Timo Keskiniemi

Measuring Performance in Go

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

2 February 2022

Abstract

Author:	Timo Keskiniemi
Title:	Measuring Performance in Go
Number of Pages:	31 pages + 1 appendix
Date:	2 February 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Katri Turunen, Technical Leader
	Roni Riska, Technical Leader
	Simo Silander, Senior Lecturer

Go is a relatively new programming language. Some of the main principles in design were high-performance networking, multiprocessing and runtime efficiency. Go is a garbage collected language.

This graduate study was done for Nokia Networks. The purpose was to study the relevant issues when measuring the performance of a program written in Go. The practical was about developing a simple measurement tool and carrying out the actual performance measurements with it. Especially the memory usage and the response times of a service were of interest. Additionally, the aim was to find out, how the garbage collection can be tuned up and what kind of consequences it has for the performance of the service.

Based on the measurements, the service does not have a memory leak and there seems to be no significant anomalies on response times. Changing the activity level of garbage collection has its pros and cons. If activity is reduced, the response times are slightly better, but memory usage is on a higher level. Corresponding to more frequent garbage collection, the service needs less memory allocated, but there is more dispersion with response times. Overall, there are no particular reason to change the default value of garbage collection.

Go has quite comprehensive tools for analyzing and profiling code. The tools can be used if there is any need to search and fix bottlenecks for better performance.

Keywords: Go, performance, garbage collection

Tiivistelmä

Tekijä:	Timo Keskiniemi
Otsikko:	Suorituskyvyn mittaaminen Gossa
Sivumäärä:	31 sivua + 1 liite
Aika:	2.2.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Tekninen johtaja Katri Turunen
	Tekninen johtaja Roni Riska
	Lehtori Simo Silander

Go on verrattain tuore ohjelmointikieli. Sen kehityksessä on pyritty panostamaan mm. rinnakkaisprosessointiin ja ajonaikaiseen tehokkuuteen. Gossa on automaattinen roskienkeräys.

Tämä insinöörityö on tehty Nokia Networksin toimeksiannosta. Tavoitteena on selvittää ohjelman suorituskykyyn ja sen mittaamiseen vaikuttavia asioita, etenkin huomioiden Gon erityispiirteet. Käytännön toteutuksena on mitattu Golla toteutetun sovelluksen suorituskykyä. Erityisesti halutaan tutkia sen muistinkulutusta ja vasteaikojen tasaisuutta. Lisäksi tavoitteena on selvittää, miten automaattiseen roskienkeräykseen voi vaikuttaa ja millaisia seurauksia sillä on sovelluksen suorituskykyyn.

Mittausten perusteella testattavassa sovelluksessa ei esiinny muistivuotoa eikä vasteajoissa ilmene merkittäviä poikkeuksia. Roskankeruun aktiivisuuden muuttamisella on sekä hyviä että huonoja vaikutuksia. Aktiivisuutta harventamalla vasteaikoihin saa lievää parannusta, mutta toisaalta muistinkäyttö kasvaa. Vastaavasti tiheämmällä roskankeruulla ohjelman tarvitsema muisti on pienempi, mutta vasteajoissa on suurempaa hajontaa. Johtopäätöksenä ei ilmennyt mitään erityistä syytä vaihtaa roskankeruun aktiivisuutta oletusarvosta.

Suorituskyvyn mittaamisen ja käsiteltävän ohjelmointikielen teoriapohjaa tarkastellessa ilmeni, että Go tarjoaa kattavat työkalut koodin analysoinnille ja profiloinnille. Näitä valmiita välineitä on mahdollista käyttää, mikäli on tarvetta selvittää tarkemmin sovelluksen suorituskyvyn pullonkauloja.

Avainsanat: Go, suorituskyky, roskankeruu

Contents

List of Abbreviations

API:	Application programming interface
CPU:	Central prosessing unit
GC:	Garbage collector
LIFO:	Last In, First Out
PID:	The identification number of the task
RSS:	Resident Set Size. The non-swapped physical memory used by the task.

1 Introduction

This graduate study was done for Nokia Networks, which is part of Nokia Corporation. Nokia Networks is a multinational data networking and telecommunications equipment company headquartered in Espoo, Finland.

As a global technology company, Nokia uses a large variety of different programming languages for various purposes. The purpose of the graduate project was to study the performance of an application code written in Go. The main focus is on memory management and the usage of garbage collection.

Go programming language, often referred to as Golang, was publicly announced in 2009 [1]. Some of the main principles in design were high-performance networking, multiprocessing and runtime efficiency [2, Chapter 1].

Concerning the application code under development there were many open questions and matters to be clarified, which were the initial motivation and premise for the study. Does Go use memory resources efficiently enough? Is there any way to control garbage collection in Go – and if there is, how it can be effectively utilized for the present needs? How well is Go suitable for a soft real-time system?

In a real-time system processing must be done within the time limits, or the system will fail. In a soft real-time system time limits are not compulsory for every task. However the more often a system is missing the time limit, the weaker the performance of the system is. [2]

One interesting and important question was to find out about the application, whether the response times of the API calls behave in a deterministic manner. The interest is thus not so much in how fast the response times are but in how stable they remain. The response times are expected to be approximately the same regardless of how long the application has been running or how many calls it has already handled. Of course it is always an objective to have an

application run as fast and efficiently as possible, but it is also essential to get its behavior to be predictable and steady.

The paper first tells about memory management and introduces related language-specific features. This is followed by providing an insight to garbage collection and how it can be tuned up in Go. The next chapter presents a tool for profiling Go code. The results of the measurements are presented after the program to be tested and the self-made testing tool have been introduced. The final chapter summarizes the study.

2 Memory Management in Go

Memory management has a significant impact to system performance. Understanding how memory is utilized helps to produce better code. This chapter first introduces the concepts of the heap and the stack. Next the Go-specific feature goroutine is explained.

2.1 Heap and Stack

A heap is a memory area which is used for dynamic memory allocation. Memory is allocated from the heap in random order. It is possible to resize elements in the heap, which makes heap allocation slower as enough memory needs to be search to hold the new data. Also access to the heap memory is time consuming, because the blocks used may not be next to each other (Figure 1). [4; 2, Chapter 2.]

A stack memory is allocated from contiguous blocks and the stack has a fixed size, which makes it faster to allocate and access. A stack-based memory allocation uses so called LIFO method. [3] This “last in, first out” principle means that the most recently allocated block is the first block to be de-allocated. That makes the stack easy and simple to implement, because only two operations are needed: push and pop [2, Chapter 2].

Stacks are usually visualized growing from the bottom up (cf. Figure 1). The next allocated memory block can only be added (push) on top of the stack. Correspondingly removing (pop) the de-allocated block has to happen starting from above.

