeded, the local node synchronizes all writes that it had not sent in previous executions of the numerical error bounding mechanism. After all remote nodes report that the received updates have been processed, the local node proceeds by bounding the order error.

4.2.3. Bounding order error

To simplify system design, Meerkat appoints a single node as the *primary* node. This node decides the commit order for all nodes. Each time a node's order error bound is reached, it synchronizes with the primary node by requesting to commit its local writes. This method is relatively old [52], and provides a single-point of failure. However, it is simple to implement and matches with the client-server model of most games, in which a server decides the final ordering of messages. On each write, Meerkat executes the following steps:

1. Send the primary node a commit request with the corresponding writes.

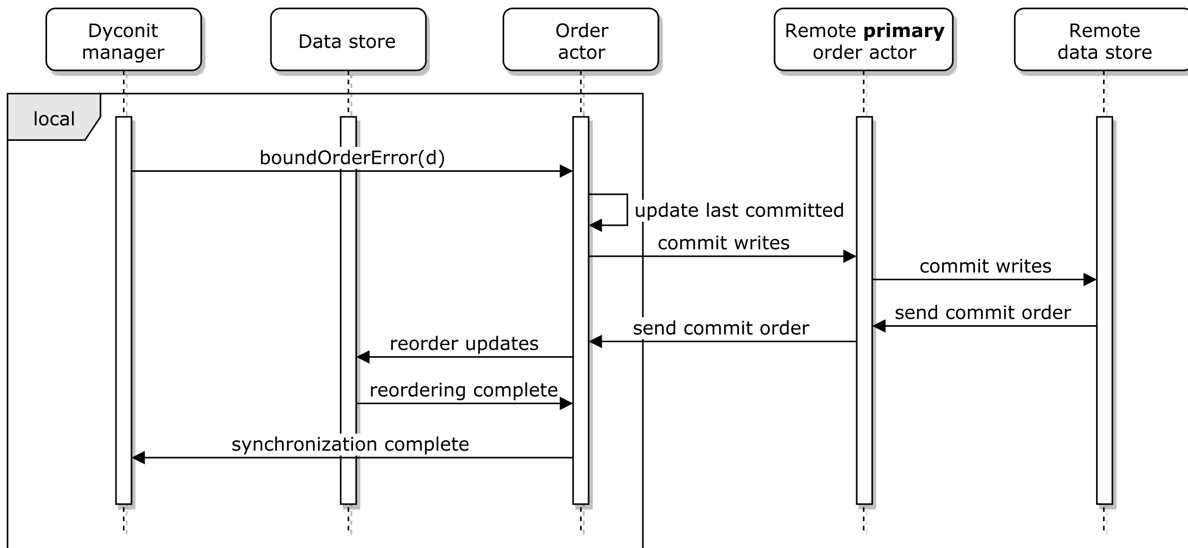


Figure 4.4: Meerkat order error bounding mechanism.

2. Receive the commit-order up to the most recently committed write on the primary node, and merge them with the local data store.

Figure 4.4 shows this behavior in the form of a sequence diagram. After bounding the numerical error, Meerkat forwards the affected Dynamic Conits to the Order actor (middle). If for any of these Dynamic Conits the number of locally uncommitted writes exceeds the set order error bound, all currently uncommitted writes are sent to the primary node. The primary node then determines the order of the writes and applies them to its local data store. The primary node keeps track of the last commit seen by the local node, and sends all writes that have been committed since. This can include writes synchronized by other nodes that have not yet been seen by the local node. Upon receiving the commits, the local node rolls back its data store and applies the commits in the order determined by the primary node. To simplify Meerkat's design, exactly the same sequence of operations is executed if the local node is the primary node. In this case, the Remote primary order actor (second from the right) and the Order actor (third from the right) are the same actor. The same holds for the Remote data store (right) and the Data store (second from the left). This causes the node to send messages to itself.

4.3. Dynamic Conit mechanisms

The previous section discussed how Meerkat bounds the inconsistency in the system such that the Dynamic Conit bounds are never exceeded. This section discusses the components and mechanisms used by Meerkat to support the Dynamic Conit features introduced in Chapter 3.

4.3.1. Dynamic bound policies (requirement 6)

Meerkat satisfies Requirement 6 by using a Dynamic bound scheduler. This scheduler updates the consistency bounds across nodes using the workload monitors that run on each node. The Dynamic bound scheduler itself runs on a single node: the primary node. This node is also in charge of assigning an order to the writes, which happens when bounding order error. For a production ready system, the primary node should be dynamically assigned at runtime to protect against system failures. To keep system design simple and focus on the Dynamic Conit mechanisms, this is not supported by Meerkat.

The Dynamic bound scheduler periodically asks all nodes for their throughput for both accesses from clients and messages used to synchronize with other nodes. Reducing the bounds on a Dynamic Conit increases the consistency between nodes, but also increases the number of messages that need to be exchanged to stay consistent. If the number of synchronization messages need to be reduced to handle an increase in the number of accesses received from clients, the dynamic bound policy can decide to increase the consistency bounds on existing Dynamic Conits. Similarly, when the number of accesses decreases, the scheduler may decrease Dynamic Conit bounds to achieve a higher level of

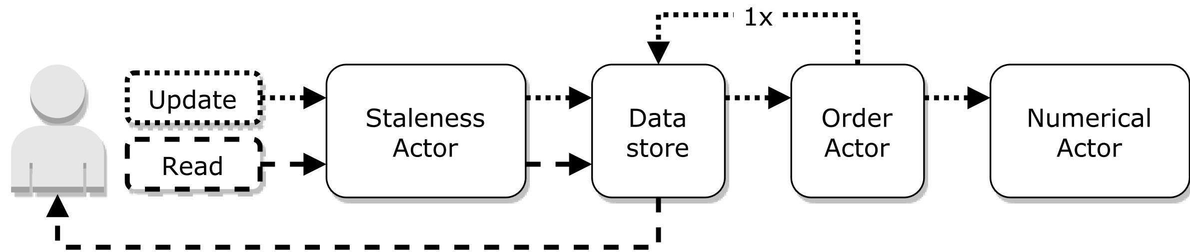


Figure 4.5: Meerkat processing pipeline. Because reads cannot increase the inconsistency in the system, they are processed differently from updates.

consistency.

The specific behavior of the Dynamic bound scheduler depends on the selected *dynamic bound policy*. This thesis presents three such policies, and discusses them in the sections below.

ADMI policy

The **ADMI** policy is a naive policy used as a performance baseline. It periodically checks the current throughput of synchronization messages. If the throughput is above the threshold, the Conit bound is multiplied by 2. Otherwise the Conit bound is decreased by 50.

This policy is derived from the additive increase/multiplicative decrease policy used in TCP congestion control. This policy linearly increases the congestion window used when transmitting data. If congestion takes place, the congestion window is exponentially decreased to prevent further escalation of the congestion.

PM policy

The **PM** policy is a more advanced policy that uses a moving average of the throughput of accesses, the number of nodes in the system, and the Dynamic Conit performance model described in Section 3.4 to calculate which Conit bounds would produce a synchronization message throughput of the desired amount. The **PM** policy calculates a lower bound for the required consistency bound, which means that the actual number of synchronization messages per second can be higher than the intended value.

PM-P policy

This policy is based on a scheduler used in ConPaas [4, 5], which is also a workflow scheduler. This policy is based on the **PM** policy. Except instead of calculating the Conit bounds such that the moving average workload would produce the desired number of synchronization messages per second, it uses linear regression to predict the workload for the next scheduling step and calculates the required Conit bounds for that workload.

The intuition behind this policy is that without anticipating the workload, the consistency bound calculation might lag behind the actual bound needed to bring the number of synchronization messages to the intended amount. Linear regression allows anticipating the update throughput based on recent throughput values.

4.3.2. Pipeline for strict and optimistic consistency (Requirement 7)

Meerkat satisfies Requirement 7 by implementing the optimistic consistency guarantees from the Dynamic Conit model discussed in Section 3.2.2. Meerkat uses a processing pipeline for incoming updates and reads. This pipeline is shown in Figure 4.5. Depending on how Meerkat is configured, it can enforce strict consistency or optimistic consistency. If only one access can use the data pipeline at the time, and the individual components process accesses synchronously, the system enforces strict consistency. If an access triggers synchronization to bound inconsistency, the pipeline is occupied for a longer period of time. This reduces availability and *blocks* incoming accesses. If the pipeline can be filled with accesses, Meerkat becomes *highly available*, no longer blocking incoming accesses. This means that the consistency bound is now optimistic: for an incoming access it is no longer guaranteed that the data is within consistency bounds. Instead, if an incoming access finds the data exceeding the consistency bounds, it is guaranteed that synchronization is currently in progress.

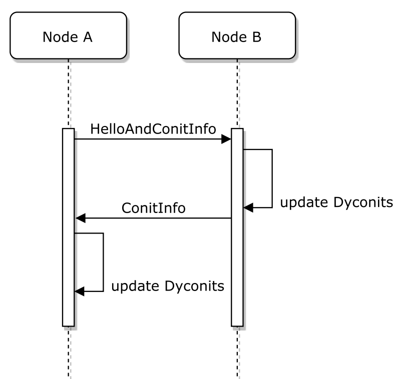


Figure 4.6: Meerkat 2-way handshake for adding nodes or Dynamic Conits. The HelloAndConitInfo message has the same content as a regular ConitInfo message, except it triggers a response from the receiving node.

From Figures 4.2, 4.3, and 4.4, only Figure 4.2 shows the difference in behavior when using strict consistency versus optimistic consistency. Figure 4.3 and 4.4 show Meerkat's behavior when using strict consistency. When using optimistic consistency, the system would continue processing the next access immediately after passing the affected Dynamic Conits to the Numerical actor or Order actor respectively. This is omitted from the figures to improve readability.

4.3.3. Dynamic Conit information message (Requirement 8)

This section discusses how Meerkat partially satisfies Requirement 8. Meerkat supports both nodes and Dynamic Conits to join and leave the system at runtime, but the mechanism used to satisfy this requirement cannot be used in a production-ready system. This section discusses this feature in three steps. First, it discusses the mechanism used by Meerkat to let new nodes join or create new Dynamic Conits. Second, it discusses a limitation of the Dynamic Conit model that is introduced by introducing this feature, which prevents it from being used in a production-ready system. Third, it discusses the wait mechanism, a simple mechanism used by Meerkat to avoid this limitation in experimental setups.

Requirement 8 is satisfied by allowing nodes to send each other Dynamic Conit information messages that specify their participation in Dynamic Conits. When a node joins an existing Conit, it must inform the current members it would like to join the Conit, and communicates its numerical error bound. It does this by sending a ConitInfo message to the current members. Upon receiving this message, existing members add the node to their set of neighbors and communicate their numerical bound to the new node before new accesses are processed. Adding the new node includes communicating all necessary writes to the new node. If a node leaves an existing Conit, it must inform the other members of the Conit, along with any writes that the leaving node did not yet communicate. Other nodes remove the node from their set of neighbors and abort all current attempts to synchronize writes to this node. Meerkat tracks for each Dynamic Conit which nodes share that Dynamic Conit. When a new node joins the system, or creates a new Dynamic Conit, it sends its Dynamic Conit information to all other nodes that it wants to share Dynamic Conits with using a two-way handshake shown in Figure 4.6. The node that joins the system or creates a new Dynamic Conit performs this handshake with all other nodes with which it shares Dynamic Conits. Leaving the system or a Dynamic Conit works analogous to joining: when a node leaves a Dynamic Conit, it sends a ConitInfo message indicating that it is no longer part of that Dynamic Conit. This will stop synchronization to this node. If the node wants to leave the system, it sends a ConitInfo message specifying that it is no longer a member of any Dynamic Conit. This will stop all synchronization to this node.

Meerkat may have been running for an arbitrary amount of time when a new node joins the system. If this node joins an existing Dynamic Conit, an arbitrary number of updates need to be synchronized before the consistency bounds of all nodes are met. Because neither the Dynamic Conit model nor Meerkat include a mechanism to merge updates, the new node may require a large amount of time to complete processing all synchronized updates. During this time, the node is unable to process accesses from users.

Meerkat has an optional wait mechanism to prevent this scenario from occurring during use. The wait mechanism makes sure nodes are never overloaded by synchronizing too many updates at once.

It does this by preventing the Meerkat nodes from accepting any accesses from users until they are connected to every other node. A different solution is required when applying the Dynamic Conit model to a production-ready system. In a real-world scenario, nodes cannot be required to all join during system startup because the game has no control over when players join or leave, or when server nodes fail.

4.4. Meerkat performance model

Meerkat implements the Dynamic Conit model, which is based on the generic Conit consistency model, but has a number of features to make it applicable to Minecraft-like games. The Dynamic Conit performance model is defined in terms of throughput: the number of messages that can be processed per second. The goal of Meerkat is to evaluate the scalability improvement in number of players when using Dynamic Conits. Therefore, it is necessary to translate the number of messages per second to a number of players. To quantify the scalability improvements when using Meerkat this section defines a performance model for Meerkat based on the performance model for Dynamic Conits discussed in Section 3.4.

Because player behavior is hard to predict and changes over time, this section takes the conservative assumption that a player sends an update to the game once every game tick. For Minecraft-like games, the tick frequency is 20Hz. This means that a player sends 20 updates per second; a player generates a workload of 20 messages per second. Let the number of players be p. Then, using Equation 3.1, this can be written as:

$$\begin{aligned} T_{max} &\geq W + S \\ W &= 20p \\ T_{max} &\geq 20p + S \\ p &\leq \frac{T_{max} - S}{20} \\ &\leq \frac{T_{max}}{20} - \frac{S}{20} \end{aligned}$$

Removing the constants by declaring new variables:

$$\begin{aligned} P_{max} &= \frac{T_{max}}{20} \\ S' &= \frac{S}{20} \end{aligned}$$

Here P_{max} is the maximum number of supported players, assuming there is no synchronization overhead. S' is the synchronization overhead expressed as the number of players. This can be written as:

$$p \leq P_{max} - S' \tag{4.1}$$

This performance model indicates that we can increase the number of supported players p by decreasing the synchronization overhead S'. The value of S (and S') depends on the workload (the number of players), the number of nodes, and the consistency bound that is used. The functions to calculate the value of S are described in Section 3.4. We use this model to quantify the scalability improvement when using the Dynamic Conit model in Chapter 6.

5

Experimental setup

This chapter discusses the setup for the experiments presented in this thesis. This thesis contains two sets of experiments. The first set of experiments evaluates the scalability of Minecraft-like games. The second set of experiments uses Meerkat to evaluate the scalability improvement for Minecraft-like games when using Dynamic Conits. Meerkat is a prototype system that implements the Dynamic Conit model.

The first section of this chapter gives an overview of all experiments presented in this thesis. Because no tools exist to evaluate the scalability of Minecraft-like games, the second section introduces Yardstick, a distributed benchmark for Minecraft-like games. The remaining sections discuss the workloads, experiment environment, metrics, and tools used in the experiments.

5.1. Experiments overview

This chapter provides a detailed description of the experimental setup used in this thesis. An overview of all experiments presented in this thesis is shown in Table 5.1. This section gives a brief description of this table and the Minecraft-like games evaluated in this thesis. The *chapter* column indicates in which chapter the results of the experiment are discussed. The *focus* column indicates what the experiment evaluates. The *system* column shows which systems are evaluated: Minecraft-like games, or Meerkat. The *workloads* column indicates the workloads used in the experiment. The workloads are discussed in Section 5.3. Meerkat can be configured to use one of multiple policies to dynamically change the consistency bounds. The *policies* column indicates which policies are used in the experiment. These policies are described in Section 4.3.1. The *metrics* column shows the main metric used to report the experiment results. The main metrics for all experiments are discussed in Section 5.5.1.

This thesis evaluates the scalability of three popular Minecraft-like games using Yardstick. The three games are:

1. **vanilla**, the official Minecraft game developed by Mojang.
2. **Spigot**, the most popular community-driven modded version of **vanilla**.
3. **Glowstone**, a from-scratch implementation that is compatible with the Minecraft protocol.

Game configurations for each game are modified to reduce the bias in the experiments while still being realistic. For instance, NPCs have been disabled (Minecraft primarily features NPCs that attack players) and join throttling has been disabled. The complete configurations for the experiments can be found in Appendix A.

5.2. Yardstick: design of a Minecraft-like game benchmarking tool

Although Minecraft is one of the most popular games of all time, no systems exist that evaluate the performance and scalability of Minecraft-like games. Furthermore, there are no large-scale, publicly available, workloads to test Minecraft-like games. To solve this problem and answer research question

Chapter	Focus	System	Workloads	Policies	Metrics
6.1.2	players vs. simulation speed	Minecraft-like games	**increasing players**	n/a	tick frequency, relative utilization
6.1.3	players vs. computation	Minecraft-like games	**increasing players**	n/a	number of utilized cores
6.1.4	players vs. network activity	Minecraft-like games	**fixed players**	n/a	MC-packet throughput
6.2.2	throughput vs. consistency	Meerkat	**stress-test**	**static**	update throughput
6.2.3	synchronization vs. consistency	Meerkat	**stress-test**	**static**	synchronization throughput
6.2.4	effect of dynamic bound	Meerkat	**increasing**, **50-player trace**	**static**, **PM**	consistency area, others
6.2.5	effect of adding nodes	Meerkat	**stress-test**	**static**	update throughput
6.2.6	effect of dynamic bound policies	Meerkat	**50-player trace**	**ADMI**, **PM**, **PM-P**	consistency area, others

Table 5.1: A summary of all experiments presented in this thesis. For each experiment: the chapter in which it is discussed, its focus, the system under test, the workloads and policies used, and the metrics used to report results.

#	Requirement	Section
1	Realistic and configurable player behavior	5.2.3
2	Scale to large numbers of players	5.2.4
3	Compatible with multiple Minecraft-like games	5.2.4, 5.2.5
4	Record complex performance metrics	5.2.5
5	Record the interaction between players and server	5.2.6

Table 5.2: Summary of Yardstick system requirements.

1, we propose Yardstick. Yardstick is a benchmark designed to evaluate the scalability of Minecraft-like games.

While Yardstick started as a part of this thesis, it is now an independent research project under the Opencraft umbrella. Jerom van der Sar, a Minecraft expert and Bachelor Honors student, now leads the project and is responsible for the further design and implementation. This seems a suitable place to thank him again for his contributions to the project (specifically for the implementation of the Yardstick *collectors* for three Minecraft-like games, the player emulation and data publishing components, and for the formulation of the relative utilization metric), and his infinite enthusiasm for Minecraft and the Opencraft project.

The remainder of this section is structured as follows. Section 5.2.1 formulates the requirements for a Minecraft-like game scalability benchmark. Section 5.2.2 discusses the Yardstick system design. The remaining sections discuss specific system components that are part of Yardstick and fulfill requirements formulated in Section 5.2.1.

5.2.1. System requirements

This section formulates and discusses five requirements for a Minecraft-like game scalability benchmark. An overview of the requirements is shown in Table 5.2.

The aim of the Opencraft project is to scale Minecraft-like games to support millions of players. Ideally, the scalability of these games is evaluated using millions of human players. However, even if this number of participants could be gathered, it would be a logistical nightmare to perform a controlled experiment. The next best option is to use emulated players that are controlled by Yardstick itself. To obtain meaningful scalability results, it is important that the behavior of the emulated players is similar to that of human players.

To benchmark scalable versions of Minecraft-like games, the benchmark should be able to control large numbers of players. Because each player communicates with the server, controlling large numbers of emulated players can be resource intensive. It is unlikely that a single system can control millions of emulated players at once.

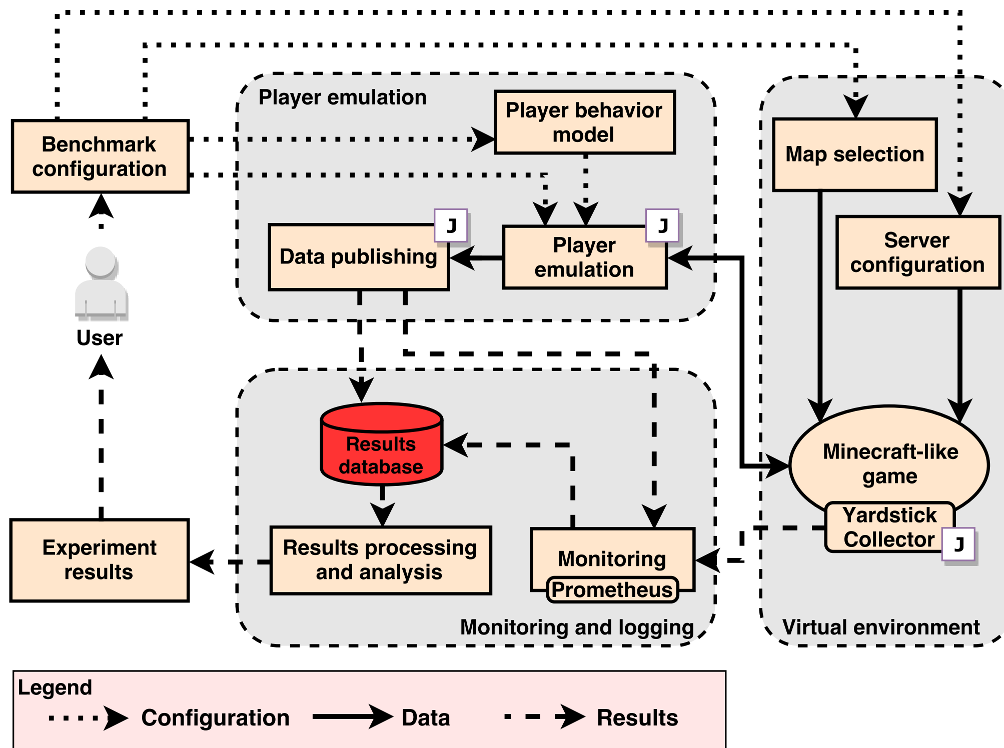


Figure 5.1: Yardstick system design. The orange boxes indicate Yardstick system components. The gray boxes group components that serve a similar purpose. Boxes marked with a 'J' are designed or developed by Jerom van der Sar.

