The Opencraft Messaging System

Jim van Vliet, Julian van Dijk, Lars de Tombe, Ruben Marang, and Wessel van de Brug

TI3806: Bachelor Thesis

Supervisor: *Thomas Overklift, MSc*
Coordinator: *prof. dr. Huijuan Wang, MSc*
Clients: *Jesse Donkervliet, MSc*
prof. dr. Alexandru Iosup, MSc

Delft University of Technology

The Opencraft Messaging System

Jim van Vliet, Julian van Dijk, Lars de Tombe, Ruben Marang, and Wessel van de Brug

TI3806: Bachelor Thesis
Supervisor: Thomas Overklift, MSc
Coordinator: prof. dr. Huijuan Wang, MSc
Clients: Jesse Donkervliet, MSc, prof. dr. Alexandru Iosup, MSc

July 9, 2020

Foreword

We chose to work on this project because all of us have played Minecraft when we were younger, and still enjoy playing it to this day. Therefore, a project centered around Minecraft seemed perfect to us. It ended up being more work than we first anticipated and made for some very long nights. But, we had fun working on it and learned a lot. There were a lot of times where ideas did not work out, or when we thought something was easy to implement but actually required many hours of work.

We could not have achieved all this without the help of a number of people, some of whom we would like to give some special thanks.

First of all, we would like to thank both Jesse Donkervliet and Thomas Overklift for their continuous help and feedback during the project. They have helped us every step of the way and were always ready to meet up for questions. Jesse shared with us much of his expertise on the design and use of experiments, which we greatly appreciate as none of us had much experience in this field. Thomas kept us on track and made sure we would work on the final report and other deadlines, preventing us from losing sight of the project goals. We would also like to thank Alexandru Iosup for helping us on occasion. His experience and expertise was very valuable to us in forming our report.

Jim van Vliet
Julian van Dijk
Lars de Tombe
Ruben Marang
Wessel van de Brug

SUMMARY

The aim of this project is to improve the scalability of the Opencraft server. The Opencraft server is based on an open-source implementation of the vanilla Minecraft server. This thesis focuses on improving the messaging system of the Opencraft server. The existing implementation of the messaging system is very basic. It went through all online players and promptly generated all messages and send them afterwards, this limits performance and scalability. We decided to replace the system with an implementation of the topic-based variant of the publish/subscribe design pattern. However, we evaluated other options as well.

We also decided to implement certain features that were deemed necessary for verification purposes. The server often varied in behaviour compared to the vanilla Minecraft server. This made message verification harder since it was not possible to compare both servers side to side. This lead to the implementation of collision, physics, and water flow, which allows anyone to verify the behaviour of the Opencraft server is correct. The implementation of the messaging system provides a variety of components that can each be configured. All different configurations were tested to find the optimal configuration for the Opencraft server.

Table of contents

Foreword . . . ii
Summary . . . iii
1 Introduction . . . 1
1.1 Context . . . 1
1.2 Report overview . . . 1
2 Problem definition and analysis . . . 3
2.1 Research questions . . . 3
2.2 Main contributions . . . 4
3 Background . . . 5
3.1 Network architecture . . . 5
3.2 Publish/subscribe design pattern . . . 5
3.3 Publish/subscribe categories . . . 6
3.4 Alternative communication options . . . 9
4 System design . . . 12
4.1 Broker . . . 12
4.2 Policy . . . 13
4.3 Filter . . . 13
4.4 Messaging system . . . 13
5 Implementation . . . 15
5.1 Policies . . . 15
5.2 Brokers . . . 15
5.3 Filters . . . 19
5.4 Messaging system . . . 19
6 Integration . . . 20
6.1 Relocation of the message generation . . . 20
6.2 Optimisation of chunk streaming . . . 20
7 Validation . . . 23
7.1 Automated gameplay testing . . . 23
7.2 Entity collision . . . 23
7.3 Block updates . . . 26
7.4 Mesa biome generation . . . 27
7.5 Multiple dimensions . . . 28
7.6 Bugfixes . . . 29
8 Code quality . . . 31
8.1 Opencraft . . . 31
8.2 Messaging system . . . 33
8.3 Testing . . . 34
9 Experiment setup . . . 35
9.1 Environment . . . 35
9.2 Metrics . . . 35
9.3 Tools . . . 36
9.4 Benchmarking method . . . 37
10 Experiments . . . 39
10.1 Messaging system latency . . . 39

10.2 Server scalability 42
10.3 Other experiments 43
11 Conclusion 45
12 Discussion 46
12.1 Software Development Methodology 46
12.2 Code change effort 46
12.3 Testing problems 46
12.4 Adaptation of Yardstick 46
12.5 Ethical implications 47
13 Future work 48
13.1 Messaging system 48
13.2 Automated gameplay testing 48
13.3 Entity-component system 48
13.4 Mathematics 48
Bibliography 49
Appendices 51
A Project info sheet 51
B (Original) Project description 53
C System requirements 54
D Bugs & missing features 56
E SIG results 58

1. Introduction

Billions of people enjoy playing video games worldwide [1]. One of the most popular games is Minecraft with around 112 million players every month in 2019 [2]. Minecraft generates a virtual world, providing landscapes, towns and other structures, in which players can interact with each other, computer controlled entities, and the world itself. The interaction with the world is what makes Minecraft unique. It allows players to add, remove, and replace any block in the world.

Currently, Minecraft-like games are not capable of accommodating millions of players, whereas Massive Multiplayer Online Games (MMOGs), are capable of that. They often allow for such a large number of players by distributing them over multiple servers. Every server simulates part of the virtual world of the game and can support thousands of players.

However, a Minecraft server can only accommodate 200-300 simultaneous players [3], before performance degrades to a point where the game becomes unplayable. While games without modifiable terrain only need to synchronize entity data to ensure consistency between players, Minecraft must also synchronize terrain data. This synchronization increases the network and computational load of the server.

This project is part of the Opencraft project, which is a long-term international research project. The ultimate goal of the Opencraft project is to build a Minecraft-like game that can support millions of simultaneous players, whom can explore and interact with each other in a single virtual environment.

Our contribution to the Opencraft project is the creation of a messaging system for the Opencraft server. Other ongoing initiatives are the development of Yardstick [3] and dynamic consistency units [4].

1.1. Context

The Opencraft server is a fork of Glowstone[1], which is an open source implementation, based on the original vanilla Minecraft server. However, Glowstone does not rely on any of the code from the vanilla Minecraft server. Furthermore, Glowstone has an MIT license which means that Opencraft can modify it without having to worry about licensing issues.

Yardstick is a tool created by researchers from the Opencraft project [3]. It is a benchmark for Minecraft-like services. We use Yardstick to evaluate the performance of the Opencraft server throughout the project.

Dynamic consistency units, which will be referred to as dynamic conits, are used to decrease the amount of communication necessary while still maintaining consistency between the player and the server. Dynamic conits solve this by omitting less relevant updates. In practice this means that these updates are sent less often, which allows the client and server to go slightly out of sync without the player noticing. The updates are sent whenever certain bounds are exceeded. These bounds are dynamic and can be adjusted at runtime.

The current implementation, as of the start of this project, features one dynamic conit per in-game chunk for each player. This means that every player receives updates about a chunk whenever the bounds of the dynamic conits associated with that chunk are exceeded. A chunk is a 256 block tall 16x16 segment of a Minecraft world[2].

1.2. Report overview

The remainder of the paper is structured as follows. Section 2 introduces the problem of the messaging system and provides the research questions. Section 3 presents the literature research on messaging

[1] https://glowstone.net/

[2] As of Minecraft version 1.12.2: https://minecraft.gamepedia.com/Chunk

techniques and design patterns. Based on this research, we propose a design for the messaging system in Section 4. Section 5 explains the actual implementation of the messaging system. Section 6 discusses how we integrated the messaging system in the existing Opencraft server. Section 7 explains the steps towards how we validated the messaging system. Section 8 discusses the code quality of both the original and new code with an evaluation of the Software Improvement Group (SIG). Section 9 explains the setup for the experiments in detail and Section 10 analyses the result of these experiments. Finally, Chapters 11 and 12 give a conclusion and discuss the findings respectively.

2. Problem definition and analysis

The goal of this project is to improve the maximum amount of players the Opencraft server can handle. We focus on message generation and distribution, aiming to reduce the computational cost of these tasks by introducing a messaging system.

In the original Opencraft server, messages are processed at multiple different moments during execution. Either they are immediately processed or stored for processing in the main game loop at a later time. Once a message is processed, an update is created and applied to the game world. On each cycle of the game loop, the world data is scanned for each player to detect updates. Messages to represent those updates are generated and sent to each player of them separately.

Because message generation is computationally intensive, multiple different processing moments are not desired. Furthermore, the generation and sending of messages is distributed throughout the code, making it harder to change and manage the code.

The validation of server functionality is another problem that has to be addressed. We need to verify that the integration of the messaging system does not limit the server's original functionality, nor prevents the implementation of missing features that are present in the vanilla Minecraft server. It is required to implement a number of these missing features, as it would otherwise be difficult to determine whether the centralised messaging system could support them.

2.1. Research questions

Based on these problems, we pose the following main question (MQ) and further divide it in three research questions (RQ).

MQ **How can the messaging system of the Opencraft server be reworked such that its scalability is increased?**

The ultimate goal of the Opencraft project is to create a Minecraft-like game that can support millions of simultaneous players. Therefore, the question we would like to answer is how we could redesign message handling in the Opencraft server to bring the Opencraft project closer to that goal.

RQ1 **How to design an extensible messaging system for the Opencraft server?**

To build the messaging system, we need to determine which design best suits the needs of the Opencraft server. The messaging system should be extensible, as we would like to be able to integrate it with other initiatives of the Opencraft project, such as the dynamic conits explained in Chapter 1.1.

RQ2 **How to validate that the integration of the messaging system does not break features of the Opencraft server nor prevents the addition of features from the vanilla Minecraft server?**

The integration of the messaging system should not break any of the existing features of the Opencraft server, this would make the Opencraft server worse instead of better, nor should it prevent the implementation of features from the vanilla Minecraft server that are currently missing from the Opencraft server.

RQ3 **Does the Opencraft server, with the messaging system, support significantly more simultaneous users than the original Opencraft server?**

Researchers from the Opencraft project have previously developed Yardstick to measure how many simultaneous players a Minecraft server could handle.

With the integration of the messaging system we hope to support more simultaneous users. To determine whether the Opencraft server performs better with the messaging system than without it, we evaluate both versions of the server using Yardstick [3], a tool previously developed by researchers from the Opencraft project to measure the performance of Minecraft-like games.

2.2. Main contributions

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

1. The design, implementation, and integration of a messaging system for the Opencraft server.
2. A validation of the correctness of the integration of the messaging system with the Opencraft server.
3. An evaluation of the performance of the Opencraft server, with and without integration of the developed messaging system.

3. BACKGROUND

In this section, we present the theoretical background for creating a messaging systems for online games. In consultation with our client we think it is best to use the publish/subscribe (pub/sub) design pattern. However, before we design our messaging system based on the pub/sub design pattern and answer the first research question, we want to compare alternatives to see if the pub/sub design pattern is indeed the best choice.

First, we explain the most relevant network architectures available for MMOGs. Then, the pub/sub design pattern and its variants are described. Afterwards, we present a number of alternative approaches and techniques that could be used together to build a messaging system. These alternatives are compared with the pub/sub design pattern to show how they compare.

3.1. NETWORK ARCHITECTURE

In an MMOG, messages need to be sent back and forth between players. We take a high level look at two common network architectures that decide what structure is used to send these messages to the players. These two architectures for multiplayer online games are client-server and peer-to-peer (P2P). The traditional client-server model consists of one (distributed) server connected to multiple clients. In a P2P model, messages are not send between client and server, but between the clients themselves. This shifts all of the responsibilities from the server to the clients [5].

This research focuses on the client-server architecture, because this is what our client wishes and is also set as a requirement. Switching to a P2P network model would also require users to install a modified Minecraft client. If users are able to use the original Minecraft clients, the threshold for playing will remain as low as possible.

3.2. PUBLISH/SUBSCRIBE DESIGN PATTERN

The pub/sub pattern enables communication between publishers and subscribers with loose coupling [6, 7]. The publishers are the senders of messages, these messages are categorized into topics without knowing who will receive it. The subscribers are the recipients of messages, they may express one or more interests in specific classes and will only receive messages they are interested in [8].

Besides publishers and subscribers, the pub/sub pattern often requires at least one broker. We define a broker to be a middle-man that manages the communication between publishers and subscribers by keeping track of the topic, or ranges of topics, subscribers are interested in. The broker distributes published messages between these subscribers.

An advantage of the pub/sub pattern is that the broker allows the publishers and subscribers to be loosely coupled. The publishers do not know who will receive their message and the subscribers do not know who sent them. The pub/sub pattern also eliminates the need to periodically check for new messages, because the messages are pushed directly to the subscribers via the broker. Another advantage is that it allows for greater parallelisation, this allows for more messages to be sent. A disadvantage of the pub/sub pattern is connected to the decoupling between publishers and subscribers. A pub/sub system designed with loose coupling cannot guarantee that a message that is meant to be received by a specific target will be delivered, because the publisher does not know whether the targeted subscriber is actually subscribed.

Eugster et al. [6] proposes three categories of the pub/sub pattern: topic-based, content-based and type-based. However, we consider the type-based category as a specialisation of the topic-based category. The topic-based category will be discussed in Section 3.3.1, and the content-based category in Section 3.3.2. We discuss the advantages and disadvantages of both of these categories in their respec-

tive sections.

Figure 1 shows how two subscribers, a broker, and a publisher would interact with each other on a high level in a pub/sub based system. In this example, subscriber 1 informs the broker it is interested in receiving messages by sending a subscribe request. Whenever a publisher sends a message, the broker forwards this message to everyone that is interested, which is only subscriber 1. After this, subscriber 2 also sends a subscribe request to the broker. When the publisher publishes another message to the broker, the broker now forwards this message to both subscriber 1 and 2, because the broker keeps track of who is subscribed and forwards messages accordingly.

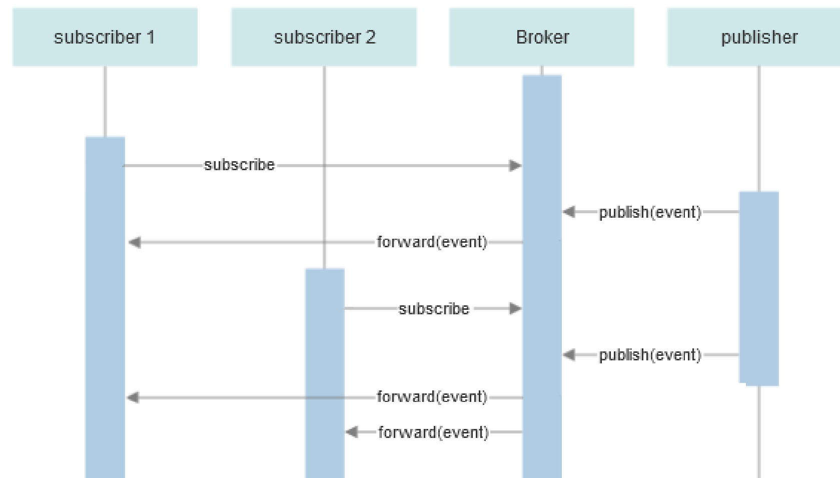


Figure 1: An example sequence diagram which shows interactions between components in a pub/sub based system

3.3. Publish/subscribe categories

As mentioned in Section 3.2, we define two types of the pub/sub pattern. However, we also recognize the possibility to combine these into a hybrid approach. In this section we describe topic-based pub/sub and a specialisation of this category, afterwards the content-based pub/sub is described. Lastly, the hybrid approach is discussed with an example.