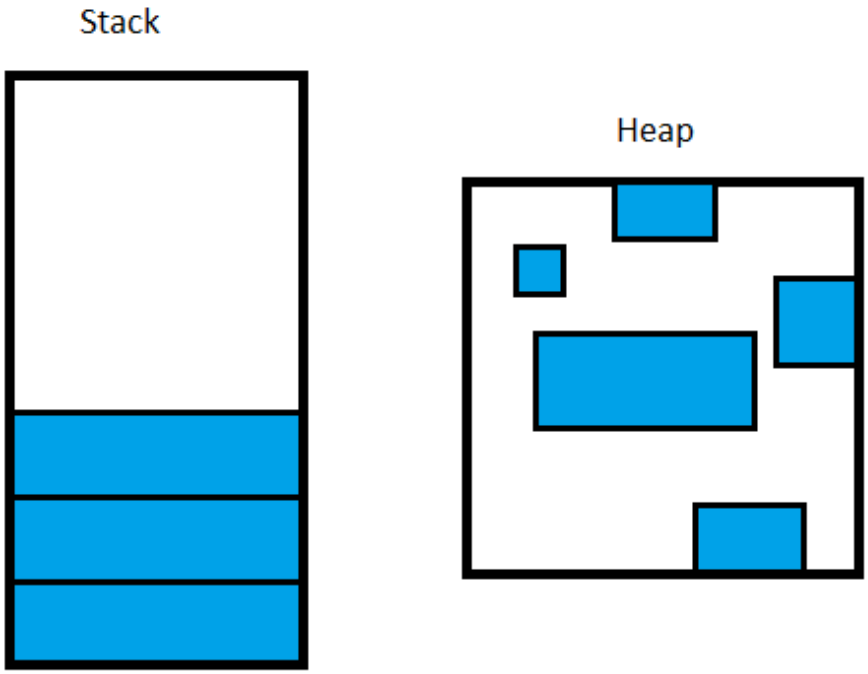

Figure 1. Stack and heap

Memory management is a vital process in Go. Features that make efficient use of memory have been developed for the language.

2.2 Goroutines

Goroutines are one of the most essential constructs in Go. A goroutine is a function or method that runs concurrently alongside other code. Every Go program has at least the main goroutine [5, Chapter 3]. The OS handles threads in other languages such as Java, but goroutines do not connect to the underlying OS.

A goroutine can be thought of as a lightweight thread. One goroutine can only have a stack size of 2 KB from the beginning, whereas threads in other languages can receive even megabytes of space. If the goroutine needs more memory, it can be allocated from another place in available memory space.

The Go scheduler manages the life cycles of the goroutines. It is able to control and even block operations as concerns to a goroutine. This helps Go to be more efficient with multiprocessing, and therefore it can be more performant in memory utilization, latency and garbage collection.

3 Garbage Collection

Garbage collection is one of the most important parts of almost every modern programming language. It has a crucial impact for the performance of the code. The purpose of garbage collection is to free up memory space that is not being used. In Go this happens in a concurrent way while a program is running.

In some languages, such as C and C++, it is the responsibility of the programmer to allocate and release memory for the objects. A typical error is to allocate memory but never release it. Then the used memory for useless objects - or amount of garbage, in other words - keeps on growing. This situation is called memory leak. Eventually the program suffers from memory leak and crashes, because there is no more memory available in the system to allocate for the new objects.

In languages such as Go and Java, the garbage collection happens automatically. This helps to avoid memory leaks, because there is no need to de-allocate memory by programmer.

The heap is the place where garbage collection takes place. The main objective is to free heap memory by deleting all those objects which are no longer needed.

3.1 Garbage Collection in Go

The functionality of the Go Garbage Collector (GC) is based on the tricolor mark-and-sweep algorithm. The GC runs as a goroutine, so it can work concurrently with the program. The algorithm visits all the accessible objects of the heap and marks them in one of these three sets: white, black or grey.

The objects of the white set have not yet been reached by the collector, and are the candidates for garbage collection. Black objects have all their fields scanned, and have no pointers to any object of the white set. An object in the gray set has one or more of its fields still need to be scanned, so it might have a pointer to object of the white set. [6, Chapter 15.1.]

All objects are white at the beginning of the garbage collection. At first, the GC visits all root objects such as global variables and other things on the stack, and marks them in gray. Then the GC starts looking if a grey object has pointers to objects of the white set. Those found children of the object under investigation will be colored grey. The original grey object is turned black. At the end of the cycle all white objects are treated as unreachable garbage. [7, Chapter 10; 8, Appendix A.]

After the algorithm has marked all the accessible objects of the heap and there are no more grey objects to examine, it sweeps the inaccessible white objects. This means that those objects are garbage collected and their memory space can be reused.

Go programs accept some parameters through the environment that can be used to tune the garbage collection.

3.2 Controlling Runtime Behavior of Go Garbage Collection

The intensity of the Go garbage collection system can be adjusted by a GOGC variable. It sets the initial garbage collection target percentage. The next GC

cycle is initiated when the heap has grown by an amount of the target percentage since the previous collection. [4]

By default, the value of GOGC is 100. If this value is set lower, it will cause the garbage collector to execute more frequently. Correspondingly setting the value larger will cause the garbage collector to execute less frequently. A value of off disables the garbage collector entirely, which might be sometimes useful for debugging. [2, Chapter 10.]

For example, there can be 4 MB of memory marked as live in the heap and GOGC has default value 100. This one hundred percent tells how much more new memory can be allocated before the next GC run starts, hence heap grows up to 8 MB. If GOGC is set to 50, heap grows to 6 MB. And if the value of GOGC is 200, the heap can grow up to 12 MB before the next GC. [5]

The number of operating system threads allocated to goroutines can be limited by a GOMAXPROCS variable. A default value for the variable is equal to the number of cores available to the application. There is no upper bound limit for this variable. It is possible to have considerable improvements when adjusting the existing number of GOMAXPROCS, if there is a parallelized and CPU-intensive function. [2, Chapter 10.]

A GOTRACEBACK variable is needed to control the generated output from a Go program when it fails due to an unexpected runtime condition or an unrecovered panic state. Setting this variable will allow to get more specific information about the goroutines that are involved to error or panic. [2, Chapter 10.]

4 Profiling Go Code

To understand the CPU and memory utilization within a program is a key factor to write faster and better code. Profiling is a process that can be used to

measure the resources utilized in a computer system. Profiling gives a better understanding of the program behavior.

There are many good reasons to profile code. In new releases of software, it is possible to check how performance has changed. Thus, it is easier to immediately react to problems with latency or how much CPU is being utilized for each task. Memory profiling can reveal incorrect usage of global variables, defective goroutines or large string allocations.

Go has a powerful inbuilt tool for profiling data. The tool called pprof is useful for analyzing and improving the memory usage of a Go application. [11, Chapter 14.]

The pprof tool can be used to collect and explore CPU profiles with stack sampling, heap profiles using allocation profiling, goroutines and their stacks, and more [6]. With pprof it is also easy to have a visualization from the analyzed data.

Figure 2 shows an example of comprehensive information that pprof provides. From a list view it is easy to find functions which consume the most system resources.

The meaning of the “flat” columns are that they give information about how much the function have allocated and held memory. The cumulative part tells the memory allocation by the function with all other functions that it has called.

Flat	Flat%	Sum%	Cum	Cum%	Name
190ms	44.19%	44.19%	190ms	44.19%	runtime.stdcall1
130ms	30.23%	74.42%	130ms	30.23%	runtime.stdcall3
20ms	4.65%	79.07%	20ms	4.65%	runtime.stdcall6
20ms	4.65%	83.72%	20ms	4.65%	runtime.(*mcache).prepareForSweep
10ms	2.33%	86.05%	10ms	2.33%	runtime.stdcall2
10ms	2.33%	88.37%	210ms	48.84%	runtime.startm
10ms	2.33%	90.70%	20ms	4.65%	runtime.semasleep
10ms	2.33%	93.02%	10ms	2.33%	runtime.pidleget
10ms	2.33%	95.35%	10ms	2.33%	runtime.doaddtimer
10ms	2.33%	97.67%	10ms	2.33%	runtime.(*profBuf).read
10ms	2.33%	100.00%	10ms	2.33%	runtime.(*addrRanges).removeLast

Figure 2. Example of function list view

In addition to the list view, there is also a graphical view (cf. Figure 3). It gives valuable information about the relations between functions.