The goal of Yardstick is to evaluate and compare the performance of Minecraft-like games. To allow performance comparisons, it is important for Yardstick to be compatible with multiple Minecraft-like games. Yardstick supports a Minecraft-like game if it can both submit a workload (by connecting emulated players) and observe its behavior (by recording a collection of performance metrics from the game server).

To evaluate the scalability of a Minecraft-like game, Yardstick should record a collection of performance metrics. Some metrics, such as the outgoing message throughput of the server, are visible to the players, but other, more complex, metrics are kept internally by the server. To record these metrics, Yardstick should be able to inspect the game state without significantly affecting the server behavior. These internal metrics can give more insight into the scalability of these games. The relative utilization is such a metric and is used in this thesis. It is described in Section 5.5.2.

By recording the interaction between the players and the server, Yardstick can export workload traces that can be replayed either on different Minecraft-like games, or on prototype systems such as Meerkat. These workloads can be used to evaluate the performance of these systems with a realistic workload, and help create reproducible experiments.

5.2.2. Design overview

An overview of the Yardstick system design is given in Figure 5.1. The Minecraft-like game is the system under test. The orange boxes marked with 'J' are developed by Jerom van der Sar. The remaining boxes indicate components developed as part of this thesis.

The Yardstick user can specify the workload that is used by changing the player behavior and other player emulation parameters such as the number of players to connect to the server, and the interval at which the players connect. The user can also configure the system under test by providing a server configuration and a map used by the server. While Minecraft-like games are typically able to generate maps at runtime, numerous highly-popular custom-made maps exist. Using the same map for multiple Minecraft-like games allows comparing the scalability of these games. Yardstick's design can be split into three parts:

1. Player Emulation (workload)

2. Virtual Environment (system under test)

3. Monitoring (collecting metrics data)

The player emulation module connects players to the Minecraft-like game server based on the configuration set by the user. Once the emulated players are connected to the server, their behavior is determined by the player behavior model. The messages exchanged between the players and the server are forwarded to the results database by the data publishing module. The behavior of these players is discussed in the next section.

The virtual world contains the Minecraft-like game, which is started with a configuration and map selection specified by the Yardstick user. To gain insight in the performance and scalability of Minecraft-like games, Yardstick provides so-called *collectors* for a number of Minecraft-like games that hook into the game, and expose complex performance metrics based on internal information to the monitoring subsystem. Section 5.2.5 describes these collectors in more detail. Section 5.5.2 describes the performance metric that the Yardstick collectors currently record.

Both the data publishing component and the Yardstick Collectors expose data to the monitoring subsystem. The monitoring component uses Prometheus to record this data and store it for later analysis. Which metrics are collected, and how Prometheus is used to collect these metrics is discussed in Section 5.5. Processing of the experiments happens after the experiment is completed. The result processing and analysis component extracts the resulting data from storage and is able to generate a collection of tables and figures for analysis by the Yardstick user.

5.2.3. Player behavior model

While research about player behavior in games is available, no model of player behavior exists for games with modifiable environments. Yardstick is designed to be extensible and allow users to add new behavior models that include modifications to the virtual environment. For games, some archives exist that contain a collection of game-traces. These traces contain information such as player movement and actions for a variety of games [28]. However, to our knowledge no large scale game-trace exists for Minecraft. For this reason, the experiments in this thesis use a synthetic workload.

Currently, Yardstick supports a simple, but configurable, player interaction model based on an existing behavior model from Second Life [38]. This model is based on observations of real players in Second Life. The movement model distinguishes between two types of movement:

1. short-distance movement.

2. long-distance movement.

Short-distance movement represent players that are in an area that contains content such as a player's house. Long-distance movement represent players that are exploring the world, looking for content. To the best of our knowledge, no realistic ratio between these types of behavior is known. Therefore, Yardstick chooses one of these two types of movement uniformly at random. Once a type of behavior is selected, the players perform this behavior for a duration selected uniformly at random between 1 and 3 minutes. When this time has expired, Yardstick again chooses a type of movement uniformly at random.

While the Minecraft modding community is large and multiple player emulation frameworks implementations already exist, we found that these frameworks are either not easily programmable, or were incompatible with modern versions of the game due to changes in the protocol used for Minecraft server-client communication. For this reason Yardstick emulates players using *custom* player emulation. Currently, the player emulation model is simple. The player emulation module ignores most incoming messages from the server, except for the map data, which is needed for player path-finding.

5.2.4. Player emulation

To benchmark scalable versions of Minecraft-like games, Yardstick should be able to emulate large numbers of players. Because each player communicates with the server, controlling large numbers of emulated players can be resource intensive. To support this, Yardstick distributes the load over multiple nodes. Yardstick can do this because the players, once connected to the server, operate independently of each other. This means that using multiple nodes to emulate players does not introduce overhead that limits scalability. Each node uses the same configuration, emulates an equal number of players,

workload	duration	player join interval	player batch size	max. players	# player emulation nodes	repetitions
increasing players	3600s	120s	10	300	5	3
fixed players	600s	1s	5	25-300:25	5	6

Table 5.3: Properties of the workloads used in the Yardstick experiments.

and connects players to the same server. Yardstick does allow the player emulation nodes to communicate. This enables complex workloads, such as a single player connecting to the server at a fixed interval, in which case the nodes need to take turns connecting new players.

A player emulation node monitors the communication between each of its players and the server. When the Minecraft-like game server communicates that the player is logged in, and both the avatar and world are loaded, the player is assigned a task based on the player behavior model that is used. The tasks, similarly to the Minecraft-like game clients, are tick based. The players are emulated by executing a single step of the their task at a fixed frequency. The communication between the players and the server is forwarded to the data publishing module that stores it with the experiment results.

5.2.5. Yardstick collector

Analyzing the external behavior of the Minecraft-like game server is enough to determine if the server scales to a certain workload, but it does not show where the scalability bottlenecks of the systems are located. To evaluate the scalability bottlenecks of these games, more advanced performance metrics should be collected that record their internal behavior. To observe this internal state, Yardstick uses *Collectors*. These Collectors consist of custom-code which is added to the server source code directly. This code exposes complex performance metrics to Yardstick's monitoring component. Alternative methods, such as the use of Java agents or an API provided by the game developers, can also be used to create Collectors. However, these alternatives are not as reliable as the current implementation. Java agents only work if the Minecraft-like game runs on the JVM, and using an API is only possible if the game developers provide it.

5.2.6. Data publishing

The data publishing module stores the communication between the emulated players and the Minecraft-like game server. Recording the interaction allows for the creation of trace-based workloads that can be replayed on other Minecraft-like game servers, or on prototype systems such as Meerkat. The data publishing stores this communication by asynchronously writing the data to the result database as separate files for each player emulation node. For each Minecraft packet that is sent or received by a player, the type of packet and a time stamp is stored. Because the volume of this data is large, it is stored using stream compression.

5.3. Experiment workloads

The experiments performed for this thesis use a variety of workloads, but also share many important properties. The Minecraft experiments use emulated players to evaluate performance. One type of behavior is used in the experiments presented in this thesis. This behavior is described in the next section. Because Meerkat is a prototype system for Dynamic Conits, and not a complete game, emulated players cannot connect to Meerkat. Instead, Meerkat is evaluated by observing its behavior while sending writes (value updates) to it. This is further discussed in Section 5.3.2.

5.3.1. Yardstick experiment workloads

Yardstick is a distributed benchmark that supports large-scale player emulation. To support coordinated player behavior, Yardstick player emulation nodes are able to communicate with each other. For example, an experiment focused on how the number of players affects server performance may let emulated players join the server in a regular interval. In this case, the Yardstick Bot Framework can internally schedule player join behavior using a round-robin approach to distribute load on Yardstick instances evenly. The Yardstick experiments use two different workloads. An overview of the workload parameters is shown in Table 5.3.

The first workload, **increasing players** lets players join over time until a maximum number of players

workload	policy	duration	bounds	synchronization throughput target	# nodes	repetitions
stress-test	**static**	300s	0-5,50-250:50	n/a	1,2,4,8,16,32	20
increasing	**static**	1500s	0, 5	n/a	1,2,4,8	1
increasing	other	1500s	n/a	200	1,2,4,8	1
50-player trace	**static**	600s	0, 5	n/a	1,2,4,8	1
50-player trace	other	600s	n/a	200	1,2,4,8	1

Table 5.4: Properties of the workloads used in the Meerkat experiments. When the **static** policy is used, a fixed bound is set. When another policy is used, the bound is dynamic and a synchronization throughput target is set.

is reached, and observes the Minecraft server during this process. For this workload, 5 player emulation nodes are used. Each node connects 2 new players every 120 seconds. The experiment duration is 1 hour, or 3600 seconds. The join interval, combined with the duration of the experiment brings the maximum number of connected players to 300. This workload is repeated three times for each Minecraft-like game.

The second workload, **fixed players** quickly connects players until a fixed amount of players has been reached. For the remaining duration of the experiment the behavior of the Minecraft-like game server is observed. For this workload, 5 player emulation nodes are used. Each node connects a new player to the server every second. The maximum number of players ranges from 25 to 300 using multiples of 25. For each combination of a Minecraft-like game and maximum number of players, the experiment is repeated 6 times.

5.3.2. Meerkat experiment workloads

A Meerkat workload consists of a collection of accesses that are submitted at a specific time. The Meerkat experiments use three different workloads. An overview of the workload parameters is shown in Table 5.4. In all experiments where a static consistency bound is used, the bound of one of the dimensions is set to a fixed value, while the others are set to infinity. While arbitrary combinations of error bounds are possible, the experiments presented here focus on the performance impact of the individual consistency dimensions.

The first workload, **stress-test**, evaluates the effect of the Conit model on the maximum throughput of the system. **stress-test** sends each node a new write as soon as it has completed processing the previous write. Because writes trigger all three consistency dimensions (in comparison to reads, which only triggers the staleness actor), this synthetic workload is the heaviest possible workload for the system. This workload is repeated 20 times for each combination of consistency bounds and number of nodes. The small bound values (0-5) have been chosen to observe scalability improvements when allowing low inconsistency in the system. The large bound values (50-250) have been chosen to observe scalability improvements when using staleness bounds that are still tolerable to players.

The second workload, **increasing**, increases the frequency of the writes over time. **increasing** sends writes with a frequency of $20 \cdot i$, where $i = 1$ at the start of the experiment, and is increased by 1 every 30 seconds. The constant of 20 writes per second is based on the Meerkat performance model, discussed in Section 4.4, where 1 player corresponds to 20 writes per second. This workload is similar to the **increasing players** workload used in the Minecraft scalability experiments, where additional players join over time. This workload has a duration of 1500 seconds, reaching a maximum of 1000 writes per second, or 50 players. This workload is used both in combination with a static consistency bound as well as a dynamic consistency bound. When a dynamic bound is used, the target synchronization throughput is set to 200 messages per second. This workload is executed once for each combination of consistency bound and number of nodes.

The final workload, **50-player trace**, evaluates a more realistic scenario by replaying messages transmitted by a Minecraft-like game server that are recorded during the Minecraft scalability experiments using the **fixed players** workload, configured to connect 50 players. In the Minecraft scalability experiments, a set of emulated players connect and interact with the Minecraft-like game server. In the **50-player trace** workload, each Meerkat node selects an emulated player and replays the `Server-EntityPositionPacket` messages this player received, by translating them into Meerkat updates. These updates are then synchronized between the multiple Meerkat nodes according to the set Dy-

Resource	Property	Value
CPU	# cores	dual 8-core
CPU	clock frequency	2.4 GHz
memory	size	64 GB
interconnect	network Architecture	FDR InfiniBand
disk	file system	network mounted storage

Table 5.5: DAS-5 node hardware configurations used for all experiments presented in this thesis.

namic Conit bounds. This scenario is similar to multiple servers communicating player locations to each other in a distributed virtual environment, which can occur when these servers share a border in a spatially-partitioned virtual world. This scenario is discussed in Section 3.3.1. Because a trace is used in which 50 emulated players connect to the Minecraft-like game server, each Meerkat node replays the position updates from $50 - 1 = 49$ other players. This means that the workload is scaled up when using multiple Meerkat nodes; running the **50-player trace** workload on four Meerkat nodes generates twice as many updates as running the same workload on two Meerkat nodes. Similar to the **increasing** workload, the **50-player trace** workload is used both in combination with a static consistency bound as well as a dynamic consistency bound. When a dynamic bound is used, the target synchronization throughput is set to 200 messages per second. This workload is executed once for each combination of consistency bound and number of nodes.

We configure the workloads to have the following properties: all workloads consist of 100% writes with a numerical weight of 1. There is always 1 defined Conit that is affected by all updates. When applied to games, the number of Dynamic Conits will likely be in the order of the number of players. However, to keep the system simple and to better observe the behavior of the Conit model, only one is used in the workloads in this thesis.

The policies used for the Meerkat experiments are **static**, **PM**, and **PM-P**. These policies are discussed in Section 3. Each policy uses a sliding window size of 10 seconds to determine the current average throughput of messages. The bounds are also updated every 10 seconds. Because we are interested in which types of tuning policies work, and not in tuning the performance of the policies, these parameters are not varied in the experiments.

5.4. Environment

Distributed systems are complex, and their performance is affected not only by their design and implementation, but also by the environment in which they are deployed. This section discusses the environment in which the Minecraft-like games and Meerkat are evaluated. First, it discusses the DAS-5: the supercomputer that is used for all experiments presented in this thesis. Second, it discusses Akka[1], a framework for building scalable distributed systems in Java and Scala, which is used to build Meerkat.

5.4.1. DAS-5 distributed supercomputer

All experiments presented in this thesis are performed on the DAS-5 distributed supercomputer [10]. The DAS-5 is a project of the Advanced School for Computing and Imaging (ASCI), and is funded by NWO/NCF. The DAS-5 is comprised of six clusters located throughout the Netherlands. It features state-of-the-art hardware, and offers special nodes that are equipped with HPC accelerators such as GPUs. The DAS-5 provides the opportunity for a large number of students and researchers to conduct experiments in a controlled and common infrastructure. Without such an infrastructure, researchers would need to use commercial alternatives. Researchers have no control over commercial infrastructures, and may not know the hardware and configurations that are used. This makes it more difficult for researchers to reproduce results. Furthermore, the lack of control also means that specialized hardware required for research might not be available.

A summary of the hardware used in the experiments presented in this thesis is shown in Table 5.5. The nodes in the DAS-5 are connected to both an Ethernet and InfiniBand network. In the experiments presented in this thesis, all nodes communicate using the high-performance InfiniBand connections. More details on the hardware of these nodes are available on the website of the DAS-5.[2]

[1] https://akka.io/
[2] http://www.cs.vu.nl/das5/

metric collected for	metric	collected by	unit
Minecraft-like game	CPU utilization	Prometheus	0-n (number of cores)
	tick frequency	Prometheus	numerical value
	relative utilization (Section 5.5.2)	Prometheus	0-100%
Yardstick	incoming/outgoing Minecraft packets	Prometheus	count
Meerkat	access throughput	Akka logging	messages per second
	sync throughput	Akka logging	messages per second
	consistency area	Akka logging	numerical value

Table 5.6: Metrics collected in experiments. All metrics are discussed in Section 5.5.1, unless indicated otherwise.

5.4.2. Akka framework

The tools available for building and debugging distributed systems are less advanced than those available for single-node systems. This means that building and debugging distributed systems is more labor intensive and error-prone. Meerkat is built on top of Akka: a framework for building scalable distributed systems in Java and Scala. Akka tries to reduce the effort needed to build scalable and reliable distributed systems. It takes care of many networking and configuration tasks, allowing the engineer to focus on the system features.

To build scalable and reliable distributed systems, Akka uses the actor model. In this model, a collection of actors run asynchronously and can be distributed across nodes. These actors perform actions based on *messages* they receive from other actors. Meerkat, a system built on top of Akka, provides its own actors that are run by the Akka framework. Akka routes the messages between the actors, taking care of synchronization, locating actors on the network, and serializing and deserializing these messages.

5.5. Metrics and data collection

Collecting data and measurements during the experiments is done either by using Prometheus or by using logging. Prometheus is the preferred method for collecting measurements. However, Prometheus has strict requirements on the format used for the data it must record. Only lists of numerical values (time-series data) with specific requirements are supported. Furthermore, Prometheus uses a polling approach which does not allow precise recording of when events are triggered.

Yardstick uses Prometheus for both hardware metrics, such as CPU and memory utilization, and system metrics, such as the rate of incoming and outgoing packets. Meerkat uses logging to collect data about the system itself, such as the number of participants in the Conit, the current consistency bounds, and the sending and receiving of synchronization messages. Hardware metrics such as CPU and memory utilization are collected using Prometheus.

5.5.1. Main metrics

This section discusses one by one the main metrics used to report the experiment results. An overview of the metrics that are used is shown in Table 5.6.

The number of utilized cores is a metric to quantify the CPU utilization. Instead of a percentage, where 0% means an idle CPU and 100% means a fully utilized CPU, a number between 0 and n is used, where n is the total number of CPU cores in the node that is used for the experiment. It is straightforward to calculate the normalized CPU utilization from the number of utilized cores: divide by the total number of cores and multiply by 100. The number of utilized cores is more informative than the normalized value. If, for example, the experiment results show that the CPU utilization does not exceed 1 core, this suggests that the system does not support parallel processing. This observation would be more difficult to make when the normalized value is used, and a value such as 12.5% is observed (which corresponds to 1 utilized core, if the total number of cores is 8).

Tick frequency is the number of ticks completed by a Minecraft-like server per second. During each tick, the server completes a simulation step of the virtual world. The value of the tick frequency is a constant determined by the game designers. All Minecraft-like games evaluated in this thesis use a tick frequency of 20Hz.

The relative utilization is a metric that is related to the tick frequency. When using a fixed tick frequency, there is a fixed amount of time between two consecutive ticks. The relative utilization quantifies

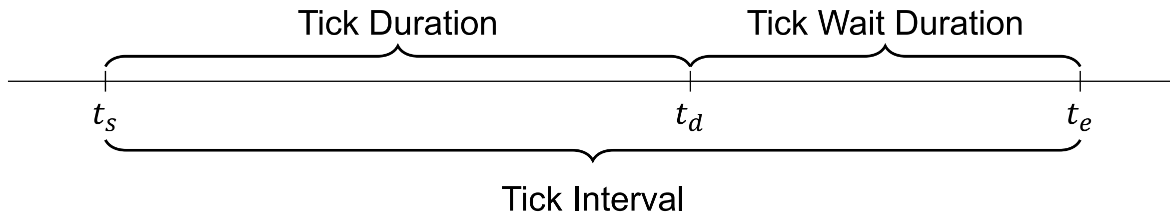


Figure 5.2: Conceptual overview of a Minecraft server tick. The Tick Duration denotes the time spent by the server simulating the virtual world. The Tick Wait Duration denotes the time spent by the server waiting before the next tick is started.

how much of that time is used by the server to perform the virtual world simulation step. If this value exceeds 100%, the processing takes more time than allocated and consecutive ticks are delayed. This suggests that the server is overloaded. This metric is explained in more detail in Section 5.5.2.

The Minecraft packet throughput captures the throughput of both the incoming and outgoing packets of the Minecraft server. This metric does not capture the throughput of *Ethernet* packets, but of Minecraft packets. These are application-level messages such as `ServerEntityPositionPacket`, which the server uses to indicate the position of an entity to a client, and `ServerChuckDataPacket`, which the server uses to communicate the layout of the virtual world to the client. To reduce the number of modifications needed to the Minecraft-like game server's code, the message throughput is captured by Yardstick's player emulation module.

The player update throughput is the metric used to report the results from the Meerkat experiments. It reports how many updates are accepted by Meerkat per second. These updates are part of the workload, which in a real-world scenario are caused by the players connected to the Minecraft-like game. A higher player update throughput indicates that more player updates can be processed in one second, and indicates better scalability.

The synchronization throughput is used to report the amount of synchronization between Meerkat nodes. To bound inconsistency, Meerkat needs to synchronize player updates received on one node to the other nodes. This means sending synchronization messages. This metric shows the total number of synchronization messages exchanged per second across all Meerkat nodes. Exchanging and processing synchronization packets is overhead caused by using a distributed system. Therefore, a lower synchronization throughput indicates better scalability.

