Mitigating garbage collection in Java microservices

How garbage collection affects Java microservices and how it can be handled

Amanda Ericson

MID SWEDEN UNIVERSITY
Department of Information Systems and Technology

Examiner: Tingting Zhang, tingting.zhang@miun.se
Supervisor: Martin Kjellqvist, martin.kjellqvist@miun.se
Author: Amanda Ericson, amer1501@student.miun.se
Program: Master of Science in Computer Engineering, 300 credits
Course: DT005A Final Project, 30 credits
Area: Computer science
Term, year: Spring, 2021

Abstract

Java is one of the more recent programming languages that in runtime free applications from manual memory management by using automatic Garbage collector (GC) threads. Although, at the cost of stop-the-world pauses that pauses the whole application. Since the initial GC algorithms new collectors has been developed to improve the performance of Java applications. Still, memory related errors occurs and developers struggle to pick the correct GC for each specific case. Since the concept of microservices were established the benefits of using it over a monolith system has been brought to attention but there are still problems to solve, some associated to garbage collectors. In this study the performance of garbage collectors are evaluated and compared in a microservice environment. The measurements were conducted in a Java SpringBoot application using Docker and a docker compose file to simulate a microservice environment. The application outputted log files that were parsed into reports which were used as a basis for the analysis. The tests were conducted both with and without a database connection. Final evaluations show that one GC does not fit all application environments. ZGC and Shenandoah GC was proven to perform very good regarding lowering latency, although not being able to handle the a microservice environment as good as CMS. ZGC were not able to handle the database connection tests at all while CMS performed unexpectedly well. Finally, the study enlightens the importance of balancing between memory and hardware usage when choosing what GC to use for each specific case.

Keywords: Garbage collector, Microservice, Docker, docker-compose, CMS, G1GC, Shenandoah GC, ZGC

Acknowledgements

I would like to express my very great appreciation to the company Knightec in Sundsvall who gave me the opportunity to perform this study and provide great assistance along the way. A special thanks to Gabriel Uppegård and Ludvig Åberg at Knightec who has given great assistance throughout the whole project and with dedication has helped with any questions and difficulties along the way.

I also wish to acknowledge the help provided by my supervisor at Mid Sweden University, Martin Kjellqvist for great advice and suggestions during this research work.

Table of Contents

List of Figures

List of Tables

Terminology

API	Application Programming Interface
CMS	Concurrent Mark Sweep
CPU	Central Processing Unit
CRUD	Create, Read, Update, and Delete
DB	Database
G1GC	Garbage First Garbage Collector
GC	Garbage Collector
JDK	Java Development Kit
JIT	Just-In-Time
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
OS	Operating System
PDF	Portable Document Format
PODA	Package Once Deploy Anywhere
RAM	Random Access Memory
Regex	Regular expression
REST	Representational State Transfer
SOA	Service-Oriented-Architechture
STW	Stop-The-World
VM	Virtual Machine
WORA	Write Once Run Anywhere
ZGC	Z Garbage Collector

1 Introduction

In the early development of new programming languages, memory was handled manually by the programmer. By example in C++, released in 1983, memory was (and still are) allocated with a ”new” statement and then freed using destructors. In newer programming languages such as Java, initially released in 1996[1], the memory management is automatic – using a daemon thread referred to as the Garbage collector (GC).

The implementation of GCs is initially based on the hypothesis” Most objects will soon become unreachable”, with the main goal being to constantly watch the memory and free space from removing objects that are out of scope or unnecessary to keep in the heap, enabling new objects to be allocated in the memory. The GC thread hence works as a helping mechanism for the programmer with the main mission to constantly prevent the heap from becoming full. [2]

Although the GC is said to be an automatic memory handler it has its faults. Sometimes the GC thread fails to remove enough data from the heap so that it becomes full – resulting in applications to crash. Also, since the initial GC algorithms many new GCs are offered by the JVM. GCs available in Java 13 are the Serial GC, Parallel GC, Concurrent Mark Sweep GC, Garbage First GC (G1GC), Shenandoah GC, and Z Garbage Collector (ZGC). Out of these G1GC is the default collector for Java 13 and 15 and Shenandoah GC and ZGC are the latest released GCs.

Today, microservices are becoming more popular and monolith systems are more and more likely to be exchanged to microservice systems. The reason being that microservices allows for a lot of benefits such as a more flexible system, better scalability and easier understanding because of each unit of the program is contained and handled one by one. However, there are problems within the fundamentals of Java which appears more frequently when using a

microservice architecture versus using a monolithic architecture. One of these problems is related to Javas garbage collection.

The GC thread generally has big impact on an applications performance, reliability and efficiency which is one of the good reasons to study GCs. Commonly, when evaluating the performance of a GC, the latency and throughput are considered. Latency being the responsiveness of an application, also described as the delay experienced by a user due to GC activity. The GC latency is based of operations where the GC needs to use all application threads to perform an operation, known as a Stop-The-World (STW) event. Throughput is instead a measurement of the workload managed by an application within a specific unit of time. To improve latency either the number of times these STW events occur needs to be reduced or the length of them. To improve throughput the workload per time unit is prioritized. Both Shenandoah GC and ZGC are experimental collectors implemented with the purpose of optimizing these parameters, in separate ways.

In 2017 a study made by Akamai it was found that every 100ms delay for a user in a web-page can decrease conversion rates by 7%, and 53% of mobile visitors will leave a page that takes more than 3 seconds to load. [3] [4] The core of all studies is that the less interactive a site becomes, the more unlikely users are to stay on the site. The interactivity of a site is highly dependent on the latency and by improving GCs the latency time can be improved. The currently used default GC in Java 13 is the G1 collector. [5] In Java 9 the motivation for changing GC were,

"Limiting GC pause times is, in general, more important than maximizing throughput. Switching to a low-pause collector such as G1 should provide a better overall experience, for most users, than a throughput-oriented collector such as the Parallel GC, which is currently the default." [6]

Hence, already at that point the benefits of using a low-latency collector were identified. Although, the performance of GCs has not been broadly compared or evaluated other than the manufacturers own evaluations. Moreover, the

evaluations made by manufacturers can often be beneficial for their own implementation, example by having a test environment favoring one of the GCs. Thus, an evaluation with equal conditions for all GCs is of great interest to fill the gap in current research, especially for the new experimental collectors in Java version 15. Also, with the current growth of microservice based systems, the need of an evaluation in a microservice environment is needed to ensure good performance, efficiency and reliability, but also for developers to be able to make reasonable decisions when choosing a GC for their purpose.

The company Knightec in Sundsvall aims to decide the best GC for a scenario including a microservice environment. This thesis aims to collect the current state-of-art regarding GCs and collect important information about GCs and what differs them from each other. Also, to test GCs in a microservice based application, in that test environment determine what may affect the performance of a GC and lastly provide a recommendation of what GC best fit in a microservice environment.

1.1 Overall aim

GC algorithms have been researched for a large extent of time, especially the Serial GC, Parallel GC and CMS GC. Lately G1GC has also been widely tested and has managed to be exchanged against the former default collector CMS since JDK13.[5] On the other hand, both ZGC and Shenandoah GC are still considered experimental collectors since they were recently released and has thus not been involved in a lot of research yet. This research aims to provide insight into these two experimental collectors compared to CMS and G1GC in a microservice environment setting. The contributions of this work also involves a recommendation of what GC works best in a microservice environment at various prerequisites.

1.2 Concrete and verifiable goals

The main goal of this research is to test and compare some of the most recent GC in a microservice environment in terms of performance, latency and throughput to determine the best option for a microservice architecture in different aspects.

Specifically, the project aims to answer the following questions,

1. Can GC be mitigated by using scheduled garbage collection and what are the trade offs of doing that?
2. How do different garbage collectors characteristics compare to each other (Serial, parallel, CMS, G1GC, Shenandoah GC, ZGC)? Specifically regarding availability in Java versions, JVM and other basic information beneficial for tests.
3. In a microservice tree with depth N, how do different garbage collectors perform compared to each other? (Serial, parallel, CMS, G1GC, Shenandoah GC, ZGC) and why? What is the worst case scenario (GC triggered everywhere) and how would this affect the application?
4. Does the presence of a database connection make a significant difference on GC performance?

1.3 Scope

The ways of tweaking the behavior of GCs are massive, both by options to the JVM but also since each GC has their own modifiable parameters, programming-wise many operations may also highly affect the performance in many ways. To limit this study a few configurations are chosen which are seen as some of the most central. The database tests are limited to only test the result of maintaining a connection to a database while running the application structure and are hence not performing any operations to the

database. The research is also limited to a microservice structured application in a SpringBoot framework and a computer using a RAM memory space of 8 GB.

1.4 Outline

The report is divided into seven chapters, following the process of the project. Chapter 2 aims to provide the reader with essential information about garbage collectors and its purpose and by that achieve a sense of understanding of the reports content. Chapter 3 presents the methodology used to achieve the results and how they were evaluated. Chapter 4 describes the implementation stages of the project, divided into three sections. Chapter 5 presents the results produced by the experiments. Chapter 6 discusses the given results and providing answers to the research questions. Chapter 7 gives conclusions and enlighten some ethical aspects of the project as well as proposes future work.

1.5 Contribution

This reports content is the result of a MSc thesis work accomplished at Mid Sweden University. All implementations are performed from scratch by the author of this report and the conclusions are a result of the tests performed in the developed environment, also by the author of this report, along with studies done in the area of GCs. All other studies used as a basis for conclusions etc. are credited in this report.

2 Theory

The theory chapter provides the reader with useful information in order to understand the content of the report. It mainly focuses on the concept of a Java heap structure, it also briefly describes the four GCs later analyzed in this report and lastly goes through the concept of a microservice architecture and tools used in this report to achieve the results.

2.1 Java heap

The Java heap is arranged into two spaces, young generation and old generation, illustrated in figure 2.1. The young generation space takes care of all newly created objects. It is divided into three segments, the Eden space, Survival 1 and Survival 2. A threshold cycle value is set which is the maximum survival cycles for an object in the young heap. When an object has survived for that amount of time in the heap it is moved to the second space in the heap which is the Old generation heap space. By example, a method call object will not survive long and will most probably never be moved to the old space, while a created cache that should be around for the entire applications lifetime will surely be moved to the old generation space at some time. [7]

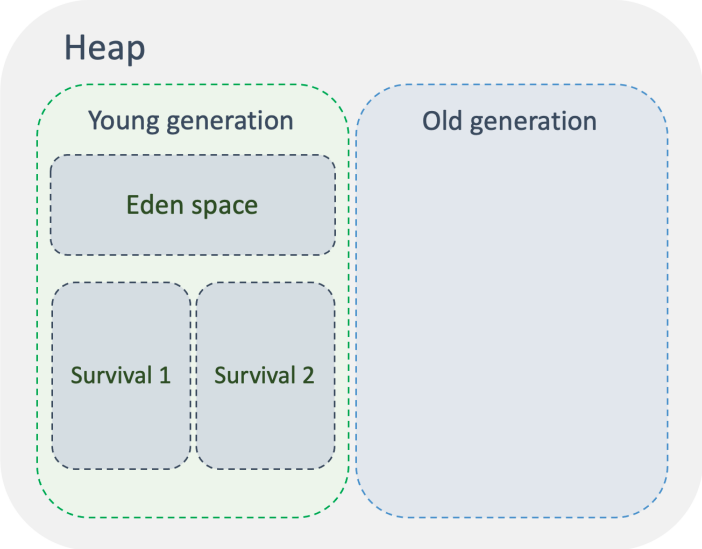


Figure 2.1: The Java heap structure

2.1.1 Java object types

Java objects are allocated in the heap of the Java memory. The garbage collector are interested in dividing the objects into the following types,

- Live objects
  An active object in the application that can be referred to by another object in the same application. E.g a class variable.
- Unreachable objects
  An object that is created during a method call and when the method is over the object becomes out of reach. [2] [8]

2.2 JVM

The Java Virtual Machine (JVM) is an abstract computing machine that translates the Java programming instructions into instructions and commands readable by the operating system. There exists a list of alternative JVM structures today but two frequently used are described below.

2.2.1 HotSpotVM

HotSpotVM is a JVM alternative implementation that currently is used as a standard in most Java systems. The architechture of HotSpotVM supports scalability and the ability to achieve high performance on small as well as large computer systems.[9]

From the perspective of tuning performance, there are three major components of the Hotspot JVM concerned,

1. Heap
2. Garbage collector
3. Just-In-Time (JIT) compiler [10]

In newer versions of the JVM, the JIT rarely needs tuning and focus is instead concentrated on the heap and the garbage collector. [9]

2.2.2 GraalVM

GraalVM is a project started by Oracle and its main mission were to become a high performance JIT compiler. [11] Using GraalVM is supposedly beneficial for microservice applications by providing 50x faster startup times and 5x smaller memory footprint. [12]

2.3 Garbage collector

The garbage collector implementation is based on the hypothesis ”Most objects will soon become unreachable”. The garbage collector works as a 'helper' for the user. It constantly watches the memory and frees space e.g. from removing objects that are out of scope, enabling new objects to be allocated in the memory. The main mission of the garbage collector thread is to constantly prevent the heap from becoming full. Because the garbage collector thread is supposed to help the programmer - the garbage collector cannot be forced to happen. [2]

The garbage collector usually takes the following steps,

1. Mark
   The marking starts from the application root and walks through all the objects in order. The GC marks objects that are reachable as live and the other are left as they are.

2. Delete
   In the second step the garbage collector deletes unreachable objects by sweeping them of the heap memory and reclaim the memory space.