A red box and a thick arrow indicate that the function has reserved more cpu resources. The percent of the occupied cpu resources at the given time are also presented.

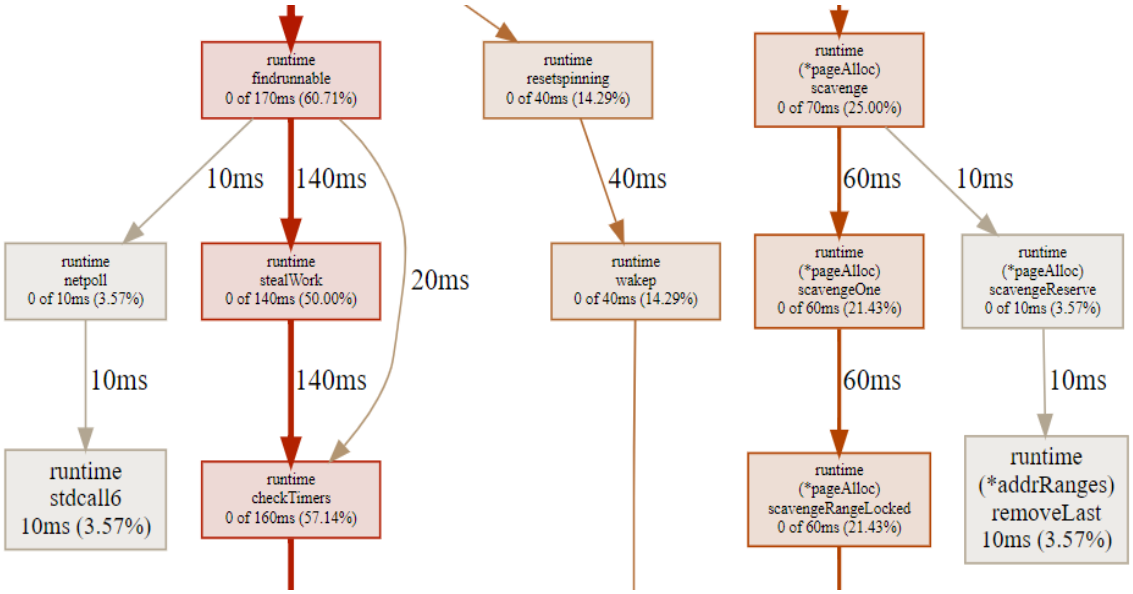

Figure 3. Part of the graphical view

Flame graphs (cf. Figure 4) can be used to visualize stack traces of profiled software [7]. The width of each segment is the CPU-time for which the function is responsible. The functions are organized vertically based on their position in the call-stack. The width of the each box represents the total amount of CPU time resources the function used. It is good to know that the colour of each segment is totally random, so there is no explanation related to performance why one segment is yellow, orange or red. [8]

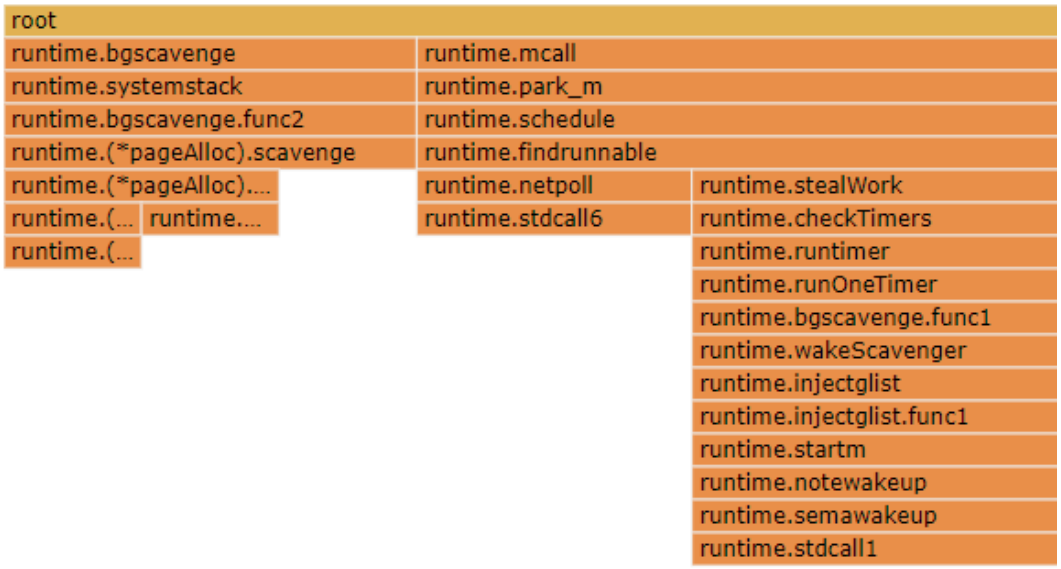

Figure 4. Flame graph view

There are multiple ways to get started with profiling: by using the -cpuprofile and -memprofile flags through tests and benchmarks, by importing “net/http/pprof” to add /debug/pprof endpoints in the service, or by starting the profile from code by calling runtime.StartCPUProfile or runtime.WriteHeapProfile.

Code tracing is a process that gives essential information related to the run-time behavior of the program: the lifetime of goroutines, the usage of the garbage collector, the number of operating system threads, the activity level of each logical processor, and so on.

The runtime/trace package can be used to collect tracing data. Tracing helps to discover performance improvements by finding hidden bottlenecks and analyzing latency of the application.

5 Measurements

This chapter first introduces the program that was tested. Then the self-made testing program is comprehensively described and examples of its code are explained. Finally, the test setups are defined in terms of memory usage, response times and code profiling.

In practical testing, the focus is on measuring the response times of the application and the usage of memory. Especially it is valuable to find out how the garbage collection affects to those factors. Changing the activity level of the GC may have a significant effect to the performance of the application. But how prominent the effect actually is, can be solved out only by measuring it.

5.1 Program Tested

The program, whose performance was measured, is a service called Argus. It is mainly written in Go. The service provides identity and access management solution in a container-based environment. The container can be deployed on a Kubernetes cluster using the Helm package manager. Argus provides a RESTful interface to communicate between the service and the outside world.

Normally, the usage of CPU is limited to ten percent for Argus. For the testing purpose this limitation is, however, overpassed. This is because the aim was to concentrate the measurements to detect performance changes caused by garbage collection. Therefore, the possibility for lack of CPU resources was minimized.

5.2 Testing Program

For measurements a configurable test program was needed. The purpose was to easily set values for certain variables such as the number of clients and the number of calls which each of client should do. In this manner it is simple to run test sets with different configurations.

The program, written in Go, was created by the author. The idea is that the testing program can be used later with other projects and measurements. For the present study it was enough to make a simple prototype, which performs the well defined task. The program is easy to be modified and extended to other uses when needed in the future.