The consistency area is defined as the integral of the consistency bound over time. The consistency area is computed using the trapezoidal rule, because it can be directly applied to discrete data. The consistency bound can be automatically adjusted by the dynamic bound policies described in Section 4.3.1. At any point in time, the consistency bound indicates the maximum allowed inconsistency in the system. The consistency area is captured to observe the behavior of the dynamic bound policies over time. The consistency area is equal to the average consistency bound multiplied by the duration of the experiment. Because all dynamic bound policies are compared using the same **50-player trace** workload, these experiments have the same duration. This means that the consistency area is proportional to the average inconsistency, but is easier to visualize.

5.5.2. Relative utilization metric (Jerom van der Sar)

To perform scalability measurements, Yardstick defines its own scalability metric based on the server tick rate. A *tick* is a simulation step of a game server that corresponds to one execution of the game loop. Most commercial games use tick-based servers that simulate (or tick) the virtual world at a regular interval.

During each iteration of the game loop (also known as a server *tick*), Minecraft processes client packets, updates entities and the environment, and transmits the new state to clients. Ticks are started at a constant frequency, called the *tick frequency*, equal to 20Hz. From this, the interval between two consecutive ticks, the *tick interval*, can be calculated to be equal to 50ms. Tick processing completes after a variable amount of time. The length of this period is called *tick duration*. Subsequently, the game loop pauses until the next tick should be started. The length of this period is equal to the tick interval minus the tick duration, and is referred to as the *tick wait duration*. Figure 5.2 illustrates a game tick. Here t_s indicates the start time of a tick, t_d indicates the start of the tick wait period, and t_e indicates the end of the tick and the start of the next tick. This is a high-level model of a tick that enables

reasoning about the performance of the game server without adding implementation-specific details. For instance, although the main operations that occur within a tick are known, the order in which these take place and the level of concurrency used depend entirely on the implementation.

From the Minecraft game loop, we determine a new metric, the *relative utilization*, which may be analogous to the utilization of the server process. We define the relative utilization as a fraction of the tick duration over the tick interval. Mathematically, this corresponds to:

$$U_r = \frac{t_d - t_s}{t_e - t_s}$$

U_r represents the relative utilization, a fraction where $0 \leq U_r \leq 1$ under normal circumstances. t_d, t_s, and t_e should be interpreted as in Figure 5.2, and are measured in millisecond precision. The tick interval in Minecraft is equal to 50ms. Considering this, the formula above can be rewritten as:

$$U_r = \frac{t_d - t_s}{50}$$

When U_r approaches 1, the tick wait duration is low and there is little leeway for additional server load. When U_r approaches zero, little to none of the capacity of the server is used. This metric effectively allows us to determine when a server is overloaded, and when there is capacity for higher server load.

5.5.3. Akka Logging

Meerkat uses logging to perform measurements during the experiments. The logging is implemented using Akka and uses dedicated actors to process log messages. Each line that should be logged is sent to a logging actor as a message. These actors are run asynchronously run by Akka to reduce the IO wait for the rest of the system. To collect the logs in a single location, the nodes in the experiment write the logs to a network mounted disk on the DAS-5.

5.5.4. Prometheus monitoring

Unfortunately, no standardized solution exists for collecting data from a multitude of metrics in a distributed system. This gives researchers the responsibility to find, deploy, configure, and operate their own set of tools. This is a task that we grudgingly accept, but take very seriously nonetheless. In the experiments presented in this thesis, a significant amount of the measurements is conducted using the Prometheus monitoring framework[3]. This framework is relatively new and is comparable to systems such as InfluxDB[4] and Graphite[5]. One major advantage of Prometheus over these other systems is that the application comes in a single binary with all its dependencies. This removes the need to install software on the nodes used for the experiments and simplifies deployment significantly.

Prometheus consists of multiple smaller applications that fulfill different roles. The main application (from here on called *Prometheus*) 'scrapes' target applications (from here on called *targets*), pulling data from different metrics and storing them in a single database. The targets monitor the system on which they run and offer the data this yields to Prometheus. For all experiments in this thesis, Prometheus runs on a separate node that has no other tasks than to log the data from the targets. A target runs on each of the other nodes involved in the experiment. Collecting game-specific data is possible by using the Prometheus API that allows arbitrary programs to expose data to the main application. An overview of the collected metrics can be found in Table 5.6.

Network activity is also monitored in the experiments presented in this thesis. Because Prometheus monitors the targets through networking, this may affect the measurements. Because the nodes in the DAS-5 are connected to both InfiniBand and Ethernet, we eliminate the effect these measurements have on the observed network activity by running the monitoring activities over Ethernet, while the system under test communicates internally using InfiniBand. Only the network device used by the system under test is considered in the experimental results.

[3]`https://prometheus.io/`
[4]`https://www.influxdata.com/`
[5]`https://graphiteapp.org/`

6

Experimental results

This chapter discusses the experiments and results for both the Minecraft and Meerkat experiments described in Chapter 5. For both sets of experiments, first the goal of each set is briefly discussed, followed by the main findings and an analysis of the results. A detailed discussion of the experiment setup can be found in Chapter 5. For a complete overview of the experiments, see Table 5.1. Next, the experiment results are presented and analyzed. Not all experiment results are shown in this chapter. For the omitted results, see Appendix B.

6.1. Minecraft scalability experiments

The two goals of the Minecraft scalability experiments are to evaluate the current scalability of Minecraft-like games, and to find out if Dynamic Conits can increase the scalability of these games. The first experiment, which shows the effect of the number of players on the tick frequency and relative utilization, covers the first goal, while the latter two experiments, which show the effect of the number of players on the CPU and network usage, cover the second goal.

6.1.1. Main findings

1. Minecraft-like games scale to hundreds of players. For higher numbers of players, the tick frequency is reduced. This motivates to look for novel scalability techniques.

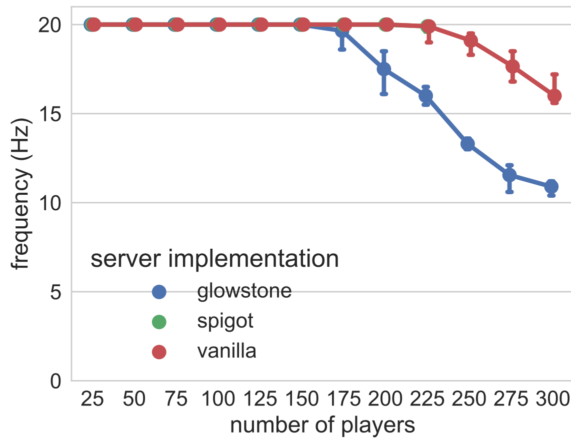
2. The CPU does not form a bottleneck for scalability. This motivates to reduce the load on resources other than the CPU to increase scalability.

3. Minecraft-like games transmit large numbers of packets which increase with the number of players. This motivates to use Dynamic Conits to reduce the number of messages by allowing bounded inconsistency between nodes.

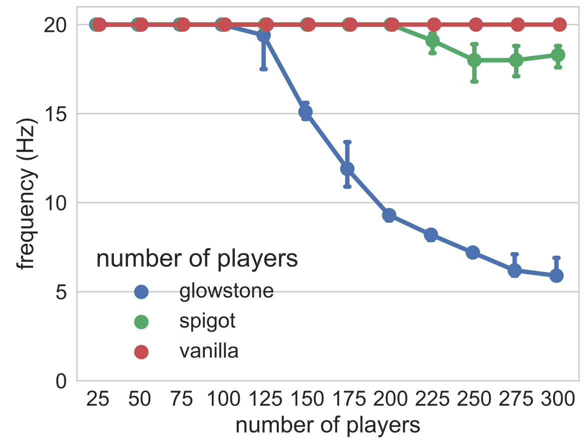
6.1.2. Analysis of finding 1

Minecraft-like games scale to hundreds of players. Using the tick frequency and novel relative utilization metrics, we observe that all the evaluated games are overloaded under at least one of the workloads. In all figures discussed in this section, the color of the curve indicates the Minecraft-like game used in the experiment. We now proceed to describe this result.

Figure 6.1 depicts the tick frequency of the Minecraft server while under the **increasing players** and **fixed players** workloads. The horizontal axis depicts the number of connected players and the vertical axes depict the tick frequency. The Minecraft-like games are all configured to run at a tick frequency of 20Hz, or 50ms per tick. Figure 6.1a shows that the tick frequency from both **Glowstone** (blue curve) and **vanilla** (red curve) drops below 20Hz during the **increasing players** workload. The tick frequency of **Glowstone** drops below 20Hz when connecting 175 players or more. The tick frequency of **vanilla** drops below 20Hz when connecting 225 players or more. The tick frequency from **Spigot** is hidden behind the red curve from **vanilla** and is limited to 225 players. The tick frequency indicates the number of simulation steps of the virtual world per second. Decreasing this value means that the

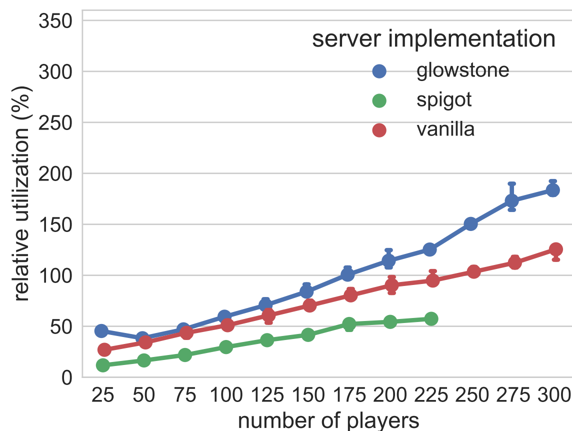


(a) Effect of the **increasing players** workload

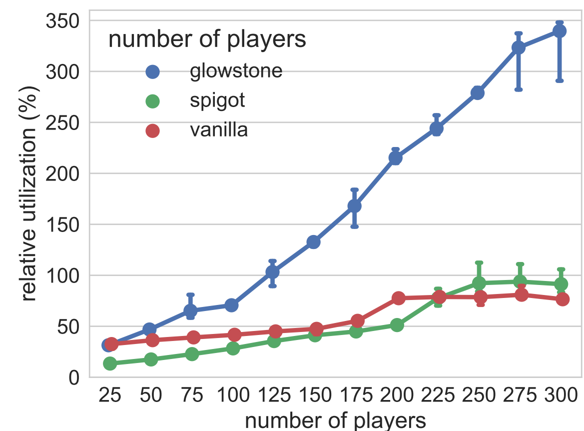


(b) Effect of the **fixed players** workload

Figure 6.1: Tick frequency of Minecraft-like game servers when varying the number of players. The markers indicate the median value, and the whiskers indicate a 95% confidence interval.



(a) Effect of the **increasing players** workload



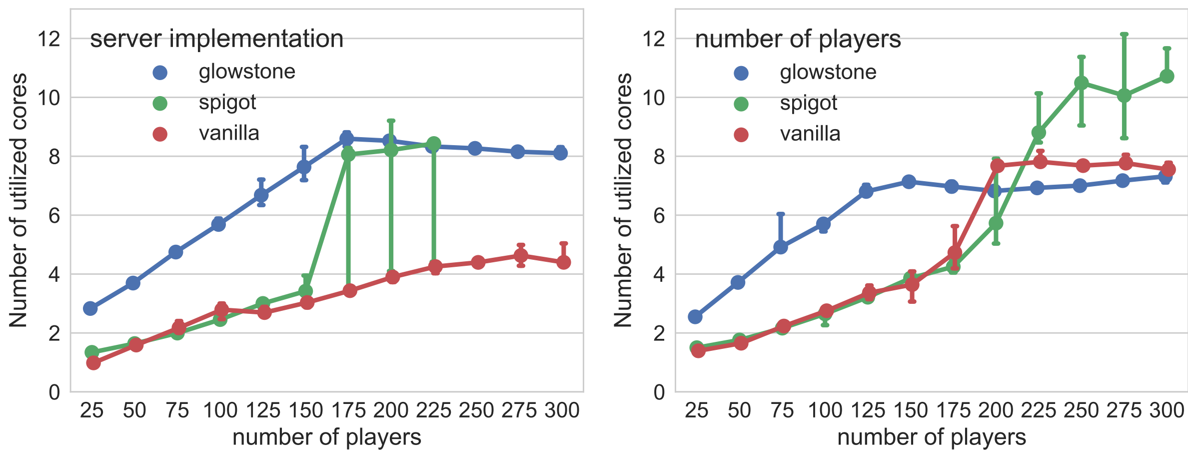
(b) Effect of the **fixed players** workload

Figure 6.2: Relative utilization of Minecraft-like game servers when varying the number of players. The markers indicate the median value, and the whiskers indicate a 95% confidence interval.

simulation of the virtual world slows down, decreasing the overall game speed. Figure 6.1b shows that both **Glowstone** (blue curve) and **Spigot** (green curve) drop below the tick frequency of 20Hz for large numbers of players while under the **fixed players** workload. **Glowstone** drops below 20Hz when connecting 125 players or more, while **Spigot** drops below 20Hz when connecting 225 players or more.

Figure 6.2 depicts the relative utilization of the Minecraft-like game servers for both workloads. The horizontal axes depict the number of players connected to the server and the vertical axes depict the relative utilization. Figure 6.2a shows that the relative utilization of both **Glowstone** (blue) and **vanilla** (red) exceeds 100% during the **increasing players** workload. The relative utilization of **Glowstone** exceeds 100% when connecting 175 players or more. The relative utilization of **vanilla** exceeds 100% when connecting 225 players or more. The relative utilization indicates how much time within a server tick is used by the server executing the game loop. A relative utilization larger than 100% means that ticks exceed their maximum duration, delaying the execution of the next server tick. Figure 6.2b shows that the relative utilization of both **Glowstone** and **Spigot** exceed 100% during the **increasing players** workload. The relative utilization of **Glowstone** exceeds 100% relative utilization when connecting 125 players or more. The relative utilization of **Spigot** exceeds 100% relative utilization when connecting 250 players or more.

Combining Figure 6.1 and Figure 6.2 shows that the decrease in tick frequency coincides with exceeding a relative utilization of 100% for each of the Minecraft-like games. This suggests that the tick



(a) Effect of the **increasing players** workload.

(b) Effect of the **fixed players** workload.

Figure 6.3: CPU utilization of Minecraft-like game servers when varying the number of players. The markers indicate the median value, and the whiskers indicate a 95% confidence interval.

frequency of the servers decrease because the duration of each individual tick is larger than its maximum duration, delaying the execution of the next tick, hereby reducing the tick frequency. Which games exceed 100% relative utilization depends on the workload. Figure 6.2a shows that both **Glowstone** and **vanilla** exceed 100% relative utilization under the **increasing players** workload. Figure 6.2b shows that both **Glowstone** and **Spigot** exceed 100% relative utilization under the **fixed players** workload This indicates that different Minecraft-like games react differently to a specific workload: **Spigot** does not exceed 100% relative utilization under the **increasing players** workload, but it does under the **fixed players** workload. **vanilla** exceeds the 100% relative utilization under the **increasing players** workload, but it does not under the **fixed players** workload.

Currently, Minecraft-like games scale to hundreds of players while state-of-the-art MMOG games scale to thousands of players. This gap, of multiple orders of magnitude, provides a clear motivation for researchers to look for novel techniques to increase the scalability of Minecraft-like games. Meerkat implements Dynamic Conits, which can be one of these techniques.

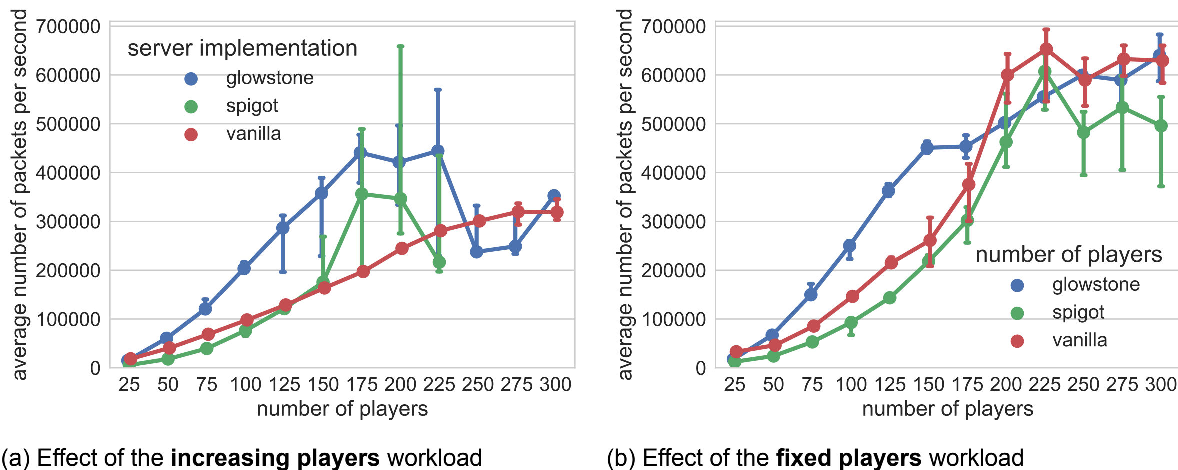
6.1.3. Analysis of finding 2

The CPU does not form a bottleneck for scalability. None of the games fully utilizes the CPU in any of the experiments. Figure 6.3 shows the number of utilized cores over the number of connected players for both workloads. The horizontal axes show the number of connected players, and the vertical axes show the number of utilized cores of the Minecraft-like game server node. We now proceed to describe this result.

Figure 6.3a shows the number of utilized cores over the number of players while under the **increasing players** workload. Between 25 and 150 players all games show a trend of an increasing CPU utilization for an increasing number of players. Between 150 and 300 players, the CPU utilization of both **Glowstone** and **vanilla** stops increasing. The CPU utilization of **Glowstone** (blue curve) keeps increasing until 175 players. For a larger number of players the CPU utilization decreases slowly and seems to diverge towards 8 cores. The CPU utilization of **vanilla** (red curve) keeps increasing until 275 players. At 300 players the utilization is slightly below the utilization for 275 players, but the variance of the utilization is increased.

Figure 6.3b shows the number of utilized cores over the number of players while under the **fixed players** workload. Between 25 and 150 players all games show a trend of increasing CPU utilization for an increasing number of players. Between 150 and 300 players, the CPU utilization of both **Glowstone** and **vanilla** stops increasing. The CPU utilization of **Glowstone** (blue curve) keeps increasing until 150 players. For a larger number of players the CPU utilization seems to be roughly constant at a value between 7 and 8 cores. The CPU utilization of **vanilla** (red curve) keeps increasing until 200 players. For larger numbers of players, the CPU utilization stays slightly below 8 cores.

Combining Figure 6.2 and Figure 6.3 shows that while the relative utilization exceeds 100% for all



(a) Effect of the **increasing players** workload

(b) Effect of the **fixed players** workload

Figure 6.4: Outgoing packet throughput on Minecraft-like game servers when varying the number of players. The markers indicate the median value, and the whiskers indicate a 95% confidence interval.

games and continues to increase with the number of players, the number of utilized cores does not, and never reaches 32 (the total number of cores in the node). This shows that the game is spending more time on each game tick without spending additional time on the CPU. We consider three explanations for this observation. The simplest explanation is that the number of computations within a game tick is not a system bottleneck. The second explanation is that the number of computations does form a bottleneck without reaching 100% CPU utilization because the game cannot be parallelized further due to inherently sequential computations. The third explanation is that the CPU does not reach 100% utilization because the games could benefit from increased parallel processing, but are not designed to do so.

Only in the second explanation does the CPU form a scalability bottleneck. However, this explanation seems unlikely because the players emulated by Yardstick move around the virtual world without interacting with the other players. This means that their player states could be updated in parallel. If CPU is not the scalability bottleneck of Minecraft-like games, other resources must be responsible for further extending the tick duration. If the network is one of these resources, Dynamic Conits can improve game scalability because they reduce the network traffic. Allowing bounded inconsistency does require additional bookkeeping and thus additional CPU usage. However, none of the games utilizes all CPU cores and the bookkeeping required by Dynamic Conits can run concurrently with the game itself. Therefore resources are available for this bookkeeping.

6.1.4. Analysis of finding 3

Minecraft-like games transmit a large number of packets which increases with the number of players. All servers show an increasing number of packets sent per second until they become overloaded, at which point the servers show erratic behavior. We now proceed to describe this result.

Figure 6.4a shows the number of packets per second transmitted by the Minecraft-like game server over the number of players while under the **increasing players** workload. The horizontal axis shows the number of players, and the vertical axis shows the number of packets per second sent by the server. The number of outgoing packets per second sent by **Spigot** (green curve) increases until 150 players. For larger numbers of players, the variance of the number of packets sent per second significantly increases and the median values (the markers on the curve) stop increasing. The same is true for **Glowstone** (blue curve) after 175 players. The number of outgoing packets per second sent by **vanilla** (red curve) increases until 275 players. Only for 300 players does the number of packets sent per second decrease slightly.