3. Compact
   Some allocated memory may have become freed while allocating other objects which leaves blank space in the memory. The compacting works by arranging all the memory spaces in order, meaning the allocated memory is located before the free space, shown in figure 2.2. The compacting step is what takes the most time in the GC process because the GC thread has to go through the complete heap step by step. [13]

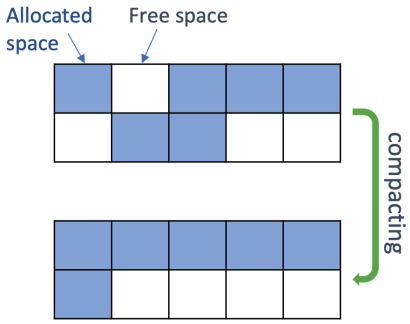


Figure 2.2: The GC compacting step

2.3.1 Collector types

There are a few basic types of garbage collectors in Java that sets the base for all new implemented versions of garbage collectors. Those are the following,

- Serial
  A single threaded collector which requires the application to pause when GC is working.

- Parallel
  A multi threaded GC which like the singe threaded GC only runs when its needed.Because of the GC running on multiple threads it is much faster than single GC and hence has shorter pause times of the application.

- Concurrent Mark Sweep (CMS)
  Runs continuously alongside the application. Does not have any pauses except for the mark and remark steps. [14]

All three collectors described above are illustrated in terms of functionality in figure 2.3. It is by the figure visible how the pause time with a single thread gives much longer pause times than with a multi-threaded GC. The CMS has the shortest pause times out of the three methods because operations are runned concurrently to the application threads and the only thing pausing the application is the mark and remark steps. [15] [14]

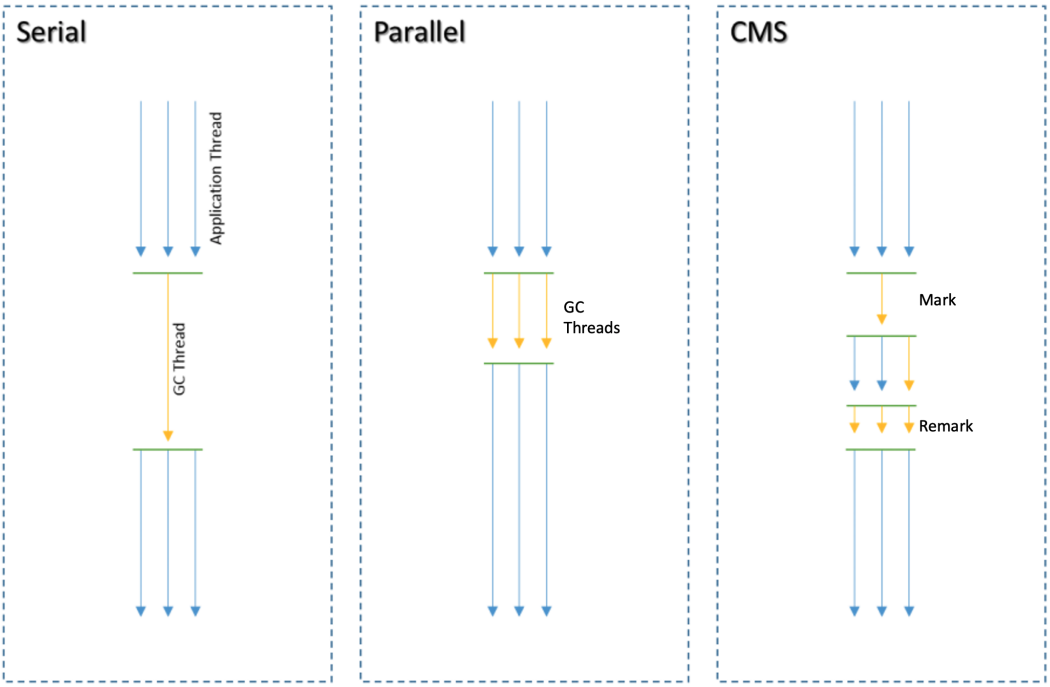


Figure 2.3: GC algorithms thread usage

2.3.2 G1

Garbage First Garbage Collector (G1GC) is a JVM GC implementation available and widely used among systems today. It was set to the default collector in Java version 13 and has been since. [16] G1GC is mainly designed for applications running on multi-processor machines with large memory space and is said to replace the CMS collector since it is more performance efficient.[14]

The G1GC heap is divided into equal sized regions with ranges of virtual memory. The collection process starts by doing a global marking phase to determine what regions that have objects that are not in use and hence can be freed. With this information G1 can start by concentrating on these areas to reclaim memory space. Focusing on the marked areas first is the origin of the G1 name. G1 also

performs heap targeting in an optimal and predictable manner using user-defined acceptable delay times.[10] [17] The heap allocation principle of G1GC is shown in figure 2.4

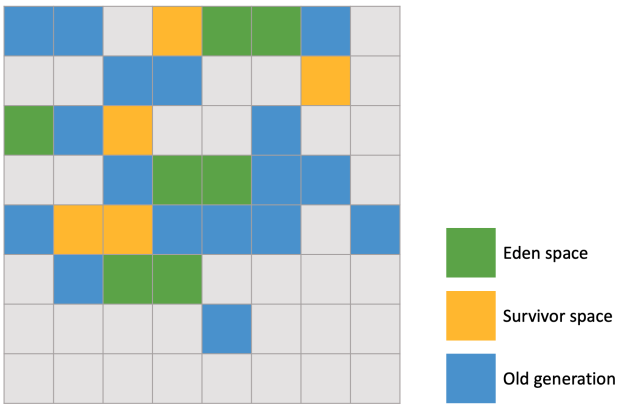


Figure 2.4: G1GC heap allocation

The goal of G1GC in the default configuration is not to maximize throughput nor to have the lowest latency but instead to produce uniform, small pause times. [16]

2.3.3 ZGC

ZGC is a GC which mainly focuses on providing low latency. The collector is developed by Oracle and is suitable for large heap applications requiring high responsiveness, e.g., a server application. [18] The low latency is achieved by providing stop-the-world phases as short as possible, regardless the size of the heap. [19]

As in most GC algorithms the first step in ZGC is marking, where objects are marked as reachable or unreachable. Unlike other GC, ZGC stores the reference state as the bits of the reference, referred to as reference coloring. Although, when using this method multiple references can point to the same object since no information about the object's location is stored. A solution to

this is multimapping, where multiple ranges of the virtual memory is mapped to the physical memory. [19]

Another important aspect in ZGC is to decrease memory fragmentation and to do this compacting is used, see figure 2.2. Despite the fact that compacting solves the memory fragmentation problem, it is very time consuming, which is not wanted considering ZGCs goal is to decrease the pause times of the application. To fix this problem the compacting is done on a separate thread alongside the application thread, using the concept of parallel GC. [20]

Still, when performing parallel relocation of objects, sometimes the application thread runs and tries to access the object in its old address. To solve this issue, ZGC uses load barriers which is a piece of code that runs when a thread loads a reference from the heap in order to access the correct object disregarding any context switches. The load barriers check the metadata bits of each reference. Some reprocessing of the object may occur in this stage and thus a new reference may be produced, a process called remapping. [19]

2.3.4 Shenandoah GC

Shenandoah GC is an ultra-low pause time GC for OpenJDK, developed by Red Hat. It runs on a concurrent GC thread and the main focus is on lowering the response times. RedHat says that, ”Garbage collecting a 200 GB heap or a 2 GB heap should have the similar low pause behavior”[21]. Although ZGC and Shenandoah GC have very similar goals of providing a low latency, high responsiveness GC, the approaches of achieving that is very different. The goal of the Shenandoah collector is to provide a GC with very low pause times and by that reduce the overall pause times made by the GC. [22] Shenandoah can in comparison to G1 relocate the objects in the memory heap without pausing the application execution. Also, Shenandoah comes with more tuning options than ZGC. [23]

2.3.5 Epsilon

Epsilon GC is another experimental GC implementation developed by ... . The GC is established to be a passive "no-op" GC, meaning it handles memory allocation but does not recycle it when objects are out of use. Instead, when the heap runs out of memory the JVM shuts down, and the application will hence crash when no more memory is available. This approach may seem useless since the purpose of a GC is often to prevent the application from crashing. But a passive "no-op" GC is actually useful in the purpose of measuring and managing application performance. In comparison to an active GC that run alongside the application a passive GC runs isolated from the application. By using a passive GC both overhead that can cause latency and reduced throughput is reduced and the performance effects of the GC on the application are removed. A passive GC like Epsilon can hence be used to observe how GC affects an applications performance and also what memory threshold there is by watching when it runs out. [22]

In a case where performance needs to be used to its full potential Epsilon GC might be an option. If the applications memory and garbage usage is known Epsilon can be considered as a viable option. [24]

2.4 Microservice

Microservices can be seen as a variation of an Service-Oriented-Architechture (SOA) that builds an application out of many small independent services. [25] Each communicating an individual task and together creating an application. In comparison to monolithic architectures, microservices are easier to scale and are also more reliable in the sense of if one service or node fails the complete system does not have to shut down, which is the case in a monolithic system, see figure 2.5. [26] [27] [28]

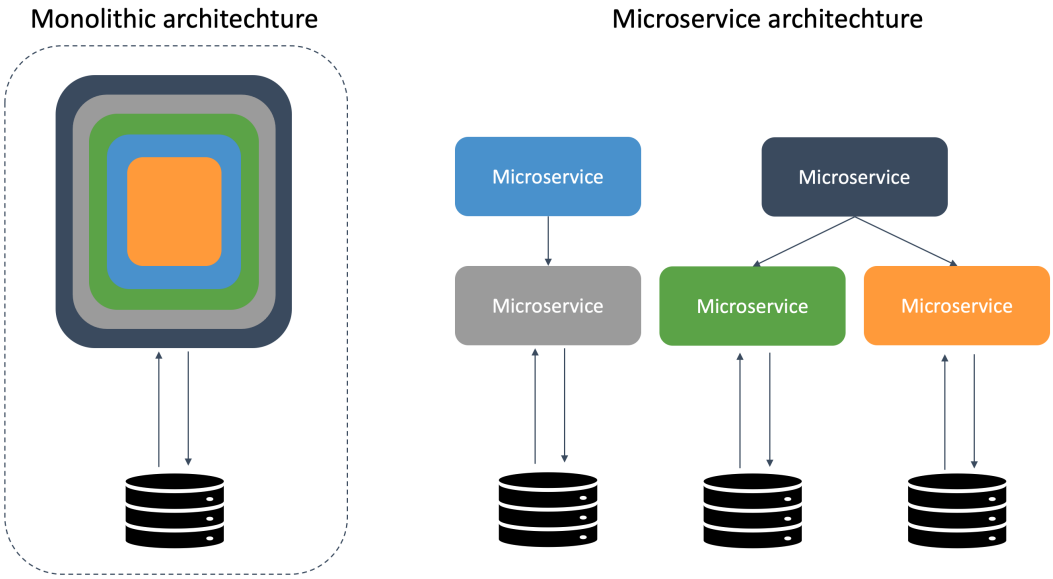


Figure 2.5: System architectures

2.5 Docker

Docker is an open platform to simplify the deployment, running and delivery of applications. The delay between writing code and running it can be reduced with Docker. Docker is often described as a set of tools that allows developers to create, publish and run containers. [29]

2.5.1 Container

Containers is self-contained pieces of software containing all necessary libraries, system tools, code and run time (e.g Java), in short everything needed to run a program. In comparison to a Virtual Machine (VM) environment where each virtual space needs a guest OS, a container doesn't and can by that save both disk and processing space. [29]

2.5.2 Docker compose

Docker compose is a tool used to create applications. In a docker compose file all services are defined and multiple containers can thus be executed at the same time, forming a microservice architecture. [30]

2.5.3 WORA

Write Once Run Anywhere (WORA) is referred to as Javas promise of being able to write an application once, compile it to byte code and then run it anywhere as long as a JVM is available, see figure 2.6. Although, usually a certain directory, library or system tool is needed which makes the transfer between platforms somewhat difficult - with the goal of still preventing the application from breaking. [31]

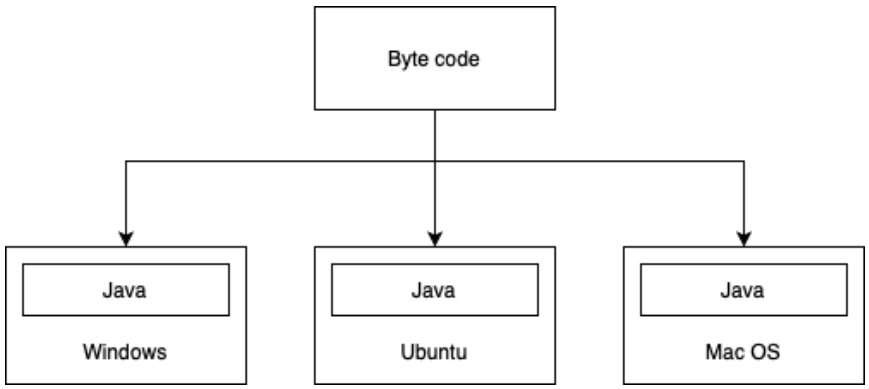


Figure 2.6: Write Once Run Anywhere using Java

2.5.4 PODA

Package Once Deploy Anywhere (PODA) is another promise saying that if an application is put in a container with all its libraries, system tools etc. The container can then be deployed anywhere as long as it is available, which is the exact concept of a docker container, see figure 2.7. [31]

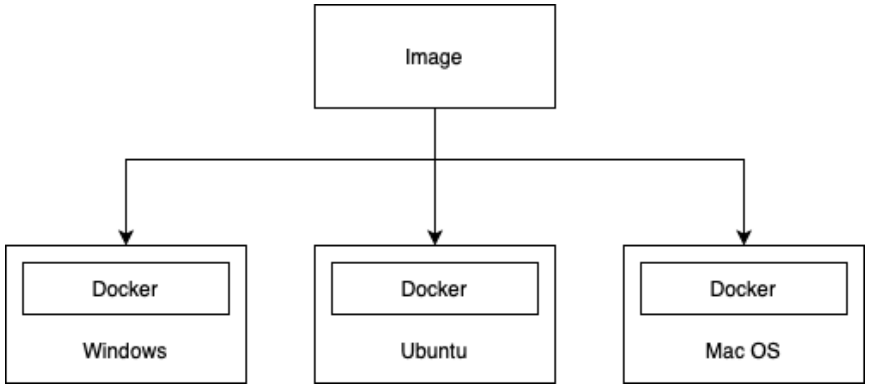


Figure 2.7: Package Once Deploy Anywhere using Java

3 Method

The following chapter presents the methods and choices made in the project. The method were divided into three phases, individually explained.

