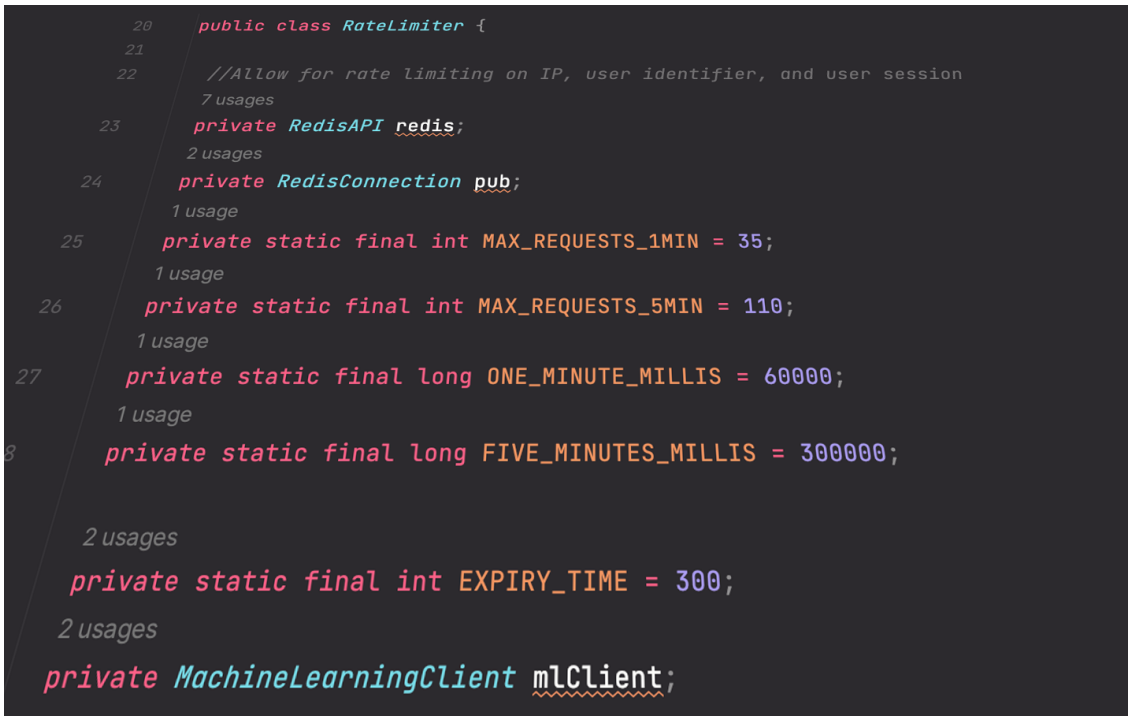
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Gateway Request Analyzer

Software as a Service

Degree project report in Computer Engineering

Emil Berzelius
Leonard Bagiu

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Degree project report 2023

Gateway Request Analyzer

Software as a Service

Emil Berzelius
Leonard Bagiu

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Gateway Request Analyzer
Software as a Service

Emil Berzelius
Leonard Bagiu

Technical Supervisor: Peter Moberg
Supervisor: Panagiotis Strikos, Department of Computer and Network Systems, Computer Science and Engineering
Examiner: Lars Svensson, Department of Embedded Electronics Systems and Computer Graphics

Degree project report 2023
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Class variables of `RateLimiter`.

Typeset in LaTeX
Gothenburg, Sweden 2023

Gateway Request Analyzer
Software as a Service
EMIL BERZELIUS
LEONARD BAGIU
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

This report describes the production of going from a proof-of-concept "rate-limiter" to a Software as a Service. A rate-limiter is meant to block spammers, bots, web scrapers, or general overuse for an application. The purpose of this degree project is to construct a rate-limiter possible to place on top of any application while still giving a customer nearly full control over its behavior. Most rate-limiters today are either custom-made for a particular purpose and therefore expensive, alternatively, black boxes placed on top of an application, black-box meaning an outside observer has no control or insight regarding its behavior. This degree project began as a proof of concept developed by students at Chalmers University of Technology but ends close to a Software as a Service, able to scale based on traffic. The Gateway Request Analyzer consists of multiple components, each based on docker images and independent from each other in their construction. Additionally, the Gateway Request Analyzer has been purposefully designed as to allow a third party to continue its development.

Keywords: Rate-limiter, Proxy, SaaS, Vertx, Asynchronous, Single-threaded, spammer, bot.

Acknowledgements

First and foremost, we want to thank Peter Moberg. Not only for giving the best available feedback possible at every turn, but also for sticking with us when things did not go as planned. He has been an invaluable resource for this project and our own personal development. We also want to thank Panagiotis Strikos for supervising this project, remaining supportive and by our side even though our focus was split at times. Finally, we would like to thank Lovisa Rosin & Viktoria Hagenbo for being collaborative in their co-project, and providing us with necessary access during the merge phase to test the GRA towards an outside project.

Emil Berzelius, Leonard Bagiu, Gothenburg, 06 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

API	Application Programming Interface
AWS	Amazon Web Service
Bash	Bourne-Again SHell
CPU	Central Processing Unit
cURL	Client For URL
DNS	Domain Name Service
ECS	Elastic Container Service
GRA	Gateway Request Analyzer
HTTP	Hypertext Transfer Protocol
I/O	Input and Output
IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
JWT	JSON Web Token
ML	Machine Learning
MVP	Minimal Viable Product
RSA	Rivest-Shamir-Adleman Encryption
SQL	Structured Query Language
SaaS	Software as a Service
TCP	Transmission Control Protocol
URL	Uniform Resource Locators
VPN	Virtual Private Network
YAML	Yet Another Markup Language