Combining Figure 6.2a and Figure 6.4a shows that for both **Glowstone** and **vanilla**, the decrease in number of packets sent per second coincides with a relative utilization greater than 100%. The relative utilization greater than 100% indicates that the server is overloaded and cannot keep up with the workload.

Figure 6.4b shows the number of packets per second transmitted by the Minecraft-like game server

packet name	frequency	percentage of total size	average packet size	σ	min	25pc	50pc	75pc	max
ServerEntityPositionPacket	45%	2%	9	35	0	9	9	9	32274
ServerEntityHeadLookPacket	18%	0%	3	38	0	3	3	4	32726
ServerEntityPositionRotationPacket	16%	1%	11	40	0	11	11	12	31618
ServerEntityVelocityPacket	5%	0%	8	146	0	7	8	8	63744
ServerEntityTeleportPacket	5%	1%	29	66	0	29	29	29	31650
ServerEntityMetadataPacket	3%	0%	25	459	0	7	7	32	62696
ClientPlayerPositionPacket	3%	0%	25	72	0	25	25	25	31622
ServerEntityStatusPacket	1%	0%	5	0	0	5	5	5	34
ServerSpawnObjectPacket	1%	0%	68	663	0	55	55	55	62236
ServerPlayBuiltinSoundPacket	1%	0%	23	176	0	22	22	22	31618
ServerChunkDataPacket	1%	93%	31643	9323	0	31066	31080	31613	251826
ServerMultiBlockChangePacket	1%	0%	37	284	0	20	26	41	32704
ServerBlockChangePacket	1%	0%	11	147	0	10	10	10	31614

Table 6.1: Network packet distribution for **vanilla** experiment with 200 players using the **fixed players** workload.

over the number of players while under the **fixed players** workload. The horizontal axis shows the number of players, and the vertical axis shows the number of packets per second sent by the server. All games show an increasing number of packets sent for an increasing number of players. The number of packets per second sent by **Glowstone** (blue curve) keeps increasing with the number of players. The number of packets per second sent by **Spigot** (green curve) and **vanilla** (red curve) also increases with the number of players, but stops doing so after 225 players.

Combining Figure 6.2b and Figure 6.4b shows that the reduced increase in number of packets sent per second by **Glowstone** coincides with the game exceeding 100% relative utilization. This can be seen by the blue curve exceeding 100% in Figure 6.2b, and the change in slope in the blue curve in Figure 6.4b. The reduction in number of packets sent by **Spigot** also coincides with the game exceeding 100% relative utilization. This can be seen by the whiskers of the green curve exceeding 100% in Figure 6.2b, and the erratic and lower values of the green curve in Figure 6.4b. The reduction in packets sent by **vanilla** does not coincide with the game exceeding 100% relative utilization.

However, when combining Figure 6.3b and Figure 6.4b we observe that the reduction in number of packets sent coincides with a stagnation in the CPU utilization of the game. **vanilla** seems unable to use more than 8 CPU cores. This can be seen in Figure 6.3b, where for 200 players or more the red curve is very close to, but stays below, a utilization of 8 cores.

Table 6.1 shows a summary of the network activity from one of the repetitions of the **fixed players** workload. Most network traffic is caused by the server communicating player location data to the clients. Only packets that have a frequency of 1% or more have been included in the table. All packet names that start with ‘Server’ are sent by the server. Similarly, packet names that start with ‘Client’ are sent by the client. The first row of the table shows that 45% of the packets exchanged between the server and clients are `ServerEntityPositionPacket` packets. These packets communicate the location of an entity from the server to a client. The top five rows in the table show packets related to entity positioning. These packets account for almost 90% of all network traffic between the server and the clients. The location of an entity in the virtual world is not always relevant to other players, motivating efforts to research novel techniques to allow (bounded) inconsistency between players and decrease the amount of data that needs to be communicated. By defining a Dynamic Conit on the data that specifies the location of entities in the virtual world, the number of packets that need to be communicated can be reduced.

6.1.5. Discussion

The experiments in this thesis that evaluate the scalability of Minecraft-like games use a player emulation model that is part of the Yardstick benchmarking tool and is based on player movement in Second Life. However, a more accurate movement model called SAMOVAR is available based on player movement traces obtained from World of Warcraft [60]. Changing the player behavior, including player movement, could affect the workload on the server and the number of players that it can support. To realistically simulate player behavior, more is needed than an accurate player movement model. Real players of Minecraft-like games are likely to modify the virtual world, which is a feature that other games (such as Second Life and World of Warcraft) do not offer. Changing the virtual world requires additional communication, and likely increases the amount of `ServerBlockChangePackets` and `ServerMultiBlockChangePackets` transmitted by the game.

Similar to entity locations, modifications to the virtual world are not always relevant to other players, motivating efforts to research novel techniques to allow (bounded) inconsistency between players and decrease the amount of data that needs to be communicated. Dynamic Conits reduce the amount of network traffic by allowing bounded inconsistency between nodes. This would affect the number of `ServerBlockChangePackets`, `ServerMultiBlockChangePackets`, and `ServerEntityPositionPackets` because these packets are used to update the state of the game at the client. Dynamic Conits can also be used to reduce the number of `ServerChuckDataPackets`. These packets are used to communicate the state of the virtual world to the clients. These packets are retransmitted each time a client connects, even if this client was recently connected to the server. If a Dynamic Conit is defined that is affected by changes to the virtual world, clients could request only those updates from the server, instead of the complete state of the virtual world. In cases where a client has not connected to a server for a long period of time, the client has likely missed many updates. Here communicating merged updates or the state of the virtual world might be more efficient. This motivates researching the combination of such techniques with Dynamic Conits, as introduced in Section 3.2.

Both experiments experienced problems when connecting players to the servers. During the experiment running the **increasing players** workload, **Spigot** regularly disconnected or refused some of the players that tried to connect. This explains why the green curve in the figures that visualize the **increasing players** workload does not continue until 300 players: this number of players was never reached before the end of the experiment. During the experiments running the **fixed players** workload, the join interval was set at 1 second. This short interval was problematic for all of the Minecraft-like game servers. All servers occasionally took longer than 1 second to inform Yardstick that the new player was connected. However, if the server does not respond in time, Yardstick considers the attempt to let the new player join as failed, sending a `Disconnect` packet to the server. Such packets seem to be ignored by the server while the player is still logging in (and thus not officially connected). This results in the player still connecting to the server eventually, but being considered failed, and thus left idle, by Yardstick. In future experiments this interaction between Yardstick and the Minecraft-like games should be improved to have more control over the (number of) connected players.

6.2. Meerkat experimental evaluation

From the results presented in the previous section, we conclude that there is ample room for improvement of the scalability of Minecraft-like games. The Minecraft scalability experiments show that the games scale to hundreds of players. The CPU is never fully utilized, but the network utilization of the games are high, and continues to increase with the number of players. To reduce the network utilization of these games, we propose a novel technique: Dynamic Conits. Dynamic Conits allow bounded inconsistency between nodes, reducing the amount of synchronization required. This section evaluates Meerkat, a prototype system that implements Dynamic Conits, in two steps.

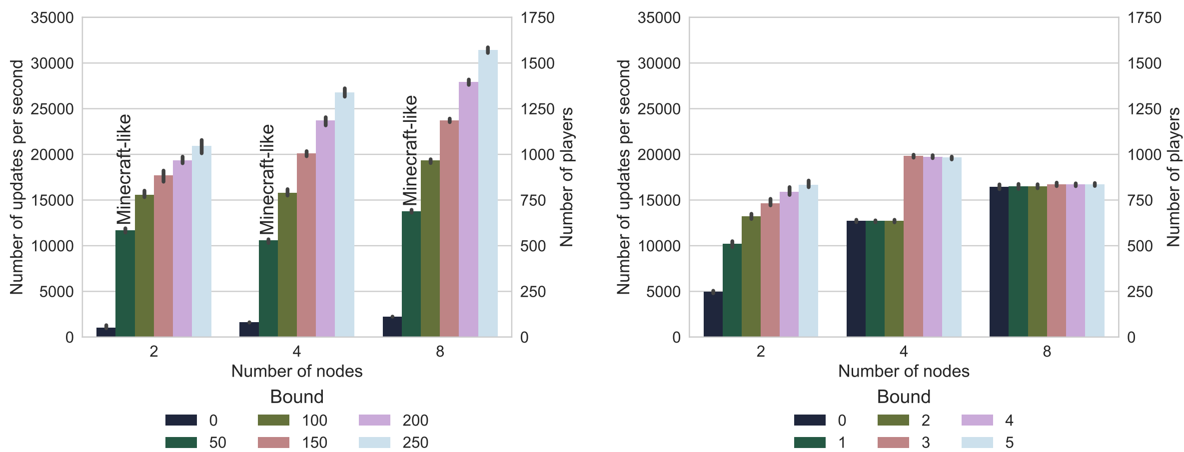
The goal of the first two experiments is to evaluate the impact of different consistency bounds of the traditional Conit model on update throughput and synchronization throughput. A higher update throughput indicates that a higher number of updates can be processed in the same amount of time. The Conit model can increase the update throughput by reducing the frequency of synchronization. For Minecraft-like games, increasing the update throughput can mean increasing the number of events the game can process in a single tick. If the number of events per time unit is constant, a higher update throughput can reduce the tick duration, reducing the relative utilization.

The goal of the remaining experiments is to evaluate the features of the Dynamic Conit model, and their ability to improve the performance of Minecraft-like games. Specifically, the experiments focus on dynamically changing Dynamic Conit bounds, and the effect of letting new nodes join a system under heavy load. Limiting the amount of synchronization messages by dynamically changing consistency bounds can prevent Minecraft-like games from exceeding 100% relative utilization. This can allow the game to scale to a larger number of players at the cost of increased inconsistency.

6.2.1. Main findings

1. Using Dynamic Conits' bounded inconsistency can double the number of supported players.

2. Using Dynamic Conits' bounded inconsistency can significantly decrease the amount of synchronization between nodes.



(a) Effect of staleness bounds on update throughput (b) Effect of numerical bounds on update throughput

Figure 6.5: Effect of staleness and numerical consistency bounds on update throughput. The top of the bars indicate the median value, and the whiskers indicate a 100% confidence interval.

3. Dynamically changing the Dynamic Conit bounds can prevent high workloads creating large numbers of synchronization messages, and can reduce inconsistency under low workloads.

4. Adding new nodes to a Dynamic Conit during high workload can cause system stalls.

5. Dynamic bound policies can control how much consistency is reduced and how much consistency is lost.

6.2.2. Analysis of finding 1

Setting higher consistency bounds using Dynamic Conits' bounded inconsistency can double the number of supported players as calculated by Meerkat's performance model discussed in Section 4.4. This is a promising result for Minecraft-like games because it suggests that Dynamic Conits can also significantly increase the maximum number of players supported in Minecraft-like games. We now proceed to describe this result.

Overall, Figure 6.5 shows that the system throughput or number of supported players can increase significantly while keeping the consistency bounds low, limiting the amount of inconsistency. This is a promising result for Minecraft-like games, because large inconsistencies could decrease the experience of players, and would thus not be a viable solution to scale these games. We now discuss each sub-part of Figure 6.5 in turn.

Figure 6.5a shows the throughput or number of players supported by Meerkat as a function of the number of nodes and the staleness bound. The horizontal axis shows the number of nodes sharing the Dynamic Conit, and each group shows a number of staleness bounds. The bound of 50ms corresponds to Minecraft-like games; this is because these games synchronize every tick, and their tick duration is 50ms. The horizontal axis shows the number of updates per second processed by Meerkat. It also shows the corresponding number of players based on Meerkat's performance model, where each player provides a workload of 20 messages per second. The figure shows that higher staleness bounds significantly increase the throughput or number of players supported by the system. The effect of the increased staleness bound is greater for a larger number of nodes. When using 2 nodes (left group) the number of players increases from roughly 600, when using a consistency bound of 50ms, to roughly 1000, when using a consistency bound of 250ms. When using 4 or 8 nodes (middle and right group, respectively), a staleness bound of 250ms achieves more than 2 times the throughput, compared to a bound of 50ms. This is a promising result for Minecraft-like games, because delays of 250ms still allow player actions without negative impact on gameplay experience [17].

Similarly to Figure 6.5a, Figure 6.5b shows the throughput or number of players supported by Meerkat. The Figure is of the same type as Figure 6.5a, except the numerical error bound is used instead of the staleness bound. Here, there is no Minecraft-like game equivalent, because these games

do not use numerical error bounding. We analyze this figure in two steps. First, we analyze the behavior observed when sharing the Dynamic Conit with 4 nodes (the middle group) and 8 nodes (the right group). Second, we analyze the behavior observed when sharing the Dynamic Conit with 2 nodes (the left group).

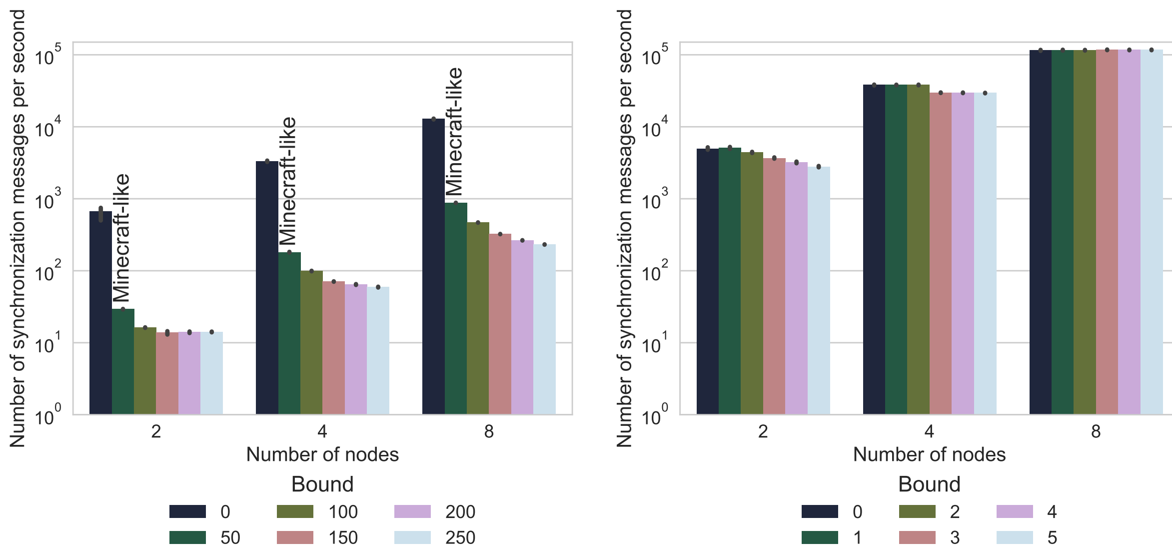
The middle and right groups in Figure 6.5b show a Dynamic Conit shared between 4 nodes and 8 nodes, respectively. When sharing a Dynamic Conit with 4 nodes, the same throughput is observed for any numerical error bound between 0 and 2. When a numerical error bound of 3 is used, the throughput increases. Similar to a numerical error bound between 0 and 2, using a numerical error bound between 3 and 5 results in equal system throughput. This is expected behavior, because the synchronization behavior for a Dynamic Conit with a numerical bound is not only determined by the value of this bound, but also by the number of nodes sharing the Dynamic Conit. The numerical error bound specifies how many writes may be unseen by any node. If a Dynamic Conit is shared with 4 nodes, then for each node there are 3 other nodes that may have seen writes the 4th node has not. Therefore, with a numerical error bound of 3, each node can accept 1 update without synchronizing, while guaranteeing the consistency bound. However, if a numerical error bound of 2 is used, accepting 1 update without synchronizing can cause the consistency bound to be exceeded. This means that, if 3 nodes share a Dynamic Conit with a numerical error bound of 2, each node must synchronize every update it receives immediately. This effectively rounds the numerical error bound of 2 down to 0. Similarly, for a Dynamic Conit shared between 8 nodes (the right group), no increase in throughput is observed when increasing the numerical error bound. This is because all numerical error bounds shown in Figure 6.5b are effectively rounded down to 0. For a Dynamic Conit shared with 8 nodes, all numerical error bounds lower than 7 are effectively rounded down to 0. See Section 3.4.2 for more details on this behavior.

The left group in Figure 6.5b shows a Dynamic Conit shared between 2 nodes. When sharing a Dynamic Conit with 2 nodes, the behavior of the numerical error bound is more intuitive. A numerical error bound of 0 (black bar) corresponds to synchronizing every update with the other node immediately, a numerical error bound of 1 (dark green bar) corresponds to synchronizing with the other node every 2 updates, and so on. Comparing these bars shows that a numerical error bound of 1 doubles the throughput to 10000 updates per second. Further increasing the numerical bound shows diminishing returns, as can be seen from the sub-linear increase in bars in the left group. However, a numerical error bound of 4 results in a throughput that is more than three times higher than the throughput observed when using a numerical error bound of 0. This is a promising result for Minecraft-like games, because low inconsistency can significantly increase the throughput. For instance, a player that sees another player building a structure in the distance may not be interested to receive updates about every change made to that structure. Here, a numerical error bound between the client and the server can be used to synchronize only every three or four updates.

6.2.3. Analysis of finding 2

Setting higher consistency bounds significantly reduces the number of synchronization messages exchanged between nodes at the cost of increased inconsistency. This is a promising results for Minecraft-like games because it can reduce the amount of synchronization between servers and clients. For our analysis, we focus on the application of Dynamic Conits between servers, as described in Section 3.3.1. Although Minecraft-like games currently do not have distributed server setups, this is likely to change in the future because all state-of-the-art MMOGs use such setups. We now proceed to describe this result.

Figure 6.6a shows the number of synchronization messages per second exchanged by the nodes as a function of the number of nodes and the staleness bound. The horizontal axis shows the number of nodes sharing the Dynamic Conit, and each group shows a number of staleness bounds. The vertical axis shows the number of synchronization messages per second. Note that the vertical axis uses a logarithmic scale. Comparing synchronizing every update as soon as it arrives (black bar) with synchronizing updates once every 50ms (dark green bar) shows that Minecraft-like games, which communicate once per 50ms, already reduce synchronization significantly compared to systems that use strict consistency. But Dynamic Conits can reduce the amount of synchronization further. For instance, a Minecraft-like game running on 8 nodes would exchange close to 1000 messages per second to synchronize updates (right group, dark green bar). When using a staleness bound of 250ms, this synchronization can be reduced to roughly 200 messages per second (right group, light gray bar).



(a) Effect of staleness bounds on synchronization throughput

(b) Effect of numerical bounds on synchronization throughput

Figure 6.6: Effect of staleness and numerical consistency bounds on synchronization message throughput. The top of the bars indicate the median value, and the whiskers indicate a 100% confidence interval.

Similarly to Figure 6.6a, Figure 6.6b shows the number of synchronization messages per second exchanged by the nodes. The figure is similar to Figure 6.6a, except that a numerical error bound is used instead of a staleness bound. Here, there is no Minecraft-like game equivalent, because these games do not use numerical error bounding. The number of synchronization messages exchanged is much higher than in Figure 6.6a, because the numerical error bound that is used is low. Because this experiment uses the **stress-test** workload, the most intensive possible workload, the number of synchronization messages is also high. When sharing the Dynamic Conit across two nodes (left group), the numerical error bound can significantly decrease the amount of synchronization messages exchanged between the two servers. The bars show a linear decrease in height. Because the vertical axis has a logarithmic scale, this means the linear increase in numerical error bound exponentially decreases the number of synchronization messages between the nodes. When running the Dynamic Conit on 4 or 8 nodes, the small numerical error bounds have little effect. This is consistent with the results shown in Figure 6.5b and the Dynamic Conit performance model described in Section 3.4, which describes that a numerical error bound becomes less effective when increasing the number of nodes.

Figure 6.6 shows the number of synchronization messages exchanged between nodes for a varying number of nodes, using multiple staleness and numerical error bounds. The result shows that the consistency bound can be used to control the number of synchronization messages. This is a promising result for Minecraft-like games, because it gives the game developer the tools to quantify how much consistency must be reduced to lower the synchronization overhead.

6.2.4. Analysis of finding 3

Dynamically changing the Dynamic Conit bounds can prevent high workloads creating large numbers of synchronization messages, and can reduce inconsistency under low workloads. We now proceed to describe this result.