3.1 Research method

The research methodology used follows an adaptation of the design process. The adaptation was done because of the time span and project expectations, and also to fit the overall scope of the project. Figure 3.1 visualizes the research method steps. The colors of the figure defines three phases of the project where the orange color defines the first phase, yellow the second phase and green the third phase. The steps taken in each phase are described in the sections below.

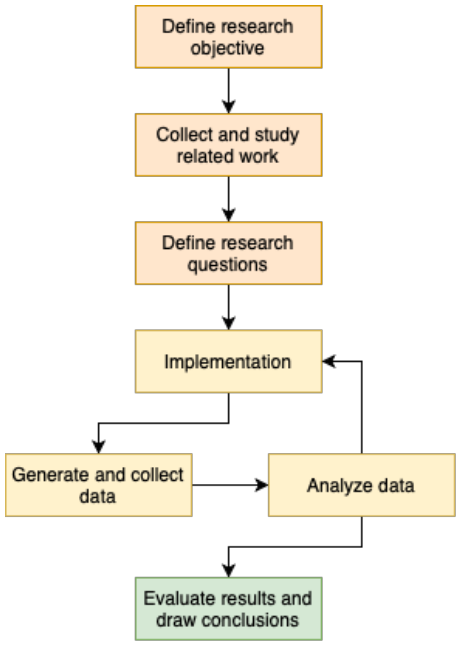


Figure 3.1: Illustrated research method

3.2 Phase 1: Defining the project scope

In the starting phase of the project the background of GCs where investigated to get a grasp over the area. Current research state and recently released articles were investigated to set a starting point for the project and decide what to examine. The research questions were established along with the research method and what steps to take in order to achieve answers to the research questions.

3.3 Phase 2: Implementation and testing

In the second phase a test environment were developed to simplify the actual testing of the GCs. The environment consisted of a REST API application producing GC log files which were entered into a parsing application that transformed the logs to JSON format. The JSON data were then used to retrieve the most vital information saved in excel files to draw diagrams which lastly formed the GC report. The report held information regarding some general GC statistics about the total time the services were working and data about the different pause time values. The distribution of pause times over time were shown as well as the percentage distribution of each kind of the pause types. The report also covered data about the heap usage over time, both before GC and after GC and lastly data about the cause of the GC pause. Information about the time to perform the requests were also collected in the client application.

3.4 Phase 3: Evaluation of results

In the third phase the GC reports generated from the developed analyzing tool were analyzed. The analysis finally lead up to conclusions about what GC fits best in a microservice environment.

The main parameters considered in the evaluation were latency and memory

management.

3.4.1 Latency

The latency in GC is manly based on two types of pause events.

1. Stop-The-World (STW) pause
2. Concurrent pause

STW events are GC pause events that needs all GC treads to perform its tasks. The second type is a concurrent pause that performs its tasks simultaneously along the application threads and only occupies one or a few threads for the GC to perform its tasks.

Each GC handles these pause times differently. E.g CMS collector uses both pause types while parallel GC uses only STW pause time events. The parallel collector although uses multiple threads while performing an STW pause event while the serial GC only uses one thread on STW pause events, resulting in extended pause events. An applications response times is highly dependent on the STW pauses, which are perceived as a delay for the user. To reduce the latency, either the number of times STW events occur or the length of them must be reduced.

3.4.2 Memory management

At each request the application creates a lot of new objects and once a request has been serviced many of these objects are of no use. At this point the GC removes these unused objects from the heap. The heap usage hence reveals a lot about the collectors performance.

4 Implementation

In the following chapter a description of the implementation process is presented. The implementation is divided into four stages related to the research questions.

4.1 Stage 1: Initial GC testing

In the first testing step a few initial tests were done to get a grasp of the GC characteristics, differences and tuning options.

The test-bed for these tests were developed as a simple application where an object were created which were directly set to null where after the garbage collector were called manually by a System.gc() call, the process is shown in figure 4.1.

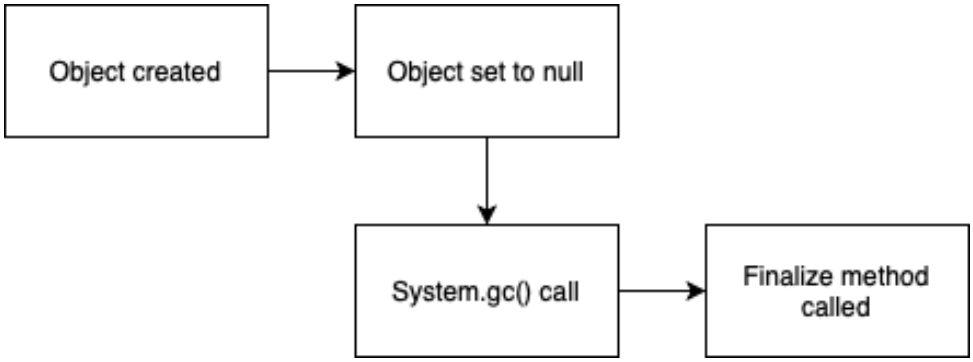


Figure 4.1: A flowchart of the application manually calling the GC

A method overriding the finalize method were implemented in the object to be able to see at what object the GC were called and if it were.

```
public class CallGC {
    @Override
    protected void finalize() throws Throwable
```

```
    {
        System.out.println("GC was called on object: " + this);
    }
}
```

A method were created where 100 million string instances were created in a loop structure to try and fill the heap. The program works by creating a string at every loop iteration and appending the string instance in a Java HashMap structure. The outOfMemory exception is captured in a try-catch statement and a message is outputted in the console.

```
private static Map<String, String> stringContainer
    = new HashMap<>();

try{
    for (int i = 0; i < 10000000; i++) {
        String newString = stringWithPrefix + i;
        stringContainer.put(newString, newString);
    }
} catch (OutOfMemoryError e){
    System.out.println("Out of memory, GC failed");
}
```

4.2 Stage 2: GC characteristics

The same application were then used to do some initial tests to both confirm and also find out new information about the collectors. The collectors were tested using different heap sizes to find a fitting size to use in the later tests, the collectors were also tested in both Graal VM and Hotspot JVM to find out if they were compatible in that environment or not.

GC logs were also produced and analyzed to be able to parse the logs into informative reports for easier extraction of information.

4.3 Stage 3: Microservice GC test

The implementation is based on creating a microservice environment and testing the GCs by tuning parameters that might affect the performance.

The test environment consisted of a web based REST API application written in Javas framework SpringBoot. The API was written with CRUD operations and a docker compose file were used to serve multiple instances of the same API on multiple ports.

A client were created using Pythons library http.client used to trigger the CRUD operations in the microservice. Each service makes a call to the next service using docker compose environment variables and the Java RestTemplate method. The following creates a microservice tree structure visualised in figure 4.2.

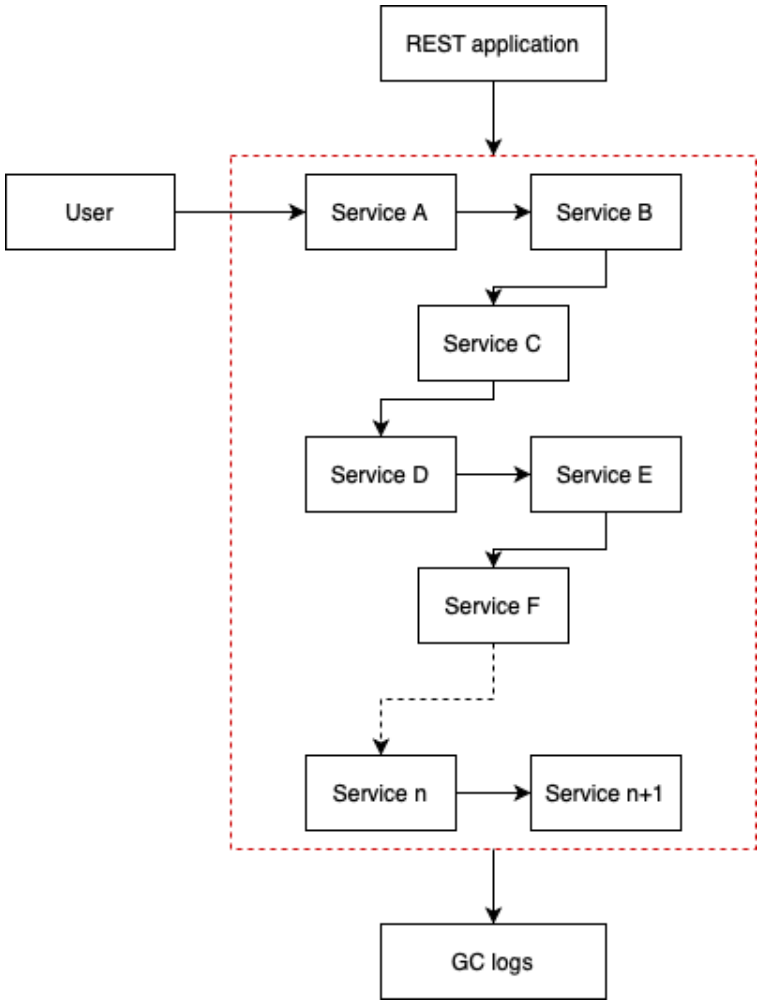


Figure 4.2: Microservice architecture created with Docker compose

Each services GC operations were saved in a separate log file which were analyzed and parsed into JSON format using Regex. The JSON file were then used to draw diagrams and create Excel files which were finally collected in PDF reports, with individual configurations depending on the GC, to create an easy understandable format in order to draw conclusions. The overall program structure is shown in figure 4.3.

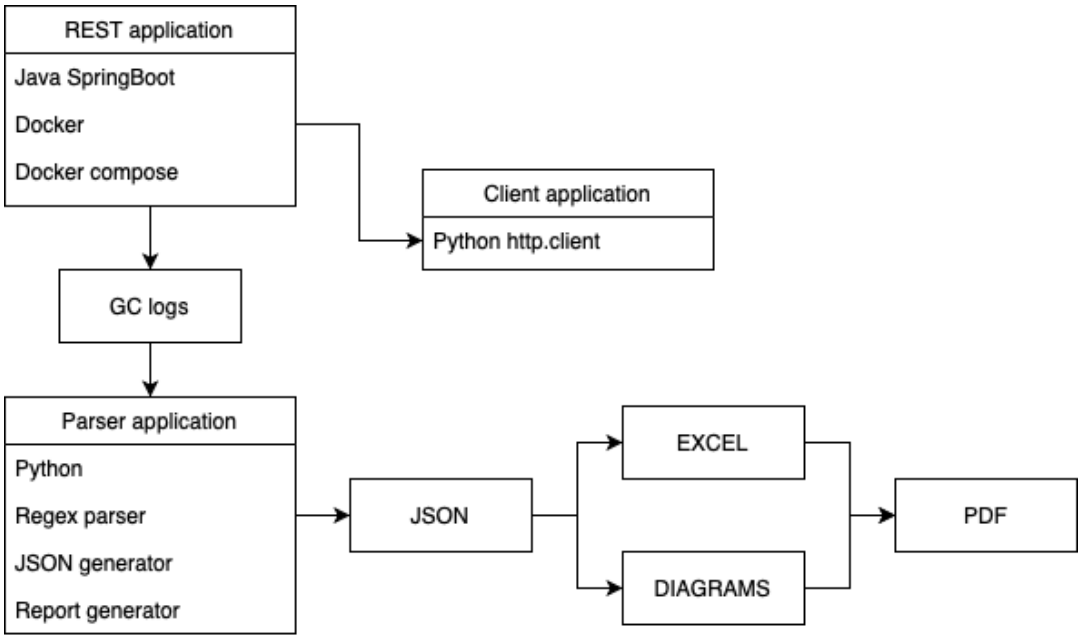


Figure 4.3: Test environment overview

4.4 Stage 4: Microservice with Database connection

The same test environment as in stage 3 where then used but this time with a connected database. The database tests were configured by connecting a locally hosted MySQL database accessed with SpringBoot JPA. Then creating a Database (DB) container in the docker compose file for an easy setup.

```
db:
    image: mysql:8.0
    ports:
      - 127.0.0.1:3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=
      - MYSQL_DATABASE=
      - MYSQL_USER=
      - MYSQL_PASSWORD=
      - MYSQL_ROOT_HOST=
```

4.5 Configurations

ZGC is available in the JDK as an experimental feature and can be enabled by enabling experimental VM options. Since Shenandoah GC is discontinued in JDK 9-10 and 12-14, and JDK 11 requires opt-in during build time the tests were performed on an alternative JDK option JDK 13 configured as in table 4.1.

Table 4.1: Enable GC

	Use
CMS	-XX:+UseConcMarkSweepGC
G1GC	-XX:+UseG1GC
ZGC	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC
Shenandoah GC	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC

Except for configuring what GC to use, the heap size were also configured to ensure equal tests for all collectors. The log files were outputted to an individual file specific for each service. The specifications made can be seen in table 4.2

Table 4.2: JVM configurations

Max heap size	-Xmx500m
Min heap size	-Xms128m
GC log to file	-Xlog:gc*:file=gc.log

Since Shenandoah GC is not avaliable in OpenJDK13 an alternative were used by specifying the following in the docker file to retrieve the AdoptOpenJDK13 image, *adoptopenjdk:13-jre-hotspot*.

5 Result

In this chapter the results of the implemented tests are presented. The structure of the chapter is based on the research questions and hence presents the results provided for each implementation step.