Contents

List of Acronyms **ix**

List of Figures **xiii**

1 Introduction **1**
1.1 Goals . . . 1
1.2 Limitations / Demarcations . . . 2

2 Method & Workflow **3**
2.1 Planning phase . . . 3
2.2 Implementation phase . . . 3
2.3 Merge phase . . . 4

3 Background **5**
3.1 Eclipse Vert.x . . . 5
3.2 Communication . . . 5
3.3 Cryptography . . . 6
3.3.1 Asymmetric Cryptography . . . 6
3.3.2 JWT . . . 6
3.3.3 OAuth 2.0 . . . 6
3.4 Redis . . . 7
3.4.1 Pub/Sub mode . . . 7
3.4.2 Cluster mode . . . 7
3.5 Docker . . . 8
3.5.1 Docker Compose . . . 8
3.6 Git . . . 8
3.6.1 Git submodules . . . 8

4 Design **9**
4.1 First model . . . 9
4.2 Second model . . . 10
4.3 Final model and design of the GRA . . . 11
4.4 Authentication Server . . . 12
4.5 GRA Proxy . . . 13
4.5.1 The GraClient Class . . . 14
4.5.2 The AuthClient Class . . . 15
4.6 GRA Server . . . 15

4.6.1 The GraServer Class 16
4.6.2 RateLimiter 17
4.6.2.1 The checkDatabase method 17
4.6.2.2 The setSortedBlocked method 18
4.6.2.3 The getSaveState method 18
4.6.3 MachineLearningClient 18

5 Results **21**
5.1 Testing for a normal user 22
5.2 Testing for a malicious user 22
5.3 Testing the performance of the GRA 23
5.4 Reconnectivity 23

6 Discussion **25**
6.1 Notable priorities and critical decisions 25
6.2 Future improvements 26
6.2.1 customer application 26
6.2.1.1 Statistics and customization 26
6.2.2 Security and authentication 26
6.2.3 Deploying the service 27
6.2.4 Improvements to GRA Server 27
6.2.5 Testing 27

7 Conclusion **29**

Bibliography **31**

A Appendix 1 **I**

List of Figures

4.1 Early model of the GRA 9
4.2 Model on how to integrate the GRA Server with a closed proxy . . . 10
4.3 Second model of the GRA 11
4.4 Final model of the Gateway Request Analyzer 12
4.5 UML diagram of GraProxy 14
4.6 UML diagram of GRA Server 16

A.1 Test data emulating a normal user I
A.2 Test data emulating a malicious user, using the same search query, user, IP address and session I
A.3 Test data emulating a malicious user, using different search queries and different sessions & IP addresses I
A.4 Test data emulating randomized requests, used for testing system capacity II

1
Introduction

During the course DAT067, held by Sakib Sistek, an opportunity was offered to construct a rate-limiting application over the span of two months. What a rate-limiting service does is place a proxy server between a user and an endpoint, and control the incoming flow to the endpoint in order to block spam, bots, or regular users trying to reach the endpoint in excess. This is done by looking at several parameters, such as IP, user ID, or session tokens. The company issuing this task wanted to fill a gap in the current market of rate-limiting. An already existing application, called CloudFlare [1], is one of the current market leaders for this purpose but does have some flaws according to the company. CloudFlare does block spam and bots in an efficient way, but the customer using CloudFlare is not able to specify certain aspects of the rate-limiting. What the company wanted to achieve is a rate-limiting application that can implement both user-specified parameters and machine learning to adapt the application according to customer needs.

After two months, the allocated group created a proof of concept for a rate-limiting application without any machine learning involved [2]. At that point, the application was able to apply simple rate-limiting by looking at either IP addresses, session tokens, or a username passed in the header of an HTTP request. However, the proof of concept application was restricted to running on a local machine at this point. Thus, the remainder of the project was split up between two different degree projects. The first, which this report will describe, is to reconstruct the proof of concept into a Software as a Service (SaaS), allowing the product to be deployed as a cloud-based service to be used by any user desiring to subscribe to the product. The second is the missing machine-learning algorithm that will be implemented as a part of the application before deployment, concluding both degree projects into one final product, called the Gateway Request Analyzer (GRA).

1.1 Goals

For the SaaS to be secure, the code will need to be split into different servers, each filling a respective role for the application as a whole. The main objects are:

- Customer interface that allows registering, and forwarding the data to the proxy server
- Authentication server that:
  - Generates public/private keys, issuing the public keys on an open API

  - Issues authentication tokens to the proxy that can be verified by the GRA Server
- Updating the GRA Server so that it:
  - Properly fetches a public key from the Authentication server
  - Verifies tokens to assure secure connections from the proxy
  - Reliably and safely exchanges information with the rate-limiting algorithm
- A proxy server that:
  - Forwards a user's requests toward a desired endpoint
  - Obtains updates from the customer front end regarding current registered customers
  - Communicates with the GRA Server and save currently blocked users
  - Obtains tokens safely and reliably from the Authentication server
  - Handles re-connectivity issues

1.2 Limitations / Demarcations

The construction and development of a useful and valuable rate-limiting ML application that would be placed on top of the GRA Server is outside of the scope of this project. Integration with an already working ML application is however desirable and one of the main aspects of the GRA Server itself.

2
Method & Workflow

The workflow to be used to develop the GRA application is inspired by the agile methodology. The agile manifesto is based on twelve principles [3] and aims to achieve adaptability and quality. There are many different implementations of agile, differentiating how often meetings should be held and how to continuously define goals and improvement. The product owner of the project will provide aspects of both a classic product owner [4] and a technical supervisor. Many of the crucial decisions regarding architecture and which tools to use will be made by or as an agreement in discussion with him.

2.1 Planning phase

During the first 3 weeks of the project, most of the focus will revolve around planning and research. Firstly other similar products will be researched, attempting to use their software products to form intuition and a bird's eye view of the main parts required for the GRA. During this phase, several high-level abstraction models, as seen in Chapter 4, will be generated to produce an agreed-upon vision. Furthermore, the models need to be based on assumptions about who a potential customer might be. These assumptions were in large part generated by the project owner.

2.2 Implementation phase

The implementation should span somewhere around 60-70% of the total time. While implementing the GRA, some parts of the agile manifesto [3] are deemed irrelevant, most notably customer interaction. Instead of following a specific agile methodology, an organic workflow inspired by agile is expected to reveal itself throughout the project. From the beginning, a few simple principles were agreed upon between the team members and the product owner.