After completing the measurements, the test program prints on the screen the most essential numbers e.g. mean values and the total execution time of the test set. The test program collects measurement data and saves it to a text file. Therefore, if the user wants, it is possible to analyse the data by using office productivity software such as Excel. In this manner it is quite straightforward to visualize numerical data and calculate various interesting statistical information, if needed.

A command line interface was used to run the testing tool, because there was no real need for graphical user interface. For simplicity, there are some hardcoded sections in code to minimize mandatory options and parameters to run the program.

For example, it is assumed that in a database of the service there already exists one particular test user, which is created via an initial configuration file when the service is started. The username and password of the test user are hardcoded to the source code of the testing tool. Otherwise all the information should be given every time when the tool is used to run test sets.

Of course hardcoding - especially hardcoding passwords - is a bad coding practice and generally it should be avoided as much as possible. This time a

violation of that important principle was made purposefully to take some shortcuts to keep development and using of the testing tool as simple as possible.

To run response time tests a user has to define

- a number of clients
- a number of requests per client
- a cluster IP address of a Kubernetes pod.

Listing 1 shows an example of a command that sets two clients to send ten requests each. The cluster IP address of a Kubernetes pod is needed to reach the container, which holds the running Argus service.

```
$ go run . -c 2 -r 10 -a 10.40.0.101
```

Listing 1. An example of a command to run response time tests

To run memory usage tests a user has to define

- a number of clients
- an execution time in minutes
- a name of the process
- a cluster IP address of a Kubernetes pod.

In Listing 2 one client is set to send requests for an hour and a process called argus is set to be monitored. When running a memory usage test, there is no need to define a number of requests per client. The duration of the measurement is given as an execution time in minutes and it depends on how long it is needed to measure memory usage. A client or clients keep sending requests throughout the measurement.

```
$ go run . -c 1 -t 60 -p argus -a 10.40.0.101
```

Listing 2. An example of a command to run memory usage tests

From Listing 3 one can get an overall picture of what the testing tool actually does during measuring response times. The first task is to start as many sessions as the user has defined the number of clients. Technically all the sessions are created by using just a one test user. With “defer” keyword in the code, it is possible to set something to happen just before exiting the function. It is used to delete all sessions lastly.

Clients are started as goroutines. That is because they need to run in parallel. Each client has its own individual session ID, which is needed to send requests to patch each session information. Actually the durations of these patch operations are the ones the tool is measuring.

After clients have finished their job, all results are delivered through channels. Then program calculates an average response time from all the results and prints it on the screen. Finally, the results are written in a text file.

```
func runResponseTimeTest(clients int, requests int) {
	fmt.Println("Measuring response times...")

	createFile(respfile)

	sessionIds := startSessions(clients)

	defer deleteSessions(sessionIds)

	ch := make([]chan []Measurement, clients)
	for i := 0; i < clients; i++ {
		ch[i] = make(chan []Measurement)
	}

	startTime = time.Now()

	for i := 1; i < clients + 1; i++ {
		wg.Add(1)
		go runRespClient(i, requests, sessionIds[i-1], ch[i-1])
	}

	var results [][]Measurement

	for i := 0; i < clients; i++ {
		result := <-ch[i]
		results = append(results, result)
	}

	wg.Wait()

	average := calculateAverageResponseTime(results, int64(clients*requests))
	fmt.Println("Average response time: " + fmt.Sprint(average))

	for i := 0; i < clients; i++ {
		writeToRespFile(results[i])
	}
}
```

Listing 3. A main function for measuring response times

The results are saved to the text file by using an easy syntax. Listing 4 shows a data structure, which is used in an every individual measurement. It saves which client has sent the request, which request it was, time elapsed so far, and the response time of that request. All this information is saved to the text file for later use.

```
type Measurement struct {
	Client int
	Request int
	Timestamp float64
	ResponseTime int64
}
```

Listing 4. Defined data structure for measurement

In Listing 5 one can see a behaviour of the client. Basically it just sends a request and marks the results. And then it repeats the same procedure as long as there are requests left. At the end results of the measurements are sent through a channel back to the function, which originally started the client.

```
func runRespClient(client int, requests int, sessionId string, ch chan
[]Measurement) {
      defer wg.Done()
      var measurements []Measurement
      var measurement Measurement
      for i := 0; i < requests; i++ {
            respTime, timestamp := request(sessionId)
            measurement.Client = client
            measurement.Request = i + 1
            measurement.Timestamp = timestamp.Seconds()
            measurement.ResponseTime = respTime.Microseconds()
            measurements = append(measurements, measurement)
            if client == 1 {
                  runCounter(i + 1, requests)
            }
      }
      if client == 1 {
            fmt.Println("")
      }
      ch <- measurements
}
```

Listing 5. Running client for response time measurements

Of course one of the most crucial parts in the testing tool is the moment when the response time is really measured. The exact moment is introduced in Listing 6.

At first a starting time is saved just before a request. Then the request is sent and response received. Immediately after that the next timestamp is taken. The response time can now be calculated as a difference between the two times.

```
start := time.Now()
resp, err := client.Do(req)
stop := time.Now()
respTime := stop.Sub(start)
```

Listing 6. Measuring response time

Listing 7 shows that running memory test is a slightly different compared to response time tests. Starting and deleting sessions works just like before and clients run as goroutines.

For monitoring the usage of memory, the pidstat command is used. The user has to give the name of the process. It is possible to get process identification number by using the name. With that PID the actual monitoring can be started.

The output of the pidstat command is used as a result. And once again, the results are written in text file for the later use.

```
func runMemoryTest(clients int, exectime int, pname string) {
	fmt.Println("Running memory test...")

	createFile(memfile)

	sessionIds := startSessions(clients)

	defer deleteSessions(sessionIds)

	pid := getPid(pname)

	startTime = time.Now()

	for i := 1; i < clients + 1; i++ {
		wg.Add(1)
		go runMemClient(i, exectime, sessionIds[i-1])
	}

	go runTimer(exectime)

	results := getResults(pid, exectime)

	wg.Wait()

	writeToMemFile(results)

	fmt.Println("Finished.")
}
```

Listing 7. A main function for memory usage tests

Before the tool can start measuring, it needs a process identification number. It can be found by using pidstat command by using the name of the process (cf. Listing 8).

```
func getPid(pname string) string {
	cmd := exec.Command("pidstat", "-C", pname)

	stdout, err := cmd.StdoutPipe()
	if err != nil {
		log.Fatal(err)
	}

	if err := cmd.Start(); err != nil {
		log.Fatal(err)
	}

	data, err := ioutil.ReadAll(stdout)
	if err != nil {
		log.Fatal(err)
	}

	if err := cmd.Wait(); err != nil {
		log.Fatal(err)
	}

	v := strings.Fields(string(data))
	pid := v[len(v)-8]

	return pid
}
```

Listing 8. Getting process identification number

The actual measurement is started in function, which is introduced in Listing 9. The PID, which was accessed earlier, is now used to get information from the right process. It is hardcoded that pidstat takes a sample every 60 seconds. The user has given the execution time in minutes. When the time is up, results are returned as a string.