5.1 Stage 1: Initial GC testing

The results of the first tests were simply confirmatory of all four collectors (CMS, G1GC, Shenandoah GC and ZGC) freeing the expected heap space when an object were set to null. The System.gc() call did not affect the probability of the GC being used in this case.

A test of filling the heap space were conducted using G1GC. After about 30 million instances the heap was filled since the GC was not able to free enough space. At 30M instances an exception were made saying the memory was filled. The generated exception is shown in figure 5.1.

```
Exception: java.lang.OutOfMemoryError thrown from the
UncaughtExceptionHandler in thread "main"
```

Figure 5.1: A generated Java outOfMemory exception

5.2 Stage 2: GC characteristics

The results of the GC characteristics analysis and tests are presented in figure 5.2. It was found that Shenandoah GC are not supported in JDK13 which led to the use of an alternative JDK13 version compatible with Shenandoah GC called *adoptopenJDK13* in the following tests. Moreover, GraalVM were

not compatible with either CMS, Shenandoah GC or ZGC and were thus not further evaluated in the upcoming tests.

	Serial	Parallel	CMS	G1	Shenandoah	Z	Epsilon
Single threaded	✓						
Multi-threaded		✓	✓	✓	✓	✓	✓
Parallel threads		✓	✓	✓		✓	
Concurrent threads			✓	✓	✓	✓	
Works well on heaps > 4GB				✓	✓	✓	
Pause times < 500ms		✓	✓		✓	✓	
Pause times < 10ms					✓	✓	
Low latency goal					✓	✓	
Passive GC							✓
Low GC overhead	✓	✓					✓
Performance efficient							✓
Experimental					✓	✓	✓
Available in Graal VM	✓			✓			
Available in Hotspot VM	✓	✓	✓	✓	✓	✓	✓
Available in OpenJDK	✓	✓	✓	✓		✓	✓

Figure 5.2: GC characteristics

5.2.1 GC log format

The generated GC log file were structured in a way that each row in the file corresponded to a operation done by the GC threads. In figure 5.3 a row in the log file is shown and in figure 5.4 the structure in the produced JSON file is shown. A more detailed example of the GC log files is shown in appendix A.

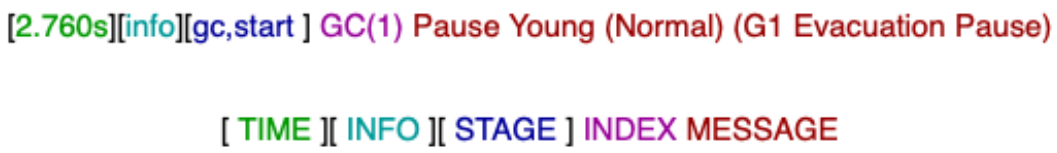


Figure 5.3: A row in a G1GC log file generated by the JVM

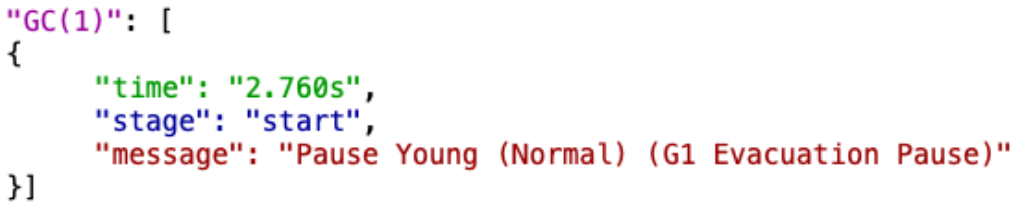


Figure 5.4: A JSON object created from the G1GC log file

5.3 Stage 3: Microservice GC test

The following section presents the results of the test in the microservice environment. The tests were performed on a computer with a 8 GB RAM space. The initial tests were to establish the best test environment for all collectors. A full GC log report generated by the parser is shown in appendix B.

5.3.1 Environment settings

In table 5.1 the number of manageable instances depending on the number of services in the microservice-tree. At a number of instances above the values in table 5.1, the application called for a OutOfMemory exception. The table values is an average of 10 equal tests.

Table 5.1: No. of manageable instances for each GC

	1 service	**3 services**	**5 services**	**10 services**
CMS	5.0 M	3.0 M	1.7 M	0.3 M
G1GC	5.0 M	3.4 M	0.7 M	0.2 M
ZGC	3.2 M	1.2 M	0.5 M	0
Shenandoah GC	4.8 M	3.0 M	0.2 M	0

Equations (5.1)-(5.4) describes the maximum number of allocated instances in a heap for each GC. Where n is the number of services available in the environment. The equations are only based on the number of services from 1 up to 10 services, stepping up one service at a time. The equations might not be correct for a number of services above 10.

$$CMS \begin{cases} if(n = 1) : n * 5 * 10^6 \\ else : n * 5 * 0.2^{\frac{n}{3}} * 10^6 \end{cases} \tag{5.1}$$

$$G1GC \begin{cases} if(n = 1) : n * 5 * 10^6 \\ else\ if(1 < n < 4) : n * 5 * 0.23^{n-2} * 10^6 \\ else : n * 5 * 0.23 * 0.13^{\frac{n}{5}} * 10^6 \end{cases} \tag{5.2}$$

$$ZGC \begin{cases} if(n = 1) : n * 3.2 * 10^6 \\ else\ if(1 < n < 10) : n * 3.2 * 0.5^n * 10^6 \\ else : 0 \end{cases} \tag{5.3}$$

$$ShenandoahGC \begin{cases} if(n < 3) : n * 4.8 * 0.4^{n-1} * 10^6 \\ else\ if(2 < n < 10) : n * 4.8 * 0.21^{n-2} * 10^6 \\ else : 0 \end{cases} \tag{5.4}$$

5.3.2 Microservice tree test

Since it was found from above tests that neither Shenandoah GC or ZGC were able to handle 10 services or above using a RAM space of 8 GB the following

tests were established at 5 services with an allocation rate of 100K instances per request. The number of requests at each test were set to 10. All results are an average of 10 equal tests.

With the above configurations a test were conducted to review the size of the services in the docker container structure, see table 5.2. The average service size is the average Docker image size at the creation of the services and the maximum service size describes the maximum image size at runtime.

Table 5.2: Docker image size for services

	Avg. service size [MiB]	**Max service size [MiB]**
CMS	125	165
G1GC	115	185
ZGC	300	420
Shenandoah GC	250	300

The time to perform the requests were calculated in the client application from the initial call until the last response. The total elapsed time for 10 GET requests are presented in table 5.3.

Table 5.3: Elapsed time for 10 GET requests

	Total request time
CMS	7.57 s
G1GC	9.58 s
ZGC	9.09 s
Shenandoah GC	5.19 s

The number of collection phases were extracted from the log files and are summarized in table 5.4. Concurrent pauses being events that partially pauses the application by using one or multiple application threads, and STW pause events fully pausing the application.

Table 5.4: No. of GC collection phases

	Concurrent	STW Pause	Total
CMS	112	226	338
G1GC	159	200	359
ZGC	79	45	124
Shenandoah GC	71	44	115

In figure 5.5 the total time of the GC collection phases were calculated and divided by the total GC collection phase time, the percentage of each phase is shown for each GC. G1GC definitely stood out by having the highest pause time among all collectors.

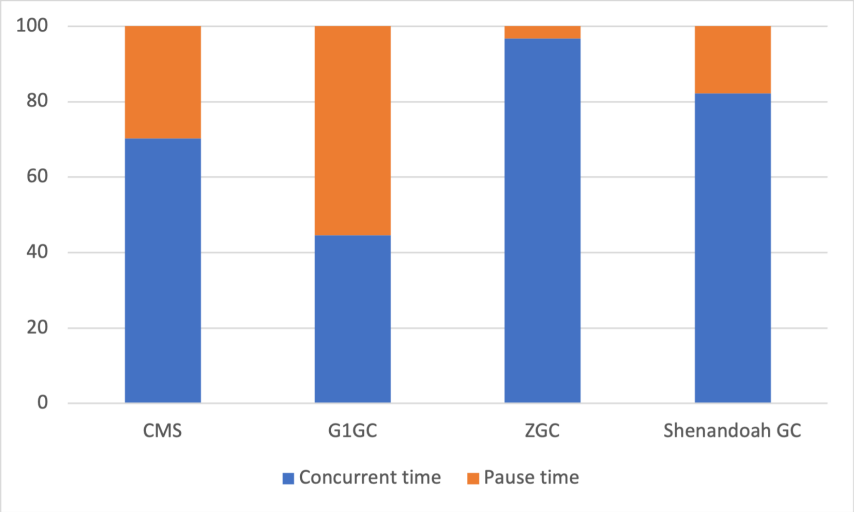


Figure 5.5: Average GC times

The total time values of the GC pause, concurrent and total times are presented in table 5.5. Table 5.5 shows that G1GC definitely has the highest amount of STW pause times, whereas CMS has the highest total GC work time. ZGC has a very small amount of STW pause time followed by Shenandoah GC which total GC time also is the lowest.

Table 5.5: Total GC stats

	Total STW	**Total concurrent**	**Total GC**
CMS	6.4 s	18.4 s	24.8 s
G1GC	11.7 s	15.5 s	27.2 s
ZGC	0.3 s	9.0 s	9.3 s
Shenandoah GC	0.9 s	4.5 s	5.4 s

The distribution of pause- and concurrent times differ among the collectors, seen by the plots in figure 5.6 - figure 5.9. The figures display two graphs per service where the top graphs showing the STW pause time distribution and the bottom graphs showing the concurrent time distribution, each figure containing graphs using a certain GC. The x-axis describes the time interval of the GC events in ms and the y-axis describes the length of the pause in ms. Observe that the y-axis scale are not equal for all graphs and that the x-axis in figure 5.8, STW pause time graphs, has a different starting value than on the others. By the figures it is visible that Shenandoah GC definitely has a more stable distribution of pause times than the others, although the amount of pause times and concurrent times are very evenly distributed in both Shenandoah GC and ZGC.