3.3.1. Topic-based

The first category of pub/sub explained is topic-based. A topic is a subject in which a subscriber could be interested. Publishers send a message about a certain topic, which is then broadcast to all the subscribers that are interested in this topic. Subscribers can subscribe to one or more topics and then receive messages from all these topics. The broker keeps track of who is subscribed to which topics and broadcasts the messages received from publishers to the correct subscribers [6, 9, 10].

One of the advantages of the topic-based category is that routing is simple, it is possible to multicast to a group of subscribers that have matching topics, this can be done with a simple table lookup [11]. Multicasting allows for communication within or to a group in a network via a single transmission of data through the source. Topic-based pub/sub also makes parallelization easier, because the topics can be processed separately. A disadvantage of topic-based pub/sub is that its expressiveness is limited, a subscriber has to subscribe to a topic even if he is only interested in a specific part of the topic, this can

cause inefficient use of bandwidth [6, 9, 12]. For example, a user could express interest in a certain stock, but cannot limit received updates to the stock prices above or below some value [13].

To get a better understanding of how a topic-based approach would work, we provide a simple example (see Figure 2). In this example we have two topics (T1 and T2) and three subscribers (S1, S2 and S3). Subscribers S1 and S2 are subscribed to T1 and subscribers S2 and S3 are subscribed to T2. When a publisher publishes a message to T1, the broker receives the message for this topic and then broadcasts this message to the interested subscribers (S1 and S2). If a publisher then sends a message to T2, the broker will broadcast the message to subscribers S2 and S3. The example in Figure 2 only uses one publisher, however multiple publishers that publish to different topics are also possible. The same holds for the broker, the example only shows one but the use of multiple brokers is possible. A real world example of a successful system that uses the topic-based pub/sub approach is Twitter, which is able to support a large amount of monthly users [9].

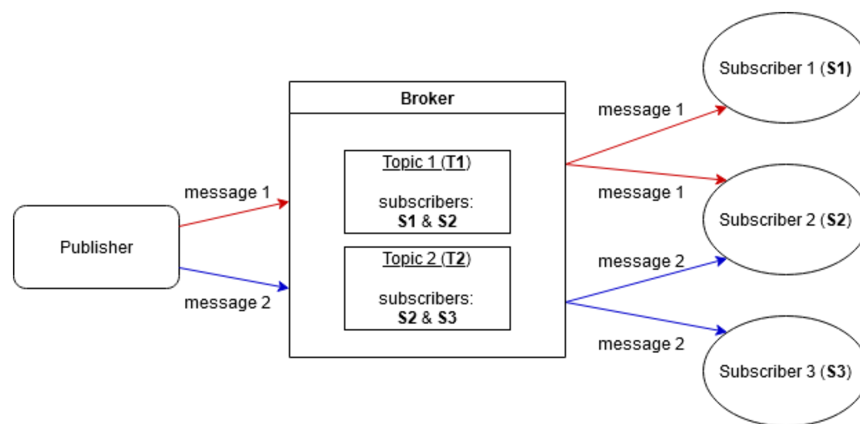


Figure 2: example message exchange for topic-based pub/sub

We see the type-based pub/sub approach as a specialisation on the topic-based category. Type-based uses object types as topics, these types are derived from the programming language that is used. This close integration with the programming language requires a smart relation between the object types and events, which would allow for the support of both a static and dynamic type-check of the publications and subscriptions [12]. The intention behind the use of type-based pub/sub is to provide type safety and encapsulation, while keeping the same efficiency in filtering and routing components [14].

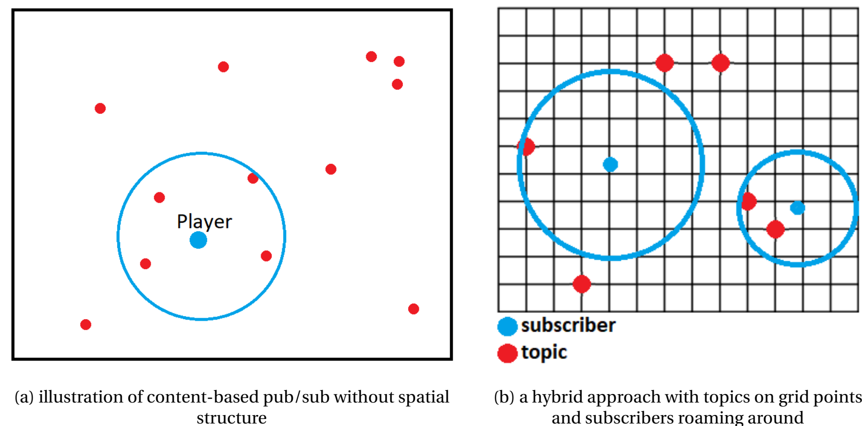
3.3.2. Content-based

The content-based variant is different from the topic-based variant, because subscriptions are based on the events themselves. These events are not classified by specific topics or topic names, but according to their content and attributes. Subscribers define ranges of events they are interested in. These ranges come in a variety of different forms, including some that allow for continuous selection criteria such as the euclidean distance between objects. Alternatively, ranges could be defined using SQL or even a dedicated subscription language [6]. These ranges can be changed during runtime depending on what the subscriber needs.

An advantage of content-based pub/sub is that it adds flexibility. With the the added expressiveness and the possibility for filtering on multiple dimensions, it is not necessary to have specific knowledge

of subjects. Because content-based pub/sub is more expressive than topic-based, it is possible to implement the topic-based approach with content-based pub/sub, while it is not possible the other way around [11]. While this category is more expressive it is also more complex, because checking which subscriber should receive which message is a lot more complicated compared to topic-based pub/sub.

An example of a scenario where the content-based approach can work well, is when there are no spatial structures. These spatial structures are often attached to coordinate systems. Content-based pub/sub does not try to define topics, but it looks at the content and attributes. We provide an example of how this would work without a spatial structure, which is illustrated in Figure 3a. In this example, there is one player in a 2d area, defined with a 2d vector that uses continuous values. Besides the player, there are also other entities in this field, the player is only interested in an entity if it is within a certain range. If there are four entities within range of the player, the player will receive information from the broker about these four entities. If a player would move away from an entity it would no longer receive messages (from the broker) about this entity, the same holds for when an entity would move away from the player. Lastly it would also be possible for the player to change its range to receive information from the broker about a larger (or smaller) area around the player.



(a) illustration of content-based pub/sub without spatial structure

(b) a hybrid approach with topics on grid points and subscribers roaming around

Figure 3: a) shows an example for content-based pub/sub and b) shows an example of a hybrid approach

3.3.3. Hybrid

Besides only using the topic-based or content-based (or in some cases even type-based) variant, it is possible to combine multiple variants into a hybrid solution. A situation where a hybrid approach could work well, is when there is a need for discrete topics and continuous subscriptions.

An example that illustrates such an situation is shown in Figure 3b. In this example the topics are specific points on the grid in a 2d space. If a publisher wants to publish a message that is relevant for that point in the grid it does so by publishing to the corresponding topic. However a subscriber is not bound to this grid and can thus be anywhere in this 2d space. The subscribers will use ranges to determine what they are interested in and thus look at the content around them, which represents the content-based approach. The subscriber will only receive updates, send by publishers, that are currently in its range.

3.4. Alternative communication options

In this section we will analyse five alternatives and additions to pub/sub that could be used to build a messaging system. Since our client has a preference for the pub/sub design pattern, these alternative communication options are compared to pub/sub to show how they compare. Some of the alternatives mentioned below can be difficult to compare since they work at different abstraction levels.

3.4.1. Message passing

The first of these alternatives is message passing, which is a primitive approach to communication [6]. It requires direct communication between the sender and receiver of messages. The communication can be performed either synchronously or asynchronously.

In synchronous communication, the sender of the message awaits a response from the receiver. This waiting is undesirable, because time could be wasted when the receiver takes too long to respond to the message.

Asynchronous communication allows a sender to perform tasks that are independent of the receiver's response. Instead, the sender periodically checks whether a response is available or awaits the response once no more independent tasks are left to be performed.

Message passing differs from pub/sub in that all participants have to manually manage their communication with others. Meaning that all participants are dependent on each other. It also requires both the sender and the receiver of a message to be available at the same time for communication to occur.

3.4.2. Remote Procedure Call

Remote Procedure Call (RPC) is a form of inter-process communication (IPC), in that different processes have different address spaces. It allows messages to be transferred to other address spaces, for execution by another process or machine, without extensive knowledge of the protocols in between. It does this by omitting unnecessary protocol layers while making the transfer.

RPC allows for communication between multiple systems on a network and is able to send the minimally required information per client since the server can handle per client requests. It does however require extensive coupling between the caller and executor. This will create a lot of overhead when implementing this method into the current code base. The extensive coupling will decrease maintainability in a project of which the maintainability is already really low. It would be a perfect candidate to replace the pub/sub method if it could be coupled loosely.

gRPC[3] is an open source RPC framework. This framework is used by for example Netflix in their Ribbon[4] implementation. Ribbon is a client side IPC library used for load balancing, fault tolerance, multiple protocol support in an asynchronous and reactive model, cashing and batching. Besides Netflix other companies and universities have also successfully integrated this library in their systems.

3.4.3. Shared virtual memory

Shared virtual memory (SVM) is a single address space which multiple processors can access directly. Thus data can naturally transfer between processors [15]. SVM is a communication system that when implemented will have less problems accessing data, however it still has problems with caching.

The main difficulty when building a SVM is that the memory coherence problem needs to be solved. The memory coherence problem entails that the memory in user applications may be adjusted by the data recovery process (retrieving inaccessible data) or accesses happening in a concurrent fashion [16].

[3] `https://grpc.io/`

[4] `https://github.com/Netflix/ribbon`

Disadvantages of SVM include the reduction of system stability and more overhead in the system causing it to run slower.

Compared with pub/sub, SVM stores all data in a big shared memory where clients can pull information from. This makes it difficult to only convey the minimally required information for a client. The client will most likely pull all data from the shared virtual memory where it would check for the information itself. It will render the same result, but it costs significantly more processing power due to the loss of context requiring additional computation to retrieve client information.

JavaSpaces is used for simplifying the creation of applications in distributed systems [17]. "JavaSpaces' support for asynchronous and loosely coupled communication can be used to simplify creation of advanced services in dynamic network-centric environments. In such environments clients and services come and go all the time and system-components may dynamically be added and removed." explained Engelhardtsen and Gagnes [18].

3.4.4. Message queuing

In practise message queuing can be used together with pub/sub [6]. Although, this is not necessary, because message queuing can also be used together with other messaging systems. In a message queue, whenever a consumer node sends a request to the producer this request can enter a queue. These messages can then be processed by multiple message handlers. The order in which these messages are processed generally follows a first in first out (FIFO) or priority based order. Similar to pub/sub the consumers and producers do not need to be aware of each other, because the message queue acts as a middleman. However, the consumer side of the message queue is generally not asynchronous, because of the need to consume the message queue in order [6].

A message queue on its own uses a queue to manage the messages that need to be consumed. This queue is accessed by multiple consumers which will act as handlers for the messages. Even though there are multiple available consumers, there is still only one queue active. This queue can become a choke point in our application, because the consumers need to retrieve the messages from the queue in a thread-safe manner.

Oracle[5] uses an implementation of message queuing. They explain their implementation as: "Advanced Queuing provides the message management functionality and asynchronous communication needed for application integration. In an integrated environment, messages travel between the Oracle database server and the applications and users" [19].

3.4.5. Observer pattern

In the observer pattern, observers are notified by subjects about changes to data they are interested in [20]. Observers need to register their interest with subjects before they can start receiving this data.

The observer pattern differs from the pub/sub method in a few things. The first being that the functionality of the broker is integrated in the publisher itself and is thus not loosely coupled [6]. The publisher would have to keep track of the information of all subscribers that subscribed to it. Secondly, the observer pattern does not allows publishing on a topic or content basis. It publishes to all subscribers in its list. The observer pattern does, however, have a single node where all messages pass through, namely the publisher itself. So, the publishers handles the job of both the publisher and the broker components of the pub/sub design pattern.

An example of how the observer pattern could work in practice is as follows. Assume that there are two observers (A and B) and one subject (X). The subject X will manage it's own observers by itself to send messages. In the beginning the list of observers of the subject is empty, so if an event occurs no observers

[5] https://www.oracle.com/nl/index.html

will be notified. Once observer A subscribes to subject X it will be notified of all events for subject X. Since observer B is not subscribed to subject X, it will not be notified of any future events of subject X, unless it were to subscribe to subject X as well.

3.4.6. Comparison

Our system has to work with a lot of people that each require different information depending on their location and interests. Message passing does not work well with a lot of clients, because they are all dependant on each other. Remote Procedure Call is not desired, because of its extensive coupling and the overhead it would create with implementing it in the existing code base. Shared Virtual Memory stores all its data in a shared memory that everyone accesses, this would send all messages in the system to the memory, which all the clients have to pull from. This would result in clients retrieving messages which are not relevant to them, we want to reduce the messages for each clients thus this is not desired.

The message queue has the same problem as Shared Virtual Memory, because it uses a single queue, all messages are send to this queue. Thus these clients would also get messages that are not relevant to them. Lastly, the observer pattern is not loosely coupled like the pub/sub pattern. In order to send only the relevant data to each client, the publisher would need to keep track of which client to send its messages to, which would unnecessarily complicate the management of messages. The pub/sub pattern is able to handle a lot of clients and more easily only send the relevant data to all the clients. Thus compared to the alternatives, the pub/sub pattern would seem to be best suited for our needs.

4. SYSTEM DESIGN

After researching different approaches and conversing with the client, we have decided to implement the messaging system based on the pub/sub design pattern. In particular, we have chosen to use a topic-based variant.

The messaging system, as we define it, consists of a number of independent interfaces. This allows it to be extended without having to make changes to existing code files. We start this chapter by explaining the three main components of the messaging system: the broker, the policy, and the filter. Then, we show how these components are assembled into a single unit.

4.1. BROKER

The broker manages the distribution of messages between publishers and subscribers. To ensure compatibility with the existing code base, we have chosen to allow publishers and subscribers to interact with the broker without requiring them to implement a specific interface. Publishers can at any time publish a message related to a chosen topic, without having to register themselves with the broker. Subscribers, whom do have to register, can do so by providing a callback to be called whenever a message is published on a topic they have expressed their interest in (Figure 4).

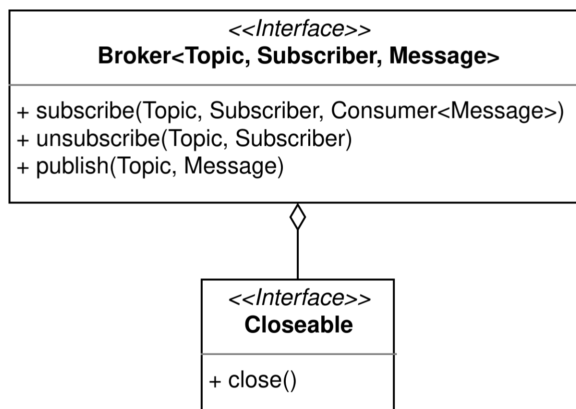


Figure 4: Broker UML

We use an explicit unsubscribe method, instead of returning a subscription object that can be cancelled by the caller, such that subscribers do not have to manage their own subscriptions. Centralizing these subscriptions makes it easier to guarantee thread-safety as well. The data can be stored in specialized structures or be protected using locks. It is important for the broker to be thread-safe as it allows code that uses the broker to be parallelized.

As complex brokers may need to perform additional tasks, such as setting up web-sockets and gracefully shutting down threads. We have chosen to let the broker extend Java's Closeable[6] interface. This extension allows the implementation and use of these brokers without exposing their implementation details to other parts of the system.