- A weekly meeting every Friday with the product owner
- Defining the upcoming week's goals in unison with the product owner
- To define tasks independent from other tasks, enabling parallel progress
- To place tasks in different meaningful categories:
  - Started
  - In testing
  - Shown to other team member
  - Ready to demo

  - Approved by the product owner
- Working software as the primary measure of progress

The abstract model of the GRA application will be revisited during the implementation phase and updated with relevant details. Furthermore, the details of the implementation will be kept as notes and changed almost on a weekly basis. The reason for this is that the priorities should change based on the expected time of completion, unexpected technical challenges which had to be addressed in a certain order, and other unexpected factors. However, the overarching vision and goal should not change much throughout the implementation phase.

2.3 Merge phase

Spanning the final 6 weeks of the project, the merge phase will hold the purpose to integrate the GRA with the Machine Learning (ML) application, making them able to run in the same environment. This will enable testing and fine-tuning of the two projects together. During this phase, an additional weekly meeting with the other group will be added to the workflow, as well as including both groups in the already existing weekly Friday meetings with the product owner. This should be done to enable a smooth transition from two separate projects into one unified application.

3
Background

In the following sections, explanations will be given for all relevant technology used within the report.

3.1 Eclipse Vert.x

Eclipse Vert.x [5] is set up to run on the Java Virtual Machine(JVM) and is a toolkit focused on using resources efficiently through reactivity. Vert.x becomes more reactive by focusing its execution on blocking I/O calls and is fully asynchronous. This combination allows the application to only run when necessary and to perform multiple tasks concurrently. The smallest deployable component of vert.x is called a verticle, and is a single-threaded execution environment, allowing better possibilities for scaling applications through niched components. When deployed, a verticle is able to share information internally through an event loop, that can be reached anywhere within the application.

3.2 Communication

Hypertext transfer protocol (HTTP) [6] is an application layer protocol [7] and has become the foundation of data transfer over the internet. HTTP is split into two parts, a request and a response, which constitute the basis for HTTP communication. A response will always be sent with a status code following a request, containing a status code for whether the request was successful or not. By using this "handshake" to confirm message statuses, the sender will be able to adapt its functionality according to whether a request was accepted or not on the receiving side. Hypertext transfer protocol secure (HTTPS) has the same functionality, but with added security measures in the form of message encryption, and is the market standard today with roughly 80% of websites using it [8].

A WebSocket, just like HTTP is an application layer protocol and is a form of connection that requires validation for its initial connection using HTTP, but then remains open for communication. The communication will in turn be full-duplex using TCP, ensuring that all messages are received between the two parts. If one of the two parts closes the connection, the connections is closed for both parties. A WebSocket will, in most cases, perform its data transfer as a buffered stream.

A proxy is an intermediate server between a user and an endpoint. A proxy will perform an additional task between a user requesting to access a website and reach-

ing it. An example of this is a VPN, that masks the IP address of a user from their Internet Service Provider in order to increase anonymity.

3.3 Cryptography

Cryptography is a technology used to safely encrypt or sign some form of data.

3.3.1 Asymmetric Cryptography

Asymmetric cryptography uses two separate keys to encrypt internet traffic, called a public and private key. These are related and are generated through mathematical formulae so that the private key is the only way to decrypt a message encrypted by the public key, and vice versa.

There are two common ways of applying asymmetric cryptography. The first is called public-key encryption and has a user encrypt a message with the public key of the recipient. The message can only be decrypted by the recipient's private key, which secures safe information transfer if the private key has been kept secret. The second is encrypting a message using the sender's private key, also called a digital signature. Anyone holding the sender's public key will be able to decrypt the message in this format, validating the sender's identity.

Most key-generating tools will allow the user to generate a key of bit sizes 512, 1024, 2048, 3072, and 4096. Larger bit sizes will grant higher levels of security but at the cost of slower computing times.

3.3.2 JWT

JavaScript Object Notation (JSON) allows changing an object into structured key value pairs. In turn, these can be converted into String values, which are used by a majority of data transfer methods. JSON Web Token (JWT) [9] is an open standard for securely packaging data formatted in JSON, adding additional information such as expiry time.

It digitally signs tokens sent between the parties with either a shared secret or a public/private key pair. A signed token can then be verified for specific access within an application. For example, a user with or without admin access to a website would have different tokens. Much like JSON, JWT is a carrier of information rather than having any functionality on its own.

3.3.3 OAuth 2.0

OAuth 2.0 [10] is an authentication protocol focused on using a third party to authenticate a user toward an endpoint. In order to do this, an access token protocol needs to be used to make sure public/private keys or a shared secret are used. OAuth 2.0 uses different grant types depending on desired accessibility of a client wanting

to be authenticated. The first type, called the Authorization Code Grant [11], has the client exchange an authorization code for an access token.

Generally, the user is redirected when trying to access the endpoint and can acquire an authorization code from that side. When redirected back, that code can be exchanged for an access token. An authorization code grant is most intended for users trying to access sites through third-party verification. The second type is called a Client Credentials Grant [11] and requires a secret to be validated between the client and endpoint for a token to be issued. The Client Credentials Grant is most used in machine-to-machine interactions. OAuth 2.0 is able to issue a refresh token in addition to an access token, that can be used to refresh the access token if an expiry time was issued.

3.4 Redis

Redis [12] is an in-memory data store, adaptable to be used as a database, cache, streaming engine, and message broker. Rather than having all data sorted into tables, like an SQL-based database, Redis uses data structures to store data in-memory. Furthermore, the data being stored can be adapted to fast-access or permanent storage according to the users' needs. Another feature of Redis is being able to design its data storage horizontally through hash-based sharding, called clustering. These clusters can reach the sizes of millions of nodes for larger data networks.

3.4.1 Pub/Sub mode

Redis has a built-in pub/sub service implementing the Publish/Subscribe messaging paradigm [13]. The pub/sub message paradigm allows a publisher to send a message to a channel rather than to a specific receiver. Furthermore, subscribers decide which channels to listen to and will be notified when a message is published on said channel. This allows for passing information between nodes in scale-able applications without a need to be aware of each other's existence as long as the channel is agreed upon.