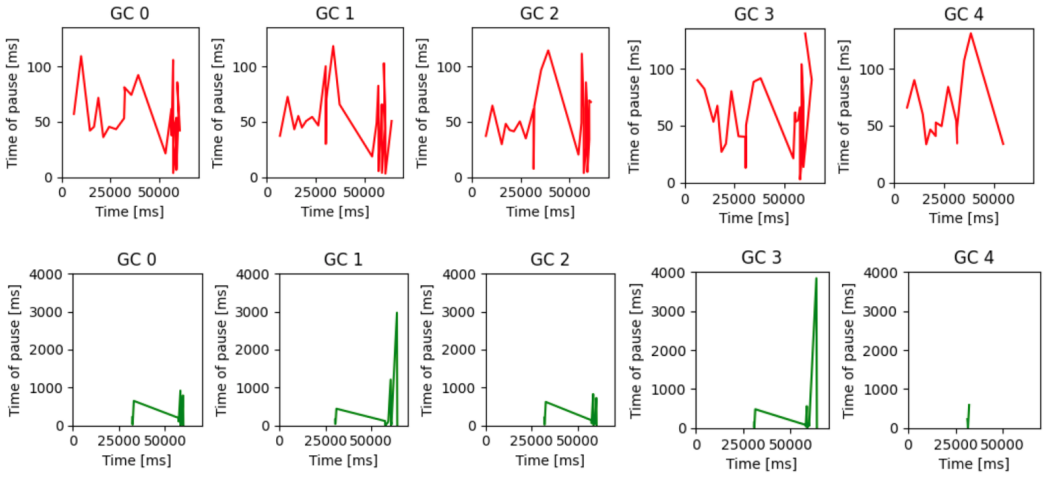


Figure 5.6: CMS pause and concurrent times

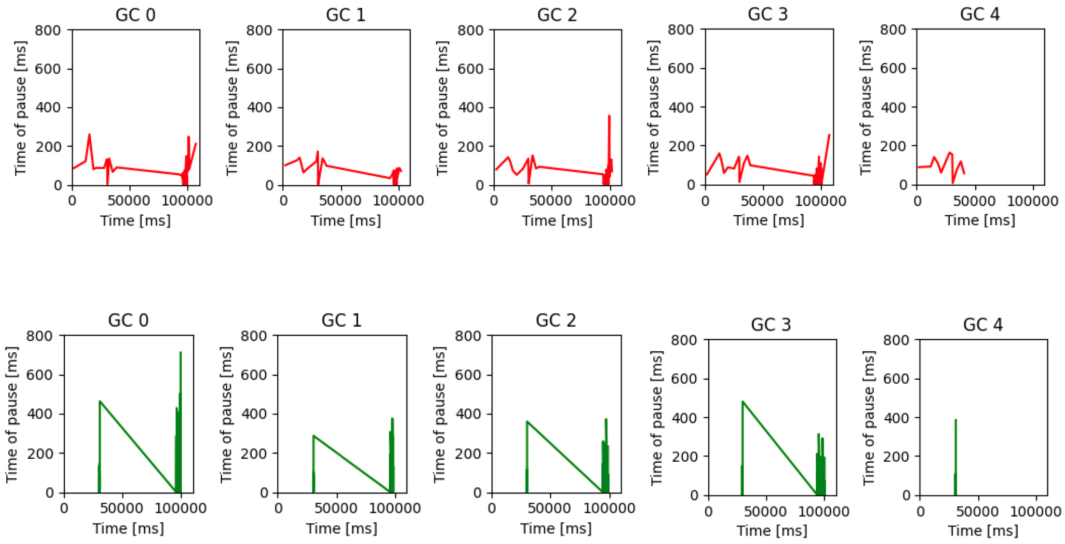


Figure 5.7: G1GC pause and concurrent times

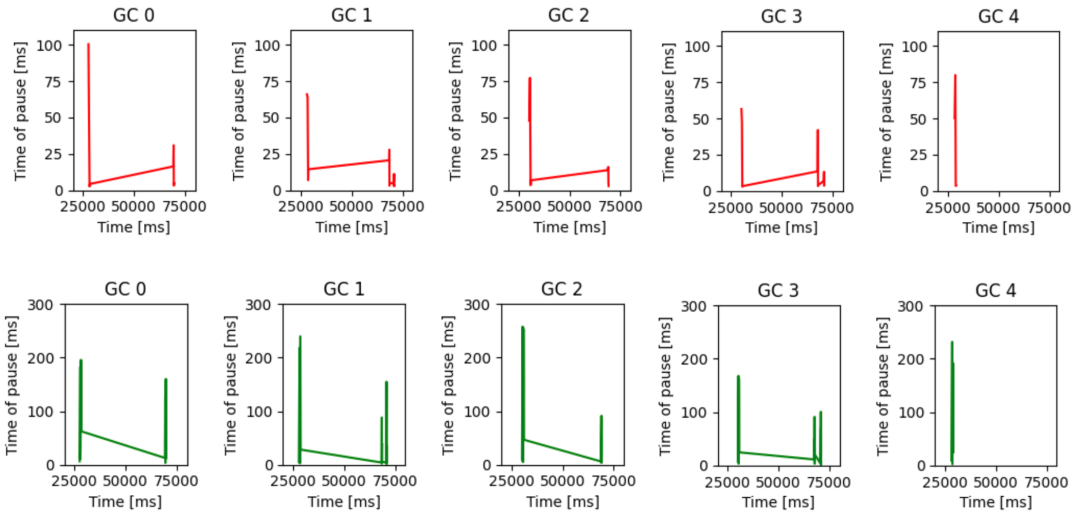


Figure 5.8: Shenandoah GC pause and concurrent times

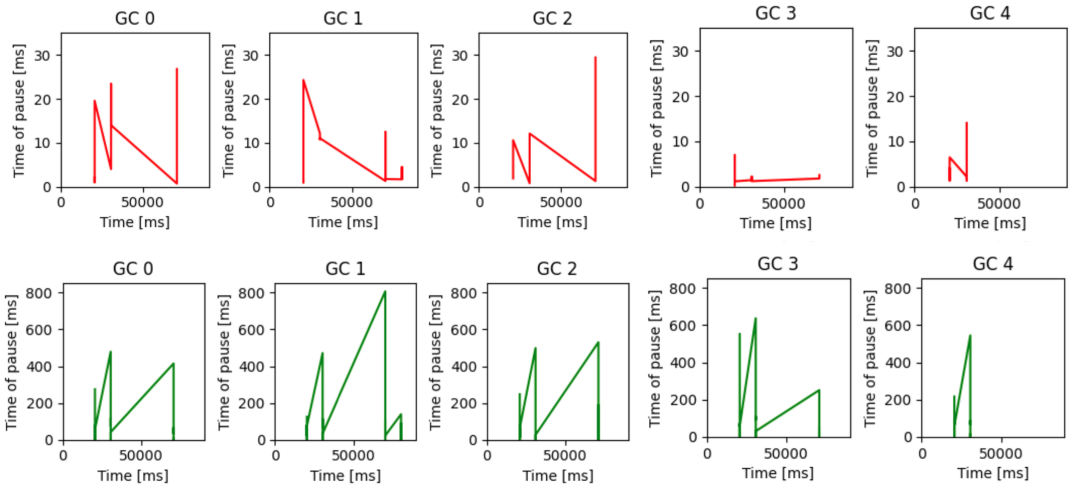


Figure 5.9: ZGC pause and concurrent times

5.3.3 Heap usage

At each request the application creates a lot of new objects and once a request has been serviced many of these objects are of no use. At this point the GC removes these unused objects from the heap. The main goal of the GC is to free as much unnecessary memory usage as possible. In figure 5.10 - figure 5.13 the top graphs shows the heap usage (the heap size) before the GC ran and the bottom graphs show the heap usage (the heap size) after the GC ran. In all graphs in figure 5.10 - figure 5.13 the x-axis indicates the time at which the GC event ran in seconds and the y-axis indicates the heap space in MB. Observe that the y- and x-axis scale are not the same in all figures.

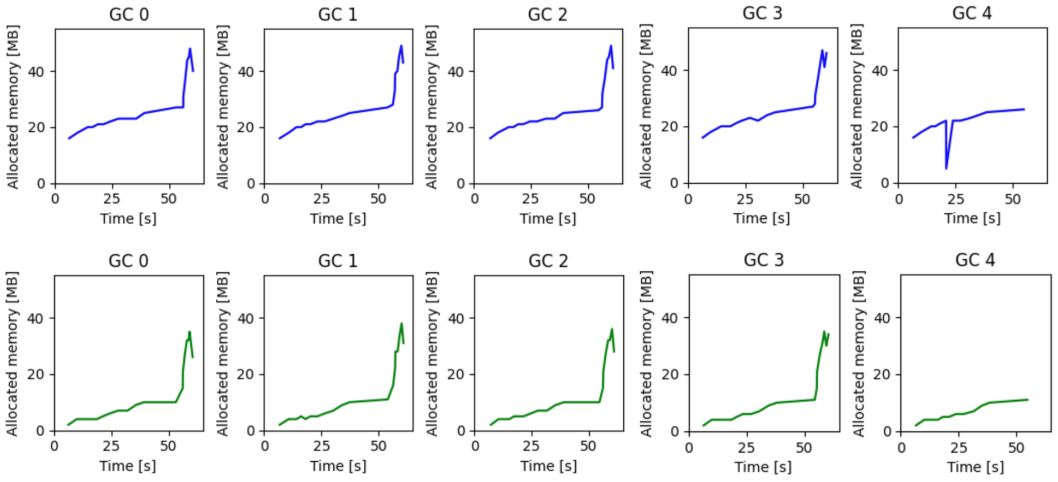


Figure 5.10: CMS heap usage

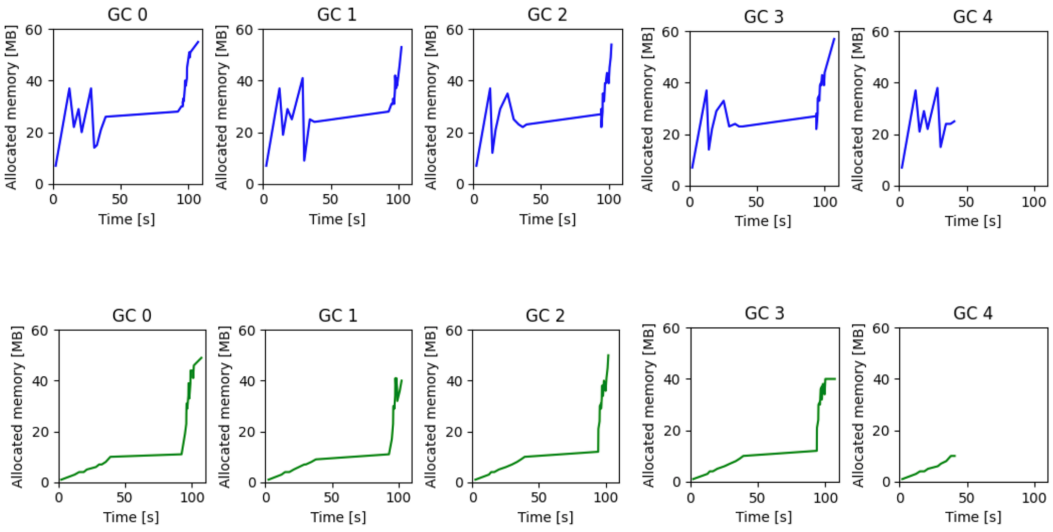


Figure 5.11: G1GC heap usage

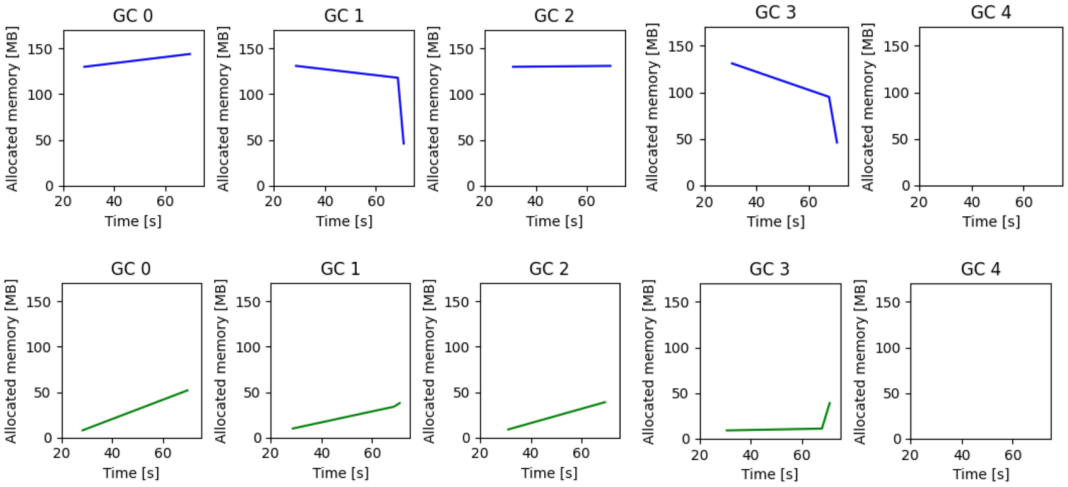


Figure 5.12: Shenandoah GC heap usage

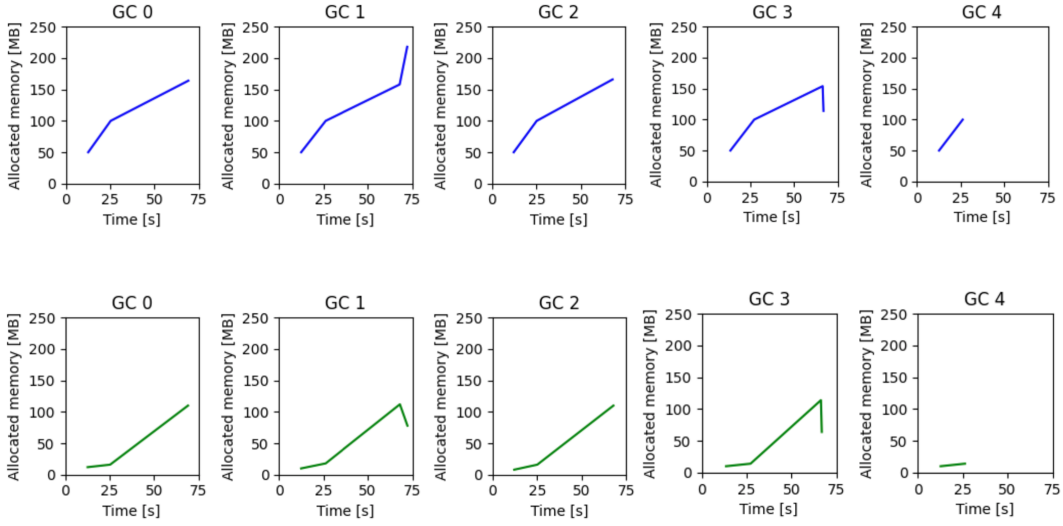


Figure 5.13: ZGC heap usage

5.3.4 Scenario performance

A worst case scenario where the number of GC work times per GC multiplied by the average pause time were calculated. The results are presented in table 5.6

Table 5.6: Worst case GC pause time

	Worst case
CMS	13.97 s
G1GC	35.83 s
ZGC	0.89 s
Shenandoah GC	2.49 s

5.4 Stage 4: Microservice with Database connection

The following tests were conducted using a database connection. The test environment consisted of 5 microservices and an allocation rate of 100K instances per GET requests. In each test 10 GET requests were performed by each GC.

In table table 5.7 the elapsed time to complete 10 GET requests with and without a database connection is shown. As seen in the following tables no data were captured for the ZGC collector. That with the reason that ZGC called a NullPtrException resulting in one or multiple services being unable to maintain connected.

Table 5.7: Elapsed time for 10 GET requests with DB connection

	Total time
CMS	17.95 s
G1GC	19.69 s
ZGC	-
Shenandoah GC	93.44 s

Considering the decrease in manageable allocated instances per service the Docker image size were examined to see if that had any influence on the crash. The image size of each collectors images is shown in table 5.8. The average service size is the average Docker image size at the creation of the services and the maximum service size describes the largest image size at runtime. It is visible by table 5.8 that ZGC had significantly larger images than the other collectors.

Table 5.8: Docker image size with DB connection

	Avg. service size [MiB]	**Max service size [MiB]**
CMS	200	240
G1GC	225	280
ZGC	550	700
Shenandoah GC	350	380

Figure 5.14 shows the average pause time percentage against the concurrent time made by the GC, using a database connection. Concurrent pauses being events that partially pauses the application by using one or multiple application threads, and STW pause events fully pausing the application.

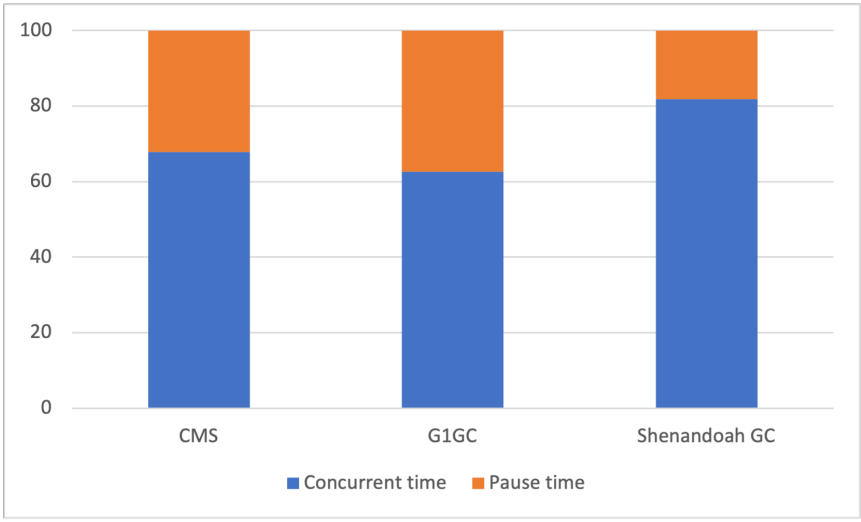


Figure 5.14: Average GC times with DB connection

The number of GC phases extracted from the GC log files are presented in table 5.9.