The reason the broker interface, and all the other components of the messaging system, are defined using generics instead of polymorphic types is that different implementations may be stricter on the type

[6] https://docs.oracle.com/javase/8/docs/api/java/io/Closeable.html

of topics, publishers, subscribers, and messages they can handle. It protects programmers from passing invalid or incompatible objects to their methods.

4.2. Policy

The policy dictates to what topics users should subscribe and publish (Figure 5). It is important to keep the topics used by the system flexible, as it allows us to tailor it to the game environment without coupling the game and messaging system too heavily, and makes it easier to integrate other projects like the dynamic conits discussed in the problem analysis (Chapter 2).

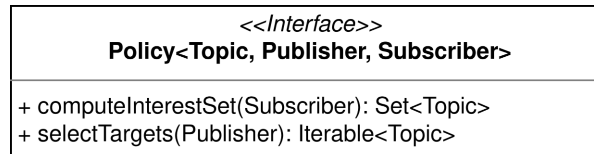


Figure 5: Policy UML

4.3. Filter

The filter allows specific messages to be filtered out before being forwarded to a subscriber (Figure 6). It can be used to reduce the number of messages send to players, or to prevent a feedback loop from disrupting the player experience. For example, a player who is sent the position updates they originally authored may experience rubber banding. Meaning that their in-game character will move back-and-forth between its current position and the position received from the server. The greater the latency between the server and the client, the greater that effect becomes.

Figure 6: Filter UML

4.4. Messaging system

The messaging system manages the interaction between instances of the aforementioned interfaces (Figure 7). The goal of this component is to simplify the integration with the existing code base without infringing on the flexibility provided by the interfaces we designed. The components are connected as follows:

- On update, the system wraps the callback provided by the subscriber in a function which used the filter to determine whether a received message should be forwarded to the subscriber. Then, the policy is used to determine to which topics the subscriber should subscribe and subscribes them to each of them via the broker.
- On broadcast, the policy is used to determine to which topics the provided message should be published. Then, the message is published to each of those topics via the broker.

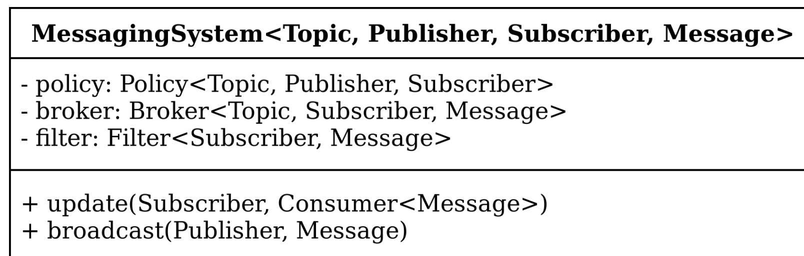


Figure 7: Messaging System UML

5. IMPLEMENTATION

This chapter focuses on the implementation of the messaging system and its components. We also discuss the addition of a specialized executor aiding the generation and distribution of chunk information.

5.1. POLICIES

While the messaging system allows us to swap between different policies, we have chosen to only include a single policy with the current version of the system: the chunk policy. Alternative policies could use unique identifiers for entities or block positions as topics instead of chunks, allowing more fine-grained control over whom receives what message.

The chunk policy determines the topics users should subscribe and publish to based on the users position in the virtual world. It accepts players, the in-game object, as subscribers and all entities, blocks, and chunks as publishers. It determines the interest set of subscribers by retrieving their position and computing the set of chunks within their view distance. The target of any publisher is the chunk in which they reside. As the positioning of each player and the separation of the virtual world into chunks are already available, there is very little additional computation needed to determine the corresponding interests and targets.

5.2. BROKERS

We developed two different types of brokers and one specialization that functions as an extension of the other broker implementations. These are the channel brokers, the Java Message Service (JMS) brokers, and the asynchronous broker.

5.2.1. CHANNEL BROKERS

The channel brokers are written in such a manner that they can be combined with any implementation of the Channel interface (Figure 8). A channel manages the distribution of messages between subscribers of the same topic. Each channel yields different performance characteristics and safety guarantees, but the basic idea is that there is a one-to-one mapping from subscribers to callbacks that needs to be updated and iterated as efficiently as possible.

Figure 8: Broker UML

Unsafe The unsafe channel stores its subscription-callback pairs in a hash map, and provides no thread-safety at all. It should only be used in single-threaded scenarios or by external synchronization of the accesses to the channel.

Read-Write The read-write channel uses an hash map as well, but provides thread-safety through the use of a read-write lock. It locks the write-lock whenever a user subscribes on unsubscribes from the channel, and locks the read-lock whenever a message is published to the channel. Thereby allow-

ing many messages to be published simultaneously while limiting the number of simultaneous changes that can be made to the underlying hash map.

Concurrent The concurrent channel is similar to the unsafe and read-write channels in that it stores the subscriber-callback pairs in a map. However, instead of relying on the user of the channel or using locks to guarantee thread-safety, it uses a concurrent hash map. Allowing both subscriptions to be updated and messages to be published simultaneously without corrupting the underlying data structure.

Guava The Guava[7] channel is based on Google's library of the same name. In particular, the channel is an adapter to the EventBus class. Which, in turn, uses a map of copy-on-write arrays to store subscriber-callback pairs. This adaptation requires the construction of a Guava listener for each of the subscriber-callback pairs.

Now that the channels are properly introduced, its time to discuss the brokers themselves. These are very much like the channels. Similar to the channels, each broker has different performance characteristics and safety guarantees. The broker represents a one-to-one mapping from topics to channels, providing methods to update and access those channels when needed.

Unsafe The unsafe broker stores topic-channel pairs in a map. Similar to the unsafe channel, it uses a hash map and provides no thread-safety. It should only be used in situations where the broker is guaranteed to be accessed by a single thread at a time. It could be iterated simultaneously, but cannot be modified during iteration.

Read-Write The read-write broker stores topic-channel pairs in a hash map and provides thread-safety using a read-write lock. The read-lock is locked whenever a publisher publishes a message to one of the topics, while the write-lock is only locked whenever a subscriber needs to update their subscriptions. Similar to the read-write channel, it allows many messages to be published simultaneously while preventing subscribers from updating their subscriptions at the same time.

Concurrent The concurrent broker stores topic-channel pairs in a concurrent hash map, which is inherently thread-safe. It allows both simultaneous publishing of messages and simultaneous updating of subscriptions.

5.2.2. JMS brokers

The JMS brokers are based on the Java Message Service library, which is part of the enterprise edition of Java (Java EE). It defines an interface for sending messages either point-to-point or via a publish/subscribe system[8]. JMS requires an external system to handle the management and distribution of messages. This section explores JMS itself, explains the adaptation of the JMS interface to the broker interface defined in Chapter /4, and presents two external pub/sub systems which are used to implement the JMS brokers.

While JMS provides both point-to-point and publish/subscribe communication options, the focus in this section is on the latter as it better matches the design of the messaging system. JMS uses message producers and consumers as publishers and subscribers. A message producer has the ability to publish messages to a specific topic, while the consumers can receive from a topic. These consumers can both be durable and non-durable. Durability means that messages that are sent while the consumer is inactive,

[7] https://github.com/google/guava

[8] https://docs.oracle.com/cd/B19306_01/server.102/b14257/jm_create.htm

not available to receive messages, are stored and sent whenever they becoming active. It ensures that consumers do not miss any messages.

The adaptation of JMS to the broker interface requires a number of workarounds. JMS defines both publishers and subscribers, the producers and consumers, to be interested in only a single topic. However, the broker interface allows publishers to publish messages to, and subscribers to subscribe to, many different topics. This mismatch is resolved by generating a message producer for each topic instead of each publisher, and a message consumer for each topic-subscriber pair.

Another incompatibility between the broker's subscriber and JMS's consumer is that the consumer is pull-based, meaning that it needs to be polled for messages, instead of push-based, which means that messages are received via a callback. This problem is resolved by adding a message listener, provided by JMS, to automatically forward messages received by the consumer to the callback of the subscriber.

The last mismatch between JMS and the broker interface is their definition of topics. JMS requires all topics to be strings. However, the broker interface allows any object to be used as topic. A one-to-one mapping from objects to strings is used to resolve this issue. Whenever a new topic is introduced to the broker, it generates a unique string and stores it in the map. Then, the correct string is retrieved from the map whenever a topic needs to be passed to JMS.

Creating producers and consumers is only possible using a connection to the external system which JMS uses to manage and distribute the messages. Establishing such a connection is done via a connection factory. Each external system defines its own connection factory with different configuration options. The connection with the external system is kept alive while the broker is in use, but closed whenever an exception occurs or the broker is closed.

As we would like the JMS brokers to be thread-safe as well, it is required that the subscribe, unsubscribe, and publish methods cannot be called simultaneously, as each of them can cause the creation and destruction of message producers or consumers. The thread-safety is ensured via a reentrant lock, which is a lock that can be acquired multiple times by the same thread, allowing it to make nested, or even recursive, calls to methods that each need to acquire the lock.

The communication between JMS and the external system is performed over the network. Meaning that all messages published by producers and received by consumers need to be serialized and deserialized. A codec interface is defined to allow the serialization of messages to be altered without requiring changes to be made to the JMS brokers themselves (Figure 9).

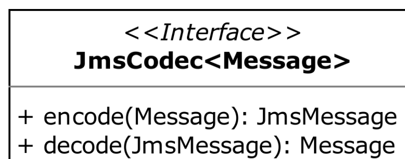


Figure 9: Codec UML

As this project requires the broker to process messages from the flow-network library, we implemented the codec interface using the partial codecs that were already available in the codebase. These partial codecs are each capable of encoding and decoding a single type of message. The composite codec, as we so aptly named it, was constructed as a composition of these partial codecs. Mapping each message type to one of the them. Unfortunately, not all of the partial codecs were completely implemented, which we therefore did ourselves.

We provide an abstract JMS broker that implements all the aforementioned features and workarounds. It requires no more than a connection and codec to function, the first of which is retrieved from the external systems discussed int the next paragraph. The codec can be swapped out depending on the use case of the broker. But, in the Opencraft server, the composite codec presented in the previous paragraph is used.

ActiveMQ Apache ActiveMQ[9] is an open source messaging and Integration Patterns server. It uses the wire protocol AMQP[10] to communicate with brokers running in separate processes. It supports multiple strategies ranging from clustered brokers to dynamic broker creation and discovery. Currently, there are two version available: ActiveMQ 5 and ActiveMQ Artemis.

We choose to use ActiveMQ 5, which works with JMS version 1.1. ActiveMQ Artemi uses JMS version 2, but does not seem to be feature complete. We implement the ActiveMQ broker through our own JMS broker. The connection the abstract JMS broker uses is provided by ActiveMQ via its connection factory. In order for the connection factory to establish a connection, an ActiveMQ server must be running. This server can run either locally or on a remote computer. An URI must be provided to the connection factory to select the correct server, and an additional username and password can be provided to access protected servers.

As each broker instance establishes its own connection, multiple brokers could connect to the same ActiveMQ server, or connect to different ActiveMQ servers to prevent interference from each other.

RabbitMQ RabbitMQ[11] is an open source message broker. It uses the wire protocol AMQP, the same as ActiveMQ. However, in contrast to ActiveMQ, it can be used as a distributed server as well. The broker implementation is furthermore identical to that of ActiveMQ. It provides a connection factory that is used to establish the connection required by the abstract JMS broker class, and multiple broker instance can connect to the same or different RabbitMQ servers.

5.2.3. Asynchronous broker

The asynchronous broker is not an implementation of a broker, but rather an augmentation of another broker. Instead of providing a mapping from topics to channels, or providing an adaptation of an existing library or framework, it extends the functionality of the other brokers by executing their publish methods asynchronously. It holds a reference to another broker and publishes tasks asynchronously by enqueuing them for processing via a thread pool executor[12].

A thread pool executor allows a number of threads to be allocated once and then reused, thereby reducing the overhead of executing tasks asynchronously. The tasks themselves are distributed over the threads allocated by the executor via a blocking queue. The allocation of threads can be handled using at least two different strategies: fixed or cached. In fixed allocation, a constant number of threads is allocated once and kept alive while the executor is running. In cached allocation, threads are allocated whenever they are needed, and kept alive for a constant duration of time after executing their last task. When a new task arrives, that duration is reset.

We have chosen to use a cached thread pool executor, as it minimizes the number of threads required in situations where few messages are to be published. However, the number of threads allocated

[9] `https://activemq.apache.org/components/classic/`

[10] `https://www.amqp.org/`

[11] `https://www.rabbitmq.com/`

[12] `https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html`

should be limited as well. Allocation of threads is rather expensive, meaning that adding more of them than strictly necessary can lead to sub-optimal performance. The number of threads allocated by the thread pool executor is dictated by the number of publishing tasks it has stored in its queue. The use of a counting semaphore allows us to keep track of the number of messages being published, and blocks the publishing of another message once the maximum number of messages has been reached. Limiting the size of the queue itself without the use of a semaphore would also have been possible, but required a customized discarding policy[13].

5.3. Filters

Similar to the policy, we have decided to only implement a single filter. More extensive filters may be required to solve bugs such as players not moving when an explosion goes of near them, or player positions stored on the client and server slowly drifting apart due to floating-point errors. Both of which may be resolved by creating a filter that can distinguish between messages authored by a player or by the server. However, solving these problems is out of the scope of this project.

The feedback filter prevents messages from being returned to their original author. This is required to prevent players from rubber banding. That is, it prevents the server from sending players messages they originally send themselves, which may cause a feedback loop as previously discussed in the paragraph on the filter's design. This particular implementation does so by mapping the message classes that need to be filtered to getters for their authors. We define a getter to be a function that can retrieve a small piece of information with little required computation. When a message is to be send to a player, the filter retrieves the message's author and checks whether the receiving player is the author. If so, it prevents the message from being send. If not, it allows the message to be forwarded to the player.

5.4. Messaging system

The messaging system is the component which brings all the others together. It is build using composition and provides the methods defined in the chapter on system design (Chapter 4). The update method of the messaging system computes the topics a subscriber is interested in using the policy, augments the subscriber's callback using the filter, and subscribes them to topics and unsubscribes them from topics via the broker. The broadcast method of the messaging system uses the policy to determine the target topics of a publisher and publishes its message to those topics via the broker.

The messaging system keeps track of all the topics subscribers are subscribed to, such that the number of interactions with the broker can be reduced. It does so by comparing the stored topics of a subscriber and the topics computed via the policy to determine to which of them to subscribe, and from which of them to unsubscribe.

[13] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.DiscardPolicy.html

6. INTEGRATION

Implementing the messaging system itself and integrating it with the existing codebase are two very different challenges. This chapter focuses on the latter, explaining what changes we have made to the code to take full advantage of the messaging system. This integration lead to a refactor of the chunk streaming code as well, which sends players complete chunks, instead of separate blocks, whenever they enter a new part of the virtual world.

6.1. RELOCATION OF THE MESSAGE GENERATION

The main benefit and challenge of integrating the messaging system with the existing codebase is the relocation of the message generation from the GlowPlayer class to the GlowWorld class.

The benefit of this relocation is a reduction in the number of messages generated. Each player used to generate and send update messages for blocks and entities themselves. Meaning that these messages were generated many times over. However, the messaging system allows a generated message to be distributed to all interested players without regenerating it for each player. Therefore, in a scenario where all players need to receive the same message, the number of messages generated is reduced by a factor equal to the number of players on the server.

The challenge of the relocation is the extensive refactoring required to move the code around. As the codebase is rather large and highly coupled, it is hard to extract pieces of code and integrate them somewhere else. Additionally, most of the messaging code in the original Opencraft server is distributed over many different, and often large, classes. Making it difficult to find out what code should be moved where. Chapter 8 explores these and other code-quality issues in more detail.