Figure 6.7 shows the player update– and synchronization throughput in Meerkat when running the **increasing** workload on two nodes while using a fixed numerical error bound of 5. This workload is completely synthetic and the configuration can be compared a scenario where two servers run a distributed virtual world and share their data. Each server is connected to an increasing number of players. For this workload, the static bound of 5 is not enough to keep the number of synchronization messages low. The horizontal axis shows time. The vertical axis shows the player update– and synchronization throughput in messages per second in the top figure, and the numerical error bound in the bottom figure. The number of player updates increases over time, which can be seen from the upward

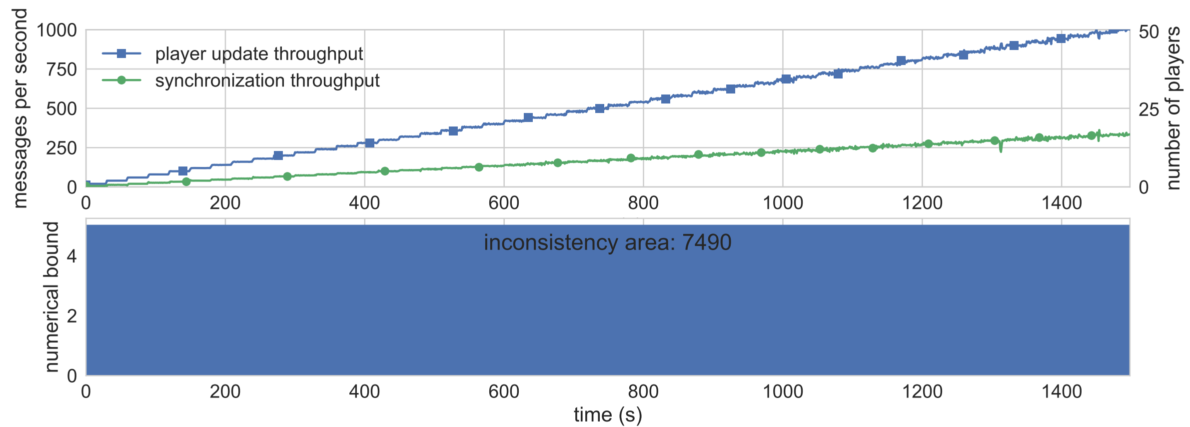


Figure 6.7: Meerkat throughput when using a fixed numerical error bound of 5 during the **increasing** workload.

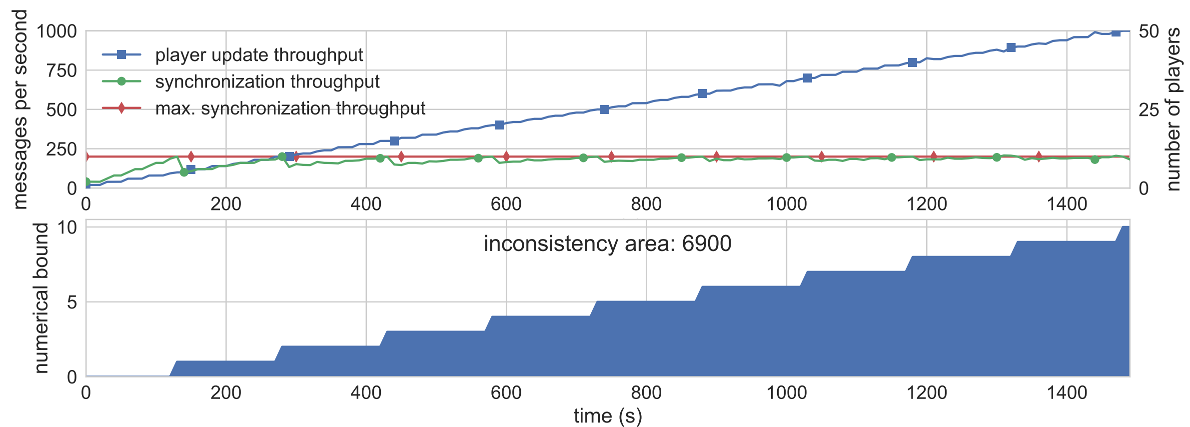


Figure 6.8: Meerkat throughput when using the dynamic **PM** numerical error bound during the **increasing** workload.

slope in the blue curve. The number of synchronization messages is proportional to the number of player updates because a static bound is used. Therefore the green curve also has an upward slope. From this workload, it can be seen that the number of synchronization messages will only increase with the number of players, generating an increasingly large overhead. This can be prevented when using dynamic consistency bounds.

Similar to Figure 6.7, Figure 6.8 shows the player update– and synchronization throughput in Meerkat when running the **increasing** workload on two nodes. This workload is completely synthetic and the configuration can be compared a scenario where two servers run a distributed virtual world and share their data. Each server is connected to an increasing number of players. Instead of a static consistency bound, the **PM** policy is used to configure the numerical error bound dynamically. The **PM** policy has been configured to keep the number of synchronization messages below 200 per second. Initially, the synchronization throughput (green curve) increases with the player update throughput (blue curve), but when the set limit is reached (red line) the **PM** policy increases the numerical error bound from 0 to 1 (blue curve in bottom figure). Because the workload keeps increasing (blue curve in top figure), the **PM** policy keeps increasing the numerical error bound (and decreasing consistency between nodes) to prevent the synchronization throughput from exceeding 200 messages per second. This result is promising for Minecraft-like games because it enables the games to keep overhead limited while scaling to larger numbers of players. However, in practice the numerical error bound is limited because the inconsistency may not decrease player experience.

Figure 6.9 shows the behavior of Meerkat when running the **50-player trace** workload on two nodes while using a fixed numerical error bound of 5. The horizontal axis shows time, and the vertical axis shows the player update– and synchronization throughput in the top figure, and the numerical error bound in the bottom figure. Contrary to the **increasing** workload, the **50-player trace** workload reduces in intensity over time. This results in both the player update throughput (blue curve) and synchronization

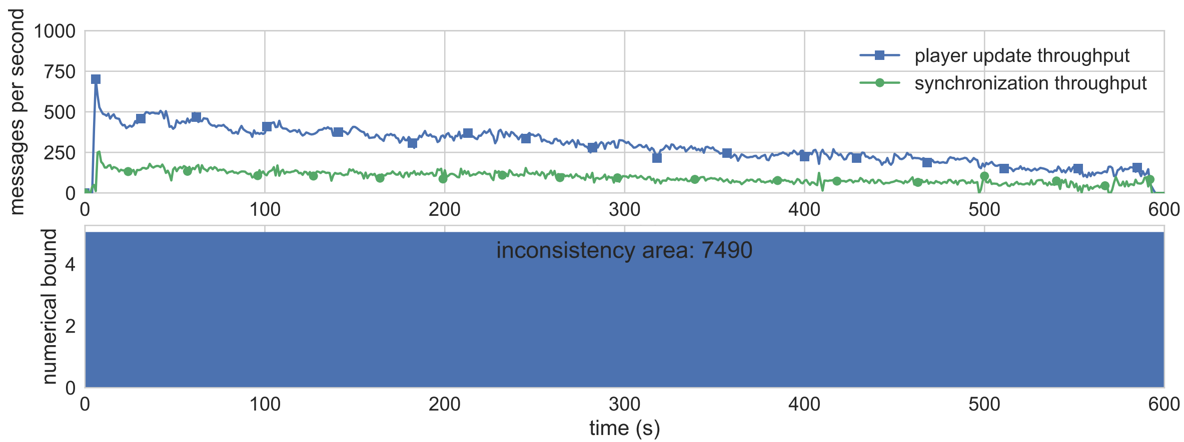


Figure 6.9: Meerkat throughput when using a fixed numerical error bound of 5 during the **50-player trace** workload.

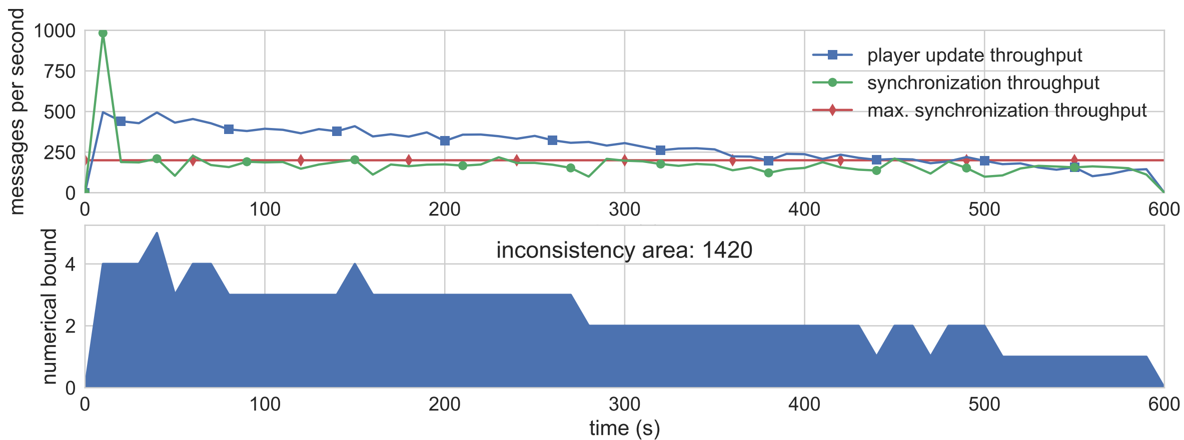
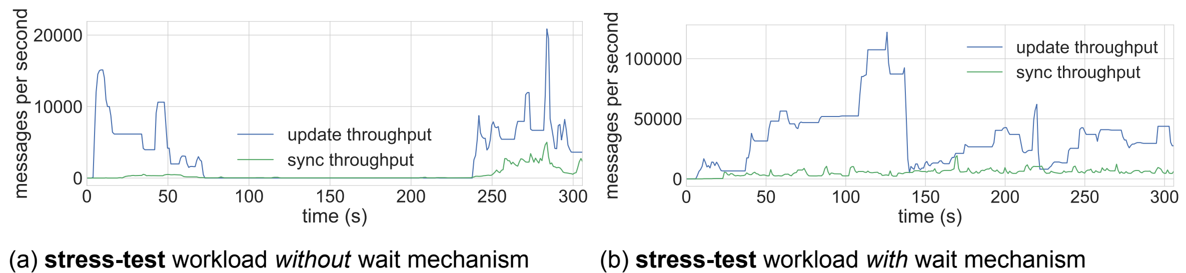


Figure 6.10: Meerkat throughput when using the dynamic **PM** numerical error bound during the **50-player trace** workload.

throughput (green curve) decreasing over time. Because of the static numerical error bound of 5 (blue curve in bottom figure), the synchronization throughput is below 200 messages per second for almost the entire duration of the experiment.

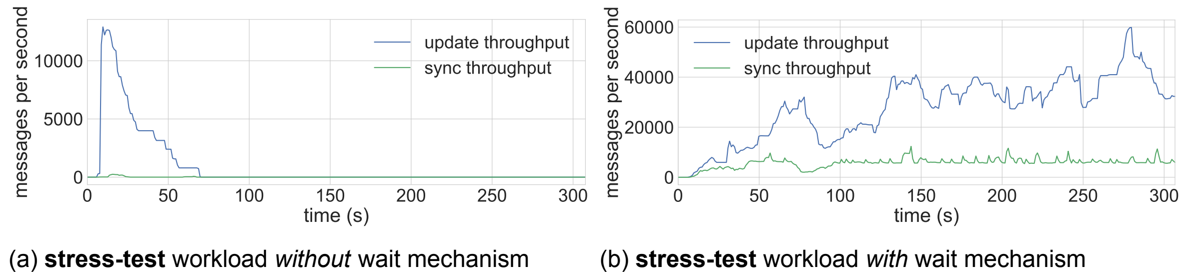
Similar to Figure 6.9, Figure 6.10 shows the behavior of Meerkat when running the **50-player trace** workload on two nodes. The horizontal axis shows time, and the vertical axis shows the player update– and synchronization throughput in the top figure, and the numerical error bound in the bottom figure. Instead of a static bound, the **PM** policy is used to configure the numerical error bound dynamically. Because the workload is low, the **PM** policy can decrease the numerical error bound below 5 (blue curve in bottom figure) without exceeding 200 synchronization messages per second (red curve in top figure). Because the workload decreases over time, the **PM** policy further decreases the numerical error bound over time, increasing the consistency between nodes. This is a promising result for Minecraft-like games because it means that the games do not constantly have to be inconsistent; some Dynamic Conits can have a default bound of 0, only increasing that bound when the workload becomes more intense.

Combining Figure 6.8 and Figure 6.10 shows the stabilizing behavior of the **PM** policy. This policy shows opposite behavior for the two workloads. The **increasing** workload shows an increasing number of updates over time. To keep the number of synchronization messages stable over time, the **PM** policy increases the Dynamic Conit bounds over time, decreasing both the number of synchronization messages exchanged between Meerkat nodes and the consistency between nodes. Exactly the opposite happens for the **50-player trace** workload. Here the **PM** policy decreases the consistency bounds over time, increasing both the number of synchronization messages exchanged between Meerkat nodes and the consistency between nodes.



(a) **stress-test** workload *without* wait mechanism (b) **stress-test** workload *with* wait mechanism

Figure 6.11: Effect of wait mechanism on throughput under **stress-test** workload on 4 nodes



(a) **stress-test** workload *without* wait mechanism (b) **stress-test** workload *with* wait mechanism

Figure 6.12: Effect of wait mechanism on throughput under **stress-test** workload on 16 nodes

6.2.5. Analysis of finding 4

Introducing new nodes during high workloads can cause system stalls. Under high workloads, many updates are processed by all nodes. When a new node is added to the Dynamic Conit, it contacts the other nodes in the Dynamic Conit to exchange updates such that the system stays consistent within bounds. Because many updates are being processed by all nodes, the current Dynamic Conit nodes are flooded with updates from the new node when it joins the Dynamic Conit. Because updates are processed individually, nodes need to first process all synchronized updates before accepting new updates. This can cause a system stall. In Minecraft-like games, such a scenario can occur when a new client connects to a server. The server has likely processed many updates before the new player connects. The client then has to process all synchronized updates before the player can start the game. This is a strong motivation to support merged updates or communicating data in the Dynamic Conit model. We now proceed to describe this result.

Figure 6.11 shows Meerkat system throughput under the **stress-test** workload on 4 nodes with and without the wait mechanism. The wait mechanism causes nodes to only start accepting updates after they are connected to all other nodes in the Dynamic Conit. Without the wait mechanism, nodes accept updates while connecting to the other nodes. The horizontal axes show time, and the vertical axes show message throughput in messages per second. In both Figure 6.11a and Figure 6.11b, the update throughput (blue curve) is unstable. However, with the wait mechanism, the throughput always stays above zero, whereas without the wait mechanism, the throughput halts completely after roughly 60 seconds. After roughly 240 seconds the synchronized updates been processed and system activity is resumed.

Figure 6.12 shows the Meerkat system throughput under the **stress-test** workload on 16 nodes with and without the wait mechanism. Similar to the experiment run on 4 nodes, running **stress-test** without the wait mechanism causes a system stall. This can be seen in Figure 6.12a, where the update throughput (blue curve) drops to zero after roughly 75 seconds. The stall takes more than 225 seconds, which means the system remains blocked for the remainder of the experiment.

6.2.6. Analysis of finding 5

Dynamic Conits' dynamic bound policies can control the consistency of the system and the number of synchronization messages exchanged between nodes. This is a promising result for Minecraft-like games because it gives game developers the mechanisms to control the consistency and synchronization throughput that are best for the game and the type of data communicated. We now proceed to describe this result.

The performance of the policies is compared using the actual synchronization throughput, the syn-

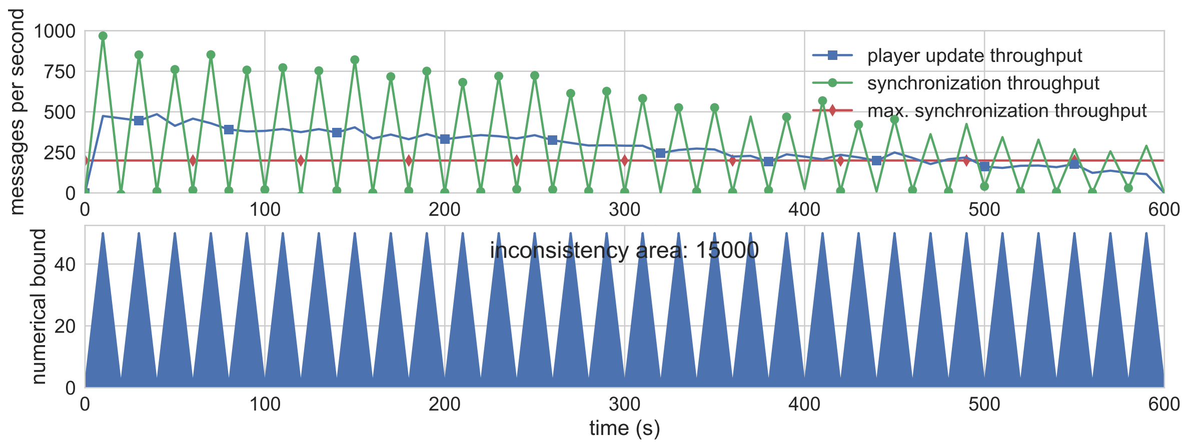


Figure 6.13: Meerkat throughput when using the dynamic **ADMI** policy during the **50-player trace** workload.

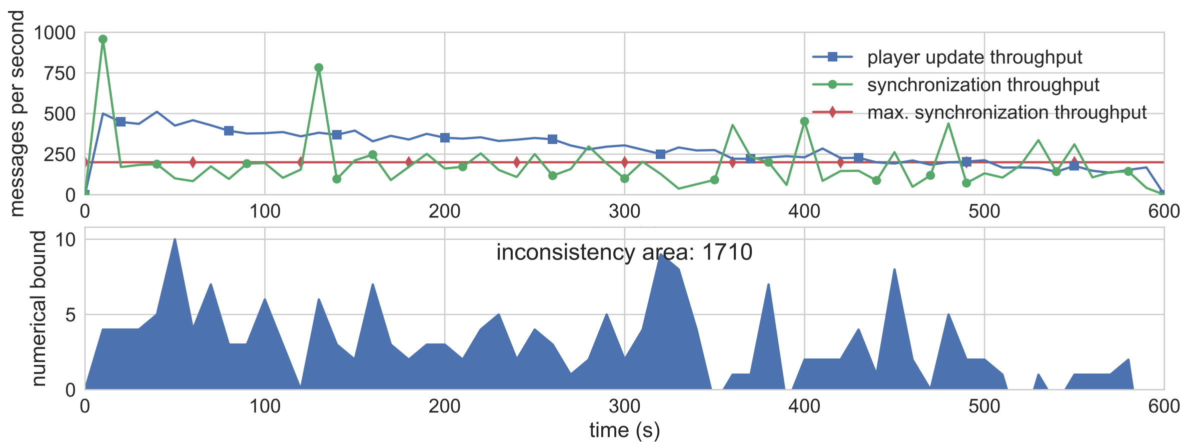


Figure 6.14: Meerkat throughput when using the dynamic **PM-P** policy during the **50-player trace** workload.

chronization throughput target, and the *inconsistency area*. The inconsistency area is the integral of the allowed inconsistency in the system over time. We compute this integral using the trapezoidal rule because it maps directly to discrete data. Because all dynamic bound experiments use the same **50-player trace** workload, they have the same duration. This means that the consistency area is proportional to the average inconsistency, but is easier to visualize.

Figure 6.13 shows the **50-player trace** workload using the **ADMI** policy (described in Section 4.3.1). The horizontal axes shows time. The top vertical axis shows the throughput of both updates and synchronization messages. The red line indicates the target throughput for synchronization messages. The bottom vertical axis shows the numerical bound that is set by the policy. The **ADMI** policy's smallest bound increase is 50, which does not work well for this low workload; a bound of 0 is too strict, but a bound of 50 is too high. The resulting behavior is oscillation, resulting in a total inconsistency area of 15000 over the complete experiment (blue area).

Figure 6.14 shows the **50-player trace** workload using the **PM-P** policy (described in Section 4.3.1). The figure is of the same type as Figure 6.13, except the **PM-P** policy instead of the **ADMI** policy. This policy tries to predict the workload using linear regression on the workload of the last ten seconds. The performance of this policy is better than that of the **ADMI** policy. The number of synchronization messages (green curve) is closer to the target (red line), and the total inconsistency area is an order of magnitude smaller than than of the **ADMI** policy on the same workload.

Figure 6.10 shows the **50-player trace** workload using the **PM** policy (described in Section 4.3.1). This policy calculates the required consistency bound for the current workload using the performance model described in Section 3.4. It outperforms both the **ADMI** policy and the **PM-P** policy on the **50-player trace** workload. The number of synchronization messages (green curve) is closer to the target (red line), and the inconsistency area of 1420 is the smallest of the three policies (blue area).

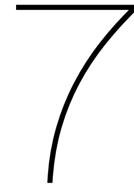
Although the **PM** policy is the clear winner in this case, different behavior might be visible for other workloads. While a comprehensive comparison of these policies is out of scope of this thesis, the differences in their behavior show that different methods for controlling the inconsistency in the system and synchronization messages is possible.

6.2.7. Discussion

The results presented in this chapter show that the value of error bounds have a significant impact on the number of supported players, but increasing the numerical error bound has the most impact when the number of nodes sharing the Dynamic Conit is low. Sharing Dynamic Conits with few nodes matches the application to Minecraft in which Dynamic Conits are shared between server-client pairs.

The Dynamic Conit bound experiments show that the number of synchronization messages can be kept stable by changing the consistency bounds. In practice, to prevent decreased user experience, an upper bound on the consistency bounds must be set. This means that Dynamic Conits allow the game scalability to stretch with the number of players, but only up to a certain point.