Table 5.9: No. of GC collection phases

	Concurrent	**STW Pause**	**Total**
CMS	324	470	794
G1GC	208	260	468
ZGC	-	-	-
Shenandoah GC	168	101	269

In figure 5.15 - figure 5.17 the distribution of pause times are shown. The top graphs in each figure representing the STW pause times, and the bottom graphs in each.figure representing the concurrent times. The x-axis describes the time interval of the GC events in ms and the y-axis describes the length of the pause in ms. Observe that the y-axis scales are not equal for all graphs.

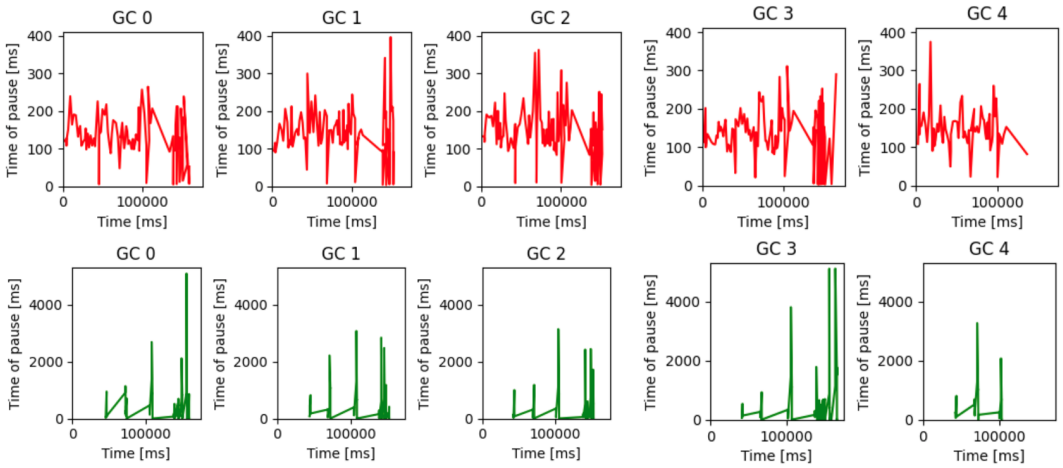


Figure 5.15: CMS pause and concurrent times

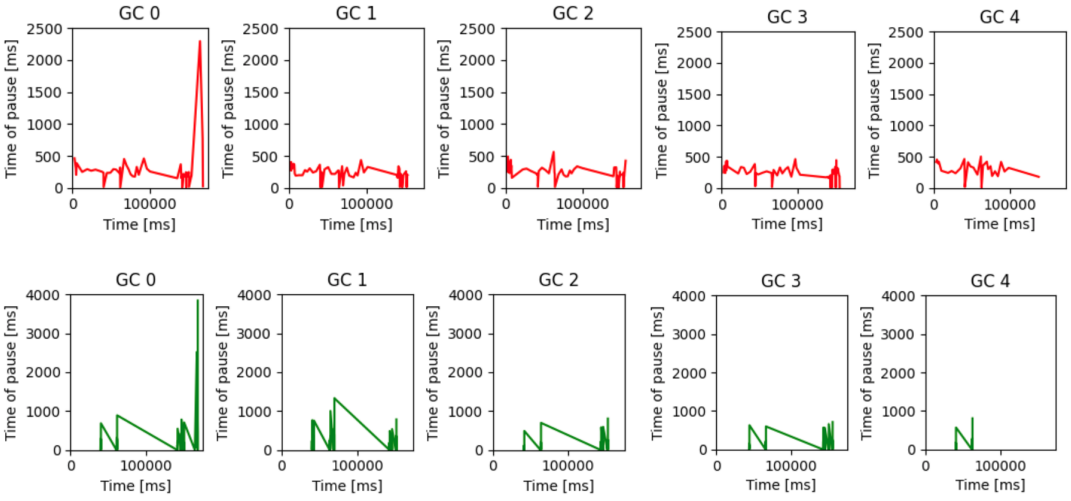


Figure 5.16: G1GC pause and concurrent times

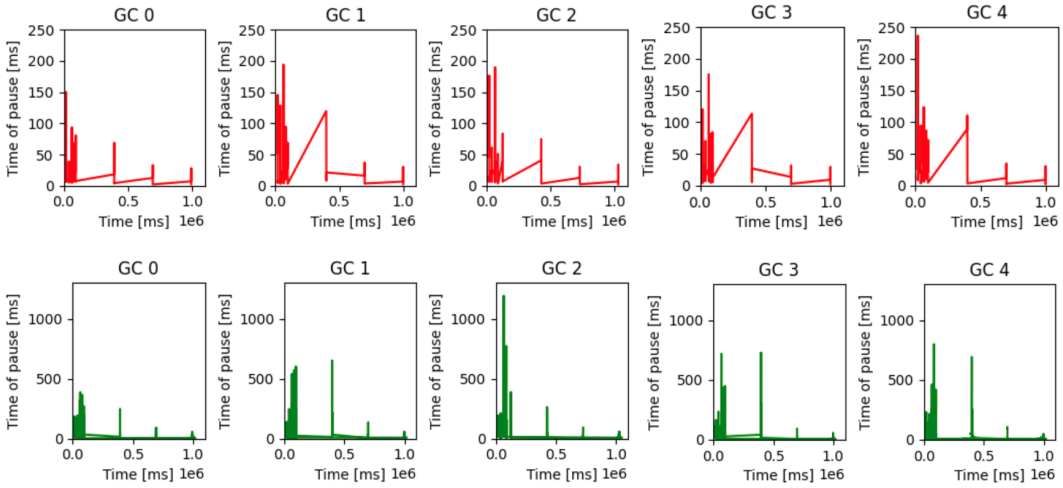


Figure 5.17: Shenandoah GC pause and concurrent times

5.4.1 Heap usage

The heap usage for the GCs using a database connection is shown in figure 5.18 - figure 5.20. The top graphs shows the heap usage before the GC ran and the bottom graphs show the heap usage after the GC ran. In all graphs in figure 5.18 - figure 5.20 the x-axis indicates the time at which the GC event ran in seconds and the y-axis indicates the heap space in MB. Observe that not all figures have the same scale on the y-axis.

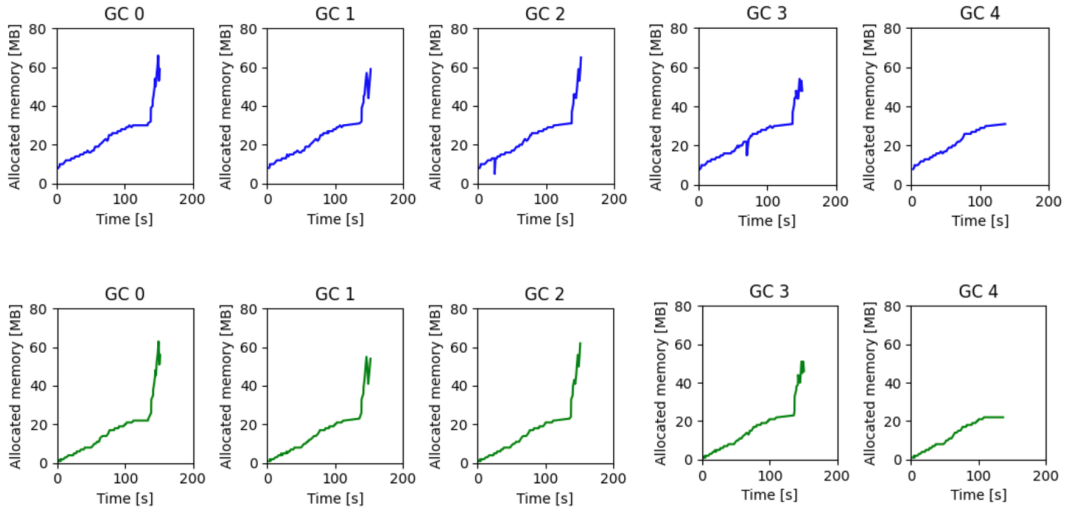


Figure 5.18: CMS heap usage with DB connection

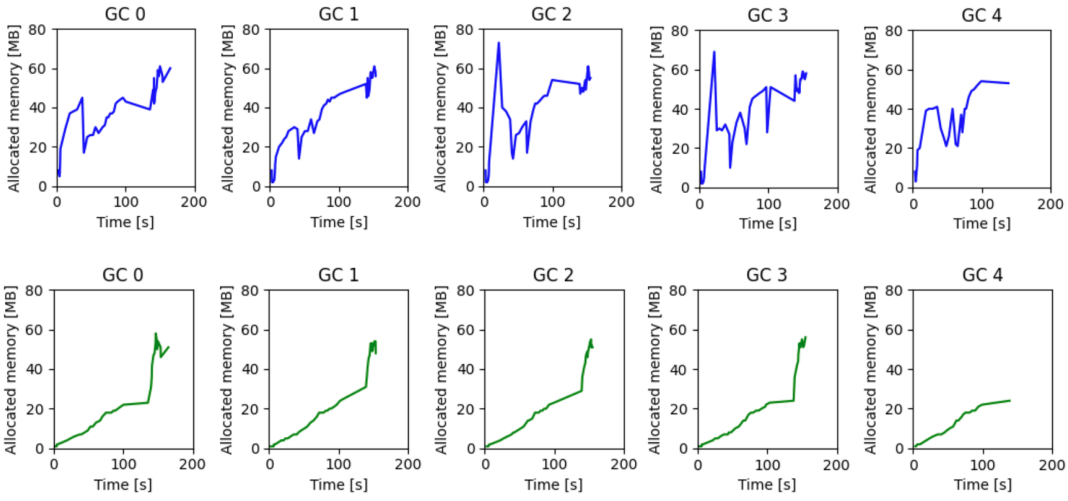


Figure 5.19: G1GC heap usage with DB connection

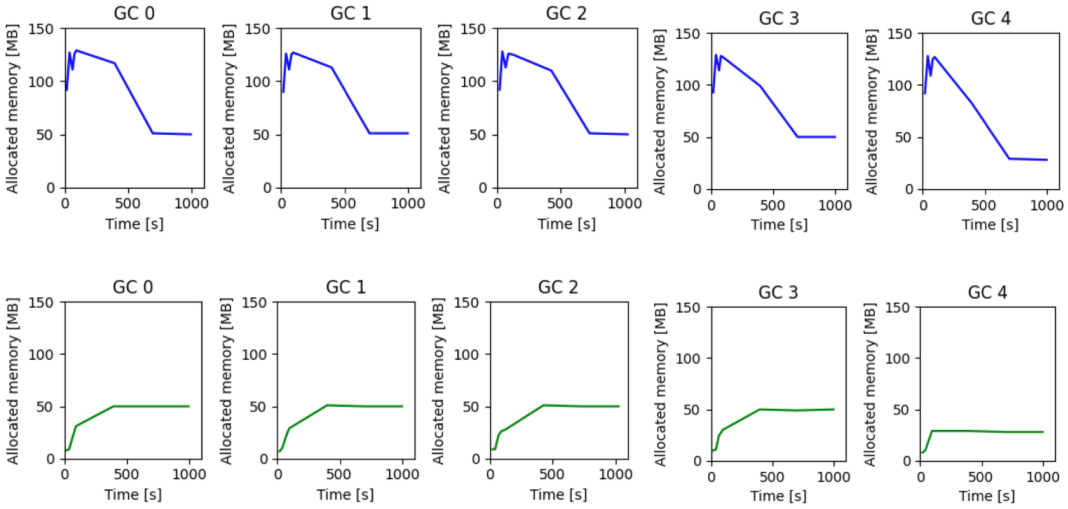


Figure 5.20: Shenandoah GC heap usage with DB connection

6 Discussion

The following chapter discusses the obtained results presented in the previous chapter.

6.1 Stage 1: Initial GC testing

Since the objects were set to null the garbage collector were definitely expected to be executed, which they also were. Which confirmed the GC activity. Even though an explicit System.gc() call is not recommended because it muddles with the JVM GC activity. For cases like testing it anyways can be a good advantageous functionality to try and force the GC at a point in the application for evaluation purposes. A System.gc() call does however not guarantee that the GC will run. The System.gc() call can instead be seen as a recommendation to the GC to run at a specific part of the program.

Filling the heap were produced as a proof of concept model to prove that the GC does have opportunities for improvement. Despite the GC purpose of replacing manual memory handling, errors still occur, proven by the received OutOfMemory exception.

6.2 Stage 2: GC characteristics

The GC characteristics table containing information about the GCs were created to form an initial understanding and overall grasp of the attributes of the GCs and their differences. By figure 5.2 ZGC and Shenandoah GC is shown to have GC STW pause times below 10ms which by later tests were not the case in a microservice environment.

Regarding the GC log files it was clear that the log files were structured in a way that were capable of parsing. The log files also without any added

justifications, apart from using the detailed GC log version, consisted of the most vital information necessary for a somewhat in-depth analysis.

6.3 Stage 3: Microservice GC test

Many times performance evaluations are configured in an environment with high specifications of performance, available memory space etc. It is clearly shown that when limiting these parameters and performing the test in a more real-world system, the tests of GCs does not always fulfill the expectations. It also goes to show that some GCs are yet not developed to work completely immaculate in microservice environments.

Primarily, evaluating the manageable instances for each GC there is a huge decrease in the convenient instances at the increase of number of services. The collector handling the increase of microservices best by the number of instances are CMS, since it is able to handle the highest amount of instances. Also, CMS along with ZGC has the most stable decrease of manageable instances at the increase of services. Both G1GC and Shenandoag GC made a huge drop of manageable instances at about 5 services. This may be an indication of the GCs not being able to handle a microservice environment since most microservice environments consists of more than 5 services, despite the fact that all microservices may not be allocating memory in the same phase as in the conducted tests. Although, at 10 services both ZGC and Shenandoah threw a OutOfMemoryException as soon as the heap were trying to allocate any memory at all.