The relocation of the messaging code also lead to the introduction of a new class: the area-of-interest. This class keeps track of a user's position and view distance, and each world holds an instance of this class for each of its players. That way, each world can determine which chunks and messages to share with each player. Thereby replacing much of the tracking of chunks and entities that was done by the players themselves.

6.2. OPTIMISATION OF CHUNK STREAMING

While a player is moving around in the world, chunks that have come within view distance of the player are serialized and sent to the player. Chunks that have not yet been explored, have to be generated before they can be serialized and sent. The chunk generation and the serialization of the chunk data have a heavy computation cost. Since the chunk streaming process is all done on the main game loop, the processing of other game components is significantly slowed down. Especially when a large amount of chunks need to be generated and sent.

To fix the chunk streaming slowing down the main game loop, the chunk generation and sending is offloaded to a different thread. This means that the computation cost of generating chunks and serializing the chunk data does not interfere with the main game loop. The exact implementation of the chunk streaming process is explained later in this subsection.

Another issue with chunk streaming in the original Opencraft server, is the order in which chunks are sent to the players. Chunks are prioritised based on the distance between the chunk and the player. Chunks with a lower distance are sent first, such that the player can interact with their immediate surroundings. The problem with how the original Opencraft server handles the prioritisation, is that it is performed per player, instead of on the entire collection of chunks to be sent. Figure 10 illustrates the problem with this approach. This figure shows the chunks within view distance of player A (on the left) and player B (on the right). The chunks around each player marked with an L have been loaded. The

numbers inside the chunks indicate in the distance between the chunk and the player. For simplification purposes all distances are whole numbers, but it is based on the euclidean distance.

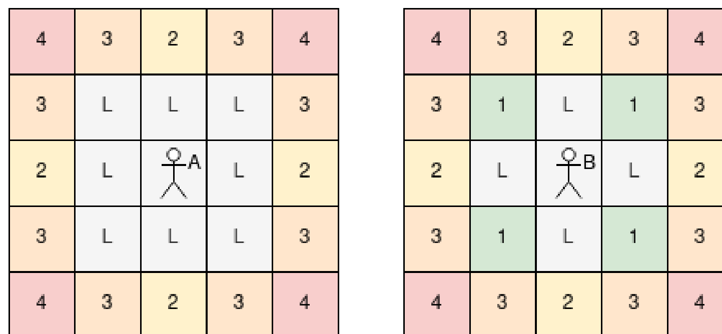


Figure 10: This figure shows the chunks that are within view distance of two players. The numbers in the chunks indicate the loading priority. Chunks marked with an L have already been loaded.

The original Opencraft server handles the situation in Figure 10 as follows. Assume that player A joined the server before player B. Both players receive their chunks ordered by distance, but player *A* receives *all* their chunks before player *B* starts to receive theirs. This happens because the chunk streaming process is done for each player individually and the order in which the players joined the server determines the order in which the chunk streaming processing is performed. In other words, players have priority over other players that joined after them. Combined with the heavy computation cost for generating previously unexplored chunks and late joining players will have to wait a significant amount of time before receiving new chunks.

To solve this issue, the chunk streaming process is now centralized so that all the chunks to be generated and sent chunks are prioritised based on distance alone. Thus the join order does not influence the order in which chunks are sent. Whenever a chunk comes within the player's view distance, a chunk streaming task is created for that chunk. This task is then enqueued to be executed by a thread pool. This thread pool is also how the chunk streaming process is offloaded from the main thread. The thread pool prioritises chunk streaming tasks based on the distance between the chunk and the player that it is being streamed to. Once a tick, the tasks for streaming chunks outside of a player's view distance are dequeued, because the chunk data associated with the task has become irrelevant, and the priorities of remaining enqueued tasks are recomputed.

A custom made sortable blocking queue is responsible for handling the prioritisation process of the chunk streaming tasks. This queue is what allows the tasks to be reprioritised, removed, added and sorted efficiently as a single transaction. Meaning that no other thread can interact with the queue while these tasks are being performed. Existing implementations of blocking queues can not do all of these steps at once without externally locking the queue, and thereby incurring additional synchronization overhead.

To do the chunk streaming process asynchronously, it is required that the chunk generation process can handle asynchronous access. This had to be implemented, since the chunk generation of the original Opencraft server is not capable of handling asynchronous access. To enable asynchronous access to the

chunk generation process, a simple reentrant lock is used that ensures all operations occur sequentially.

Technically it should be possible to asynchronously generate chunks as long as the chunks being generated are not close to each other. However, as the population of chunks requires many neighbouring chunks to be edited at the same time, performing chunk generation in parallel is non-trivial. Population is the process of adding terrain features like caves, lakes, and structures to the virtual world. It would require many chunks to be locked in a specific order to prevent deadlocks.

With these changes to the chunk streaming process the situation in Figure 10 is handled as follows. All chunks are streamed in order of distance, independently of the player that needs to receive them. So player B on the right, receives the chunks with distance 1 first. After that both players receive the chunks with distance 2 at around the same time. This goes on until both players have received all chunks within their view distance.

7. VALIDATION

Large parts of the codebase are currently untested and poorly documented. Those parts are often highly coupled with other classes and are deemed necessary for making changes to the server. This means that additional code can not always be guaranteed to work if it has to work together with a highly coupled class. Our approach to the validation of these changes is discussed in Section 7.1. Most of our changes should mimic the behaviour of the original Opencraft server or, if possible, the behaviour of the vanilla Minecraft server. Which is easier to verify by comparing them side by side, instead of testing every possible game state. In the following table the added features and their respective validation targets are mentioned.

Features	Validation target
Entity collision	Used to prevent the physics from causing never ending updates
Explosions	Used to test the server with a short burst of messages
Falling blocks	Test propagating update messages
Water flow	Test propagating update messages
World generation	Used to make sure that would not be any abnormalities
Multiple dimensions	Test if the broker does not break when switching dimensions

7.1. AUTOMATED GAMEPLAY TESTING

One of our ideas was to use a bot to test the current Opencraft server implementation in order to find bugs easier. The bot we finally decided to use was an extraction of the bot used in Yardstick itself. This bot implements a few simple features and is mostly incomplete, but it would be fine for testing purposes. We wanted to create an interface for the Minecraft bot and test with multiple bots at the same time and all give them their own separate space to keep operations from interfering with each other. Bots would be moved to these spaces by letting them call '/teleport'. This was unfortunately not possible due to the servers' way of handling players.

The Opencraft server assumes that the player's actions are correct and expects the player to do all simulations for itself. The server does not verify any action of the player. This makes verification of behaviour difficult if not problematic. A bug in the bots' behaviour or player actions that should not be permitted would not be caught by the server. This often resulted in strange behaviour where the bot could break blocks behind other blocks or ignore walls. This discovery lead to the decision to stop the development of a bot verification testing utility. It would require a big structural rewrite on the server side before the testing tool could be used. That rewrite is out of scope for our project.

7.2. ENTITY COLLISION

Entities in the Opencraft server were implemented poorly. Entities would float due to the lack of gravity, which in turn meant that entities were not affected by ground friction. The code for gravity acceleration calculations was present, but the gravity vector was set to $(0,0,0)$. The reason for the vector being $(0,0,0)$ was unclear, until we added a negative acceleration value to the y-axis. Entities did not yet support collision and would fall through the ground, like in Figure 11. This behaviour made manual testing and

Figure 11: A figure showing the effects of having no gravity

debugging a lot harder. Entities are often used to check consistency between players. It is an rudimentary check that allows for quick consistency comparison. This pushed us to fixing the collision code.

Collision does not work as intended when turning on gravity, as seen in Figure 12. The then implemented collision detection method applied the current velocity to the entity location and checked that location for the presence of a solid block. This construction allowed entities to pass walls when traveling fast enough within one tick. It also meant that entities would not stand perfectly on the ground since collision would occur within the length of one velocity tick. The implementation brought along other issues such as suffering from fall damage in water and falling through the void check which would result in the entity not dying.

We improved the entity collision system to prevent entities from falling eternally and in turn generating an overabundance of messages which made it a lot harder to debug the messages. The collision issue was partially solved by using Axis Aligned Bounding Boxes (AABB)[14] to check for collision. At first, the collision detection checked all subsequent locations between the entity before velocity application and the entity after velocity application. the code will stop collision checking when we encounter a solid block. This blocks' hitbox will be used in combination with the entity's hitbox to calculate the overlap between hitboxes. This overlap will then be used to push the entity back that amount in order to ensure that the entity is not actually in the block itself. This ensures collision detection as shown in Figure 13, but this solution was not made for collision on the x and z axis. It also became apparent that the current block bounding box implementation always returned an 1x1x1 bounding box.

The collision not working on the x and z axis pushed us towards implementing a solution that does. We were already improving collision and thus it seemed better to implement it correctly and document it well. The code now uses Swept Axis Aligned Bounding Boxes (Swept-AABB)[21] check for collision. This algorithm calculates the percentile of velocity on the entity that can be executed before collision. A response will be calculated for the leftover percentile. In our implementation we use the slide response specified in the specifications of swept-AABB[21]. It calculates the leftover velocity by multiplying the velocity with the leftover percentile. The leftover velocity will be projected to a vector that is perpendicular to the surface normal of the collided surface. This velocity will then be used to repeat this process

[14]https://en.wikipedia.org/wiki/Minimum_bounding_box#Axis-aligned_minimum_bounding_box

Figure 12: A figure showing the effects of turning on gravity

in order to prevent accidental collision during sliding. The process will halt when the left over velocity's size is equal to 0.0.

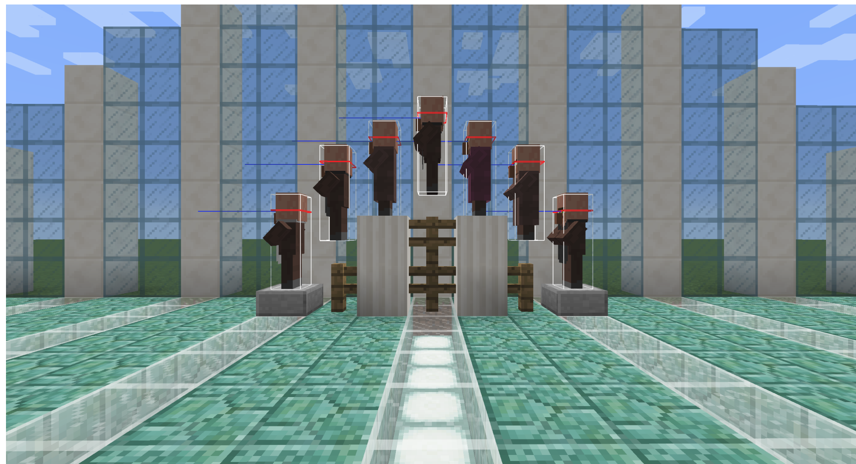
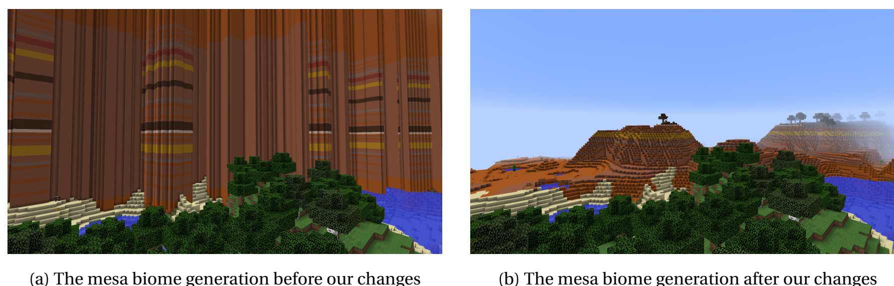


Figure 13: A picture showing the new implementation of in game entity gravity

The other issue we encountered in the code was the missing of correct bounding box definitions for blocks. Every block was assigned a standard bounding box of size-1. This meant that fences, pressure plates, snow, doors, gates, etc. were not working as intended for entities. We resolved this by creating a 'BlockBoundingBoxes' class that builds the required bounding boxes for a block and returns them. The server now supports custom bounding boxes for blocks that are not of size 1 in the game as seen in Figure 13.



(a) The mesa biome generation before our changes (b) The mesa biome generation after our changes

Figure 14: a) shows a picture of the previous mesa biome generation and b) shows a picture of the current mesa biome generation

7.3. Block updates

We found out that not all block updates were working correctly. This resulted in a lot of missing block update messages. These block updates are essential in testing and validating our messaging system. If these updates do not work we will not be able to test whether the updates are sent from and to the broker correctly. The three major block updates we focused on were the falling blocks, explosions and the water flow.

7.3.1. Explosions

Minecraft is a game that has a lot of explosions. Creepers, TNT, and Ghasts are all based on explosions and their functionality would be highly limited without them. We noticed during development that explosions were not functioning correctly. Velocity would often be applied to entities worldwide and only to one specific direction. Getting knocked back into an explosion instead of away from an explosion felt counter intuitive as player. Knocking back entities that did not even have the explosions within loaded view distance created some strange situations as well. It seemed logical to fix explosions such that they could be used to test large amount of block updates. Our assumption was that blowing up a lot of TNT is one of the easiest ways to generate a significant amount of load on a server on your own. We were proven right, after fixing the aforementioned issues, the fix greatly increased the debugging and development time of the messaging system.

7.3.2. Falling blocks

Blocks like sand, red sand and gravel are blocks that will fall down when there is no supporting block below them. It is possible for a falling block to float in the air when they are first generated in the world and have not been interacted with. The simple falling functionality of these blocks was working correctly. When the falling block was placed on air or when the block below a falling block was removed it did fall to the ground.

The problem we found, was that a falling block was only updated when a block directly below had been removed. While a floating sand block should also fall down when any adjacent block is updated. So if for example, a sand block were to be floating in the air, it should fall down if a block was placed or removed adjacent to it, not only if a block below was removed. Furthermore, if there would be a group of sand blocks floating in the air and one block were to fall down, the whole group of sand connected to that block should also fall down. This missing behaviour resulted in a lot of block update messages to be missing and could thus not be validated.

This problem was resolved by using a method from the GlowBlock class that is called whenever a block is updated within a 3 by 3 radius of the current block. So whenever a block within a 3 by 3 radius

has been updated, the current falling block will check the block that was updated is adjacent to him and update himself accordingly. This will also propagate to other falling blocks, because if a block where to fall down it will also cause an update.

7.3.3. Water flow

Besides falling sand, the water flowing mechanism was also not working properly on the original Opencraft server. This resulted in infinite water pools not forming and water not spreading when a block next to the source block is broken. Both of these mechanisms can result in multiple update messages being sent to the subscribers. In order to validate that the messages are actually sent over the messaging system these block update messages need to be sent over the system. Validating this together with our messaging system is important to know if the system works correctly. That is why we decided to implement this next to our existing project.

First the water spreading issue was tackled. Here the wrong block was being checked as the target block, the block where we need to flow towards. This meant that water would not flow when the water was 1 block deep. Checking the right block fixes this issue and ensures that water flows in all situations.

Next up was the issue that two water blocks could not combine to make a new source block. In order to make this work we needed to perform a couple of checks on the flowing block, the target block and the surrounding blocks. Both the target and the flowing block need to be water and the flowing direction needs to not be down. After these checks we need to count how many of the blocks surrounding the target block are source blocks. If this amount is two or more, we can conclude that the target block must be a source block as well and the strength is set to that of a source block.