Scalability improvements using bounded order error are not evaluated because of technical difficulties. Even when using a simple commit algorithm as discussed in Section 4.2.3, the implementation turns out to be non-trivial.



Conclusion and future work

The gaming industry is a young but very large industry, generating more than 20 billion dollars in revenue in 2016 in the United States alone [23]. To appreciate the scale of this industry, consider that the total box office revenue[1] in 2016 in the United States was less than 12 billion dollars [46]. This large revenue is only possible because of the large number of people that buy and play video games.

While games provide amusement for millions of people, games are also used for other important purposes. Games that are designed with a purpose other than entertainment are called *serious games*. Serious games can be used for societal goals such as training professionals [69] and treating trauma patients [29, 45] by simulating real-world scenarios, and educating people by presenting content in an immerse form [2, 51, 53, 57].

The large population of gamers enables the growth of the game industry, but also poses challenges for game developers. One of these challenges is that players want to play together with friends, share experiences and meet new people. To meet these challenges, researchers look for novel techniques to create fresh content at a massive scale [32] and increase the performance of games and game platforms to scale games to massive numbers of players [33, 42, 62]. Increasing the scalability of games is an active topic of research.

Minecraft is a game that can be used both for amusement and as a serious game. It can do this because of a novel feature that allows players to modify the entire virtual world. A large Minecraft modding-community has evolved online, modifying the game even further and sharing custom-built worlds and games built within Minecraft itself. Additionally, Microsoft acquired Mojang, the developer of Minecraft, for 2.5 billion dollars in 2014 [1] and has released an *education edition* of Minecraft that is used in primary schools to teach a variety of subjects such as history, anatomy, digital logic, and economics using custom Minecraft worlds [2].

Unfortunately, the large scale of the industry does not match the scalability of Minecraft. It is difficult to scale games to millions of players, which are the number of players in popular MMORPGs, and Minecraft's modifiable world poses additional, novel, scalability challenges. Because of Minecraft's unique modifiable environment, massivizing it is a challenge. In games, only dynamic data is communicated between players. Static content such as textures (the paint on a player's weapon or tool) and models (a building or vehicle) is stored locally on the player's device. Dynamic content, such as player behavior (e.g., movement, combat) and its consequences (e.g., finding an item, gaining experience points) has to be communicated between players. Minecraft features a virtual environment that is completely modifiable by players, which means that more data is dynamic, and has to be communicated between players.

Our goal is to enable massive amounts of players to explore Minecraft's virtual environments to explore, create, and learn. This thesis makes several contributions to this end.

[1] The revenue from selling tickets in movie theaters.

7.1. Main contributions

This section discusses the main contributions in this thesis, and how these contributions address the research questions posed in Chapter 1.

How to assess the scalability of Minecraft-like games? (RQ1)

To improve the scalability of Minecraft-like games it is important to understand the scalability bottlenecks of these games. However, no benchmarks exist to assess the scalability of these games. This thesis designs and uses Yardstick, the first distributed-large scale benchmark for Minecraft-like games. The thesis evaluates the scalability of three popular Minecraft-like games using Yardstick. The development of Yardstick and the evaluation of Minecraft scalability continues as a separate project under the Opencraft umbrella and is lead by Jerom van der Sar.

Using Yardstick we find that Minecraft-like games scale to hundreds of players. When the server cannot handle the number of connected players the tick frequency of the server drops below its intended value and the relative utilization, a novel scalability metric, exceeds 100%. Minecraft-like games use concurrency, but CPU utilization does not form the bottleneck for the scalability of the game. We find that servers of Minecraft-like games transmit a large number of packets that increases with the number of players. The application of Dynamic Conits could reduce the number of packets transmitted by Minecraft-like game servers by allowing bounded inconsistency between nodes.

How to adapt the Conit consistency model to apply to Minecraft-like games? (RQ2)

The Conit model allows bounded inconsistency between nodes in a distributed system. The Conit model balances generality and practicality. To keep the model general, it does not consider system properties such as the network layout or system dynamics such as the change in connected nodes or dynamic workloads. In principle, the Conit model can be applied to the servers in a distributed virtual environment, but a number of limitations prevent it from being applied to real-world systems.

In this thesis, we design a new consistency model called Dynamic Conits that is based on the Conit consistency model and can be applied to Minecraft-like games. This is the first attempt to modify the Conit consistency model such that it applies to games.

The Dynamic Conit model differs from the original Conit model by supporting additional mechanisms.

1. Dynamic Conits can change their consistency bounds at runtime, adapting to dynamic workloads.
2. Dynamic Conits support optimistic consistency, remaining highly available, but synchronizing updates as soon as consistency bounds are violated.
3. Dynamic Conits can be created or removed during runtime, allowing bounded inconsistency to be enabled or disabled for new players and servers that join the system.
4. Dynamic Conits can be multi-hop, a mechanism to create a network of Dynamic Conits, forming a new logical network over the distributed system. This allows the use of Dynamic Conits without requiring all-to-all communication.

We also conjecture that a mechanism to transfer data instead of updates is needed to bootstrap new nodes in a network. How this mechanism should work and what the effects are on the consistency guarantees are left for future work.

How to design a system that improves the scalability of these games? (RQ3)

This thesis uses the KISS principle to reduce the amount of complexity introduced in the system design phase. Because building distributed systems is labor intensive and error-prone, it is important not to introduce additional complexity where it is not needed. To this end, Meerkat uses multiple simple algorithms and single-purpose components to implement the Dynamic Conit model.

To keep Meerkat generic and applicable to Minecraft-like games, the system is designed to synchronize updates based on incoming accesses. This means that the game has to inform Meerkat of which Dynamic Conits to create and how to configure the allowed inconsistency for each of these Dynamic Conits. Because Meerkat is responsible for synchronizing data across nodes, the game does not have to know about how updates are synchronized.

Meerkat is a system that implements the Dynamic Conit model. Nodes that run a Meerkat instance connect to each other and synchronize data between each other with bounded inconsistency. Using policies that dynamically update the consistency bounds, Meerkat enables system maintainers to control the scalability/consistency trade-off in the system.

How to evaluate such a system experimentally? (RQ4)

The Dynamic Conit model is a consistency model that allows bounded inconsistency between nodes. Because implementing the Dynamic Conit model in practice introduces a number of technical challenges, we evaluate its scalability improvements by building a prototype system, avoiding further challenges introduced by combining the Dynamic Conit model with an already existing Minecraft-like game.

We evaluate Meerkat using using real-world experiments. Because no large scale traces of Minecraft workloads are publicly available, the experiments use a combination of synthetic workloads and trace-based workloads obtained from Yardstick. The **stress-test** workload is synthetic and is the heaviest possible workload for Meerkat. We use this workload to evaluate possible improvements in the system throughput when using Dynamic Conits. The **50-player trace** workload is a trace-based workload created from the outgoing server traffic of a Minecraft server while hosting 50 players. This workload is used to evaluate the reduction in synchronization messages when using static and dynamic consistency bounds. The experiments are run on the DAS-5, a distributed super computer for computer science research designed by the Advanced School for Computing and Imaging [10].

This is the first evaluation of scalability improvements for Minecraft-like games using the Conit or Dynamic Conit model. We find that low consistency bounds greater than zero can significantly improve system throughput. This means that throughput can be improved without introducing large inconsistency between nodes, motivating the use of Dynamic Conits. For any static consistency bound, a workload exists that causes the system to exchange many synchronization messages. Using dynamic consistency bounds this can be prevented by increasing the consistency bounds under high workloads and decreasing the consistency bounds under low workloads. We also find evidence for the need of a data transfer mechanism. Exchanging large amounts of updates at once can halt the processing of new updates, freezing the system. How to combine such a mechanism with Dynamic Conits is left for future work.

7.2. Future work

This thesis is part of the ongoing Opencraft research project. Opencraft aims to massivize Minecraft-like games, bringing entertainment and education to millions of people around the world. This thesis, Meerkat, and Yardstick are products of the Opencraft project. However, Minecraft-like games do not yet support millions of players; there is still work to be done. This thesis presents a number of promising results for Minecraft-like games. Building on these results, we identify three research directions that can be explored. These are discussed below.

Realistic player behavior

The experiments using the **50-player trace** workload give insight in the scalability improvements of using Dynamic Conits, but this is a trace based on structured player behavior. In future work, it would be interesting to test these policies with more diverse traces from Minecraft. For instance, traces from popular Minecraft servers, education servers, or modified servers may show different player behavior. This might lead to changes in update propagation, possibly changing the effect of using Dynamic Conits.

We conclude that 45% of Minecraft packets transmitted by the Minecraft-like game server is caused by sending entity position data. However, using more realistic workloads can give different results, because the players interact with the virtual world by placing and removing blocks, starting redstone simulations, or setting off chain reactions. Intuitively, these dynamic activities are also communicated between players using other types of packets, reducing the relative frequency of the entity position data packets.

Evaluating other Dynamic Conit mechanisms

The findings in this thesis give insight into how Dynamic Conits can be applied to Minecraft-like games. However, Meerkat is a prototype system that cannot be directly applied to games. There are still a number of modifications that need to be designed, implemented, and evaluated before the full

benefits of applying Dynamic Conits to games can be assessed. In this thesis, we conclude that policies that change Dynamic Conit bounds at runtime can keep the number of synchronization messages low. However, a number of Dynamic Conit mechanisms are not evaluated in this thesis. Furthermore, the novel mechanisms defined in the Dynamic Conit model are not comprehensive; additional mechanisms may be required to efficiently apply Dynamic Conits to Minecraft-like games.

The experiments presented in this thesis use a mix of the strict consistency and optimistic consistency settings from Meerkat. The **stress-test** experiments use strict consistency because it provides an inherent rate-limiting system that would have to be implemented separately when using the optimistic consistency. The other Meerkat experiments use the optimistic consistency setting. Future research could look at how often the consistency bound would be exceeded when using the optimistic consistency setting.

Meerkat supports nodes and Dynamic Conits being added to the system at runtime by using Conit-Info messages that communicate Dynamic Conit information to other nodes. In practice, new nodes constantly join the network. These nodes could be either servers that are started to handle increased workload, or clients from players that start the game. New nodes can either create a new Dynamic Conit with other nodes, or join an existing Dynamic Conit. This thesis concludes that the wait mechanism is not sufficient to prevent synchronizing large numbers of updates from freezing the system in practice, because nodes cannot be forced to join the system during the initial setup. Currently, the Dynamic Conit model does not provide a solution to this problem. This thesis suggests transferring a snapshot of the system state or merging updates as possible solutions, but evaluating these solutions is left to future work. Additionally, evaluating the overhead of creating and removing large numbers of Dynamic Conit in real-world systems can teach us more about the overhead of using Dynamic Conit to synchronize data between nodes.

The classic Conit consistency model assumes all-to-all communication between the nodes that share Conits. In MMOGs, this behavior is undesirable because not all nodes share the same data, and the large number of nodes in the system. Most commercially successful games work via a server-client model. Compatibility between Dynamic Conits and the client-server model increases the applicability of the model. To this end, this thesis proposes to extend the Conit model with the multi-hop feature. Multi-hop Conits is the notion that only a subset of the nodes in the network share a particular Dynamic Conit. By overlapping the nodes that share Dynamic Conits, updates can be propagated from one Dynamic Conit to the next. This allows synchronization between nodes that are not directly connected to each other over multiple hops. Multi-hop Dynamic Conits are a conceptual contribution. Evaluating the effect on the consistency bounds and technical difficulties are left to future work.

Combining the Dynamic Conit model with a Minecraft-like game

The next step for the Opencraft project is to combine the Dynamic Conit model with a Minecraft-like game. This thesis designs and evaluates Meerkat, the first system that implements the Dynamic Conit model. Meerkat is a prototype system that evaluates the scalability improvements when using the Dynamic Conit model and its mechanisms. The thesis concludes that Dynamic Conits can reduce the communication required between nodes in a Minecraft-like game, improving its scalability.

Combining the Dynamic Conit model with a Minecraft-like game allows evaluating the scalability improvements in a more realistic scenario. Furthermore, it exposes potential technical concerns, such as the implementation difficulty of the various Dynamic Conit mechanisms. The resulting scalable Minecraft-like game can be further used as a research platform, facilitating the evaluation of other novel scalability techniques for Minecraft-like games.

Bibliography

[1] Minecraft to join Microsoft, 2014. URL https://www.prnewswire.com/news-releases/minecraft-to-join-microsoft-275112831.html.

[2] Minecraft: Education Edition, 2017. URL https://education.minecraft.net/.

[3] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37–49, 1995. doi: 10.1007/BF01784241. URL https://doi.org/10.1007/BF01784241.

[4] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*, pages 31–40. ACM, 2012.

[5] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In Filip De Turck, Luciano Paschoal Gaspary, and Deep Medhi, editors, *2012 IEEE Network Operations and Management Symposium, NOMS 2012, Maui, HI, USA, April 16-20, 2012*, pages 204–212. IEEE, 2012. ISBN 978-1-4673-0267-8. doi: 10.1109/NOMS.2012.6211900. URL https://doi.org/10.1109/NOMS.2012.6211900.

[6] Trevor Alstad, J. Riley Dunkin, Simon Detlor, Brad French, Heath Caswell, Zane Ouimet, Youry Khmelevsky, and Gaétan Hains. Game network traffic simulation by a custom bot. *9th Annual IEEE International Systems Conference, SysCon 2015 - Proceedings*, pages 675–680, 2015. doi: 10.1109/SYSCON.2015.7116828.

[7] Morgan G Ames and Jenna Burrell. 'Connected Learning' and the Equity Agenda: A Microsociology of Minecraft Play. In Charlotte P Lee, Steven E Poltrock, Louise Barkhuus, Marcos Borges, and Wendy A Kellogg, editors, *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW 2017, Portland, OR, USA, February 25 - March 1, 2017*, pages 446–457. ACM, 2017. ISBN 978-1-4503-4335-0. doi: 10.1145/2998181. URL http://dl.acm.org/citation.cfm?id=2998318.

[8] Grenville J Armitage. An experimental estimation of latency sensitivity in multiplayer Quake 3. In *11th IEEE International Conference on Networks, ICON 2003, September 28 - October 1, 2003 Sydney, NSW, Australia.*, pages 137–141. IEEE, 2003. ISBN 0-7803-7788-5. doi: 10.1109/ICON.2003.1266180. URL https://doi.org/10.1109/ICON.2003.1266180.

[9] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013. doi: 10.1145/2447976.2447992. URL http://doi.acm.org/10.1145/2447976.2447992.

[10] Henri E Bal, Dick H J Epema, Cees de Laat, Rob van Nieuwpoort, John W Romein, Frank J Seinstra, Cees Snoek, and Harry A G Wijshoff. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *IEEE Computer*, 49(5):54–63, 2016. doi: 10.1109/MC.2016.127. URL https://doi.org/10.1109/MC.2016.127.

[11] Richard A. Bartle. *Designing virtual worlds*. New Riders Pub, 2004. ISBN 9780131018167. URL https://books.google.nl/books?redir_esc=y&id=z3VP7MYKqaIC&q=chapter+3#v=snippet&q=chapter3&f=false.

[12] David Bermbach and Jörn Kuhlenkamp. Consistency in Distributed Storage Systems - An Overview of Models, Metrics and Measurement Approaches. In Vincent Gramoli and Rachid Guerraoui, editors, *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*, volume 7853 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2013. ISBN 978-3-642-40147-3. doi: 10.1007/978-3-642-40148-0_13. URL http://dx.doi.org/10.1007/978-3-642-40148-0_13.

[13] Ashwin Bharambe, John R Douceur, Jacob R Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Peer-to-Peer Games. pages 389–400, 2008.

[14] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A Distributed Architecture for Online Multiplayer Games. In Larry L Peterson and Timothy Roscoe, editors, *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings.* USENIX, 2006. URL `http://www.usenix.org/events/nsdi06/tech/bharambe.html`.

[15] Ashwin R Bharambe, John R Douceur, Jacob R Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In Victor Bahl, David Wetherall, Stefan Savage, and Ion Stoica, editors, *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*, pages 389–400, New York, New York, USA, 2008. ACM Press. ISBN 9781605581750. doi: 10.1145/1402958.1403002. URL `http://portal.acm.org/citation.cfm?doid=1402958.1403002http://doi.acm.org/10.1145/1402958.1403002`.

[16] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 398–407. ACM, 2007. ISBN 978-1-59593-616-5. doi: 10.1145/1281100.1281103. URL `http://doi.acm.org/10.1145/1281100.1281103`.

[17] Mark Claypool and Kajal T Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006. doi: 10.1145/1167860. URL `http://doi.acm.org/10.1145/1167860`.

[18] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008. URL `http://www.vldb.org/pvldb/1/1454167.pdf`.

[19] Susan B Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *ACM Comput. Surv.*, 17(3):341–370, 1985. doi: 10.1145/5505.5508. URL `http://doi.acm.org/10.1145/5505.5508`.

[20] Raluca Diaconu and Joaqu□́Keller. Kiwano: A scalable distributed infrastructure for virtual worlds. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 664–667. IEEE, 2013. ISBN 978-1-4799-0836-3. doi: 10.1109/HPCSim.2013.6641489. URL `http://dx.doi.org/10.1109/HPCSim.2013.6641489`.

[21] Raluca Diaconu, Joaquin Joaqu□́Keller, and Mathieu Valero. Manycraft: Scaling Minecraft to Millions. In *Annual Workshop on Network and Systems Support for Games, NetGames '13, Denver, CO, USA, December 9-10, 2013*, pages 1:1—-1:6. IEEE/ACM, dec 2013. ISBN 978-1-4799-2961-0. doi: 10.1109/NetGames.2013.6820617. URL `http://dl.acm.org/citation.cfm?id=2664635http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6820617`.

[22] Herman A. Engelbrecht and Gregor Schiele. Transforming Minecraft into a research platform. In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pages 257–262. IEEE, jan 2014. ISBN 978-1-4799-2355-7. doi: 10.1109/CCNC.2014.6866580. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6866580`.

[23] ESA. 2017 Essential Facts About the computer and video game industry. Technical report, 2017. URL `http://www.theesa.com/wp-content/uploads/2017/04/EF2017_FinalDigital.pdf`.

[24] Travis Faas and Chaolan Lin. Self-Directed Learning in Teacher-Lead Minecraft Classrooms. In Gloria Mark, Susan R Fussell, Cliff Lampe, M. c. schraefel, Juan Pablo Hourcade, Caroline

Appert, and Daniel Wigdor, editors, *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017, Extended Abstracts.*, pages 2569–2575. ACM, 2017. ISBN 978-1-4503-4656-6. doi: 10.1145/3027063.3053269. URL http://doi.acm.org/10.1145/3027063.3053269.

[25] Hector Garcia-Molina and Boris Kogan. Achieving High Availability in Distributed Databases. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 430–440. IEEE Computer Society, 1987. ISBN 0-8186-0762-9. doi: 10.1109/ICDE.1987.7272409. URL https://doi.org/10.1109/ICDE.1987.7272409.

[26] Seth Gilbert and Nancy A Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

[27] Wojciech M Golab, Xiaozhou Li, and Mehul A Shah. Analyzing consistency properties for fun and profit. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 197–206. ACM, 2011. ISBN 978-1-4503-0719-2. doi: 10.1145/1993806.1993834. URL http://doi.acm.org/10.1145/1993806.1993834.

[28] Yong Guo and Alexandru Iosup. The Game Trace Archive. In *11th Annual Workshop on Network and Systems Support for Games, NetGames 2012, Venice, Italy, November 22-23, 2012*, pages 1–6. IEEE, 2012. ISBN 978-1-4673-4576-7. doi: 10.1109/NetGames.2012.6404027. URL http://dx.doi.org/10.1109/NetGames.2012.6404027.

[29] Corentin Haidon, Adrien Ecrepont, Benoit Girard, and Bob-Antoine J. Menelas. A Driving Simulator Designed for the Care of Trucker Suffering from Post-Traumatic Stress Disorder. In *Serious Games and Edutainment Applications*, pages 411–431. Springer International Publishing, Cham, 2017. doi: 10.1007/978-3-319-51645-5_19. URL http://link.springer.com/10.1007/978-3-319-51645-5_19.

[30] Maurice Herlihy and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972.

[31] Ibe van Geel. MMOData blog: MMOData Charts version 4.1 is Live !, 2013. URL http://mmodata.blogspot.nl/2013/12/mmodata-charts-version-41-is-live.html.

[32] Alexandru Iosup. POGGI: generating puzzle instances for online games on grid infrastructures. *Concurrency and Computation: Practice and Experience*, 23(2):158–171, feb 2011. ISSN 15320626. doi: 10.1002/cpe.1638. URL http://doi.wiley.com/10.1002/cpe.1638.

[33] Alexandru Iosup, Siqi Shen, Yong Guo, Stefan Hugtenburg, Jesse Donkervliet, and Radu Prodan. Massivizing online games using cloud computing: A vision. In *2014 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, pages 1–4. IEEE, jul 2014. ISBN 978-1-4799-4717-1. doi: 10.1109/ICMEW.2014.6890684. URL http://ieeexplore.ieee.org/document/6890684/.