The pidstat command is useful for the purpose here, because it can monitor any individual process, which is managed by the Linux kernel. Now it is deployed to give information about Argus process every minute. The pidstat command provides several information about the process, but what is of interest is a value of RSS, resident set size. It is the non-swapped physical memory used by the process. The output of the command is generated line by line, minute by minute, so it is easy to handle and save to a text file.

```
func getResults(pid string, exectime int) string {
	cmd := exec.Command("pidstat", "-r", "-p", pid, "60",
fmt.Sprint(exectime))

	stdout, err := cmd.StdoutPipe()
	if err != nil {
		log.Fatal(err)
	}

	if err := cmd.Start(); err != nil {
		log.Fatal(err)
	}

	data, err := ioutil.ReadAll(stdout)
	if err != nil {
		log.Fatal(err)
	}

	if err := cmd.Wait(); err != nil {
		log.Fatal(err)
	}

	return string(data)
}
```

Listing 9. Setting pidstat command to gather results

A behaviour of the client is very straightforward (Listing 10). It just keeps sending requests as long as there is time left. When running memory usage tests, there is no need to measure response times.

```
func runMemClient(client int, exectime int, sessionId string) {
	defer wg.Done()
	for {
		_, timestamp := request(sessionId)
		if timestamp.Minutes() > float64(exectime) {
			break
		}
	}
}
```

Listing 10. Running client for memory usage tests

In summary, the testing program is developed to cope with a very well-defined task and it is easy to use for it. However, the program has been designed with its effortless further development in mind.

5.3 Response Times

To measure the response times of the application one would want to have several clients to make a high number of calls to the application under testing.

The clients should run in a parallel fashion to simulate a real situation with a heavy usage. There can be, for example, ten clients and each of them does ten thousand patch request via API. A time between each request and response are measured and saved to the file.

The results can be used to calculate what is the average response time of that certain API call. A minimum and maximum values of the measured times are valuable information to detect whether there occur some odd hiccups in the performance. It is also interesting to know the total execution time of the test set.

The aim is to find out what kind of role the garbage collection plays in terms of response times. That is why the same tests have to be run with different activity levels of GC. The intensity of garbage collection can be set with the GOGC variable. The default value is 100, which means the activity level of one hundred percent.

The test sets can be executed with the different values of GOGC, e.g. 50 and 150, to see how response times varies from the default test setup. It is a reasonable assumption that GC does have some effect for response times. But if there are no significant difference, it is a valuable result too.

5.4 Memory Usage

One of the main concerns is whether the memory usage keeps on growing when the application is running. This kind of behavior is indicating that there might be a memory leak, which should be fixed before letting the production code to be published.

To avoid such problems one would want to measure memory usage in a longer period of time than in the previous test sets. There is a need for a several clients, which will keep the application busy, just like before. This time the total

execution time for test is set. It can be one hour or more, whatever seems to be enough for the useful results.

When a test set is running, the test program records every minute the usage level of memory. After the clients stop making requests, it is good to make sure that memory originally allocated to the application is de-allocated back to the OS.

The pidstat command is used for collecting information about the task. The user must give a name of the task as one of the initial parameters. Using the task name, the test program will get a PID, the identification number of the task being monitored. With this process id the actual gathering of information can be done.

The pidstat can be set to report memory utilization at appropriate intervals. The most interesting value is RSS, resident set size, which is the non-swapped physical memory used by the task in kilobytes. The value of RSS is recorded every minute as long as the measurement lasts.

5.5 Profiling

The pprof tool can be used to profile the production code. The main purpose of the profiling is to find out which is a portion that the garbage collection needs from the whole memory usage or response times.

For profiling, the test setup is fairly similar to what it was when measuring response times. Yet again, with different values of GOGC it is possible to get information about how the performance of the application varies when the activity level of the garbage collection is changed.

Based on the results from the previous measurements and profiling a fairly comprehensive overview of the performance of the application can be formed.

Due to limitations of resources regarding the graduate study, code profiling was not actually performed. Outlining the process what could be done was, however, useful for potential future additional research.

6 Results

The measurements made can be divided into two categories: response times and memory usage. In both sections there was an interest to find whether garbage collection affects performance or not. The activity of garbage collection was altered with different values of GOGC: 0, 50, 100, 150 and 200.

First, the results of the response time measurements are presented. The response times were measured with a one client and two clients. Finally, the results related to memory usage are reported.

6.1 Response Times

Response times were measured by using one and two clients. In both cases every test was run five times with 6000 requests. Then those five test run were combined and actual results calculated by using them all. Thus, there were 30 thousand individual measurement results for every test scenario.

6.1.1 Response Times with One Client

It is the easiest to visualize the effects of changing the GOGC value when the measurements are done with just one client. Garbage collection causes delays in response times, which can be seen as peaks in Figure 5 below.

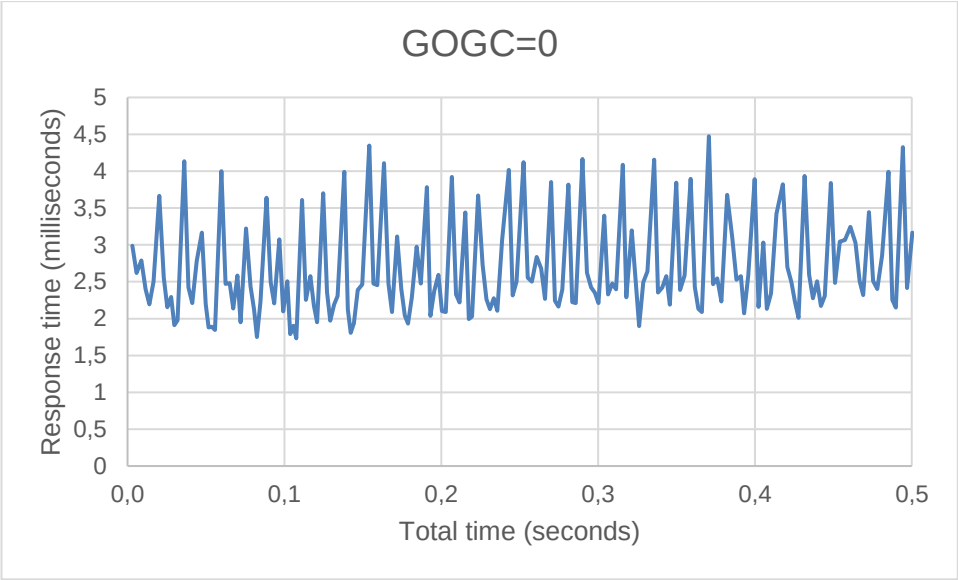

Figure 5. Response times from the first half second when the value of GOGC is 0.

With the default value of GOGC, the peaks are clearly rarer as can be seen in Figure 6.