When both of these issues were fixed we discovered that the ocean generation and update propagation was wrong, which caused lot of lag when an update was happening. This was because in the original Opencraft implementation all water block updates were propagated across the entire ocean, where some caves were generated within water. These blocks would be filled up which would take a large amount of time in which the server was not responsive. We fixed both these issues by making sure the block updates do not propagate across all water blocks when one block is updated. Besides this we also made sure that the terrain generation would not spawn any caves within an ocean which reduced the messages being sent and made this look a lot more like default Minecraft.

7.4. Mesa biome generation

The mesa biome was not being generated properly on the original Opencraft server. We stumbled across this problem by flying around whilst we were stress testing the version of Opencraft we started with. The problem with this is that when a biome is not generated properly there can be some messages that aren't sent when they should be or messages that are sent when they shouldn't be. This does not give a realistic representation of how the messaging system can perform. Because of this we decided to implement the correct mesa biomes in order to validate whether the messages related to the mesa biome are handled correctly.

In the original implementation, the terrain would be generated as shown in Figure 14.a. Here, the hard clay blocks of the mesa biome are generated from surface level to the sky/build limit. The same problem occurred in the mesa forest biome, except coarse dirt blocks were generated until the sky/build limit. Besides the blocks not being in the right place, the sand was also missing on the bottom of the canyons. These problems lead to debugging, since finding the problem in the code was not straightforward. However, in the end we did manage to fix these problems.

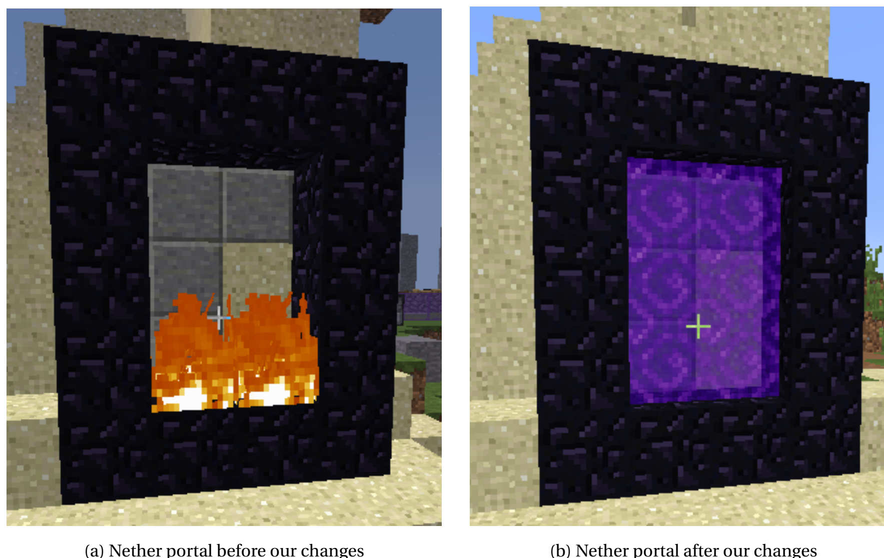
In our implementation, the mesa bryce and the mesa forest biomes are generated correctly as shown in Figure 14.b. The bottom surface of the mesa bryce biome is covered by red sand, which stops at a

certain height, while the canyons use different colored layers of hard clay. The biome generation is now similar to that of the default Minecraft.

7.5. Multiple dimensions

We not only looked at the scale of biomes, but also on the scale of the world itself. Each world has its own broker that handles the messages that have to be send to players in that world. Therefore when a player travels from one world to another, chunk streaming and generation should keep working. When a player travels to another world like the nether, all chunks should be unloaded from the previous world and the new chunks should be loaded from the new world. However, when trying to validate if this was working correctly we found out that nether portals did not work properly. Thus in order to test whether travelling between worlds was working as intended and that the messages are correctly handled in our messaging system, we also implemented the portal creation feature.

The original implementation of the nether portal as shown in Figure 15.a was flawed and did not function. At first portal blocks where placed in one pillar on top of each other as long as the block they were replacing was air. The portal blocks that were placed also did not have the correct texture. After the blocks where placed no checks where made whether the portal was still intact and should or should not be removed.



(a) Nether portal before our changes

(b) Nether portal after our changes

Figure 15: a) shows how creating a nether portal looked before b) shows a picture of the current nether portal creation

There are multiple things that went wrong here. First of all, when trying to create a portal by placing portal blocks, the game has to check if there is a valid portal. A valid portal consists of a rectangle construct made from obsidian, this construct should have a space of at least 2 blocks of air wide and 3 blocks of air high between the obsidian blocks. The portal cannot be obstructed by other blocks or liquid. Furthermore a portal block should have a texture to indicate that the portal is active, this portal block should have the correct orientation, as a portal can be placed along both the x and z axis. Lastly when one portal block is removed or an obsidian block that forms the outer layer of the portal is broken, all the portal blocks inside this portal should be removed, because the portal is not valid anymore.

The nether portals in our implementation as shown in Figure 15.b checks if the portal is valid, uses the correct portal textures and supports various portal sizes. It is also important that the portals are able to transport an entity to the nether, so this was also implemented in order to make sure that all entities are able to travel to another dimension. We did not implement the other features related to portals, like linking portals between dimensions and the creation of a matching portal in the nether. These extra features are not needed for the validation of messages related to travelling between dimensions.

After we implemented the nether portals, we could test whether travelling to and from the nether was working as intended. When we were testing this, we did find a mistake related to chunk loading. To know what new chunks have to be loaded, we need to keep track of the player's previous location. The reason this is required is because we have to know if the player has moved from its old position. The problem was that the previous location of each player was stored in the worlds itself, thus when a player traveled to another world the chunks in the old world did not get unloaded properly. This happened, because when the player traveled to another world, the old world stopped updating the local previous location of the player.

This was solved by storing the players from the previous pulse in a list. With the use of the previous players we can find out if a player has joined or left the world by comparing it with the current players. We know a player has left if the player is in the previous players list but not in the current list of players. With this information we can make sure all chunks are properly loaded and unloaded when a player joins and leaves a world.

7.6. Bugfixes

During testing we played through the basic Minecraft behaviour on the Opencraft server. This uncovered some bugs or features that were not implemented. We chose to fix a couple of these because we thought they would be rather easy to fix and improve the overall quality of the Opencraft project. We did not spend a large amount of time on any of these bug fixes, however it does improve the ability to play survival mode on the Opencraft server, which should be a goal for the Opencraft team. And it reduces the change of unexpected behaviour happening during testing. Other bugs we found but did not fix will also be mentioned here.

7.6.1. Underwater mining

The Opencraft project did not implement a penalty for mining under water. We implemented this by adding a penalty variable that is 1 by default. When the player's head is under water it multiplies this by 5 and when the player is not touching the ground it multiplies the penalty by 5 again. This penalty is used to calculate the amount of ticks needed to break a block. With the current implementation this will take longer when the player is under water. The numbers used are based on the Minecraft Wikipedia[15].

Besides a mining penalty for mining under water there should also be a penalty for mining under lava and for different status effects (e.g haste and mining fatigue). Mining under lava has the same implementation as for under water except that your player takes damage whilst doing so. The haste status effect increases your mining speed whilst the mining fatigue status effect decreases the speed. The numbers used for these status effects are also based on the Minecraft Wikipedia.

7.6.2. Banner placement

Another minor bug we found, was that placing banners didn't work. Banners consist of two different types, wall banners and standing banners and both types didn't work. Whenever a banner was placed it caused an exception. When placing a banner, something went wrong with setting the block type and

[15] https://minecraft.gamepedia.com/Breaking

caused the banner placement to fail. This was fixed, by making sure the correct type of banner was selected when placed. Now we can place both standing and wall banners that also face the correct direction.

8. CODE QUALITY

When we started this project, we did not start with a clean slate. There were already around 80000 lines of code present. This section will focus on both the code quality of the original code and the code of our messaging system.

The Opencraft section will explain the problems from the existing code. When we were working with the existing code, we found a lot of problems related to the code quality. Besides our own analysis and finds, our code quality was also measured by a third party.

The code quality was measured by the Software Improvement Group (SIG) who work with a star based grading system. The star grade for the entire system is based on metrics SIG provides. These metrics include duplication, unit size, unit complexity, unit interfacing, module coupling, maintainability, volume, component balance, component independence. The SIG report show how these metrics and our test coverage have improved compared to the original version with a rating between the 0.5 and 5.5 stars.

8.1. OPENCRAFT

The Opencraft part of the project is seen as the entire code base of the original server. Much of this code was not touched by us unless it was related to our messaging system. The original code of Opencraft had some major problems that need to be fixed. As most of these problems are not related to our messaging system and take too much time for us to fix, we will not be able to improve these problems ourselves.

The problems mentioned in this section include god classes, the problem regarding inheritance, the missing mathematical classes, the test coverage of the project and the large methods which have too much responsibilities.

8.1.1. GOD CLASSES

The project has a couple of god classes, classes that has too much influence over the code, which makes the project hard to work with. Some of these classes also have really large methods. The method with the largest lines of code in this project has 469 lines. There are also some classes that just have a list of game objects which already makes the class quite big before any functionality has been implemented. Or classes that implement the Glowkit/Bukkit interface which are forced to implement a large number of methods. These classes are hard to fix, because these interfaces are necessary to support Glowkit/Bukkit plugins.

There are however other classes that are easily fixable, these classes have one or a couple very big methods. If these methods are split into multiple methods and move them to other classes. This will decrease the unit size of the classes and improve the maintainability of the project as a whole.

Another way to fix the god classes easily is to make use of proxy methods. The way this works is keeping the methods that exist in the classes the same but implementing the functionality in another class so we have a clearer division of tasks within the classes. We used both of these in order to improve the maintainability of our code base.

8.1.2. INHERITANCE VERSUS COMPOSITION

In Opencraft, all the entities use inheritance. This can become problematic when the complexity increases. For instance, if a new horse and minecart entity were to be created that both have storage functionality. Then in the inheritance model you would either have to duplicate this functionality or implement this functionality in the entity class, since the entity class is the earliest common ancestor. The downside of adding this functionality to the entity class, is that it results in all sub classes having this functionality, even though they do not require it.

A better solution to this problem is with the use of composition instead of inheritance [20]. Instead of having an 'is-a' relationship, you have a 'has-a' relationship with composition. For instance, if new entities for the horse and minecart with storage functionality were to be added to a system that uses composition. Then, the problem can be solved by creating a new entity that is exactly the same as the horse and minecart entities, with the only difference being that they both have a new 'storage' component. As previously mentioned inheritance does not have such a simple solution to this problem.

In the SIG report, this problem is reflected by the module coupling metric. The module coupling of the original code was given 1,7 stars. This means that a lot of the classes were strongly coupled together which makes it very hard to modify, test and analyze the code. There are some obvious classes that can be changed in order to improve this rating.

The classes GlowPlayer, GlowServer, GlowEntity, GlowBlock are intertwined with nearly everything in the application. These classes, including a number of base classes for entities and blocks, have huge dependency graphs. To reduce this, we would propose the introduction of an ECS, such that the definition of game objects can be simplified and the large inheritance structures can be removed.

An Entity Component System [22] (ECS), which is an advanced technique commonly used in game design, follows the composition over the inheritance principle. The core difference between a composition based approach such as ECS and an inheritance based approach, is that the data is not directly tied to the entity itself. This data-centered approach allows ECSs to make more optimizations.

Because the inheritance problem is so deeply rooted in the entire code base we will not be able to improve the module coupling score by much. The implementation of an ECS is not related to our messaging system and would require too much work. When we will write new code however, we will make sure that the different components will be loosely coupled and that the module coupling score for this code will be higher.

8.1.3. MATHEMATICS

We noticed that the mathematics classes were not implemented in a practical way. This resulted in a very high number of parameters per method. The SIG score of our project reflected this as well, as the score for the unit interface was only 2,1 stars.

One class that can help with mathematics and improve the unit interfacing score from SIG, is the addition of proper Vector classes (Vector$(I|F|D)$(2|3|4)). This would dramatically reduce the number of parameters on all methods. Currently parameters such as x, y and z are all given as parameters to multiple methods, while a single Vector3D class would suffice. There are also libraries that add similar mathematical capabilities that could further improve the unit interfacing score, a good library for this is JOML[16].

However, some of these methods are part of the Glowkit API. Meaning that we cannot change them without losing compatibility with existing plugins. It should be possible to create an adapter for each Glowkit interface, but this would defy the purpose of increasing readability. Improving the unit interfacing score of the original opencraft code has almost nothing to do with our messaging system and is out of the scope of this project. We will make sure however that code related to our messaging system will have smaller interfaces.

8.1.4. UNIT SIZES

When we first looked at the code we discovered that some methods were not implemented they way proper code styling would indicate. The project has some large methods which make the project hard to

[16] `https://github.com/JOML-CI/JOML`

work with. The method with the biggest lines of code in this project has 469 lines of code. The responsibility of these methods is too large which means that they need to be divided into multiple methods. This will result in methods generally having only one responsibility.

There were also some methods that had a large unit complexity, which means that they have a high amount of execution paths. A high unit complexity will often result in harder to understand code and will always result in more test cases for that one method. A high unit complexity often goes hand in hand with a high unit size. This is generally bad code styling and should be solved the same way a high unit size is solved, splitting the methods.

The SIG report we received agreed with this statement and said that the unit size and complexity of the opencraft project were pretty bad. The score for unit size was only 2,5 stars and the unit complexity score was 2,1 stars. The adjustment mentioned above should be able to decrease the average length of methods and decrease the unit size.

8.2. Messaging system

The messaging system is the part of the project that we were in charge of. This was entirely tested and implemented by us and we tried to make sure that the code quality was as high as possible. This was reflected when we received the SIG report over the code of the messaging system (Appendix E). The SIG gave us 4.2 stars for the new code quality. This is a pretty good mark, however there were still some parts that were not up to the standard we had in mind. These sections of the SIG report were unit size and unit complexity. These sections were improved by resolving two issues where the code quality was low. We improved this quality in the collision detection and chunk streaming.

8.2.1. Collision detection

The collision detection code can be split into two separate parts: The block bounding boxes and the collision method. The block bounding boxes contributed to large unit size due to the large amount of edge cases that can be found in the boundingboxes of Minecraft. A lot of blocks have different shapes with no real explanation as to why exactly those shapes. This lead to a large switch statement that addressed all those cases. the SIG addressed the switch statement in their report saying that it has a high McCabe complexity and unit size. We decided to fix this issue by using sets for matching an splitting the method up into functions.

The second issue was the collision method. The collision code was all put into one method where every axis was addressed multiple times in doubles. This gave the method a big unit size and also a big McCabe complexity according to SIG due to the amount of if statements present. The doubles were switched to vectors which reduced the amount of values used by a factor of 3. Vectors also allowed for the use of vector specific methods which allowed the code to be shortened even more. Some parts could be described mathematically without using if statements after making some extra vector utility functions. This eventually allowed the method to be lowered in complexity. Part of the code as however harder to refactor due to the fact that it requires a lot of Boolean operations to calculate the closest surface normal with which collision occurred. This code was put in a separate method with a clear description of what it does. Refactoring the method any further from this point would not increase the code's readability and instead just be an example of chasing the metric.

8.2.2. Chunk streaming

One of the methods that we changed and implemented for our messaging system was the chunk streaming method. Whenever a player moves to a new chunk, it has to check what new chunks to load and what old chunks to unload according to the view distance of the player. This method became quite large,

because it has to do relatively much. The SIG report, showed that for the new code the unit size was not as high as we wanted (3,2 stars). One of the refactoring candidates was the chunk streaming method.

We decided to completely refactor the chunk streaming method. Instead of one method that does everything, we decided to split the functionality into multiple parts and write more compact code. First of all we split the chunks that have to be loaded and unloaded into two separate methods. Furthermore, the way these methods work are relatively similar, they both have a double for loop to check for new or old chunks. We found that we could create another method that can be used both for the loading of new chunks and the unloading of old chunks, this way only one double for loop is required and code duplication is reduced. With this refactor the unit size of the chunk streaming is reduced by quite a bit.