3.4.2 Cluster mode

Redis cluster mode will run several instances (nodes) of Redis and allow for sharing of data between them automatically. A cluster is divided into 16384 hash slots, and each node is responsible for a sub-part of them. To calculate the hash slot of a key the CRC16 of the key modulo 16384 [14] is used. A cluster containing 3 nodes will divide the slots accordingly:

- Node A contains hash slots from 0 to 5500.
- Node B contains hash slots from 5501 to 11000.
- Node C contains hash slots from 11001 to 16383.

Furthermore, Redis cluster mode implements a master-slave model to ensure fail safety. Each node will create a read-only replica (slave), which will take over as a master node in case its master fails.

3.5 Docker

Docker [15] is a tool that can package an application or a process and all of its dependencies into a container. A container is an isolated process containing its own file system, kernel namespaces and cgroups [16]. This allows for a good test environment since the container will be able to run on local machines, virtual machines, or deployed to the cloud as is. A container is based on an image containing all of the files needed for the application. The rules for what is included in an image are specified in a Dockerfile, unique to each application.

3.5.1 Docker Compose

Docker Compose is a tool for defining and running multiple containers. Docker Compose also permits the creation of a virtual network between the containers in the Compose. The rules for a Docker Compose are specified in a YAML file, allowing it to execute all of its goals with a single command line. According to the docker official website "Compose works in all environments: production, staging, development, testing, as well as CI workflows" [17]. Furthermore, Compose has commands which allow:

- Starting, stopping and rebuilding services
- Viewing the status of running services
- Streaming the log output of running services
- Running a one-off command on a service

3.6 Git

Git [18] is a version control system developed by Linus Torvalds with the purpose of allowing contributions from several members of a team to a project. The concept of Git is that a project is constructed within a shared repository, holding the project in its entirety.

3.6.1 Git submodules

A repository within Git can contain external repositories as a reference to be used within the users' repository, called a submodule. This allows for using already finished external code within a project, allowing several parties to work separately towards a common final project, even when working in different repositories.

4
Design

This chapter aims to describe the technical design of the GRA both in different stages of development, and the final result.

4.1 First model

Figure 4.1 represents the first model produced by the GRA. It was made during the planning phase and was based on the initial research where other similar products were tested. In this model, the blue parts represent what was thought to be the minimal viable product (MVP) and would mostly include large-scale improvements of the already existing programs. The most notable difference when comparing this model to the final model in Figure 4.4 on page 12 is the existence of the customer application and the choice of how to apply color. The purpose of the coloring during the later models was rather to differentiate running applications. When producing the first model there was no regard for the runtime environment, which ended up being docker containers, and no understanding of how to implement authentication between programs. However, it fulfilled the purpose of creating a united vision of what the project would look like and was deemed good enough for a basis on which to start.

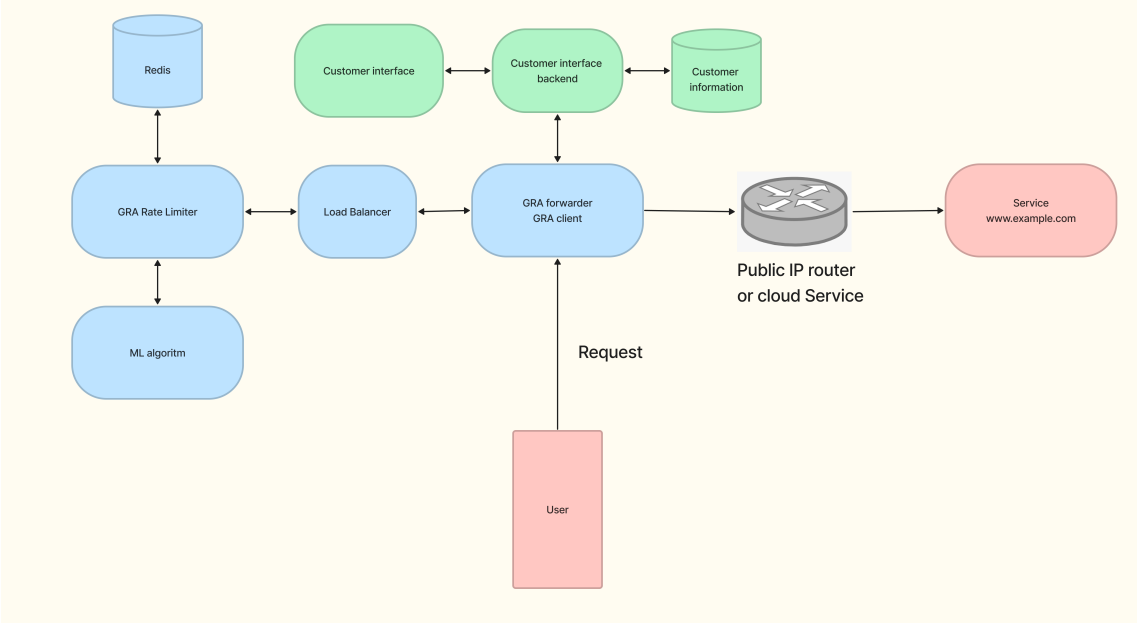


Figure 4.1: Early model of the GRA

The most important aspect of the first model was placing the proxy between the customer application and its respective APIs. During the planning phase, the potential customers were placed into three categories:

- A custom-developed proxy.
- Open source proxy (Nginx, HAProxy).
- Closed proxy - Cloud load balancer (AWS, Azure, Google Cloud, etc) or a hardware load balancer.

The first two would have been able to integrate the GRA Server in their own proxy by making calls directly to the GRA Server from it. However, most customers would fall into the third category where they are assumed to use cloud services for their APIs or use a hardware load balancer on top of their own data center. The only way to allow for a customer in the third category to use the GRA would be to place it in between the customer app and their APIs. This decision was mostly informed by the product owner and visualized by him as seen in Figure 4.2.