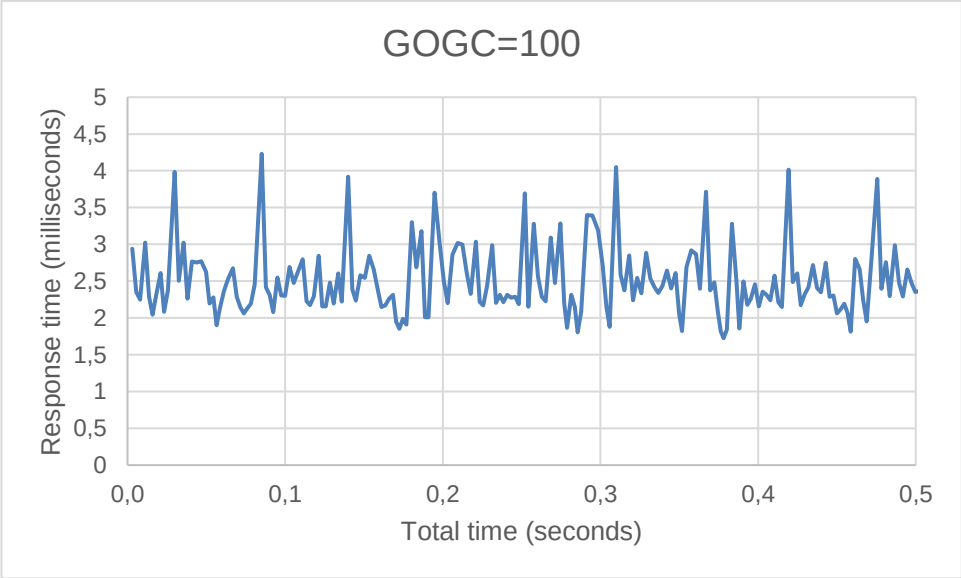

Figure 6. Response times from the first half second when the value of GOGC is 100.

Figure 7 shows that if the value of GOGC is set to 200, there are definitely less interference caused by garbage collection. The difference between values 100 and 200 is maybe not as significant as between 0 and 100, but the change is still visible.

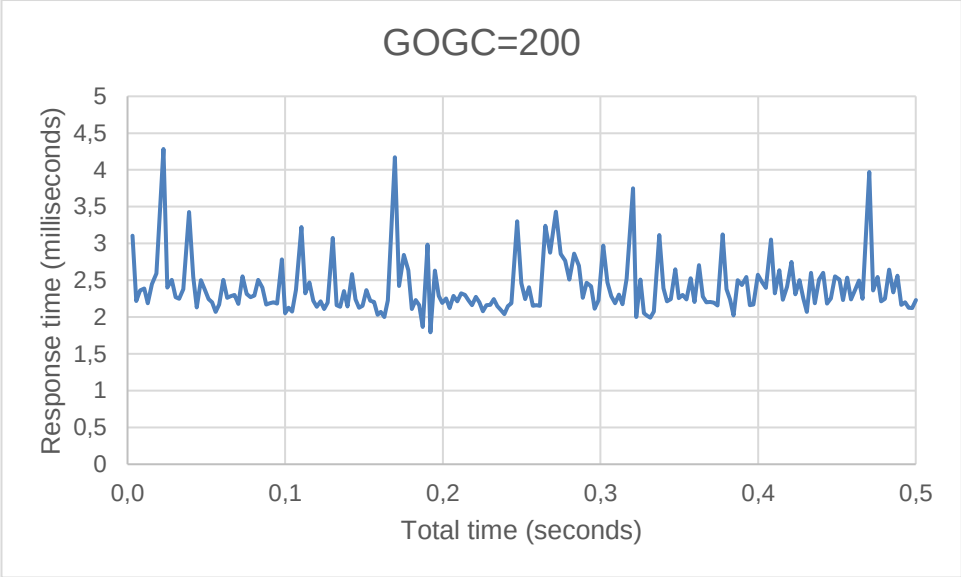

Figure 7. Response times from the first half second when the value of GOGC is 200.

With one client the service is able to handle all the requests in a good order and fast. Next, the results measured with two clients are presented.

6.1.2 Response Times with Two Clients

The idea of using two clients is to keep the program to be measured in a constant hurry, with both clients sending requests all the time and the program trying to send responses as soon as possible.

With a variety of different statistical numbers it is possible to evaluate the performance of the program. The numbers are gathered in Table 1.

Table 1. Statistics for different values of GOGC

GOGC	0	50	100	150	200
Arithmetic mean	8878	8797	7641	7479	7426
Median	9034	8956	7602	7489	7460
Minimum	3164	3235	3113	3057	3120
Maximum	22698	27885	21647	17596	17752
Standard deviation	1125	1163	1144	1053	1021

The different values are examined in more detail below. The values for the arithmetic mean and median in Figure 8 are first presented.

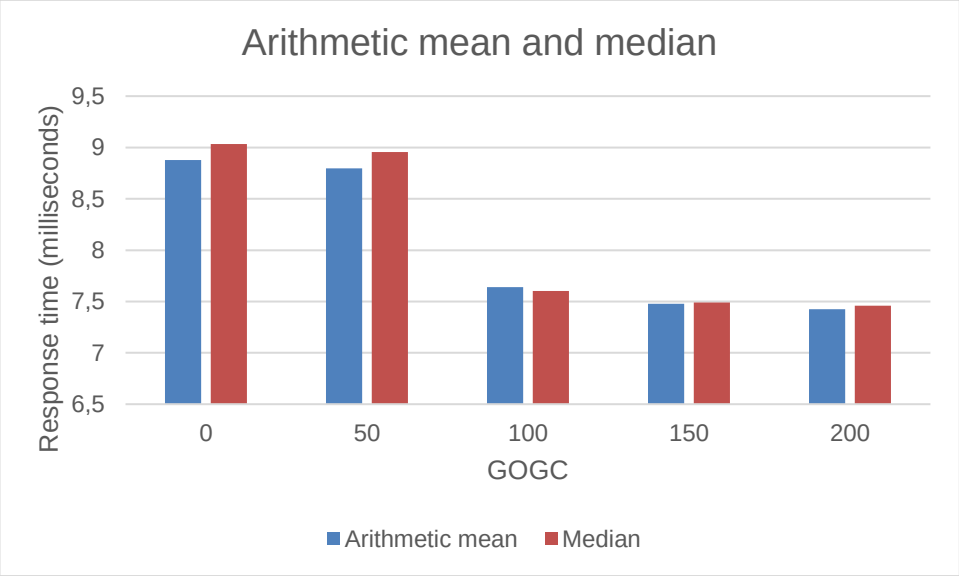

Figure 8. Two average values for response times: arithmetic mean and median.

The arithmetic mean and median were calculated to get two different average values (cf. Figure 8). It was good to see that both values were close to each other. If they were remarkably different, it would mean that the data are far from being normally distributed. That kind of scattering with values would indicate some kind of undeterministic behaviour in the system.

Figure 9 presents the minimum and maximum values. In practice, the minimum value means the fastest response time and the maximum value the slowest.

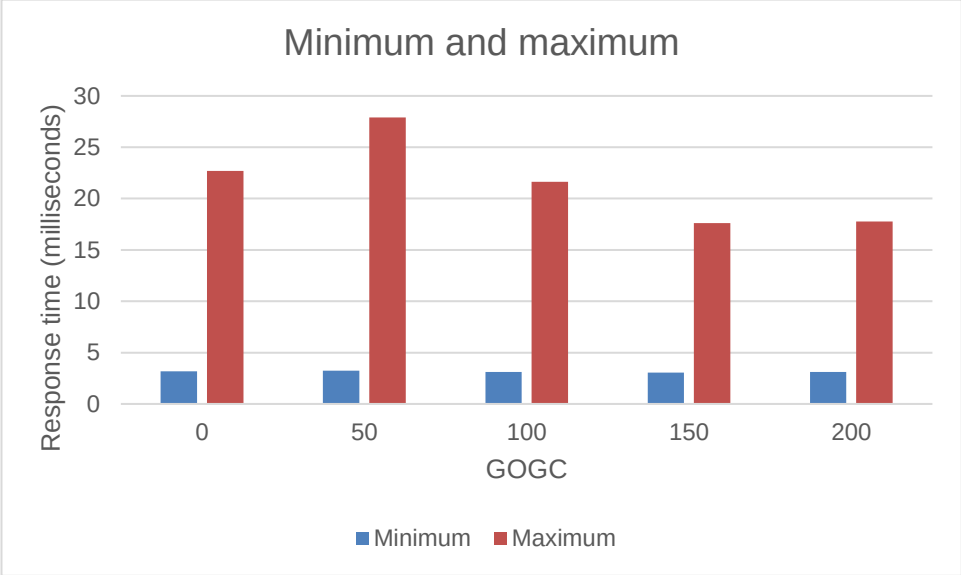

Figure 9. Minimum and maximum values for response times.