8.3. Testing

The SIG report also illustrated the test coverage of the project, which was 9%. Usually the coverage on a project is higher then this. We got the test coverage up to 13.7% by testing the features we added and writing tests for some code we adjusted. We wanted to make sure that what we implemented was also tested correctly, because it makes adjusting and maintaining the code easier. We managed to achieve a 90% test coverage on the code of the messaging system.

9. Experiment setup

This chapter focuses on the setup of the experiments that are used to measure how well the messaging system and the Opencraft server perform. The main focus of these experiments is on the different broker implementations and their impact on the latency and tick rate. We want to know the performance characteristics of the different brokers and whether the integration of the messaging system in the Opencraft server has improved its scalability.

In this chapter, we explain the environment in which the experiments are run. Afterwards, we explain the various metrics and data that are collected while running the experiments. Furthermore, we explain the tools required for running and evaluating the experiments. Lastly, we explain how the benchmarks are generated, ran and visualized.

9.1. Environment

This section discusses the environment in which the Opencraft project is evaluated. Discussing the environment is important, because the environment influences the results of the experiments. So in this section it is explained why the experiments are run on the DAS-5 supercomputer. Other environment details, such as the Java version used to compile the Opencraft server, are also explained.

9.1.1. DAS-5

All the experiments are run on the DAS-5 distributed supercomputer [23]. The DAS-5 distributed system is designed by the Advanced School for Computing and Imaging (ASCI) and is distributed over six clusters from the participating universities and organizations. DAS-5 has approximately 200 compute nodes available for use and uses the CentOS[17] Linux operating system.

The reason for using DAS-5 for running the experiments, as suggested by our client, is that they are run in a controlled and stable environment. The nodes of DAS-5 are homogeneous[18] on the TU Delft cluster we use. The experiment results are thus comparable even if they are run on different nodes.

Each experiment is run on a node of DAS-5. To ensure that users do not interfere with each others experiments, the Slurm[19] job scheduler is used. With Slurm, users can reserve nodes for running experiments. Furthermore, it allows users to setup experiments that need to be executed on a distributed system. One of the goals of Opencraft is to achieve scalability using distributed systems, thus running the experiments on the DAS-5 distributed supercomputer provides relevant experiment data that shows the potential performance on a distributed system.

9.1.2. Other environment details

To compile the Opencraft server, the messaging system and the messaging system benchmarks we use java 8. Furthermore, to run the banchmarking scripts we use python 3.6 and bash. The ActiveMQ broker uses ActiveMQ 5.15.13 and the RabbitMQ broker uses RabbitMQ 3.8.5.

9.2. Metrics

We require a number of metrics to quantify the performance of the messaging system and the Opencraft server during the experiments. This section first discusses the round-trip time, which is a metric for the messaging system. Afterwards, the tick rate and relative utilization are discussed, which are metrics for the performance of the Opencraft server.

[17] https://wiki.centos.org/action/show/Manuals/ReleaseNotes/CentOS7
[18] https://www.cs.vu.nl/das5/special.shtml
[19] https://www.schedmd.com/

9.2.1. ROUND-TRIP TIME

The round-trip time (RTT) is measured as the time between the broadcasting of a message and the receiving of the response. It is an important metric of the messaging system, as it is an indication of how long a subscriber would have to wait on a message that was published.

9.2.2. TICK RATE

The tick rate, or tick frequency, is the amount of ticks processed per second. A tick is one cycle of the main game loop of the server. Every tick, the game world simulation advances a little bit, the behaviour of entities are updated, the player is affected by effects such as hunger. Minecraft has a default fixed tick rate of 20 ticks per second, we refer to this as a tick rate of 20. If the server is unable to keep up with all the changes in the game it is possible that the tick rate drops below 20. For this reason the tick rate is a good metric to analyze the server performance.

9.2.3. RELATIVE UTILIZATION

As mentioned in Section 9.2.2, the tick rate of Minecraft starts at a fixed rate of 20, this means there is an interval of 50ms in between the start of two consecutive ticks (tick interval). A tick can be split into two sections, the tick duration and the tick wait duration. The tick duration is the time it takes to process one tick. After this the game loop pauses and waits for the next tick to start, this is the tick wait duration.

The relative utilization, which is the term used by Jerom van der Sar in the Yardstick paper [3], can be defined as how much of the server process is being used to process a tick. The formula for the relative utilization is:

$$\frac{t_d - t_s}{t_i}$$

t_d indicates the end time of the tick duration, t_s indicates the start time of the tick and t_i indicates the tick interval, which is 50 ms as mentioned previously.

The relative utilization is under normal circumstances a number between 0 and 1, because the duration of a tick is then shorter than the tick interval. If the relative utilization goes above 1, it means that the ticks run at a lower frequency and the server may slow down. When this happens, players can start to experience lag and can thus no longer smoothly play the game.

9.3. TOOLS

To run the experiments and measure the results properly, certain tools are required. Each of these tools is used for one or more of the experiments. This section discusses the Java Microbenchmark Harnass[20] (JMH), Yardstick, and a logger.

9.3.1. JAVA MICROBENCHMARK HARNASS

The Java Microbenchmark Harness is a framework that assists in the implementation of performance benchmarks in Java. A benchmark built using JMH evaluates the performance of a single code snippet by repeatedly executing it and measuring the time between its invocation and completion. It provides a number of features that allow the benchmarks to be tweaked, both in performance and accuracy.

- The benchmark can be run as a fork, meaning that it runs in its own isolated process. This reduces the previous runs and code other than the snippet being benchmarked from influencing the benchmark results.

[20] `https://openjdk.java.net/projects/code-tools/jmh/`

- The number and duration of warm-up iterations can be increased, such that the impact of branch prediction and other runtime optimizations on the benchmark results is reduced.
- The number and duration of measurement iterations can be increased, such that the standard error of the benchmark results is reduced.

9.3.2. Yardstick

Yardstick [3] is a tool to compare the performance of Minecraft-like games. Yardstick is capable of emulating players by using bots that simulate player behavior. These bots can thus be used to test the performance of the Opencraft server, without needing to use real players. While each bot is running it bot is assigned tasks according to a player behaviour model. The models will differ for the experiments based on what behaviour we want to emulate.

Although Yardstick is able to collect measurement data using Prometheus [3], this functionality is disabled in our experiment. Instead we use low-overhead logging, which is less bulky than for instance Prometheus. Furthermore, this logging collects data on the server side instead of on the client side to avoid discrepancies.

The benchmarks measurements can occasionally vary by a large margin when running the walk and interact experiment of Yardstick. Yardstick does not always produce the same workload for the server. The bots can sometimes break their only path away from spawn, which causes the bots to group together and eventually get stuck within a small area.

9.3.3. Logger

In multiple experiments some form of logging is used to collect measurements. In all of these instances the logging is done asynchronously. This means that the call to the logger is returned as soon as possible and all disk I/O is done on a separate thread. Asynchronous logging is important for reducing the amount of interference the logging itself has on the performance of the elements that are being tested.

9.4. Benchmarking method

Due to the amount of benchmarks, all with a variety of configurations, it turned out to be infeasible to run and configure the benchmarks all by hand. To give some perspective, there are nearly 300 benchmarking configurations necessary to run the experiments, not counting the discarded configurations that were deemed insufficient. Consequently, we wrote multiple benchmarking scripts in python that automatically generate the required files for running a benchmark. To run these experiments on DAS-5, we also wrote scripts that could automatically run the generated benchmark configurations. These benchmark runner scripts are also capable of automatically allocating the required nodes for each benchmark.

The configuration files for each benchmark are created using predefined templates that are instantiated with the configuration options for each benchmark. Each template is a Slurm job file that contains the Slurm configuration options for each benchmark, for instance the amount of nodes to allocate. Besides the Slurm configuration options, the templates also contain the general procedure for running a benchmark. For instance, the template for benchmarking the ActiveMQ broker specifies the procedure for setting up, running and stopping ActiveMQ.

Once the benchmarks have finished running, another python script collects the measurement data and converts it to a format that is easier to use when visualizing the data. This intermediate format is CSV, as it made it easier to manually review the data as well. Visualization of the data is done through pandas[21], which is used to parse and manage the data, and plotly[22], which is used to render the graphs. Each

[21] https://pandas.pydata.org/
[22] https://plotly.com/

graph shows a number of traces based on the mean of the measurements, including whiskers showing the 95% confidence interval.

10. EXPERIMENTS

In this chapter, we discuss the experiments conducted and the main findings that resulted from those experiments. These findings are as follows:

1. JMS and asynchronous brokers have too much overhead to be used in the Opencraft server.
2. Thread-safe channels have additional synchronization overhead while not providing any additional thread-safety. Therefore, they should not be used in combination with an already thread-safe broker.
3. The Opencraft server scales better, meaning that it supports more simultaneous players, with than without the messaging system. However, it must be configured with the correct broker, as not all of them outperform the original Opencraft server.

10.1. MESSAGING SYSTEM LATENCY

It is important to provide players with a responsive gameplay experience. This means that whenever a Minecraft player breaks a block or hits an entity the response to the actions should appear within a certain amount of time. If the player only receives confirmation that it hit another entity hundreds of milliseconds, after the action occurred, it would would dramatically diminish their gameplay experience.

To measure the responsiveness of the messaging system we measure the round-trip time between broadcasting and receiving messages. We record the round-trip time using JMH and place an artificial load on the system to simulate a more realistic use case of the messaging system.

The measurements are taken for 1 to 200 users with an interval of 25. The load that is put on the messaging system is based on the number of users used during the experiment, meaning that the load grows as the number of users grows. The load is simulated using two scheduled thread pool executors, each executing a predetermined number of actions per second. The first executes 20 subscription updates per player per second and the second performs 150 broadcasts per player per second. We use these specific values based on measurements recorded during gameplay testing. The results of these measurements can be found in Figure 16.

As the performance of the ActiveMQ broker is particularly poor, and as the RabbitMQ broker could not be ran on DAS-5 due to a missing Erlang module, we do not pursue further experimentation with any of the JMS brokers (Figure 17). The asynchronous variants of the read-write and concurrent brokers show a similar results, their initial round-trip time is much larger than that of the other channel brokers (Figure 18). However, we will continue experimenting with these in the server scalability experiment, as we expect these to scale well when used in the Opencraft server. The channel brokers using channels other than the unsafe channel will not be used in that experiment. Static analysis of the code shows that the additional layer of synchronization does not provide any additional thread-safety. Therefore, all computation and locking performed by these thread-safe channels is redundant when combined with an already thread-safe broker. The impact of these additional computations can be seen in Figure 19.

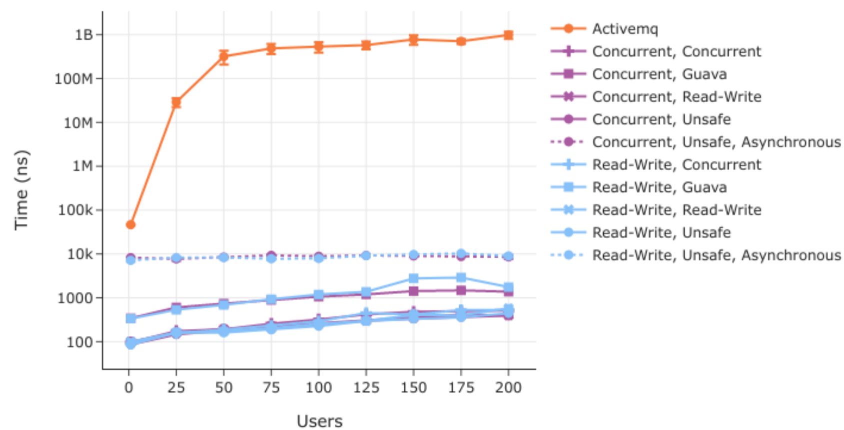


Figure 16: The round-trip time of all brokers. The whiskers represent the 95% confidence interval of the measurements. It uses a log-scale for the y-axis as the ActiveMQ broker's measurements dwarf those of the other brokers.

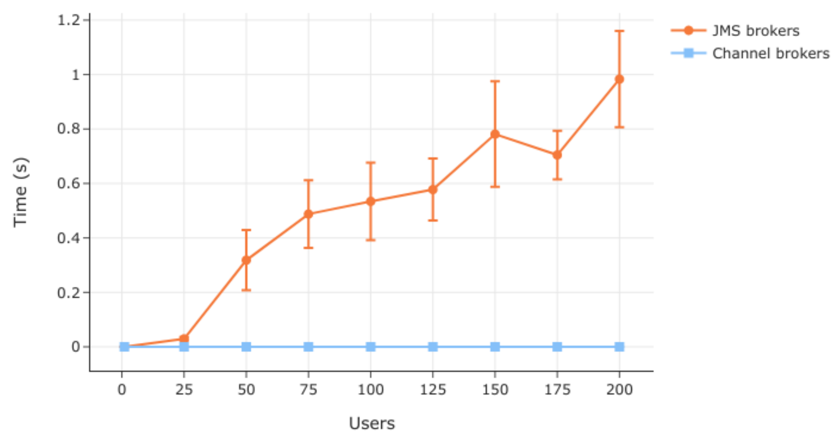


Figure 17: The mean round-trip time of the JMS brokers compared to the channel brokers. Similar to Figure 16, he whiskers represent the 95% confidence interval of the measurements.

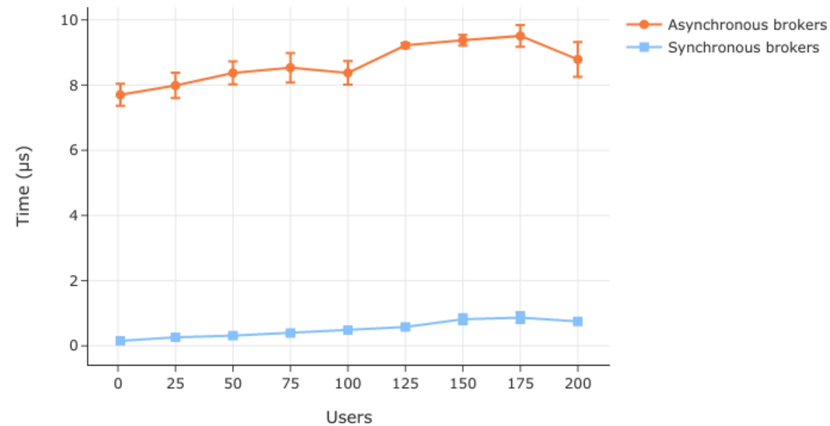


Figure 18: The mean round-trip time of all asynchronous channel brokers compared to all synchronous channel brokers. Similar to Figure 16, the whiskers represent the 95% confidence interval of the measurements.

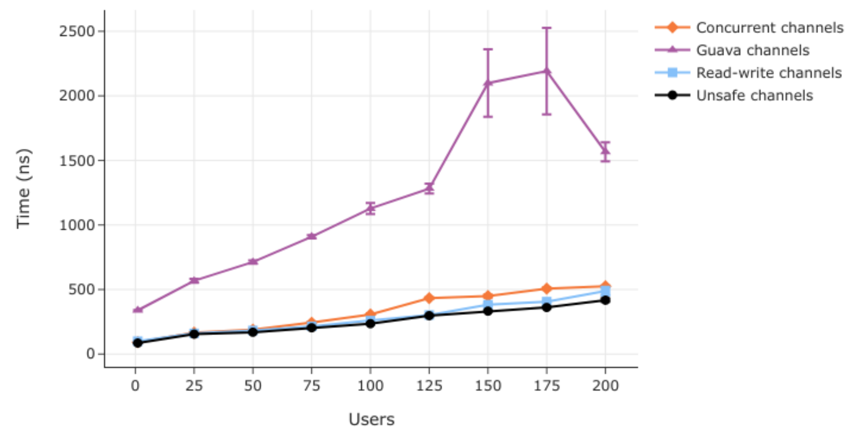


Figure 19: The mean round-trip time of all channel brokers compared to each other. Similar to Figure 16, the whiskers represent the 95% confidence interval of the measurements.