This led to evaluate the Docker image size for each GC. It was by that clear that ZGC had the largest service sizes, partly explaining the drop in manageable instances. It also correlates for the other collectors. Also, in Stage 1 when testing GC activity, the number of manageable instances were almost 10x more in the case of a monolith service architecture, not using Docker.

The time to perform the requests differed among the collectors and Shenandoah GC had the shortest request time, almost half the time as G1GC and

ZGC. CMS again performed well when comparing against the other collectors, although it being the earliest of the collectors.

Even though the request time were low the affects on the systems may differ from these results. One indication of how the GC affects the system were by looking at the GC pause and concurrent times. That information definitely gave away more about the performance. Although G1GC did not have the highest amount of STW pause times the total STW time were the highest of all collectors and the percentage stood out, being the overall highest GC percentage of STW pause time. As seen in figure 5.6 - figure 5.9 it is visible that all collectors top graphs (showing the STW pause time) has a peak at the beginning of the plot. The explanation being all collectors has an initial marking step to start off the GC process.

The heap usage also gives away a lot about the performance of the GC. Since all collectors performed under the same conditions it is clear that the collectors have differences in performance. An important aspect to be aware of is that the first request took a bit longer than the following requests, as usual in Java because Java stores information about the last request, making the following request time faster. As is shown in figure 5.10 presenting the heap usage before the GC and after GC, Shenandoah GC definitely were able to free the most memory followed by ZGC, G1GC and lastly CMS.

The worst case scenario were set up as a guideline and an indicator to what would happen if all GC pause times ended up being STW pause times. An average value of all GC pause times were used as the STW pause time value which were multiplied by the total amount of GC pause events. The results again pointed towards ZGC providing the by far lowest total time, which again implies it being beneficial for systems requiring low latency.

6.4 Stage 4: Microservice with Database connection

rThe database tests in many ways differed from the tests done without any database connection. Especially the request time differed greatly between the collectors with CMS performing all the requests at a twentieth of time compared to Shenandoah GC. Although the services weren't directly connected to the database or the application using the database the image sizes were significantly larger than when the database were not included in the system. In particular ZGC nearly doubled its average service size when having a connected database in the architecture.

ZGC did not respond well to the tests in the environment with a database connection. This is most probably due to ZGCs multi-memory mapping technique, meaning it uses more memory mappings per process than other collectors. The maximum mappings per process were adjusted as well as the heap maximum and minimum size without any differences. The reason might have been because of the use of a docker container structure. The internal docker preferences were also tweaked by lowering the resources, specifically the available memory and swap space, which also had no impact on ZGC. Thus no further data regarding ZGC were collected in this research.

The number of GC collection phases alike the image sizes increase considerably. Both CMS and Shenandoah GC more than doubled both the amount of concurrent pauses and STW pauses (CMS nearly tripled the amount of concurrent pauses). G1GC increased both STW and concurrent pause times by about 20 percent. G1GC did also improved the percentage of total elapsed STW pause time in comparison to concurrent time by about 20 percent when using a database connection.

Regarding the heap usage the database did not make any outstanding differences in comparison to not having a database connected.

7 Conclusion

The main objective for this research were to shed light on the CMS, G1GC, Shenandoah GC and ZGC performance in a microservice environment. The study was able to demonstrate the importance of the choice of the right GC and its impact on microservice applications by inspecting each collectors performance in the same environment. Furthermore, this study also managed to confirm ZGC and Shenandoah GCs latency advantage compared to the default collector G1GC in JDK13 and CMS. In the following chapter some of the final conclusions made in this thesis are presented.

ZGC showed great results regarding keeping low STW pause times and being able to free a lot of garbage objects in the heap. The results obtained aims towards ZGC being a good option for a system with high resource specifications, when opting for low response times, e.g in a web application. Although, ZGC did not perform well on either a microservice application with many services connected, neither on an application with a database connection. The results in this research leans towards the reason being ZGC requiring more memory space to build the service images when using a Docker container. If more memory space were available, ZGC may have had different results.

CMS definitely had better results than were expected in the start of the project. This with the reason of CMS being the oldest of the GCs and microservice application being a fairly new concept. CMS although showed to perform very well in the microservice environment. Both by being able to handle the most instances in larger microservice systems, having the second best request time (and best in the system using a DB connection) CMS also had a very stable pause time as well as heap usage behavior. Still, the STW pause time is quite high which has a pretty big impact on the final decision of the GC choice since it has large impact on the latency of the application.

As a final conclusion, microservice applications working on a large heap optimizing for low response times should primarily consider using ZGC and secondary Shenandoah GC. For an application running on a smaller heap space (less than 4GB) CMS could be a stable option if the system is opting to be as scalable as possible regarding number of services. If the requirement is only a few services ZGC again performs very pleasantly concerning STW pause times and can thus be considered a good option. The recommendation that can be made regarding what GC to choose in the case of a microservice environment using a database connection leans towards CMS performing the most stable results out of the collectors. Even though Shenandoah GC does have the lowest STW pause time percentile the time to perform the application requests were by far the most time consuming. Again though, if the aim is to achieve the lowest STW pause times Shenandoah GC should be used as GC. As a final note, there of course is always a balance between memory and hardware usage that should be considered for each individual case.

7.1 Ethical aspects

The ethical aspects of this research mainly relates to providing a fair comparison to all included collectors. This is done by performing the test in an environment that is equal for all GCs. It is also vital for the results to be presented in a unbiased manner and present any injustices caused by the tests. Especially since the results of this study may form the basis for decisions regarding what GC to pick for developers. The performance evaluation of GCs can be highly beneficial in several aspects. Depending on the performance feature reviewed, a better GC configuration can lead to,

- Application efficiency
- Reduced memory footprint
- Reduced CPU consumption
- Lowered hardware costs

The more efficiently we utilize the resources we already have, the more can be achieved, which is important in both a sustainable and effective perspective. By comparing the GCs with each other developers can also be helped into making better decisions when choosing a GC for their specific environment. Also here preventing rebuilding things, and by that save time, environment, money and workload. Another aspect to consider is the power consumption of a microservice application in comparison to a monolith system.

7.2 Future work

The future of GC definitely should involve developing and consider solutions suitable for microservice environment. There exists a definite need for continous and uninterrupted services in modern program architectures. The ways of adjusting the GC tuning options are massive and thus future investigations has a lot of potential improvements.

Since the ZGC had bad response when using a database connection more evaluation should definitely be done in that area. Especially since the reason of the application not responding to JVM tuning regarding memory mappings was not found. A good idea might be to perform the tests on a different OS or investigate if the use of Docker has any impact on the problem by using another microservice structure.

This study were limited to a simple application and results relating to parameters such as memory footprint and CPU usage were limited, which could be of interest in future research. Also, apart from tuning GC options; hardware setup and application properties would also be interesting to adjust. Since the results aim towards the GCs being application dependent, referring to the allocation test, the performance in other application structures is also of interest. Furthermore, the java version and JVM type would be of interest for further tests since this research did the major tests on Java version 13 using HotSpot JVM, especially since GraalVM is said to be appropriate for a microservice architecture. Regarding the hardware in the test environment

it also is an aspect that could be compared to other hardware setups. E.g by increasing or limit the memory usage or available CPU threads.

References

[1] *Java se naming and versions*, Oracle. [Online]. Available: https://www.oracle.com/java/technologies/javase/naming-and-versions.html (visited on 02/05/2021).

[2] *Java garbage collection basics*, Oracle. [Online]. Available: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html (visited on 02/02/2021).

[3] *Akamai online retail performance report: Milliseconds are critical*, Akamai. [Online]. Available: https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp (visited on 04/01/2021).

[4] Y. Einav, *Amazon found every 100ms of latency cost them 1% in sales*, Gigaspaces, 2019. [Online]. Available: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/?fbclid=IwAR33heyNAI-a827DIXCizeLtp2aN-L0iNTYoL8-vVAVf5hfQniAthUoOrL0 (visited on 04/19/2021).

[5] *Java 11 guidelines*, Gigaspaces. [Online]. Available: https://docs.gigaspaces.com/latest/rn/java11-guidelines.html (visited on 02/07/2021).

[6] *Jep 248: Make g1 the default garbage collector*, OpenJDK. [Online]. Available: http://openjdk.java.net/jeps/248 (visited on 04/12/2021).

[7] *Understanding memory management*, Oracle. [Online]. Available: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html (visited on 02/06/2021).

[8] O. Agesen, D. Detlefs, and J. E. B. Moss, *Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines*, 1998. DOI: 10.1145/277650.277738.

[9] S. Kadavath, *Understanding hotspot vm garbage collectors (gc) in depth*, DZone, 2018. [Online]. Available: https://dzone.com/articles/understanding-garbage-collectorsgc-in-depth (visited on 03/03/2021).

[10] H. Grgić, B. Mihaljević, and A. Radovan, *Comparison of garbage collectors in java programming language*, Rochester Institute of Technology Croatia, 2018. [Online]. Available: https://www.researchgate.net/profile/Aleksander-Radovan/publication/326701102_Comparison_of_garbage_collectors_in_Java_programming_language/links/5d5f9ba392851c37637370eb/Comparison-of-garbage-collectors-in-Java-programming-language.pdf (visited on 05/19/2021).

[11] *Deep dive into the new java jit compiler – graal*, Baeldung. [Online]. Available: https://www.baeldung.com/graal-java-jit-compiler (visited on 02/25/2021).

[12] *Graalvm*, GraalVM. [Online]. Available: https://www.graalvm.org (visited on 02/20/2021).

[13] P. Jayawardhana, *Jvm garbage collection and optimizations*, Medium, 2020. [Online]. Available: https://medium.com/@Pushpalanka/jvm-garbage-collection-and-optimizations-3f338c86de27 (visited on 02/05/2021).

[14] *Jvm garbage collectors*, Baeldung. [Online]. Available: https://www.baeldung.com/jvm-garbage-collectors (visited on 02/16/2021).

[15] *Type of garbage collector*, PerfMatrix. [Online]. Available: https://www.perfmatrix.com/type-of-garbage-collector/ (visited on 02/08/2021).

[16] *Garbage first garbage collector tuning*, Oracle. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/g1gc.html (visited on 02/17/2021).

[17] *Getting started with the g1 garbage collector*, Oracle. [Online]. Available: https://www.baeldung.com/jvm-garbage-collectors (visited on 02/16/2021).

[18] *The z garbage collector*, Oracle. [Online]. Available: https://docs.oracle.com/en/java/javase/11/gctuning/z-garbage-collector1.html#GUID-A5A42691-095E-47BA-B6DC-FB4E5FAA43D0 (visited on 02/04/2021).

[19] *An introduction to zgc: A scalable and experimental low-latency jvm garbage collector*, Baeldung. [Online]. Available: https://www.baeldung.com/jvm-zgc-garbage-collector (visited on 02/04/2021).

[20] *Main*, OpenJDKWiki. [Online]. Available: https://wiki.openjdk.java.net/display/zgc/Main (visited on 03/15/2021).

[21] I. Clark, *Shenandoah gc*, OpenJDK, 2021. [Online]. Available: https://wiki.openjdk.java.net/display/shenandoah/Main (visited on 02/19/2021).

[22] K. Chandrakant, *Experimental garbage collectors in the jvm*, Baeldung, 2021. [Online]. Available: https://www.baeldung.com/jvm-experimental-garbage-collectors (visited on 02/24/2021).

[23] P. Nima, *A brief overview of garbage collectors in java, because cleanliness is necessary*, Medium, 2020. [Online]. Available: https://medium.com/swlh/a-brief-overview-of-garbage-collectors-in-java-because-cleanliness-is-necessary-f3dd9babc2cb (visited on 02/08/2021).

[24] E. Goebelbecker, *Java garbage collection*, DZone, 2019. [Online]. Available: https://dzone.com/articles/java-garbage-collection-3 (visited on 02/22/2021).

[25] M. Persson, *Microservices – vad är det och hur utnyttjar du det*, Telia, 2018. [Online]. Available: https://www.cygate.se/blogg/microservices-vad-ar-det-och-hur-utnyttjar-du-det/ (visited on 03/12/2021).

[26] C. Richardsson, *What are microservices?*, Microservices, 2020. [Online]. Available: https://microservices.io (visited on 04/21/2021).

[27] *Understanding microservices*, Redhat. [Online]. Available: https://www.redhat.com/en/topics/microservices (visited on 03/12/2021).

[28] N. Alshuqayran, N. Ali, and R. Evans, *A systematic sapping study in microservice architecture*, 2016. DOI: 10.1109/SOCA.2016.15.

[29] *Docker overview*, Docker. [Online]. Available: https://docs.docker.com/get-started/overview/ (visited on 02/10/2021).

[30] *Overview of docker compose*, Docker. [Online]. Available: https://docs.docker.com/compose/ (visited on 02/11/2021).

[31] *Chapter 1. introduction to docker*, O'reilly. [Online]. Available: https://www.oreilly.com/library/view/docker-for-java/9781492042624/ch01.html (visited on 02/11/2021).

Appendices

A GC log example

A short example of the GC log file structure for each collector.