In Figure 9 one can see the minimum and maximum values for response times. It was highly expected that all the minimum values are fairly similar regardless of the value of GOGC. Basically, it gives the fastest possible response time, which is about three milliseconds in the case at hand.

The maximum values are the slowest response times. It was definitely a good result that the worst response time was under 28 milliseconds. There were 150 thousand individual requests and measured response times - and none of them was slower than that.

It seems that higher values of GOGC gives better results when looking at the maximum values of response times.

The standard deviation (cf. Figure 10) is an interesting statistical indicator. It is a measure of the amount of variation of a set of values. For response times, the lower the standard deviation, the better. It would mean that the measured

values are close to each other. In other words, the response times would behave in a deterministic way in that case.

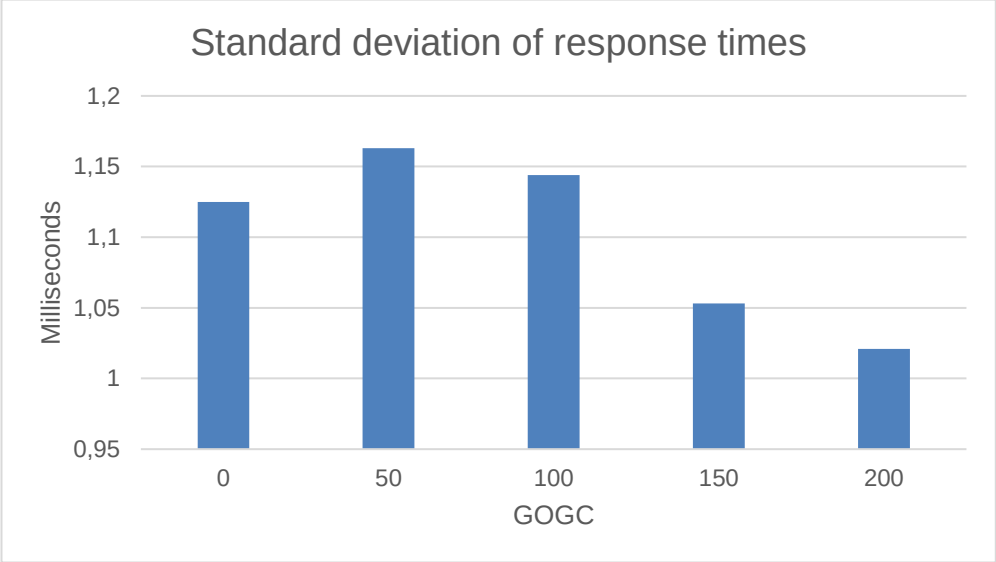

Figure 10. The standard deviations of the response times.

As can be seen in Figure 10, setting the value of GOGC higher, the standard deviation gets lower. It is interesting that GOGC value 50 gives the highest value of the standard deviations i.e. it is the worst one.

However, the best results are achieved with a high value of GOGC. The explanation is most likely that when GC is running less frequently, it interferes less the responsing of the program. Hence there are fewer larger numbers (or “spikes” seen earlier) among the response times.

Overall, the desired outcome is that response times behave deterministically. The predictability of response times is easy to determine by looking at Figure 11.

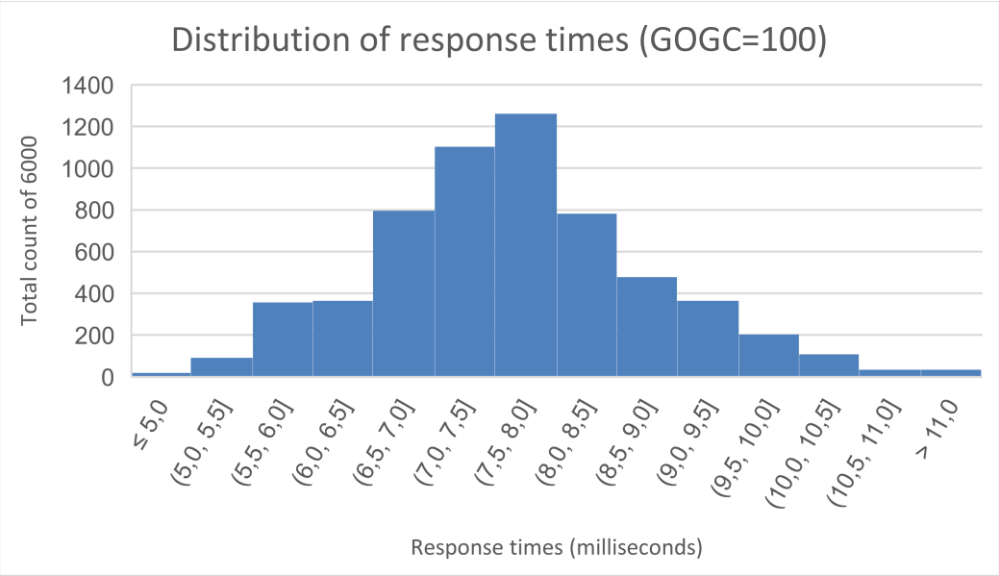

Figure 11. Distribution of response times with the default value of GOGC.

In a perfect world the response times would all be the same. In reality there is always some variety within them. Figure 11 shows that the distribution of response times forms actually a quite harmonious looking graph. The results are accumulated around the mean. The vast majority of the values are within a few milliseconds. In practice, such small differences are irrelevant.

It would have indicated some kind of problems, if the distribution had scattered to a wider area or there were more than one accumulation points.

6.2 Memory Usage

Memory usage measurements were performed with different values of GOGC. The usage level of memory was recorded every minute. Figure 12 shows the results for the first five hours.