[34] Won Kim. Highly Available Systems for Database Applications. *ACM Comput. Surv.*, 16(1):71–98, 1984. doi: 10.1145/861.866. URL http://doi.acm.org/10.1145/861.866.

[35] Sudha Krishnamurthy, William H Sanders, and Michel Cukier. An Adaptive Framework for Tunable Consistency and Timeliness Using Replication. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 17–26. IEEE Computer Society, 2002. ISBN 0-7695-1597-5. doi: 10.1109/DSN.2002.1028882. URL http://dx.doi.org/10.1109/DSN.2002.1028882.

[36] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439. URL http://dx.doi.org/10.1109/TC.1979.1675439.

[37] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi: 10.1145/279227.279229. URL http://doi.acm.org/10.1145/279227.279229.

[38] Huiguang Liang, Ian Tay, Ming Feng Neo, Wei Tsang Ooi, and Mehul Motani. Avatar Mobility in Networked Virtual Environments: Measurements, Analysis, and Implications. *CoRR*, abs/0807.2, 2008. URL http://arxiv.org/abs/0807.2328.

[39] David J Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Comput. Surv.*, 25(3):303–338, 1993. doi: 10.1145/158439.158907. URL http://doi.acm.org/10.1145/158439.158907.

[40] Elvis S Liu and Georgios K Theodoropoulos. Interest management for distributed virtual environments: A survey. *ACM Comput. Surv.*, 46(4):51:1—-51:42, 2014. doi: 10.1145/2535417. URL http://doi.acm.org/10.1145/2535417.

[41] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416. ACM, 2011. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043593. URL http://doi.acm.org/10.1145/2043556.2043593.

[42] Alexandru Losup, Ruud van de Bovenkamp, Siqi Shen, Adele Lu Jia, and Fernando Kuipers. Analyzing Implicit Social Networks in Multiplayer Online Games. *IEEE Internet Computing*, 18(3): 36–44, may 2014. ISSN 1089-7801. doi: 10.1109/MIC.2014.19. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6756709.

[43] Betty Love, Victor Winter, Cindy Corritore, and Davina Faimon. Creating an Environment in which Elementary Educators Can Teach Coding. In Janet C Read and Phil Stenton, editors, *Proceedings of the The 15th International Conference on Interaction Design and Children, IDC '16, Manchester, United Kingdom, June 21-24, 2016*, pages 643–648. ACM, 2016. ISBN 978-1-4503-4313-8. doi: 10.1145/2930674.2936008. URL http://doi.acm.org/10.1145/2930674.2936008.

[44] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, and Others. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[45] Simon Mayr, Wolfgang Horleinsberger, and Paolo Petta. The Trauma Treatment Game: Design Constraints for Serious Games in Psychotherapy. In *2014 6th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pages 1–6. IEEE, sep 2014. ISBN 978-1-4799-4056-1. doi: 10.1109/VS-Games.2014.7012171. URL http://ieeexplore.ieee.org/document/7012171/.

[46] Pamela McClintock. 2016 Box Office Revenue Hits $11.17B for Another Record Year | Hollywood Reporter, 2016. URL http://www.hollywoodreporter.com/news/2016-box-office-record-year-crosses-11-billion-959300.

[47] Mike Schramm. Chinese WoW hits 1 million concurrent players, 2008. URL https://www.engadget.com/2008/04/11/chinese-wow-hits-1-million-concurrent-players/.

[48] Stephan Mueller, Mubbasir Kapadia, Seth Frey, Severin Klingler, Richard P Mann, Barbara Solenthaler, Robert W Sumner, and Markus H Gross. Statistical Analysis of Player Behavior in Minecraft. In José Pablo Zagal, Esther MacCallum-Stewart, and Julian Togelius, editors, *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015, Pacific Grove, CA, USA, June 22-25, 2015*. Society for the Advancement of the Science of Digital Games, 2015. URL http://www.fdg2015.org/papers/fdg2015_paper_39.pdf.

[49] Stephan Mueller, Barbara Solenthaler, Mubbasir Kapadia, Seth Frey, Severin Klingler, Richard P Mann, Robert W Sumner, and Markus H Gross. HeapCraft: interactive data exploration and visualization tools for understanding and influencing player behavior in Minecraft. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games, MIG 2015, Paris, France, November 16-18, 2015*, pages 237–241. ACM, 2015. ISBN 978-1-4503-3991-9. doi: 10.1145/2822013.2822033. URL http://doi.acm.org/10.1145/2822013.2822033.

[50] Vlad Nae, Alexandru Iosup, and Radu Prodan. Dynamic Resource Provisioning in Massively Multiplayer Online Games. *IEEE Trans. Parallel Distrib. Syst.*, 22(3):380–395, 2011. doi: 10.1109/TPDS.2010.82. URL http://dx.doi.org/10.1109/TPDS.2010.82.

[51] Steve Nebel, Sascha Schneider, and Günter Daniel Rey. Mining Learning and Crafting Scientific Experiments: A Literature Review on the Use of Minecraft in Education and Research. *Educational Technology & Society*, 19(2):355–366, 2016. URL http://www.ifets.info/journals/19_2/26.pdf.

[52] Karin Petersen, Mike Spreitzer, Douglas B Terry, Marvin Theimer, and Alan J Demers. Flexible Update Propagation for Weakly Consistent Replication. In Michel Banâtre, Henry M Levy, and William M Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 288–301. ACM, 1997. ISBN 0-89791-916-5. doi: 10.1145/268998.266711. URL http://doi.acm.org/10.1145/268998.266711.

[53] Marc Prensky. Digital game-based learning. *Computers in Entertainment*, 1(1):21, oct 2003. ISSN 15443574. doi: 10.1145/950566.950596. URL http://portal.acm.org/citation.cfm?doid=950566.950596.

[54] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In Wu-chang Feng, editor, *Proceedings of the 3rd Workshop on Network and System Support for Games, NETGAMES 2004, Portland, Oregon, USA, August 30, 2004*, pages 152–156. ACM, 2004. ISBN 1-58113-942-X. doi: 10.1145/1016540.1016557. URL http://doi.acm.org/10.1145/1016540.1016557.

[55] Kjetil Raaen, Ragnhild Eg, and Carsten Griwodz. Can gamers detect cloud delay? In Yutaka Ishibashi and Adrian David Cheok, editors, *13th Annual Workshop on Network and Systems Support for Games, NetGames 2014, Nagoya, Japan, December 4-5, 2014*, pages 1–3. IEEE, 2014. ISBN 978-1-4799-6882-4. doi: 10.1109/NetGames.2014.7008962. URL http://dx.doi.org/10.1109/NetGames.2014.7008962.

[56] Michal Ries, Philipp Svoboda, and Markus Rupp. Empirical study of subjective quality for massive multiplayer games. *Proceedings of IWSSIP 2008 - 15th International Conference on Systems, Signals and Image Processing*, pages 181–184, 2008. doi: 10.1109/IWSSIP.2008.4604397.

[57] José Manuel Sáez-López, John Miller, Esteban Vázquez-Cano, and María Concepción Domínguez-Garrido. Exploring application, attitudes and integration of video games: Minecraftedu in middle school. *Educational Technology and Society*, 18(3):114–128, 2015. ISSN 14364522. URL http://www.ifets.info/journals/18_3/9.pdf.

[58] Nuno Santos, Luís Veiga, and Paulo Ferreira. Vector-Field Consistency for Ad-Hoc Gaming. In Renato Cerqueira and Roy H Campbell, editors, *Middleware 2007, ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, USA, November 26-30, 2007, Proceedings*, volume 4834 of *Lecture Notes in Computer Science*, pages 80–100. Springer, 2007. ISBN 978-3-540-76777-0. doi: 10.1007/978-3-540-76778-7_5. URL http://dx.doi.org/10.1007/978-3-540-76778-7_5.

[59] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in Warcraft III. In *Proceedings of the 2nd Workshop on Network and System Support for Games, NETGAMES 2003, Redwood City, California, USA, May 22-23, 2003*, pages 3–14. ACM, 2003. ISBN 1-58113-734-6. doi: 10.1145/963900.963901. URL http://doi.acm.org/10.1145/963900.963901http://portal.acm.org/citation.cfm?doid=963900.963901.

[60] Siqi Shen and Alexandru Iosup. Modeling Avatar Mobility of Networked Virtual Environments. In *Proceedings of International Workshop on Massively Multiuser Virtual Environments, MMVE 2014, Singapore, March 19-21, 2014*, pages 2:1—-2:6. ACM, 2014. ISBN 978-1-4503-2708-4. doi: 10.1145/2577387.2577396. URL http://doi.acm.org/10.1145/2577387.2577396.

[61] Aman Singla, Umakishore Ramachandran, and Jessica K Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *SPAA*, pages 211–220, 1997. doi: 10.1145/258492.258513. URL http://doi.acm.org/10.1145/258492.258513.

[62] Siqi Shen, Alexandru Iosup, and Dick Epema. Massivizing Multi-player Online Games on Clouds. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 152–155. IEEE, may 2013. ISBN 978-0-7695-4996-5. doi: 10.1109/CCGrid.2013.23. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6546073.

[63] Dale Skeen and David D Wright. Increasing Availability in Partitioned Database Systems. In Daniel J Rosenkrantz and Ronald Fagin, editors, *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 290–299. ACM, 1984. ISBN 0-89791-128-8. doi: 10.1145/588011.588054. URL http://doi.acm.org/10.1145/588011.588054.

[64] Mirko Suznjevic, Lea Skorin-Kapov, and Maja Matijasevic. The Impact of User, System, and Context factors on Gaming QoE: a Case Study Involving MMORPGs. In *Annual Workshop on Network and Systems Support for Games, NetGames '13, Denver, CO, USA, December 9-10, 2013*, pages 3:1—-3:6. IEEE/ACM, 2013. ISBN 978-1-4799-2961-0. URL http://dl.acm.org/citation.cfm?id=2664637.

[65] Francisco J Torres-Rojas and Esteban Meneses. Convergence Through a Weak Consistency Model: Timed Causal Consistency. *CLEI Electron. J.*, 8(2), 2005. URL http://www.clei.org/cleiej/paper.php?id=110.

[66] Francisco J Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed Consistency for Shared Distributed Objects. In Brian A Coan and Jennifer L Welch, editors, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, '99Atlanta, Georgia, USA, May 3-6, 1999*, pages 163–172. ACM, 1999. ISBN 1-58113-099-6. doi: 10.1145/301308.301350. URL http://doi.acm.org/10.1145/301308.301350.

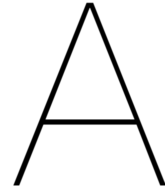
[67] E. van der Hoeven. Scaling Distributed Virtual Environments using Socially Aware Load Partitioning and Interest Management, 2017. URL https://repository.tudelft.nl/islandora/object/uuid%3A3e8a66e5-80f4-466d-b6bb-961f9896b800?collection=education.

[68] Paolo Viotti and Marko Vukolić. Consistency in Non-Transactional Distributed Storage Systems. dec 2015. URL http://arxiv.org/abs/1512.00168.

[69] Fang You, Yuxin Tan, Jinsong Feng, Linshen Li, Jing Lin, and Xin Liu. Research on Virtual Training System in Aerospace Based on Interactive Environment. pages 50–62. Springer, Cham, apr 2016. doi: 10.1007/978-3-319-40259-8_5. URL http://link.springer.com/10.1007/978-3-319-40259-8_5.

[70] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, aug 2002. ISSN 07342071. doi: 10.1145/566340.566342. URL http://portal.acm.org/citation.cfm?doid=566340.566342http://doi.acm.org/10.1145/566340.566342.

Appendices



Minecraft-like game configurations

A.1. Vanilla configurations

A.1.1. server.properties

```
#Minecraft server properties
#Thu Jun 15 08:09:40 CEST 2017
spawn-protection=2
max-tick-time=60000
generator-settings=
force-gamemode=false
allow-nether=true
gamemode=0
broadcast-console-to-ops=true
enable-query=false
player-idle-timeout=0
difficulty=0
spawn-monsters=true
op-permission-level=4
announce-player-achievements=true
pvp=true
snooper-enabled=false
level-type=LARGEBIOMES
hardcore=false
enable-command-block=false
max-players=500
network-compression-threshold=256
resource-pack-sha1=
max-world-size=29999984
server-port=25565
texture-pack=
server-ip=
spawn-npcs=true
allow-flight=true
level-name=mcpworld
view-distance=8
resource-pack=
spawn-animals=true
white-list=false
generate-structures=true
online-mode=false
max-build-height=256
```

```
level-seed=MCP 8.02
prevent-proxy-connections=false
motd=Mod Coder Pack
enable-rcon=false
```

A.2. Spigot configurations

A.2.1. server.properties

```
#Minecraft server properties
#Thu Jun 15 08:03:12 CEST 2017
generator-settings=
op-permission-level=4
allow-nether=true
level-name=world
enable-query=false
allow-flight=false
announce-player-achievements=true
prevent-proxy-connections=false
server-port=25565
max-world-size=29999984
level-type=DEFAULT
enable-rcon=false
force-gamemode=false
level-seed=
server-ip=
network-compression-threshold=256
max-build-height=256
spawn-npcs=true
white-list=false
spawn-animals=true
snooper-enabled=true
hardcore=false
resource-pack-sha1=
online-mode=false
resource-pack=
pvp=true
difficulty=0
enable-command-block=false
player-idle-timeout=0
gamemode=0
max-players=500
spawn-monsters=true
view-distance=10
generate-structures=true
motd=A Minecraft Server
```

A.2.2. bukkit.yml

```
# This is the main configuration file for Bukkit.
# As you can see, there's actually not that much to configure without any plugins.
# For a reference for any variable inside this file, check out the Bukkit Wiki at
# http://wiki.bukkit.org/Bukkit.yml
#
# If you need help on this file, feel free to join us on irc or leave a message
# on the forums asking for advice.
#
# IRC: #spigot @ irc.spi.gt
```

```
#   (If this means nothing to you, just go to http://www.spigotmc.org/pages/irc/ )
# Forums: http://www.spigotmc.org/
# Bug tracker: http://www.spigotmc.org/go/bugs


settings:
  allow-end: true
  warn-on-overload: true
  permissions-file: permissions.yml
  update-folder: update
  plugin-profiling: false
  connection-throttle: 0
  query-plugins: true
  deprecated-verbose: default
  shutdown-message: Server closed
spawn-limits:
  monsters: 70
  animals: 15
  water-animals: 5
  ambient: 15
chunk-gc:
  period-in-ticks: 600
  load-threshold: 0
ticks-per:
  animal-spawns: 400
  monster-spawns: 1
  autosave: 6000
aliases: now-in-commands.yml
database:
  username: bukkit
  isolation: SERIALIZABLE
  driver: org.sqlite.JDBC
  password: walrus
  url: jdbc:sqlite:{DIR}{NAME}.db
```


A.2.3. spigot.yml


```
# This is the main configuration file for Spigot.
# As you can see, there’s tons to configure. Some options may impact gameplay, so use
# with caution, and make sure you know what each option does before configuring.
# For a reference for any variable inside this file, check out the Spigot wiki at
# http://www.spigotmc.org/wiki/spigot-configuration/
#
# If you need help with the configuration or have any questions related to Spigot,
# join us at the IRC or drop by our forums and leave a post.
#
# IRC: #spigot @ irc.spi.gt ( http://www.spigotmc.org/pages/irc/ )
# Forums: http://www.spigotmc.org/

config-version: 11
settings:
  debug: false
  save-user-cache-on-stop-only: false
  item-dirty-ticks: 20
  user-cache-size: 1000
  timeout-time: 60
  restart-on-crash: true
```

```
  restart-script: ./start.sh
  bungeecord: false
  int-cache-limit: 1024
  sample-count: 12
  netty-threads: 4
  player-shuffle: 0
  attribute:
    maxHealth:
      max: 2048.0
    movementSpeed:
      max: 2048.0
    attackDamage:
      max: 2048.0
  late-bind: false
  filter-creative-items: true
  moved-wrongly-threshold: 0.0625
  moved-too-quickly-multiplier: 10.0
messages:
  whitelist: You are not whitelisted on this server!
  unknown-command: Unknown command. Type ”/help” for help.
  server-full: The server is full!
  outdated-client: Outdated client! Please use {0}
  outdated-server: Outdated server! I’m still on {0}
  restart: Server is restarting
commands:
  tab-complete: 0
  replace-commands:
  - setblock
  - summon
  - testforblock
  - tellraw
  log: true
  spam-exclusions:
  - /skill
  silent-commandblock-console: false
stats:
  disable-saving: false
  forced-stats: {}
world-settings:
  default:
    verbose: true
    item-despawn-rate: 6000
    merge-radius:
      item: 2.5
      exp: 3.0
    arrow-despawn-rate: 1200
    mob-spawn-range: 4
    growth:
      cactus-modifier: 100
      cane-modifier: 100
      melon-modifier: 100
      mushroom-modifier: 100
      pumpkin-modifier: 100
      sapling-modifier: 100
      wheat-modifier: 100
      netherwart-modifier: 100
```

```
      vine-modifier: 100
      cocoa-modifier: 100
    ticks-per:
      hopper-transfer: 8
      hopper-check: 1
    hopper-amount: 1
    entity-activation-range:
      animals: 32
      monsters: 32
      misc: 16
    max-tnt-per-tick: 100
    entity-tracking-range:
      players: 48
      animals: 48
      monsters: 48
      misc: 32
      other: 64
    hunger:
      jump-walk-exhaustion: 0.05
      jump-sprint-exhaustion: 0.2
      combat-exhaustion: 0.1
      regen-exhaustion: 6.0
      swim-multiplier: 0.01
      sprint-multiplier: 0.1
      other-multiplier: 0.0
    max-tick-time:
      tile: 50
      entity: 50
    random-light-updates: false
    save-structure-info: true
    enable-zombie-pigmen-portal-spawns: true
    wither-spawn-sound-radius: 0
    nerf-spawner-mobs: false
    view-distance: 10
    zombie-aggressive-towards-villager: true
    hanging-tick-frequency: 100
    seed-village: 10387312
    seed-feature: 14357617
    seed-monument: 10387313
    seed-slime: 987234911
    dragon-death-sound-radius: 0
```

A.3. Glowstone configurations

A.3.1. glowstone.yml

```
# glowstone.yml is the main configuration file for a Glowstone server
# It contains everything from server.properties and bukkit.yml in a
# normal CraftBukkit installation.
#
# For help, join us on Discord: https://discord.gg/TFJqhsC
server:
    ip: ''
    port: 25565
    name: Glowstone Server
    log-file: logs/log-%D.txt
    online-mode: false
```

```
    max-players: 500
    whitelisted: false
    motd: A Glowstone server
    shutdown-message: Server shutting down.
    allow-client-mods: true
    snooper-enabled: false
console:
    use-jline: true
    prompt: '>'
    date-format: HH:mm:ss
    log-date-format: yyyy/MM/dd HH:mm:ss
game:
    gamemode: SURVIVAL
    gamemode-force: false
    difficulty: PEACEFUL
    hardcore: false
    pvp: true
    max-build-height: 256
    announce-achievements: true
    allow-flight: false
    command-blocks: false
    resource-pack: ''
    resource-pack-hash: ''
creatures:
    enable:
        monsters: true
        animals: true
        npcs: true
    limit:
        monsters: 70
        animals: 15
        water: 5
        ambient: 15
    ticks:
        monsters: 1
        animal: 400
folders:
    plugins: plugins
    update: update
    worlds: .
files:
    permissions: permissions.yml
    commands: commands.yml
    help: help.yml
advanced:
    connection-throttle: 0
    idle-timeout: 0
    warn-on-overload: true
    exact-login-location: false
    plugin-profiling: false
    deprecated-verbose: 'false'
    compression-threshold: 256
    proxy-support: false
    player-sample-count: 12
    metrics: true
    metrics-server-uuid: d48fc677-3296-44b3-b079-1ae74a1e42c2
```

```
    gpgpu: false
    gpgpu-use-any-device: false
    run-glowclient: false
extras:
    query-enabled: false
    query-port: 25614
    query-plugins: true
    rcon-enabled: false
    rcon-password: glowstone
    rcon-port: 25575
    rcon-colors: true
world:
    name: world
    seed: ''
    level-type: DEFAULT
    spawn-radius: 16
    view-distance: 8
    gen-structures: true
    allow-nether: true
    allow-end: true
    keep-spawn-loaded: true
    populate-anchored-chunks: true
    classic-style-water: false
    disable-generation: false
database:
    driver: org.sqlite.JDBC
    url: jdbc:sqlite:config/database.db
    username: glowstone
    password: nether
    isolation: SERIALIZABLE
```


A.3.2. worlds.yml


```
general:
  sea_level: 64
overworld:
  coordinate-scale: 684.412
  height:
    scale: 684.412
    noise-scale:
      x: 200.0
      z: 200.0
  detail:
    noise-scale:
      x: 80.0
      y: 160.0
      z: 80.0
  surface-scale: 0.0625
  base-size: 8.5
  stretch-y: 12.0
  biome:
    height-offset: 0.0
    height-weight: 1.0
    scale-offset: 0.0
    scale-weight: 1.0
    height:
      default: 0.1
```