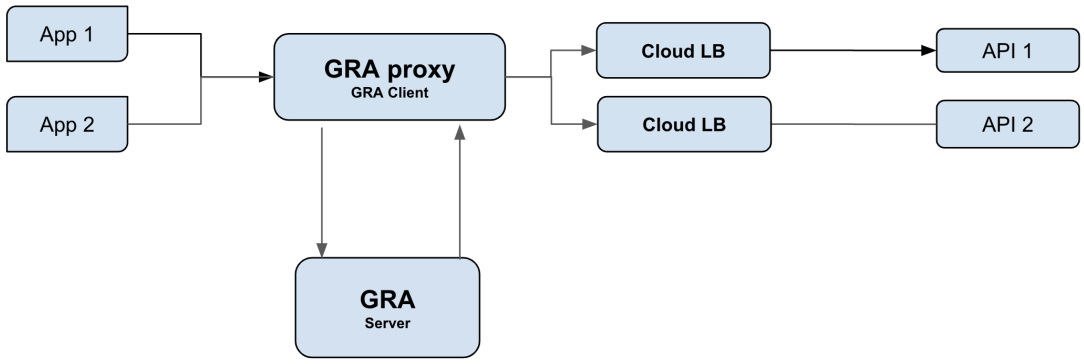


Figure 4.2: Model on how to integrate the GRA Server with a closed proxy

4.2 Second model

The second model can be seen in figure 4.3 and was produced while developing the authentication server and further granularity was added to the model using colors. When the authentication server was finished, much time was spent on documentation and retrospection. This was also a point where the finished picture started to be graspable. As seen in Figure 4.3 there were no features yet removed from the model. However, this was where a realization occurred that some parts would not be implementable during the time remaining. This meant that some parts had to be prioritized over others and this was done in unison with the product owner. An agreement was made that the runtime environment and deployment were aspects that should be of higher priority.

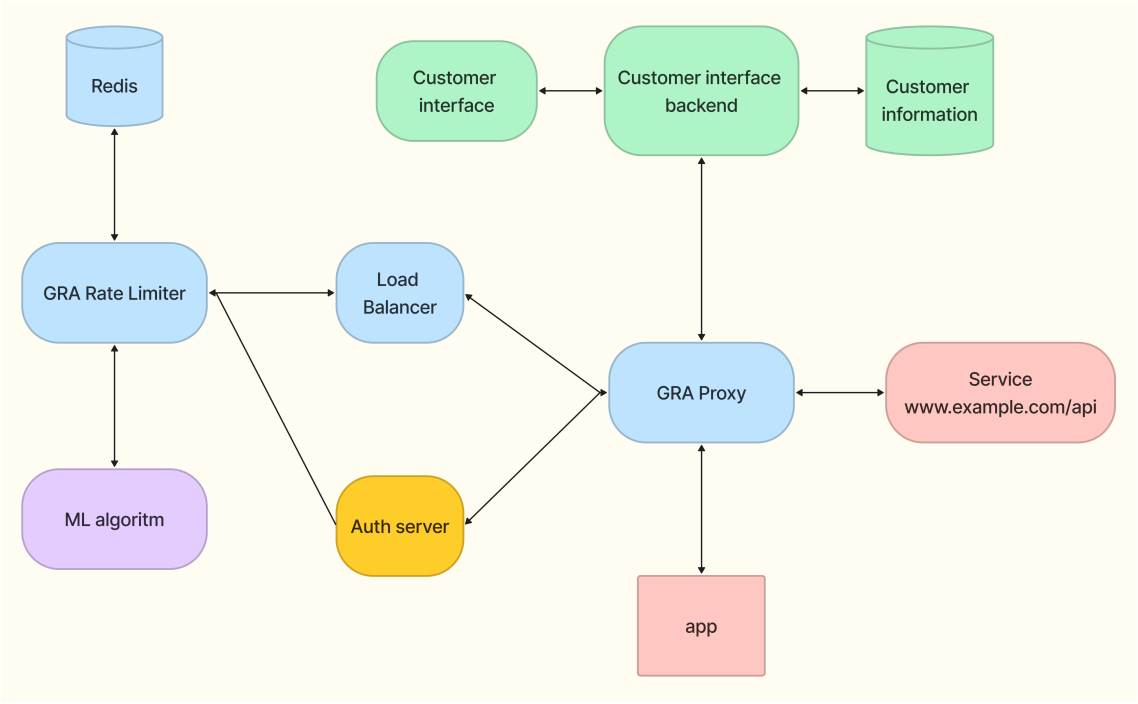


Figure 4.3: Second model of the GRA

4.3 Final model and design of the GRA

Firstly a short introduction of the final model is given and the journey of a successful request is explained. Each independent part of the GRA is then described in detail. Figure 4.4 demonstrates a model of the top-level abstraction of the entire GRA application. Each color represents a different responsibility and task carried out in order to achieve the goal of a generalized rate limiter. The red color represents what lies completely outside the control of the product, which is the customer app and its back-end APIs.The blue color represents the GRA Proxy and is the heart of the total product. The green color represents the GRA Server which is the brain of the product and is responsible for decision-making. These decisions are called actions and might block a certain IP or user. The purple color represents the ML application, which is also outside of the scope of this project but is a vital part of the value of the product. The orange color represents the part of the project responsible for authentication and safety. Furthermore, each box represents a running docker container, or several instances of themselves, communicating with HTTP or WebSockets in between each other.