A.0.1 CMS

```
[0.039s][info][gc] Using Concurrent Mark Sweep
[0.043s][info][gc,heap,coops] Heap address: 0x00000000c2600000, size:
986 MB, Compressed Oops mode: 32-bit
[6.729s][info][gc,start     ] GC(0) Pause Young (Allocation Failure)
[6.733s][info][gc,task      ] GC(0) Using 2 workers of 2 for evacuation
[6.760s][info][gc,heap      ] GC(0) ParNew: 16896K->2112K(19008K)
[6.762s][info][gc,heap      ] GC(0) CMS: 0K->94K(42368K)
[6.764s][info][gc,metaspace ] GC(0) Metaspace: 3556K->3556K(1056768K)
[6.766s][info][gc           ] GC(0) Pause Young (Allocation Failure)
16M->2M(59M) 37.321ms
[6.785s][info][gc,cpu       ] GC(0) User=0.01s Sys=0.01s Real=0.05s
[10.617s][info][gc,start     ] GC(1) Pause Young (Allocation Failure)
[10.622s][info][gc,task      ] GC(1) Using 2 workers of 2 for
evacuation
[10.685s][info][gc,heap      ] GC(1) ParNew: 19008K->1764K(19008K)
[10.687s][info][gc,heap      ] GC(1) CMS: 94K->2383K(42368K)
[10.688s][info][gc,metaspace ] GC(1) Metaspace: 5983K->5983K(1056768K)
[10.690s][info][gc           ] GC(1) Pause Young (Allocation Failure)
18M->4M(59M) 72.525ms
[10.691s][info][gc,cpu       ] GC(1) User=0.01s Sys=0.01s Real=0.07s
```

A.0.2 G1GC

```
[0.047s][info][gc,heap] Heap region size: 1M
[0.077s][info][gc     ] Using G1
[0.096s][info][gc,heap,coops] Heap address: 0x00000000e0e00000, size:
498 MB, Compressed Oops mode: 32-bit
[0.102s][info][gc,cds       ] Mark closed archive regions in map:
[0x00000000fff00000, 0x00000000fff79ff8]
[0.109s][info][gc,cds       ] Mark open archive regions in map:
[0x00000000ffe00000, 0x00000000ffe47ff8]
[0.294s][info][gc           ] Periodic GC disabled
[2.085s][info][gc,start     ] GC(0) Pause Young (Normal) (G1 Evacuation
Pause)
[2.089s][info][gc,task      ] GC(0) Using 2 workers of 2 for evacuation
[2.150s][info][gc,phases    ] GC(0)   Pre Evacuate Collection Set:
0.0ms
[2.152s][info][gc,phases    ] GC(0)   Evacuate Collection Set: 58.7ms
[2.153s][info][gc,phases    ] GC(0)   Post Evacuate Collection Set:
0.9ms
[2.155s][info][gc,phases    ] GC(0)   Other: 5.7ms
[2.157s][info][gc,heap      ] GC(0) Eden regions: 7->0(18)
[2.158s][info][gc,heap      ] GC(0) Survivor regions: 0->1(1)
[2.159s][info][gc,heap      ] GC(0) Old regions: 0->0
[2.160s][info][gc,heap      ] GC(0) Archive regions: 2->2
[2.161s][info][gc,heap      ] GC(0) Humongous regions: 0->0
[2.162s][info][gc,metaspace ] GC(0) Metaspace: 1331K->1331K(1056768K)
[2.164s][info][gc           ] GC(0) Pause Young (Normal) (G1 Evacuation
Pause) 7M->1M(34M) 78.733ms
[2.165s][info][gc,cpu       ] GC(0) User=0.01s Sys=0.01s Real=0.08s
[6.820s][info][gc,start     ] GC(1) Pause Young (Normal) (G1 Evacuation
Pause)
[6.822s][info][gc,task      ] GC(1) Using 2 workers of 2 for evacuation
```

A.0.3 Shenandoah GC

```
[0.131s][info][gc] Consider -XX:+ClassUnloadingWithConcurrentMark if
large pause times are observed on class-unloading sensitive workloads
[0.298s][info][gc,init] Regions: 1990 x 256K
[0.315s][info][gc,init] Humongous object threshold: 256K
[0.321s][info][gc,init] Max TLAB size: 256K
[0.333s][info][gc,init] GC threads: 1 parallel, 1 concurrent
[0.336s][info][gc,init] Reference processing: parallel
[0.343s][info][gc     ] Heuristics ergonomically sets
-XX:+ExplicitGCInvokesConcurrent
[0.348s][info][gc     ] Heuristics ergonomically sets
-XX:+ShenandoahImplicitGCInvokesConcurrent
[0.360s][info][gc,init] Shenandoah heuristics: adaptive
[0.415s][info][gc,ergo] Pacer for Idle. Initial: 9M, Alloc Tax Rate:
1.0x
[0.429s][info][gc,init] Initialize Shenandoah heap: 32000K initial,
6656K min, 497M max
[0.431s][info][gc,init] Safepointing mechanism: thread-local poll
[0.433s][info][gc     ] Using Shenandoah
[0.437s][info][gc,heap,coops] Heap address: 0x00000000e0e40000, size:
497 MB, Compressed Oops mode: 32-bit
[27.842s][info][gc           ] Trigger: Learning 1 of 5. Free (348M) is
below initial threshold (348M)
[27.846s][info][gc,ergo      ] Free: 348M (1393 regions), Max regular:
256K, Max humongous: 356352K, External frag: 1%, Internal frag: 0%
[27.848s][info][gc,ergo      ] Evacuation Reserve: 25M (100 regions),
Max regular: 256K
[27.850s][info][gc,start     ] GC(0) Concurrent reset
[27.852s][info][gc,task      ] GC(0) Using 1 of 1 workers for
concurrent reset
[27.856s][info][gc           ] GC(0) Concurrent reset 124M->124M(124M)
5.690ms
```

A.0.4 ZGC

```
[0.087s][info][gc,init] Initializing The Z Garbage Collector
[0.089s][info][gc,init] Version: 13.0.2+8 (release)
[0.090s][info][gc,init] NUMA Support: Disabled
[0.091s][info][gc,init] CPUs: 2 total, 2 available
[0.092s][info][gc,init] Memory: 1990M
[0.093s][info][gc,init] Large Page Support: Disabled
[0.094s][info][gc,init] Workers: 2 parallel, 1 concurrent
[0.097s][info][gc,init] Address Space: 0x0000040000000000 -
0x0000140000000000 (16T)
[0.098s][info][gc,init] Heap backed by file: /memfd:java_heap
[0.100s][info][gc,init] Min Capacity: 8M
[0.101s][info][gc,init] Initial Capacity: 32M
[0.101s][info][gc,init] Max Capacity: 498M
[0.103s][info][gc,init] Max Reserve: 36M
[0.104s][info][gc,init] Pre-touch: Disabled
[0.106s][info][gc,init] Available space on backing filesystem: N/A
[0.125s][info][gc,init] Uncommit: Enabled, Delay: 300s
[0.140s][info][gc,init] Runtime Workers: 2 parallel
[0.143s][info][gc     ] Using The Z Garbage Collector
```

B GC report example

The following is an example of a report generated with the parser application of a microservice environment with 5 services using G1GC.

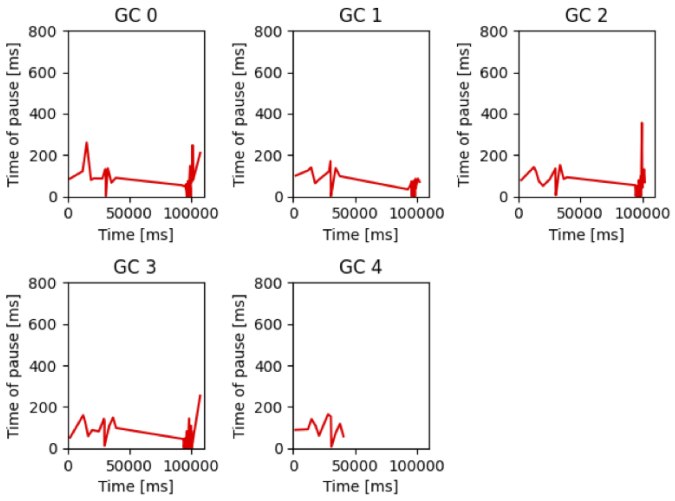
Garbage Collector Report

	GCType	Total time[s]
0	G1	107.213
1	G1	102.214
2	G1	102.008
3	G1	107.116
4	G1	40.802

Collection Phases Statistics

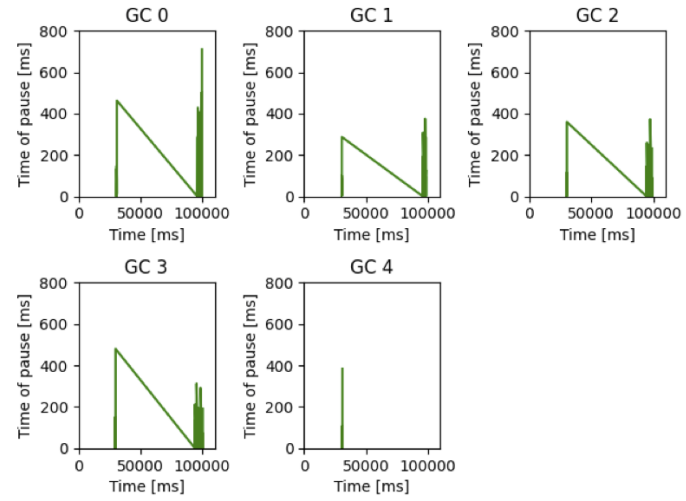
GC Pause time

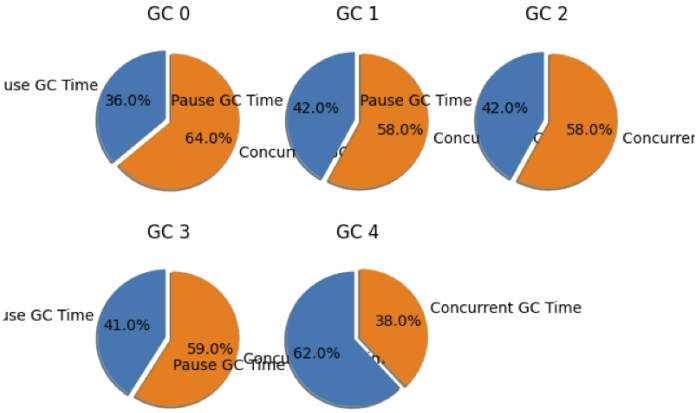
	Total [ms]	Average [ms]	Std Dev [ms]	Max [ms]	Min [ms]	Count
0	2778.55	75.10	63.31	259.94	1.55	37
1	2075.42	61.04	43.68	171.68	1.25	34
2	2479.96	65.26	63.95	356.14	1.12	38
3	2501.31	65.82	56.28	253.88	1.45	38
4	1136.25	94.69	45.19	163.78	7.85	12



GC Concurrent time

	Total [ms]	Average [ms]	Std Dev [ms]	Max [ms]	Min [ms]	Count
0	4857.66	151.80	194.36	710.35	1.23	32
1	2901.00	90.66	115.07	375.87	1.13	32
2	3365.87	84.15	104.31	372.78	1.26	40
3	3657.65	76.20	100.17	480.22	0.96	48
4	696.59	87.07	126.87	384.30	1.88	8





GC Remark time

	Total [ms]	Average [ms]	Std Dev [ms]	Max [ms]	Min [ms]	Count
0	177.52	44.38	53.78	124.85	12.78	4

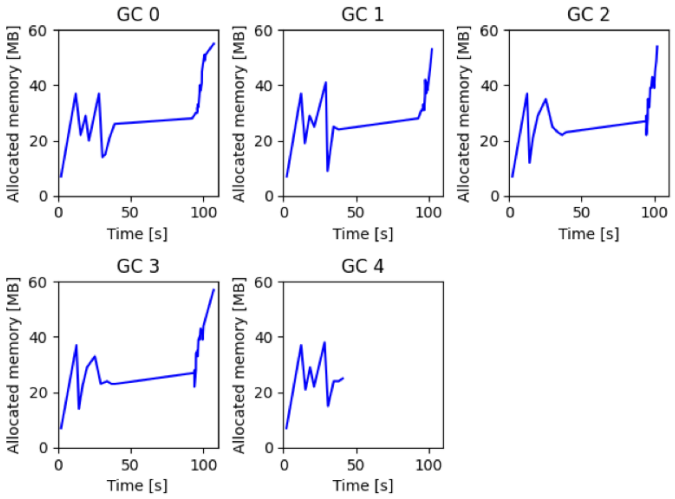
	Total [ms]	Average [ms]	Std Dev [ms]	Max [ms]	Min [ms]	Count
1	115.20	28.80	22.87	61.50	8.57	4
2	108.99	21.80	17.05	51.28	9.75	5
3	212.79	35.46	45.37	127.03	13.14	6
4	65.67	65.67	0.00	65.67	65.67	1

GC Cleanup time

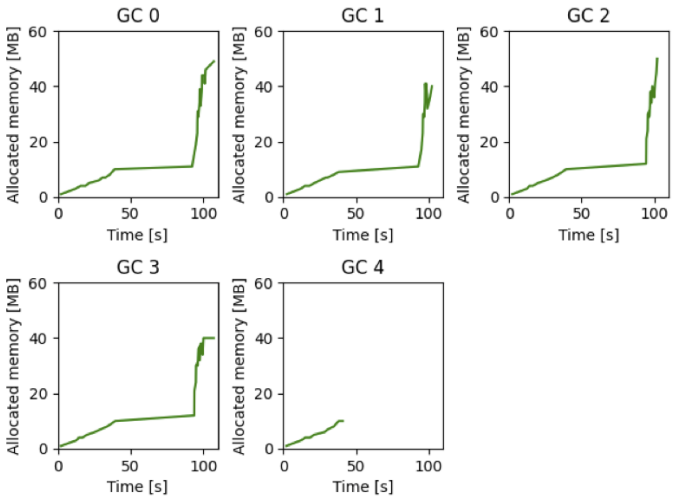
	Total [ms]	Average [ms]	Std Dev [ms]	Max [ms]	Min [ms]	Count
0	10.51	2.63	53.78	3.65	1.55	4
1	6.85	1.71	22.87	2.59	1.25	4
2	15.86	3.17	17.05	7.54	1.12	5
3	24.12	4.02	45.37	12.62	1.45	6
4	7.85	7.85	0.00	7.85	7.85	1

Graphs

Heap usage before GC



Heap usage after GC



GC Causes