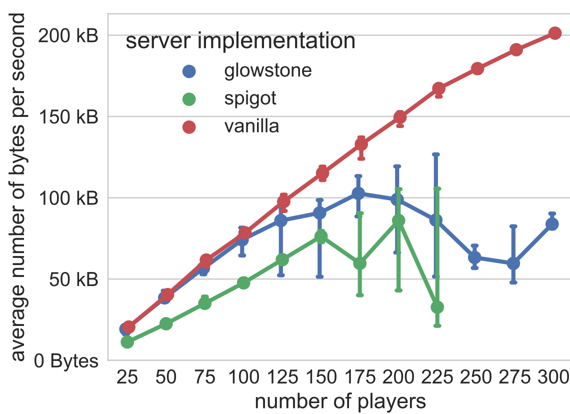
```
      flat-shore: 0.0
      high-plateau: 1.5
      flatlands: 0.125
      swampland: -0.2
      mid-plains: 0.2
      flatlands-hills: 0.275
      swampland-hills: -0.1
      low-hills: 0.2
      hills: 0.45
      mid-hills2: 0.1
      default-hills: 0.2
      mid-hills: 0.3
      big-hills: 0.525
      big-hills2: 0.55
      extreme-hills: 1.0
      rocky-shore: 0.1
      low-spikes: 0.4125
      high-spikes: 1.1
      river: -0.5
      ocean: -1.0
      deep-ocean: -1.8
    scale:
      default: 0.2
      flat-shore: 0.025
      high-plateau: 0.025
      flatlands: 0.05
      swampland: 0.1
      mid-plains: 0.2
      flatlands-hills: 0.25
      swampland-hills: 0.3
      low-hills: 0.3
      hills: 0.3
      mid-hills2: 0.4
      default-hills: 0.4
      mid-hills: 0.4
      big-hills: 0.55
      big-hills2: 0.5
      extreme-hills: 0.5
      rocky-shore: 0.8
      low-spikes: 1.325
      high-spikes: 1.3125
      river: 0.0
      ocean: 0.1
      deep-ocean: 0.1
  density:
    fill:
      mode: 0
      sea-mode: 0
      offset: 0.0
nether:
  coordinate-scale: 684.412
  height:
    scale: 2053.236
    noise-scale:
      x: 100.0
      z: 100.0
```

```
  detail:
    noise-scale:
      x: 80.0
      y: 60.0
      z: 80.0
  surface-scale: 0.0625
end:
  coordinate-scale: 684.412
  height:
    scale: 1368.824
  detail:
    noise-scale:
      x: 80.0
      y: 160.0
      z: 80.0
```

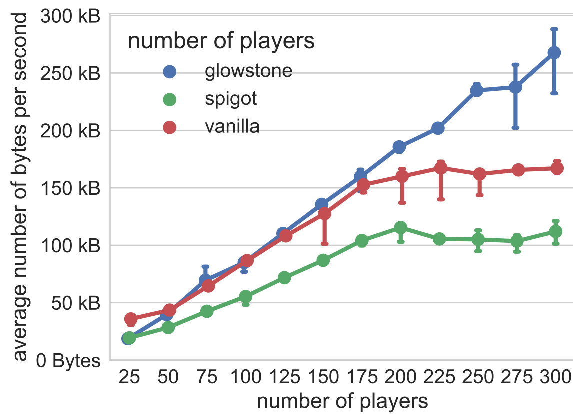
B

Experimental results

B.1. Minecraft scalability experiments

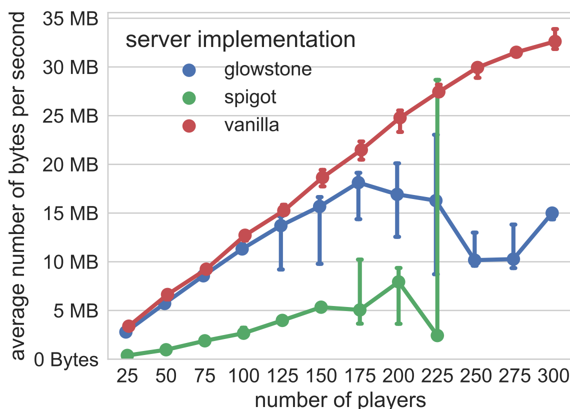


(a) Effect of the **increasing players** workload

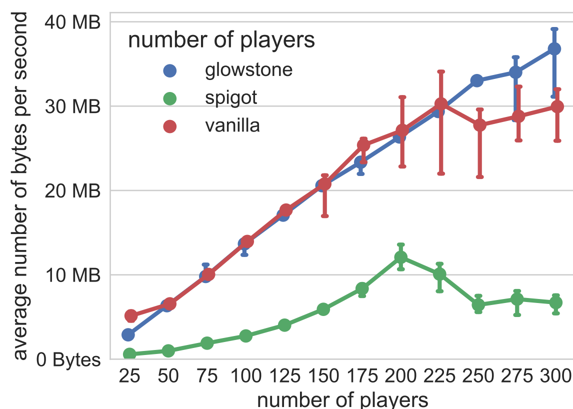


(b) Effect of the **fixed players** workload

Figure B.1: Additional experimental result: Minecraft incoming bytes throughput under the **increasing players** workload and **fixed players** workload.

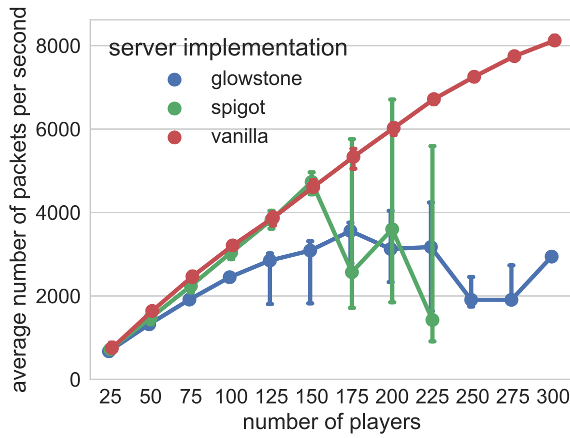


(a) Effect of the **increasing players** workload

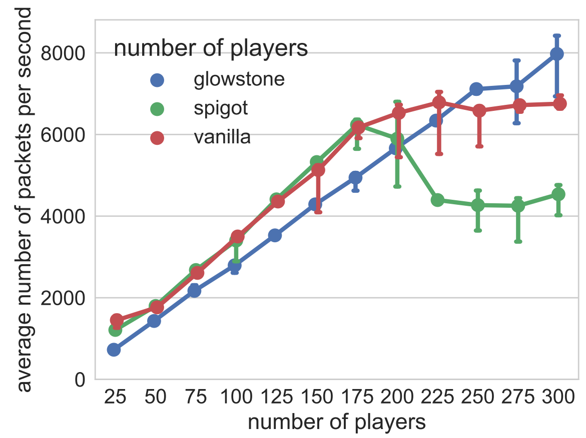


(b) Effect of the **fixed players** workload

Figure B.2: Additional experimental result: Minecraft outgoing bytes throughput under the **increasing players** workload and **fixed players** workload.

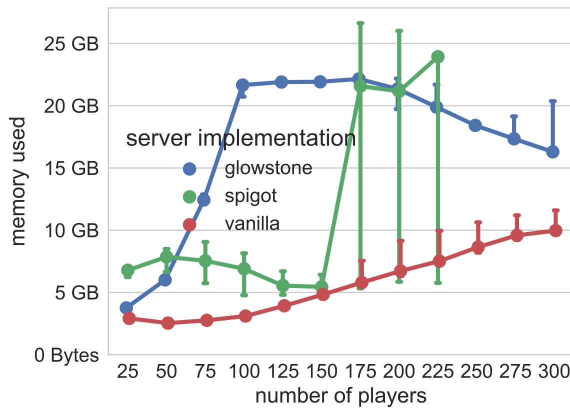


(a) Effect of the **increasing players** workload

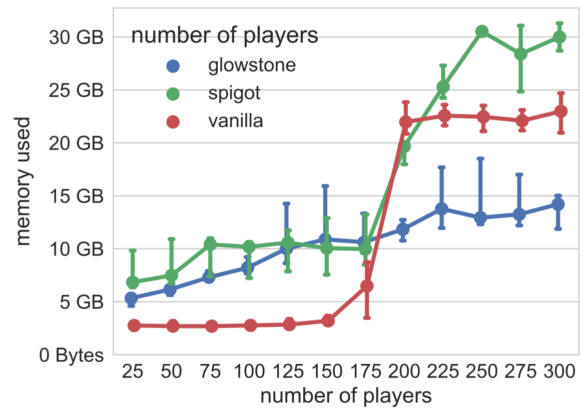


(b) Effect of the **fixed players** workload

Figure B.3: Additional experimental result: Minecraft incoming packets throughput under the **increasing players** workload and **fixed players** workload.



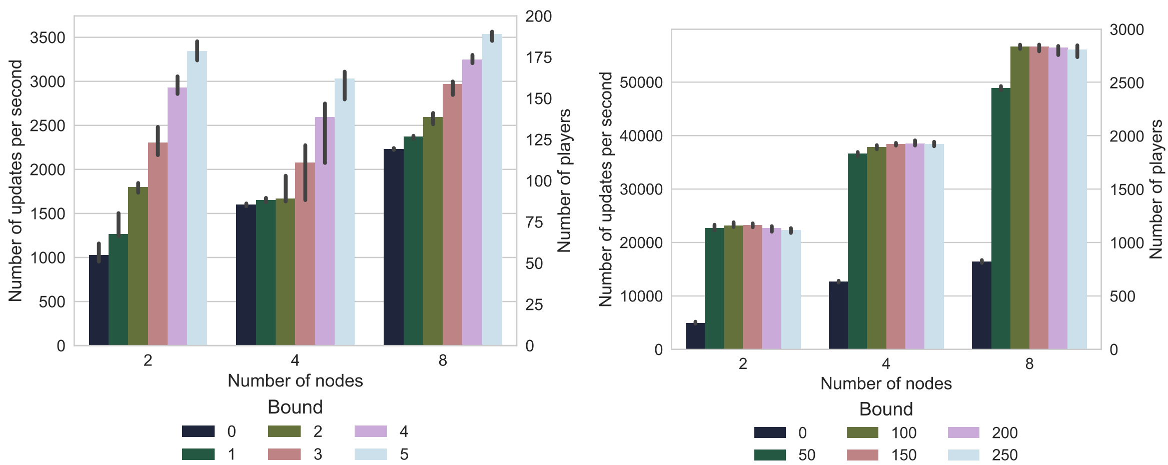
(a) Effect of the **increasing players** workload



(b) Effect of the **fixed players** workload

Figure B.4: Additional experimental result: Minecraft memory usage under the **increasing players** workload and **fixed players** workload.

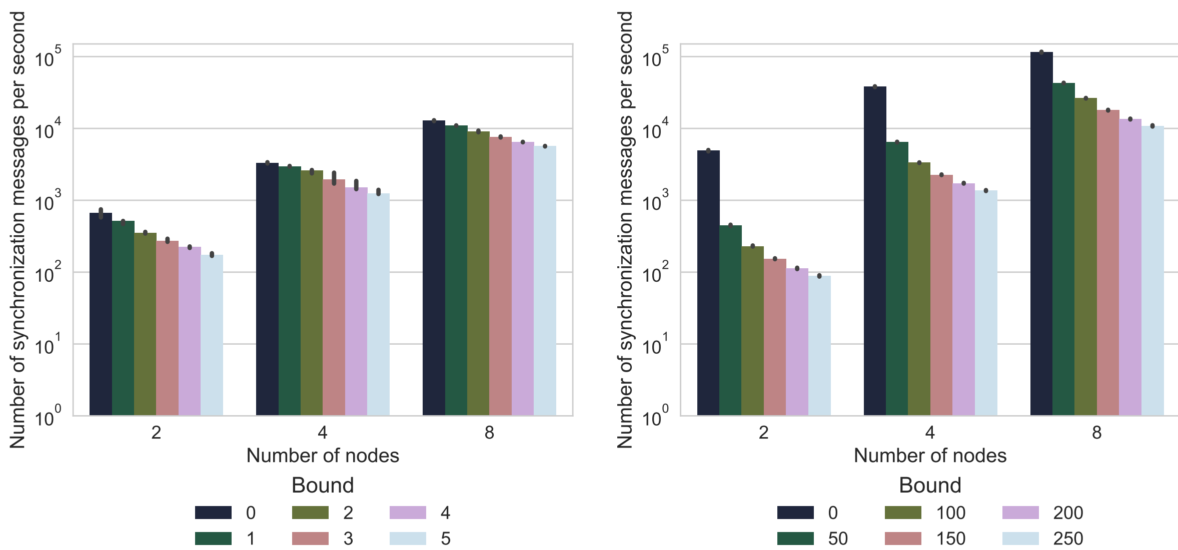
B.2. Meerkat experiments



(a) Effect of small staleness bounds on update throughput

(b) Effect of large numerical bounds on update throughput

Figure B.5: Additional result from experiment 6.2.2. The top of the bars indicate the median value, and the whiskers indicate a 100% confidence interval.



(a) Effect of small staleness bounds on synchronization throughput

(b) Effect of large numerical bounds on synchronization throughput

Figure B.6: Additional result from experiment 6.2.3. The top of the bars indicate the median value, and the whiskers indicate a 100% confidence interval.

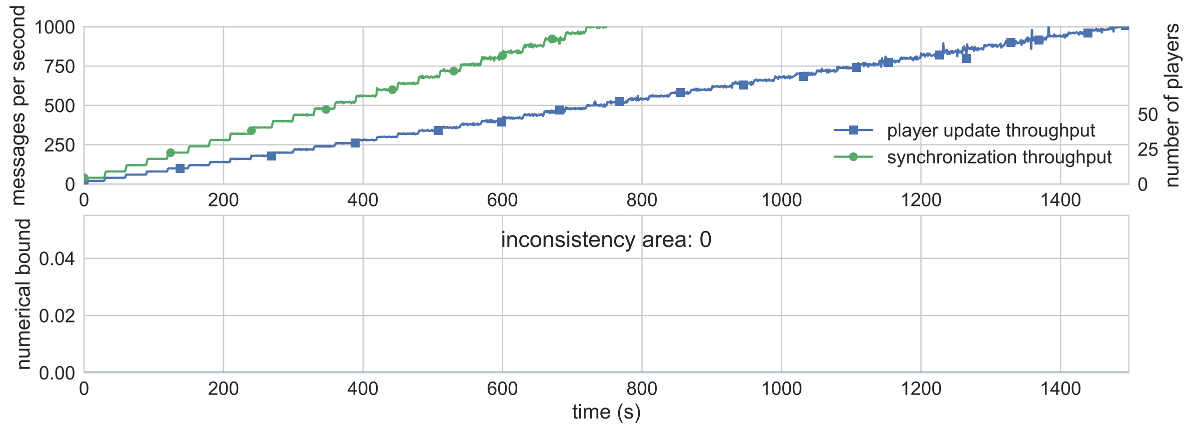


Figure B.7: Additional experimental result: using static bound of 0 on **increasing** workload on 2 nodes.

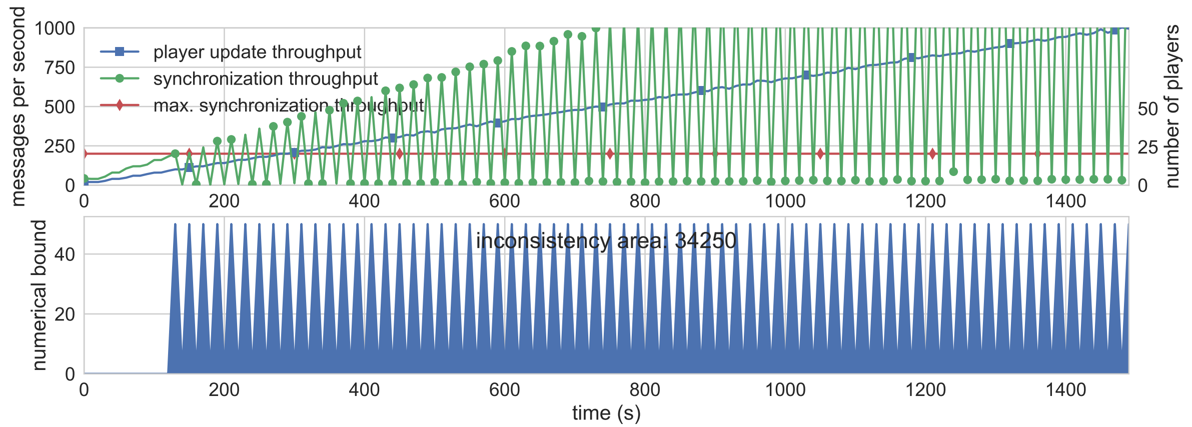


Figure B.8: Additional experimental result: using **ADMI** policy on **increasing** workload on 2 nodes.

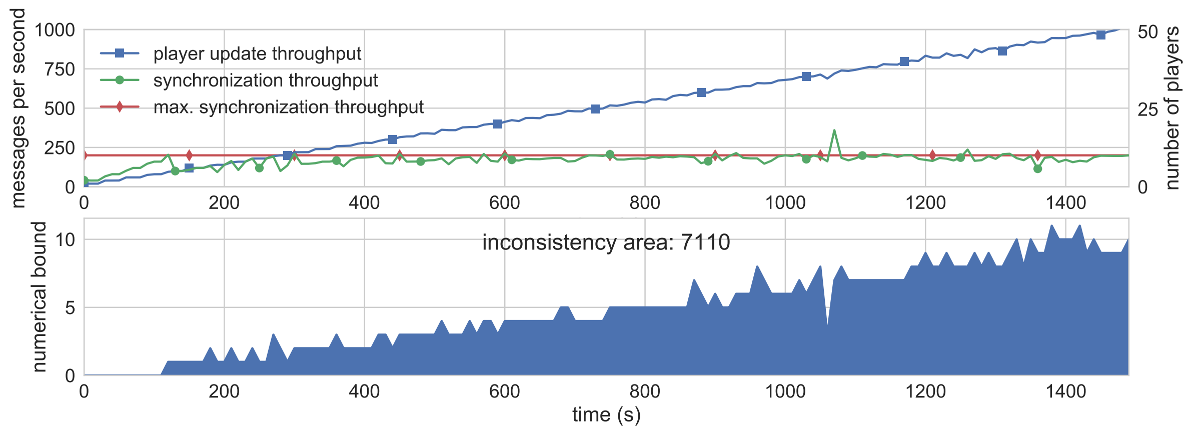


Figure B.9: Additional experimental result: using **PM-P** policy on **increasing** workload on 2 nodes.

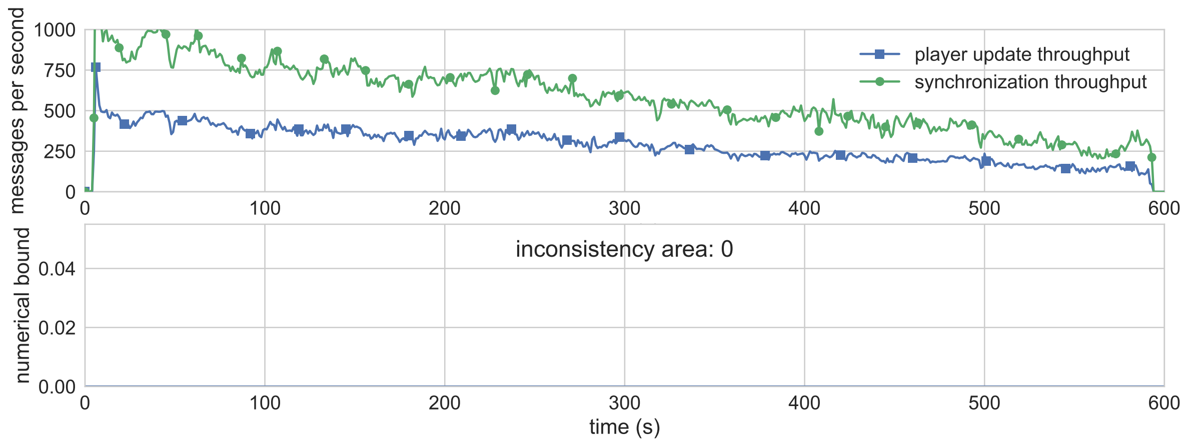


Figure B.10: Additional results from experiment 6.2.4. The effect of a numerical bound of 0 on the synchronization throughput running the **50-player trace** workload on 2 nodes.

(a) **stress-test** workload *without* wait mechanism.

(b) **stress-test** workload *with* wait mechanism.

(c) **stress-test** workload *without* wait mechanism.

(d) **stress-test** workload *with* wait mechanism.

(e) **stress-test** workload *without* wait mechanism.

(f) **stress-test** workload *with* wait mechanism.

(g) **stress-test** workload *without* wait mechanism.

(h) **stress-test** workload *with* wait mechanism.

Figure B.11: Additional results from experiment 6.2.5. The effect of the wait-mechanism on system throughput using the **stress-test** workload and a varying number of nodes.