10.2. SERVER SCALABILITY

The Yardstick paper [3] measured the maximum number of players the Glowstone, Spigot, and Minecraft servers could handle. We test the Opencraft server in the same manner, and will rerun some of the experiments from the paper to create a baseline with which to compare our results. Because, we cannot rely on outdated results, since the environment is not exactly the same, and we are working with another Minecraft version than in the Yardstick paper. However when we run the old original implementation of the Opencraft server ourselves we will have a good understanding of how our implementation compares to this.

This experiment will not be run with all the brokers, but with a selection of them. We want to compare our brokers to each other and also our best and worst broker to the original implementation. The brokers will be chosen based on the results of the previous two experiments. These brokers will also be used for the coming experiments.

After testing broker configuration in Section 10.1. We found that the following 4 broker configurations were the best performing ones when compared to the other configurations and thus would yield us the best results. We decided to test these 4 broker configurations based on the results: read-write-unsafe, read-write-unsafe-async, concurrent-unsafe-async and concurrent-unsafe. These 4 configurations and the original opencraft-dev branch will be tested with the same configuration. A benchmark class has been added that generate the relative utilization per tick and amount of players online in csv format. Every configuration will be tested with a range of 0 to 300 players with a 25 player interval. The server will be filled with players where each step increases the player count by 25. The server will be left running for 5 more minutes after reaching the player count in order to let the tick rate stabilise. These numbers will be used to verify the servers performance for the given player count.

The players are configured to execute one of two tasks and select a new one when their prior task is complete. There is a 25% chance that the bot will perform a break block task, a 25% chance that the bot will perform a place block task, and a 50% chance that the bot will perform a walk task. Where as the walk task will make the bot walk to a randomly selected location within a predefined maximum distance from the bot. All players receive independent actions which results in all players executing one of two tasks and thus creating a comparable real world players load.

The results in Figure 20 show that the asynchronous brokers perform worse than the original implementation, but the non-asynchronous brokers do not. This can be attributed to the additional overhead produced by the asynchronous brokers, but the non asynchronous brokers do not have such overhead and thus they perform better than the original implementation which can be further examined in Figure 20. The performance of the read-write broker is almost identical to the concurrent broker.The maximum capacity while maintaining 20 ticks/second is increased to 16.7% when using the read-write broker. This difference in performance grows to 23.1% when look at 300 players.

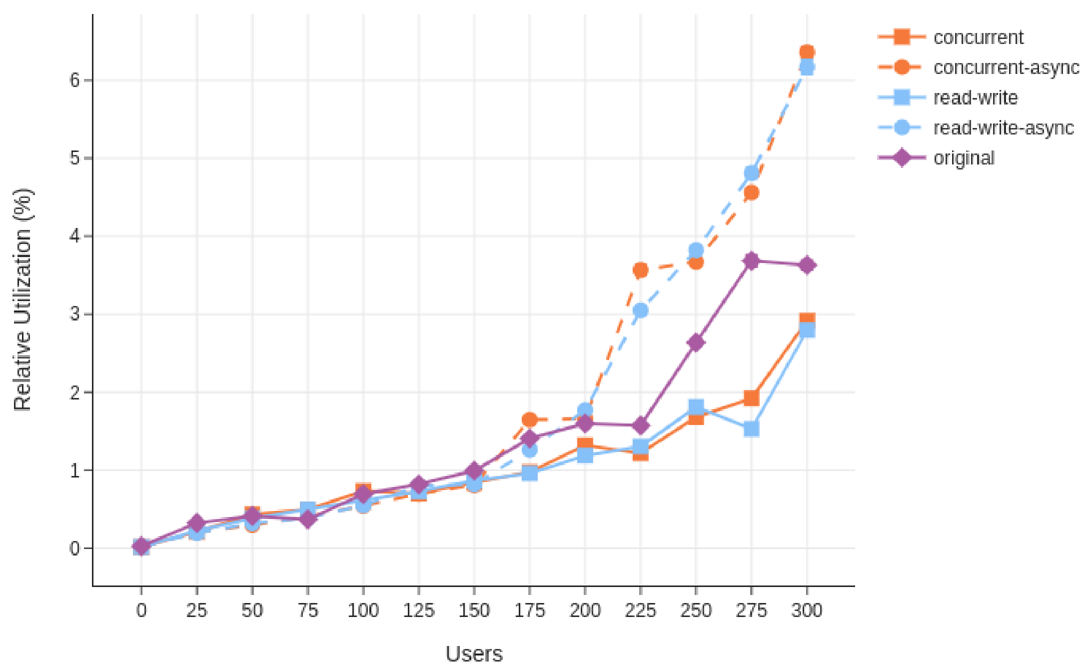


Figure 20: The relative utilization per player count for different broker configurations and the original Opencraft.

10.3. Other experiments

Originally when designing our experiments we came up with more experiments than we actually ran. Due to time constraints we were not able to do these experiments ourselves. We do believe that these experiments can provide useful information about the performance of our messaging system. This section mentions two of these useful experiments.

10.3.1. Action-reaction latency

Besides testing the latency for just the standalone messaging system, we can also test the latency of the entire Opencraft server. This gives us a more realistic view of how the server would perform under real workloads. Measuring this is done by varying the amount of bots that are active on the server at the same time. Every bot has the ability to place a block. After this is done, the bot measures how much time has passed between the placing of the block and when the block appears in game. This measurement can provide us with the processing time the server took for changing the block and sending a reply.

To measure the latency of the server a varying amount of bots will be active on the server at the same time. Varying the amount of players that are active on each experiment run is important for measuring how the latency changes depending on the amount of players that are connected to the server.

10.3.2. Asynchronous chunk generation

As mentioned in Section 6.2, we improved the chunk generation by creating the priority executor that can offload the chunk generation to a different thread from the main game loop thread. This should improve player gameplay experience, because players can perform other tasks while chunks are being generated instead of the main thread being severely slowed down due to the heavy computation cost of generation new chunks. To see how much of an improvement the priority executor made, we can test how quickly and how many chunks are loaded and generated with and without the priority executor.

The property we wanted to test is how many chunks can be generated and sent to the player each tick. This can be tested by using a large amount of bots that move in a straight line from the spawn point in different directions and thus generate and load new chunks. The measurements can be gathered by way of a logger. With these measurements we can see how fast and how many chunks are generated and loaded.

11. CONCLUSION

This section presents our answers to the research questions and the main question. The answer to each of these is explained in detail including references to the relevant chapters. The collection of these answers form the conclusion of this thesis. Since this thesis is part of the Opencraft research project, an international scientific collaboration, this collection of answers will be used to further the research of this group.

RQ1 **How to design an extensible messaging system for the Opencraft server?**

The design of the messaging system is based on the publish/subscribe design pattern. In particular, on the topic-based variant discussed in Chapter 3. Each of its components are defined using interfaces, allowing them to be replaced or improved upon without requiring alterations to existing code. Thereby making it easily extensible. This design of loosely coupled components is explained in detail in Chapter 4.

RQ2 **How to validate that the integration of the messaging system does not break features of the Opencraft server nor prevents the addition of features from the vanilla Minecraft server?**

Features of the Opencraft server are validated using gameplay testing, because it was the only viable option. Automated testing was not possible, because the Opencraft server handles messages differently than the vanilla Minecraft server does. This makes it difficult to use existing validation tools. Adapting Yardstick to function as a validation tool was not an option, since it would have taken too much time. We found out during gameplay testing that many features of the Opencraft server were incomplete or missing. To ensure that the messaging system would not prevent the implementation of these features, we have implemented a number of them ourselves. More information on these features can be found in Chapter 7.

RQ3 **Does the Opencraft server, with the messaging system, support significantly more simultaneous users than the original Opencraft server?**

The Opencraft server, with the messaging system, supports more simultaneous users than the original Opencraft server. However, its performance is highly dependent on the messaging system's configuration. Only the synchronous read-write and concurrent brokers coupled with the unsafe channel provide greater scalability than the original Opencraft server. To be exact, an increase in maximum capacity of 16.66% while maintaining 20 ticks/second. As the number of users grows to 300, this difference in performance is 23.14%.

MQ **How can the messaging system of the Opencraft server be reworked such that its scalability is increased?**

The messaging (system) of the Opencraft server can be reworked by replacing it with a messaging system structured after the publish/subscribe design pattern. It allows the messaging to be parallelized, all the while reducing the number of messaging that needs to be generated by centralizing the generation and distributed the generated messages to interested players afterwards.

12. Discussion

In the previous sections we presented the results and answered the research questions. In this section we will discuss the current state of the Opencraft project and the problems we had during development. Besides this we will also discuss the ethical implications and regression problems of this project.

12.1. Software Development Methodology

Our approach to software development was similar to scrum. Every day started with a short meeting with the developers. During these meetings, we discussed the issues to complete that day, and reflected on the issues of the day before. There were also weekly meetings with the client, and bi-weekly meetings with the coach, to keep them up-to-date, request feedback, and discuss important decisions.

All meetings were documented in Google Drive, and all issues were tracked on an issue board in Gitlab. Gitlab was also used to manage the codebase. We used its continuous integration and merge request tools to ensure stability of the project and the quality of the codebase.

12.2. Code change effort

The code quality problems outlined in Chapter 8 heavily influenced our programming efficiency. Due to the complexity and size of the existing code base, it was hard to move code around or replace existing parts. Debugging was harder as well, since it was often unclear how different parts of the code were connected. For example, a message received from a client could be handled by one of many handler classes, each of which performed completely different actions depending on the content of the message.

12.3. Testing problems

The Opencraft server's source code is poorly tested. The project had a branch coverage of 9% when we started on it. We increased this percentage in the final version to 13.7%, but we were unable to test the entire project reliably. The low branch coverage resulted in regression which could often only be prevented by extensive manual testing. However, this did not prevent the code from already containing bugs. We would often find bugs that were related to certain features not working well together. This could be attributed to each feature having been developed by an independent developer and not having any tests included.

12.4. Adaptation of Yardstick

At the beginning of the project we wanted to create an adaptation of the bot included in Yardstick, such that it could be used for testing the server. The bot was isolated from the remainder of Yardstick and edited to accommodate for the easy creation of a bot with a certain task sequence. The bot would then complete the sequence and report back to its creator when successful. However, we ran into a lot of incompatibilities on the Opencraft server side which made automated testing too time consuming and inconvenient. The bot would not be able to teleport itself on the Opencraft server, but would function correctly on the vanilla Minecraft server. Another issue on the Opencraft server was player verification. The server does not check physics data for the player, which allows the bot to fly without having the appropriate permissions. This extents to collisions as well. The bot could walk through a wall without any issues. The number of road-blocks we faced throughout the adaptation lead us to abandon the development of the testing utility all together.

12.5. Ethical implications

There are also ethical implications that need to be thought of, like with every project. These implications might be more severe for some projects then for this one, however we should still think about them. In this section we will discuss the ethical implication of the messaging system we implemented into the Opencraft project.

One of the reasons for implementing the messaging system is to make Opencraft more scalable and allow for more players to play on the server at the same time. In general we think that allowing more players to play on the same server is a good thing, however it can also raise some ethical concerns.

The big Minecraft servers are able to hold more then 100 players. The players on these servers range from young children to adults. However, with this much players on a server, moderation is required. This moderation has to be done by trusted individuals, to make sure that the players on the server don't misbehave. This misbehaviour can include cyberbullying and scamming other players in various forms. These forms of misbehaviour can cause psychological damage for the victims, especially if they are still very young. Thus when we are able to support more players on a server and more people are active at any given time, it is a lot more difficult to monitor the different players. This can lead to more misbehaviour and might cause more psychological damage.

13. Future work

This thesis is part of the Opencraft project. In particular, we have developed a messaging system for the Opencraft server. While a lot of progress has been made on the Opencraft server, a lot remains to be done as well. This section first discusses two additions to the messaging system that could be considered as future work. Then, it proposes two more extensions of the Opencraft server that would significantly improve its code quality.

13.1. Messaging system

The first extension of the messaging system would be the addition of content-based channels. While Minecraft-like games often provide a spatial structure that can be leveraged to assign topics, there are many in-game objects that could make efficient use of a content-based channel instead. For example, a content-based channel could be used to handle the communication between entities. Entities have continuous positions and their interest range is easily defined as a combination or their position and view distance.

The second extension of the messaging system is related to its integration in the Opencraft server. The messages distributed by the messaging system to subscribers are still being send by the server itself. This is mainly due to the high coupling between the networking code, particularly the GlowSession class, and the in-game player object, which is represented by the GlowPlayer class. Decoupling these classes could allow the messaging system to be ran in a separate process or even on another node in a distributed system.

13.2. Automated gameplay testing

As mentioned in an earlier chapter (Chapter 7), it should be possible to adapt the bot used by the Yardstick benchmarking tool to perform automated gameplay testing. This would require the bot's capabilities to be extended, such that it can perform precise and actions and verify that the server provides the correct response. For example, it could perform a place-block action and verify that the block appears in the virtual world.

13.3. Entity-component system

The problem mentioned in Section 8.1.2 is one we recommend the Opencraft research team fixes. If this is corrected the definition of game objects can be simplified and the large inheritance structures that are currently used can be removed.

13.4. Mathematics

Besides adding an ECS we also recommend the addition of proper mathematics classes. For example, a vector class which can be used to represent 2-dimensional and 3-dimensional positions, translation, and rotations. This would significantly reduce the number of parameters required by many methods, and would simplify computations that are currently performed separately for the x, y, and/or z components.

Bibliography

[1] Newzoo, "Newzoo's Global Games Market Report," market analysis, Newzoo, 2019.

[2] B. Gilbert, "'Minecraft' has been quietly dominating for over 10 years, and now has 112 million players every month," *Business Insider*, Sept. 2019.

[3] J. van der Sar, J. Donkervliet, and A. Iosup, "Yardstick: A Benchmark for Minecraft-like Services," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, (Mumbai, India), pp. 243–253, Association for Computing Machinery, Apr. 2019.

[4] J. Donkervliet, "Design and Experimental Evaluation of a System based on Dynamic Conits for Scaling Minecraft-like Environments," 2018.

[5] S. R. Saikia, "A Survey of MMORPG Architectures," Mar. 2008.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, pp. 114–131, June 2003.

[7] M. A. Tariq, B. Koldehofe, and K. Rothermel, "Securing Broker-Less Publish/Subscribe Systems Using Identity-Based Encryption," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 518–528, Feb. 2014.

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, pp. 332–383, Aug. 2001.

[9] Y. Zhao, K. Kim, and N. Venkatasubramanian, "DYNATOPS: A dynamic topic-based publish/subscribe architecture," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, (Arlington, Texas, USA), pp. 75–86, Association for Computing Machinery, June 2013.

[10] C. Cañas, K. Zhang, B. Kemme, J. Kienzle, and H.-A. Jacobsen, "Publish/subscribe network designs for multiplayer games," in *Proceedings of the 15th International Middleware Conference*, Middleware '14, (Bordeaux, France), pp. 241–252, Association for Computing Machinery, Dec. 2014.

[11] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pp. 262–272, June 1999.

[12] P. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pp. 611–618, July 2002.

[13] G. Mühl, *Large-Scale Content-Based Publish-Subscribe Systems*. Phd, Technische Universität, Darmstadt, Nov. 2002.