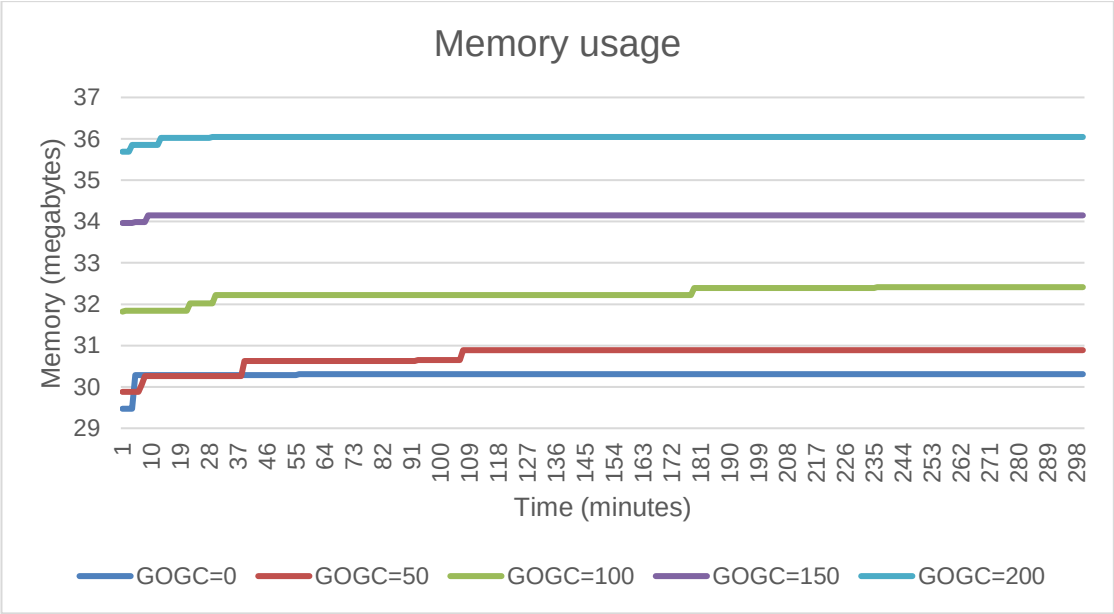

Figure 12. Memory usage for the first five hours.

Figure 12 shows that increasing the value of GOGC results in higher memory usage. The observation makes sense: there is more need for memory to handle all useless objects, when garbage collection is utilized less frequently.

Based on the measurements the most important result is that there are no memory leaks. Figure 12 shows only the first five hours, but when the measurements were continued for a longer time, the lines still remained at the same levels. If the memory usage had continued to rise, the reason for it should have been investigated and fixed.

Table 2 contains an essential comparison of garbage collection with different activity levels. The results were clear and reasonable.

Table 2. Memory usage with different values of GOGC.

GOGC	**0**	**50**	**100**	**150**	**200**
Memory (MB)	30,3	30,9	32,4	34,1	36,0
Difference (MB)	-2,1	-1,5	0,0	1,7	3,6
Difference (%)	-6,5	-4,7	0,0	5,4	11,2

The magnitude of difference in memory usage between different values of GOGC can be seen in Table 2. Memory usages are measured when they reach their highest level. Naturally, the basic level for comparing differences has been chosen to be in the default value of GOGC. Hence, all other values are compared to it.

When the garbage collection is set to be as active as possible i.e. the value of GOGC is zero, memory usage has been reduced by two megabytes, which means about seven percent reduction. When the activity level of GC is set to be less frequent, the need for memory increases. With GOGC value 200, addition for memory usage is almost four megabytes, which corresponds with eleven percent higher need for memory.

7 Conclusions

Based on the measurements made, the performance of the service is acceptable. There seems to be no memory leaks and response times are reasonable with no significant anomalies.

Overall the program written in Go is well suitable for use for a soft real-time system. The performance is efficient and reliable.

The activity of garbage collection can be adjusted with the variable GOGC. It turned out that reducing its value, the need for memory was also reduced. That is because of the accelerated GC, useless objects are cleaned more often.

Therefore those objects will not have that much time to accumulate and reserve memory space for nothing than when GC is executed less frequently.

On the other hand, response times were slighty better and steadier with higher values of GOGC. This can be explained by the fact that GC is interfering less the intended work of the program. Compared to default value, the gain is not, however, very significant.

As stated, there are some advantages and disadvantages if the value of GOGC is changed one way or another. As a conclusion, there is no particular reason to change the activity level of garbage collection from the default value.

References

1 J. Kincaid, "TechCrunch+," Verizon Media, 10 November 2009. [Online]. Available: https://techcrunch.com/2009/11/10/google-go-language/. [Accessed 21 October 2021].

2 "Javatpoint," [Online]. Available: https://www.javatpoint.com/hard-and-soft-real-time-operating-system. [Accessed 16 December 2021].

3 M. Martin, "Guru99," 7 October 2021. [Online]. Available: https://www.guru99.com/stack-vs-heap.html. [Accessed 25 October 2021].

4 D. Cheney, "The acme of foolishness," 5 January 2021. [Online]. Available: https://dave.cheney.net/tag/gogc. [Accessed 7 December 2021].

5 N. Soni, "Medium," 27 January 2021. [Online]. Available: https://medium.com/@n.nikhil.ns65/go-garbage-collector-how-does-it-work-f030cc961c46. [Accessed 14 December 2021].

6 A. Josie, "Medium," 28 October 2020. [Online]. Available: https://medium.com/swlh/go-profile-your-code-like-a-master-1505be38fdba. [Accessed 16 December 2021].

7 B. Gregg, "Brendan Gregg's Homepage," 31 October 2020. [Online]. Available: https://www.brendangregg.com/flamegraphs.html. [Accessed 29 October 2021].

8 B. Ryan, "brendanjryan.com," 28 February 2018. [Online]. Available: https://brendanjryan.com/2018/02/28/profiling-go-applications.html. [Accessed 29 October 2021].

9 M. Tsoukalos, Mastering Go - Third Edition, Packt Publishing, 2021.

10 A. Torres, Go Programming Cookbook - Second Edition, Packt Publishing, 2019.

11 B. Strecansky, Hands-On High Performance with Go, Packt Publishing, 2020.

12 B. Kommadi, Learn Data Structures and Algorithms with Golang, Packt Publishing, 2019.

13 R. Jones, A. Hosking and E. Moss, The Garbage Collection Handbook, Chapman and Hall/CRC, 2016.

14 K. Cox-Buday, Concurrency in Go, O'Reilly Media, Inc., 2017.

Examples of the measurement results

Listing 1 shows some results from a response time test. There are a client, a request, a total execution time in seconds and a response time in microseconds.

```
Client  Request  Total          Response
1       1        0.002545336    2467
1       2        0.005278193    2548
1       3        0.007725071    2118
1       4        0.010490878    2595
1       5        0.012963788    2274
1       6        0.015425537    2088
1       7        0.018045492    2551
1       8        0.020302292    2143
1       9        0.022652963    2263
1       10       0.024969174    2192
```

Listing 1. Response time test – the output of the first ten requests (GOGC=100)

The pidstat command gives information about process called argus (listing 2). The relevant column is RSS, which is the memory in kilobytes reserved by the process.

```
Linux 5.11.0-40-generic (keskinie-HP-EliteBook-725-G2)      07.12.2021
     _x86_64_   (4 CPU)

13.02.38     UID       PID  minflt/s  majflt/s     VSZ     RSS   %MEM  Command
13.03.38     999    599568      4,17      0,00 1615036   29472   0,19  argus
13.04.38     999    599568      0,43      0,00 1615036   29472   0,19  argus
13.05.38     999    599568      0,18      0,00 1615036   29472   0,19  argus
13.06.38     999    599568      0,07      0,00 1615036   29472   0,19  argus
13.07.38     999    599568      4,67      0,00 1615292   30284   0,20  argus
13.08.38     999    599568      0,17      0,00 1615292   30284   0,20  argus
13.09.38     999    599568      0,03      0,00 1615292   30284   0,20  argus
13.10.38     999    599568      0,07      0,00 1615292   30284   0,20  argus
13.11.38     999    599568      0,05      0,00 1615292   30284   0,20  argus
13.12.38     999    599568      0,03      0,00 1615292   30284   0,20  argus
```

Listing 2. Memory usage test - the output of the first ten minutes (GOGC=0)