[14] P. Eugster, "Type-based publish/subscribe: Concepts and experiences," *ACM Transactions on Programming Languages and Systems*, vol. 29, pp. 6–es, Jan. 2007.

[15] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

[16] W.-J. Huang and E. J. McCluskey, "A memory coherence technique for online transient error recovery of FPGA configurations," in *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, (Monterey, California, USA), pp. 183–192, Association for Computing Machinery, Feb. 2001.

[17] S. P. Ahuja, R. Eggen, and A. K. Jha, "A Performance Evaluation of Distributed Algorithms on Shared Memory and Message Passing Middleware Platforms," *Informatica*, vol. 29, no. 3, 2005.

[18] F. B. Engelhardtsen and T. Gagnes, "Using JavaSpaces to create adaptive distributed systems," in *Proceedings of Workshop and EUNICE Summer School on Adaptable Networks and Teleservices*, 2002.

[19] D. Bradshaw, B. Nainani, K. MacDowell, and D. Raphaely, "Introduction to Oracle Advanced Queuing," in *Oracle9i Application Developer's Guide - Advanced Queuing*, pp. 43–70, Mar. 2002.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education (US), Oct. 1994.

[21] "Collision detection and response." https://www.gamedev.net/articles/programming/general-and-gameplay-programming/swept-aabb-collision-detection-and-response-r3084/.

[22] D. M. Hall, "ECS Game Engine Design," 2014.

[23] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *Computer*, vol. 49, pp. 54–63, May 2016.

Appendices

A. Project info sheet

Title of the project: Opencraft - A scalable Minecraft-like game for millions of players
Name of the client organization: AtLarge international research team
Date of the final presentation: 01-07-2020

Description

The challenge of this project is to rework the messaging system of the Opencraft server as to improve its scalability. The research performed at the start of the project taught us about a number of design patterns that we had not heard of before, and gave us insight in a number of libraries that were required to complete the project. The development process was rather straightforward, all issues that needed to be completed were put on an issue board, and every day was initiated with a short meeting in which was reflected on the previous day and in which the upcoming day was planned out. There were weekly meetings with the client as well, to keep him up-to-date and request feedback. Unfortunately, one of our team members as injured towards the end of the project, which we handled by requesting a short extension. The final product is a version of the Opencraft server in which the messaging system is integrated. Its features were validated through gameplay testing, and its performance was evaluated using a benchmarking framework and Yardstick. We recommend the client to look into integrating an off-the-shelf entity-component system into the Opencraft server, as well as a mathematics library. These would significantly increase the code quality of the project. It would also be possible for the client to extend the developed messaging system by adding support for content-based channels (a component of the publish/subscribe design pattern). We expect future student groups working on projects provided by the AtLarge research group to continue working on the version of the Opencraft server we have developed.

Team members of the project:

Julian
roles: Software Tester / Benchmarker / Developer
contributions: Implementation / Validation / DAS-5 / Experiment: Server scalability / Yardstick bot / Validation: Physics
interests: Optimisation / Computer Graphics / Automation

Jim
roles: Benchmarker / Developer
contributions: Integration: Chunk Streaming / JMS broker / DAS-5 / Experiment: Messaging system latency / Experiment: Server scalability
interests: System Programming / Programming Languages

Lars
roles: Software Tester / Developer
contributions: Background / Validation / Gameplay testing / Code quality / JMS broker
interests: System Programming / Machine learning / Reading

Ruben
roles: Software Tester / Developer
contributions: Background / Validation / Code quality / Guava broker
interests: Data science / Software Engineering / Machine learning

Wessel
roles: Technical Lead / Project Manager / Benchmarker
contributions: System design / Implementation / Integration: Messaging system / Experiment: Messaging system latency
interests: Software Architecture / Computer Graphics

Client, coach, and contact of the project:

Client
name: ir. Jesse Donkervliet, Msc
affiliation: Atlarge Reseach group, Vrije Universiteit Amsterdam

Coach
name: Thomas Overklift, Msc
affiliation: Teaching Team, Delft University of Technology

Contact
name: ir. Jesse Donkervliet, Msc
e-mail: j.donkervliet@gmail.com

The final report for this project can be found at: http://repository.tudelft.nl.

B. (Original) Project description

B.1. About Opencraft

Computer games with modifiable terrain are growing in popularity. One such game, Minecraft, is particularly prominent—over 50 million players on more than 10 gaming platforms. A key challenging aspect of multiplayer games is scalability. How can thousands of players play together in one game-instance? How to measure how the game copes under severe load? How to improve Minecraft-like gaming architectures to support high scalability and high performance? Our Opencraft project addresses these questions. Opencraft consists of two research directions:

1. Designing and building a Minecraft-like game that uses novel scalability techniques to support millions of players. To this end, we developed Meerkat, a prototype system that uses Dynamic Conits to increase scalability using bounded inconsistency between players.

2. How to measure the performance and scalability of Minecraft-like games? How to do so for a variety of Minecraft-like servers? To this end, we develop Yardstick, a distributed benchmark for Minecraft-like games. The benchmark supports system-level metrics such as CPU and network utilization but also uses domain-specific metrics such as the server tick-rate and the relative utilization to determine the scalability of these games.

B.2. Project description

Current Minecraft-like games are limited to tens of simultaneous players. This project is focused on building Opencraft, a Minecraft-like game that supports millions of simultaneous players. Your challenge is to design and evaluate a new messaging system for Opencraft to improve its scalability. And validate whether the functionality of the messaging system works correctly.

B.3. Goal of this project

The goal of this project is to improve the current messaging system in Opencraft. The system should meet state-of-the-art software engineering standards such that it can be open-sourced at the end of the project.The system should be extensible to allow other scalability techniques to be applied later.

During this project, you will not only learn a lot about Minecraft-like games but also play a lot of Minecraft as part of debugging distributed systems! At the end of the project, all the students in the Computer Science bachelor program can simultaneously play and learn in one massive Opencraft world!

C. System requirements

To manage the requirements of the messaging system, we have chosen to use the MoSCoW method. Also, each requirement is classified as either functional or non-functional. Functional requirements are those that define a certain use-case. Non-functional requirements express qualities of the system, such as run-time performance, and constraints, such as compatibility with specific software. Based on conversations with the client we identified the following requirements.

C.1. Functional

Must

- The messaging system must have a computational complexity smaller or equal to that of the original system.
- The messaging system must be based on the pub/sub design pattern.
- The messaging system must support topic-based channels.
- The messaging system must be a standalone system.
- Each world chunk must be associated with a topic.
- Each entity must be associated with a topic.

Should

- The messaging system should have a clear interface.
- The messaging system should be integrated with dynamic conits.
- The messaging system performance should be measured on the DAS-5.
- Each world chunk should have its own topic.
- Subscribers should not have to manage their subscriptions themselves.

Could

- The messaging system could be configured without recompiling.
- The messaging system could support content-based channels.
- The messaging system could be distributed over multiple machines.

Won't

- The components not related to the messaging system won't be refactored to improve code-quality.

C.2. Non-functional

- The Opencraft server must remain compatible with the Minecraft client version 1.12.2.
- The Opencraft server must remain compatible with Yardstick.
- The messaging system must have a branch coverage of at least 70%.

D. Bugs & missing features

During this project we discovered certain bugs that had nothing to do with our specific project, which is why we did not solve them. However, we thought it might be useful to list some of them for people that are going to work on Opencraft in the future. Below there are some missing features when we compare the Opencraft server to vanilla Minecraft and some smaller bugs.

In the Opencraft project there are some core features of the Minecraft game that are not yet implemented. If the eventual goal is to have a realistic Minecraft survival experience on an Opencraft server these features need to be implemented.

D.1. Artificial intelligence

The Opencraft project does currently not have any functioning AI. There is an implementation present, but it currently only delivers AI for Zombies. The zombie AI implementation only makes the zombie run around in random patterns. Opencraft would benefit from having a simple rudimentary AI that would provide the world with some content. Passive mobs should at least wander around aimlessly instead of standing still.

Figure 21: A picture showing the current state of AI in the game. The creeper should run away from the ocelot with normal behaviour

D.2. Enderdragon

As of this moment in time the Opencraft project does not have an implementation for killing the enderdragon. This means that there is no way to come back from the end and when a player is there he won't be able to come back to the overworld. Killing the enderdragon is seen by many as finishing the game which means that as of now there is no way to finish the game on an Opencraft server.

D.3. Farm land

The Opencraft server does not support turning tilted dirt into normal dirt by jumping on it. When playing in survival mode the mechanics of farming are quite essential as it will be the main source of food. Something not working in this will withhold players on the Opencraft server from having a survival gameplay experience.

D.4. Minecarts and boats

In survival Minecraft traveling is quite a pain if you have to do it all by walking/running. That is why boats and minecarts have been implemented in the game. The Opencraft project however has not implemented these features, minecarts cannot be placed on rails and boats cannot be placed on water. This means that traveling will be a lot slower and exploring a lot of new terrain will be near impossible within a acceptable time period.

D.5. Anvil duplication

Besides all the above mentioned missing functionalities in the Opencraft project, there is also a duplication bug within the project. This bug happens when a player tries to enchant or repair an item. The anvil does not consume the tool that is being used but does make the new one. This results in the player receiving two items instead of one and because of that being able to duplicate items. The ability to duplicate items should not a part of a survival gamemode and thus should be removed.

D.6. Remaining bugs

Other than the above bugs we labeled as core functionalities there are also some smaller bugs that when fixed would improve the gameplay experience. These bugs are listed down below.

1. Shift-clicking logs into a furnace as fuel doesn't work (probably because it can go in both slots)
2. Jumping on farmland doesn't change back to dirt
3. Anvils do not consume the items when enchanting/repairing, this allows players to duplicate items
4. A player should not be able to spawn an iron golem underwater or be able to knock it back by punching
5. Placing slightly or heavily damaged anvils doesn't work, normal anvils are placed instead
6. Minecarts don't work, they can't be placed on powered rail, they don't move and the hitbox when placing on normal rail is incorrect
7. Boats don't work and can't be placed on water.
8. The ender dragon can't be killed (thus leaving the end is impossible)
9. Cactus damage radius is too big

E. SIG RESULTS

E.1. MIDTERM

System fact sheet

Name			Julianvandijk
Size			9 PY (+0,23)
Test code ratio			11,3% (+2,3)
Code touched			2.035 LOC
Code removed			273 LOC
Maintainability	★★☆☆☆	(2,4)	▲ 0,01
Volume	★★★★☆	(4,2)	▼ 0,01
Duplication	★★★★☆	(4,0)	= 0,00
Unit size	★★☆☆☆	(2,4)	= 0,00
Unit complexity	★★☆☆☆	(2,1)	▲ 0,01
Unit interfacing	★★☆☆☆	(2,0)	▲ 0,04
Module coupling	★★☆☆☆	(1,7)	▲ 0,02
Component balance	★☆☆☆☆	(0,5)	= 0,00
Component independence	☆☆☆☆☆	(N/A)	= 0,00

Figure 22: This shows the improvement of the system as a whole from the midterm in comparison with the original code

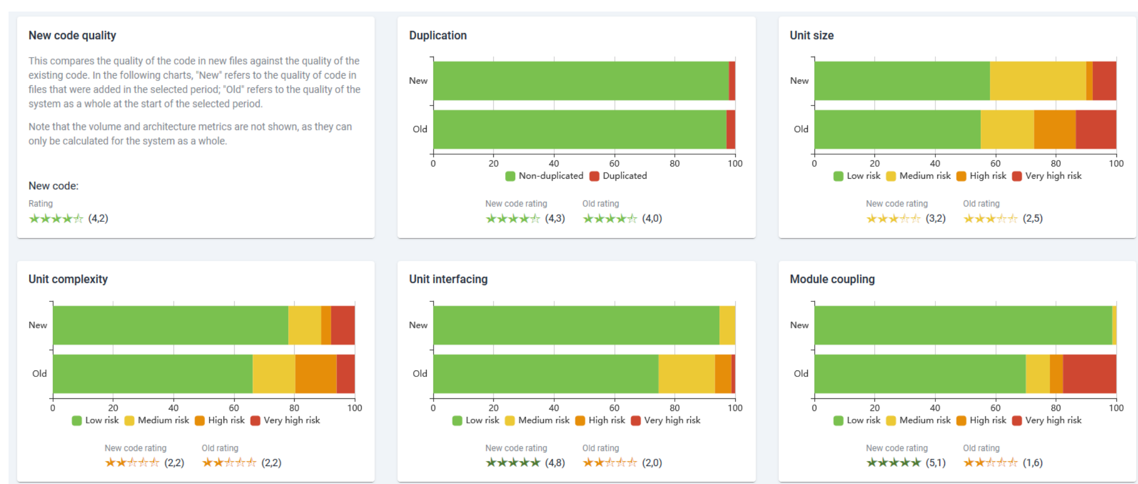


Figure 23: this shows how the quality of the new code in the midterm compares with the original code

E.2. ENDTERM

System fact sheet

Name			Julianvandijk
Size			9 PY (+0,39)
Test code ratio			13,7% (+4,6)
Code touched			4.691 LOC
Code removed			55.967 LOC
Maintainability	★★★☆☆	(2,9)	▲ 0,54
Volume	★★★★☆	(4,2)	▼ 0,02
Duplication	★★★★☆	(4,0)	▲ 0,01
Unit size	★★☆☆☆	(2,4)	▲ 0,01
Unit complexity	★★☆☆☆	(2,2)	▲ 0,04
Unit interfacing	★★☆☆☆	(2,0)	▲ 0,04
Module coupling	★★☆☆☆	(1,7)	▲ 0,06
Component balance	★★☆☆☆	(1,5)	▲ 1,10
Component independence	★★★★★	(5,1)	▲ 5,11

Figure 24: This shows the improvement of the system as a whole from the endterm in comparison with the original code

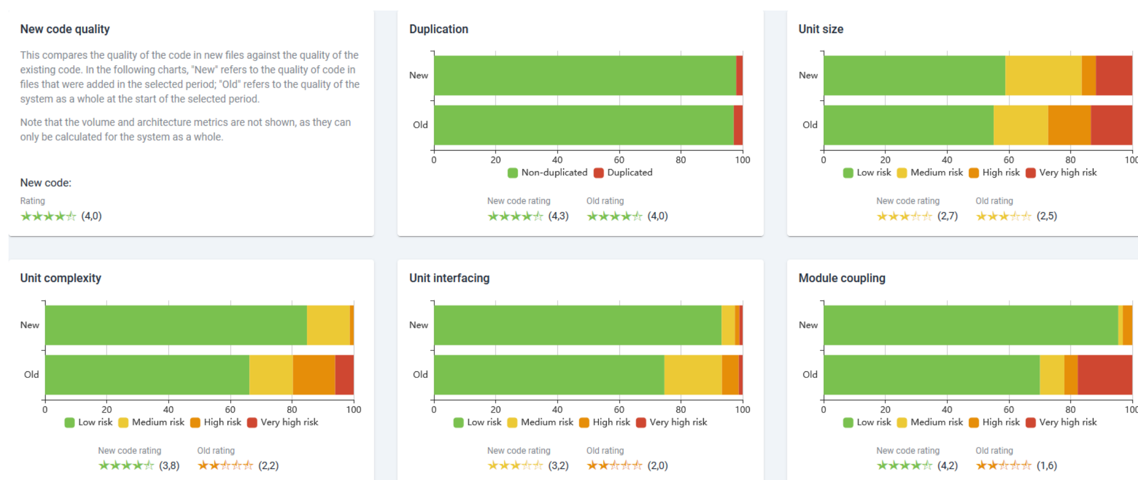


Figure 25: this shows how the quality of the new code in the endterm compares with the original code