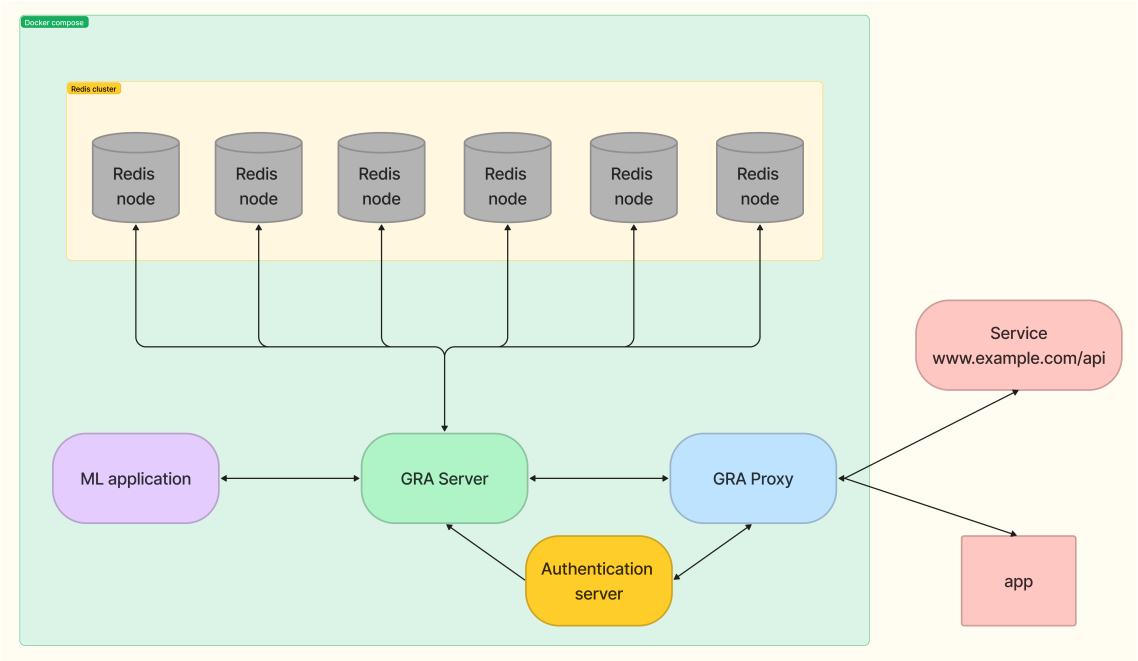


Figure 4.4: Final model of the Gateway Request Analyzer

The entire application is designed to be built using Docker Compose, which sets up an internal network between the containers and initiates each Dockerfile, building the desired image.

Assuming each program is running, the journey of a successful HTTP request starts at the authentication server broadcasting its public key to the GRA Server, making it possible to verify access tokens upon request. The GRA Proxy requests an access token, signed by the authentication server. Assuming this is in place, an HTTP request is sent from an outside app to the proxy. The GRA Proxy checks if the sender of the request is already blocked and if not, forwards information about the request to GRA Server and fetches the API (red box) back to the sender. The GRA Server first verifies the access token, using the public key fetched from the authentication server, and proceeds to check and count each parameter. The necessary information is saved in a Redis database.

Finally, the information is forwarded to the ML application which might deem an action necessary and communicates one of 3 possible outcomes back to the GRA Server. The outcomes are "OK", "block" and "verify". "OK" and "block" will allow or block a user, and the "verify" action will send back a message to the user that it needs further verification. Suppose the GRA Server makes a decision or receives one from the ML application. In that case, it is broadcast to each running instance of the GRA Server and communicated to each Proxy currently connected.

4.4 Authentication Server

The authentication server is written in Java and leverages the Vert.x library. It consists of 3 classes, `AuthServer`, `KeyGen`, and `KeyAPI`. The authentication server

implements the OAuth 2.0 design and only allows client credentials as a grant. `KeyGen` is used to generate an RSA public/private key pair and store them in a file. `KeyAPI` is an HTTP server that will respond with the RSA public key to any request sent to the Authentication Server with the path:

```
/.well-known/jwks.json
```

`AuthServer` is an HTTP server that will create and sign a JWT to clients with permitted client credentials. The allowed clients are stored inside a file and are read at the start of the program. To retrieve a token, a client sends a request to the server with the path:

```
/oauth/token
```

If the client credentials are in order, a JWT is created and signed with the RSA private key. The correct body and headers are set according to the OAuth 2.0 protocol and the `AuthServer` responds with HTTP-request such as [11]:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"jwt",
  "expires_in":3600
}
```

4.5 GRA Proxy

The Proxy is written in Java using the Vert.x toolkit and consists of 3 classes, as well as a startup class that extends `AbstractVerticle` ensuring the program runs single-threaded.

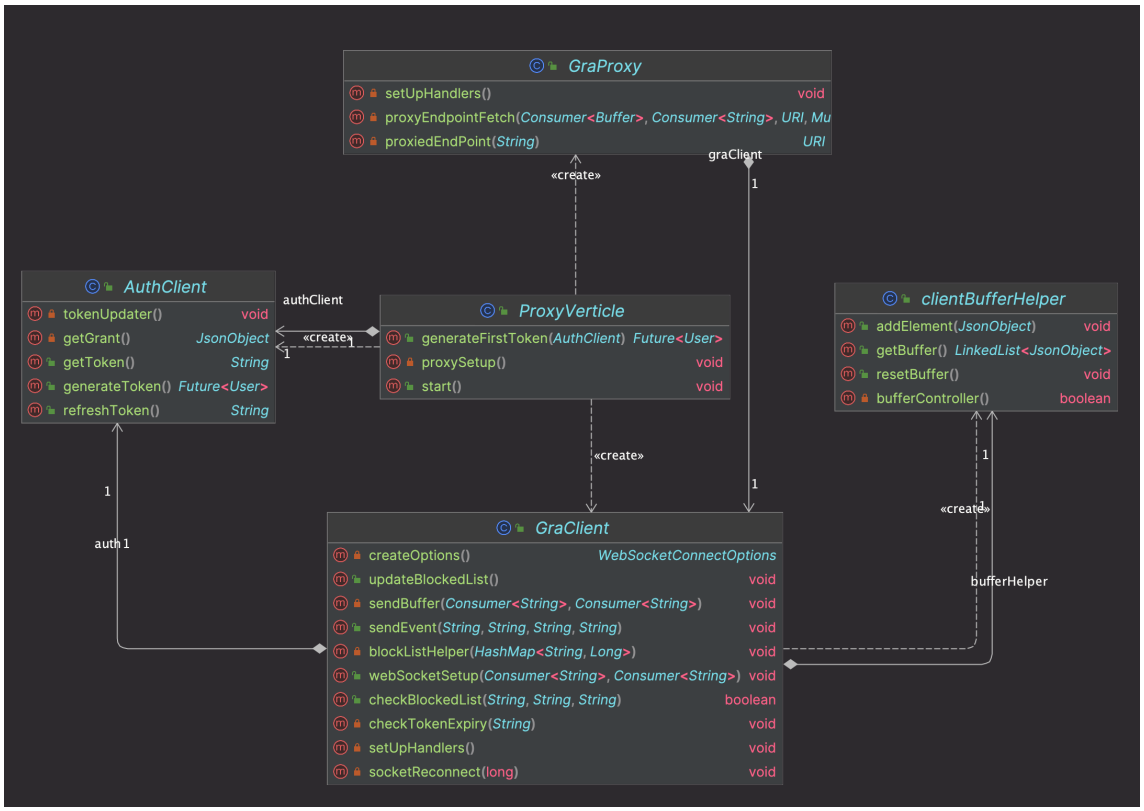


Figure 4.5: UML diagram of GraProxy

As seen in Figure 4.5 the `ProxyVerticle` initiates all other classes. The `AuthClient` is created and a first token is requested from the Authentication Server. This is done using the `CompositeFuture` feature provided by Vert.x, ensuring that a token is generated before any further code is executed. After this, the final stages of the setup are completed. This includes initiating an HTTP server, initiating the `GraClient` and initiating the `GraProxy`. The `GraClient` opens a WebSocket connection toward the GRA Server, using the access token provided by the `AuthClient`. The Proxy uses `GraClient` as a library to handle all communication with the GRA Server. It also uses the `GraClient` method `checkBlockedList` to confirm that the sender of the request is not already blocked. Finally, it uses `sendEvent` to transmit the necessary data to GRA Server and fetches the endpoint which the original HTTP request had specified. Furthermore the `GraClient` uses the `AuthClient` to fetch the current token. The `AuthClient` will generate a new token before the current one expires and is also invoked to generate a new token if any request to the GRA Server is denied.

4.5.1 The GraClient Class

The `GraClient` functions as a library to communicate with the GRA Server. It maintains a `HashMap` of each parameter which a user might be rate limited on and holds the blocked parameter as the key and a Unix time stamp [19] of the expiry time as the value. A single entry in the blocked IPs `HashMap` might look as such:

```
<"0.0.0.0", 1679319129>
```

The `GraClient` attempts to manifest a WebSocket connection to the GRA Server as soon as it is created and if unsuccessful attempts to reconnect on an exponentially increasing time interval. The method `setUpHandlers` is executed in the constructor. The 2 handlers used by `GraClient` are a binary message handler and an exception handler. The binary message handler handles 2 different scenarios: a single action taken by the GRA Server or a `saveState` object containing all actions currently being enforced. A single action will be added to its respective `HashMap` and invoke the method `updateBlockedList`. The method will remove entries from each `HashMap` where the expiry time has expired. A `saveState` object will typically be received when opening a new WebSocket connection. If a state object is received, all of the actions are added to their respective `HashMaps`. The `sendEvent` method will package a buffer of all necessary parameters and an access token and send it as a binary message over the WebSocket to the GRA Server.

To handle re-connection issues between the GRA Proxy and the GRA Server the `socketReconnect` method is used. It attempts to open a new socket connection to the server at an increasing time interval. After a failed attempt to open a connection, a new attempt will be made after 3 seconds. The time interval will double for the next attempt and if the time interval reached 60 seconds, an attempt to connect will be made every minute. In addition to this, the class `ClientBufferHelper` will gather up incoming requests while there is no open connection and store them as an `ArrayList`. If the number of requests stored crosses 3000 the `ClientBufferHelper` will no longer store requests to avoid overwhelming the main memory of the Proxy. However, 3000 is a placeholder value as the program is currently not hosted on a large server. When a connection is reestablished the linked list will be flushed and each request will be sent to the server. Finally, the buffer is emptied by the `resetBuffer` method.

4.5.2 The AuthClient Class

The `AuthClient` will read its client credentials from a file when created. The `generateToken` method will request an access token with its client credentials from the authentication server. This is done using the `OAuth2Auth` library provided by Vert.x [5]. The `AuthClient` object generates a new token before its current token expires and the `getToken` method is used by the `GraProxy` to access the currently valid token.

4.6 GRA Server

The GRA Server is written in Java, use the Vert.x toolkit and consists of four main classes, `GraServer`, `RateLimiter`, `TokenAuthorizer` and `MachineLearningClient`. In addition to this, a start-up Class ensures the application runs single-threaded and that all necessary connections are established before accepting any requests, and 2 small helper Classes responsible for holding data are used by the GRA Server.

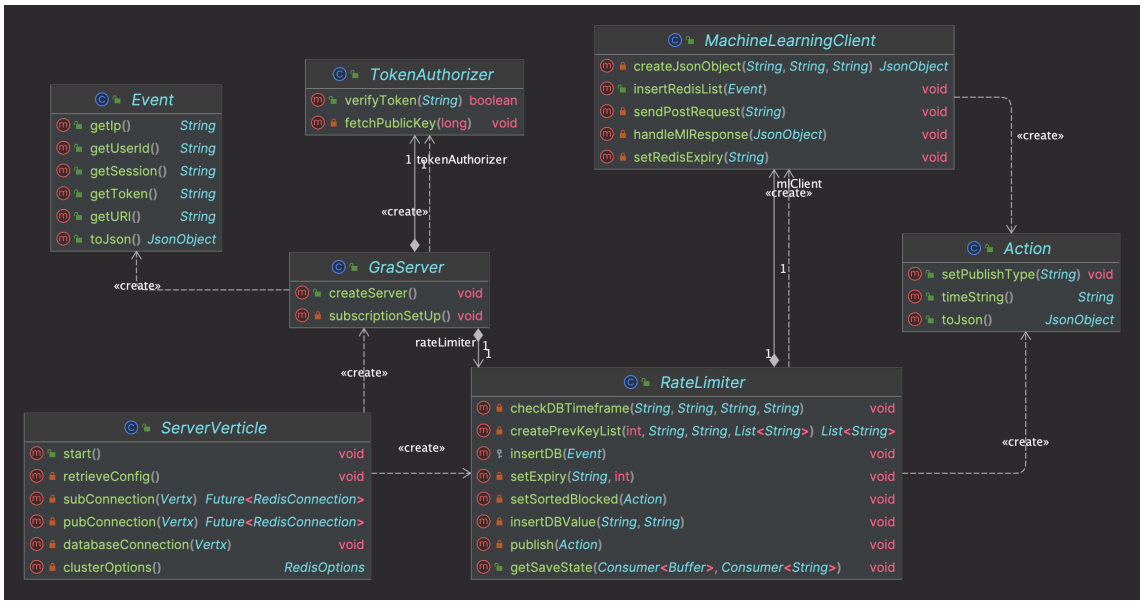


Figure 4.6: UML diagram of GRA Server

Earlier, the entire application of Figure 4.6 has been referred to as GRA Server and should not be confused with the Java class `GraServer` included in the application. As seen in Figure 4.3, `ServerVerticle` is responsible for the creation of all 3 main classes. The `ServerVerticle` will first retrieve its configurations from a file, and then the `CompositeFuture` feature will ensure all connections are established before any further code is executed. The necessary connections are: Connection to the Redis database, Redis pub channel, and Redis sub channel. On completion, the `RateLimiter` and `TokenAuthorizer` are instanced and passed as parameters to the `GraServer` constructor.

4.6.1 The GraServer Class

The `GraServer` will run its method `subscriptionSetup` to subscribe to the Redis pub/sub channel where all the running servers communicate. In addition to this, a handler is set up dictating that any message received on the channel will be serialized to JSON and sent as a binary message to all the currently connected instances of GRA Proxy. This will typically be an action regarding a user or a complete `saveState`.

The `createServer` method will start a server listening to a port and declare a Vert.x `webSocketHandler` that will accept WebSocket connections. The first thing checked inside of this handler is if a valid access token is passed in the header of the initial request. This is done by the `TokenAuthorizer` method `verifyToken`, using the RSA public key broadcast from the Authentication Server mentioned in section 4.2. Much like the `GraClient` mentioned in section 4.3.1, the `TokenAuthorizer` will attempt to fetch the public key from the Authentication Server until successful. If the token is not validated, the connection is closed and responded with status code 418, indicating to the Proxy that a new token has to be generated. After the token has

been validated, the save-state object is transmitted as a binary message to the newly connected client and the connection is added to a `HashMap` called `openConnections`.

Inside of the `webSocketHandler` a `closeHandler` and a `binaryMessageHandler` is defined. The `closeHandler` will remove the connection from the `openConnections HashMap` when a WebSocket is closed. The `binaryMessageHandler` will check and verify an access token on each incoming message and close the WebSocket if a valid token is not included. A message including a valid token will be serialized into an `Event` object and this object is passed to `RateLimiter` through the method `unpackEvent`.

4.6.2 RateLimiter

The `RateLimiter` is the brain of the application and the most extensive class of the entire project. It involves all communication with the Redis database and ML application.

4.6.2.1 The checkDatabase method

The `checkDatabase` method takes one of the parameters from the `Event` and if this value does not already exist in the database, it is saved as a key/value pair. The key of this pair is the value of the parameter concatenated with the current minute and the value is the number of requests made this minute. A first request from IP address 0.0.0.0 made at 14:17 will have a corresponding entry in the database as such:

```
<"0.0.0.0:17": "1">
```

The expiry time Redis feature is used on each key and an expiry time of 5 minutes is applied to each entry. If the key already exists the value is incremented by 1. If the value of an entry crosses the threshold of what is allowed for the duration of a minute, an action is taken and published on the Redis pub/sub channel to all servers currently running and in turn communicated back to all connected instances of GRA Proxy.

In its current implementation the `RateLimiter` has a threshold for a maximum number of requests over 5 minutes in addition to 1 minute. This is because a typical user will make requests in bursts. To check how many requests have been made in the past 5 minutes by a user the method `createPrevKeys` is used to fetch all relevant requests from the database and count the total amount. This is done on every request and if the sum crosses the 5 minute threshold on any given parameter an action is issued.

In addition to publishing single actions to the Redis pub/sub channel, the actions are added to a save state object in the database through the `setSortedBlocked` method.

4.6.2.2 The setSortedBlocked method

The `setSortedBlocked` method makes use of the `Action` class. It contains the following information about an action:

- `value` - the value of the `Action` being rate limited (user ID, IP).
- `actionType` - the type of `Action` being rate limited.
- `timeBlocked` - a Unix time stamp of when the `Action` expires.
- `publishType` - specifies if this is a `single` action or a `saveState`
- `blockSource` - specifies the source of the `Action` which will typically be the `RateLimiter` or ML application.

A single action will be added to a Redis sorted set. The key of every entry in this set is the expiry time of the action, and the value is an entire `Action` object serialized to JSON. On each entry, the Redis sorted set command `zremrangebyscore` is executed on the sorted set which will remove all entries where the expiry time has passed.

4.6.2.3 The getSaveState method

The `getSaveState` method implements the Java Consumer Interface and will fetch the `saveState`. The `saveState` is serialized to JSON and on completion is returned as a `JsonObject` to the method caller.

4.6.3 MachineLearningClient

The `MachineLearningClient` functions as a helper class to `RateLimiter` and is responsible for communication with the ML application. The `MachineLearningClient` will store relevant request values in Redis as a list to fit the ML application-API. The list holds several entries from the same user and when serialized to JSON might look as such:

```
[
    {
    "timestamp": 1683296967686,
    "userID": "abc122",
    "sessionID": "abcder",
    "expiring": 1683307767686,
    "URL": "/login"
    },
    {
    "timestamp": 1683296967572,
    "userID": "abc122",
    "sessionID": "abcder",
    "expiring": 1683307767572,
    "URL": "/search?value=abc"
    }
]
```

The `MachineLearningClient` will send a list as an HTTP request to the ML application-API when it has gathered 20 requests from the same user and will save

a maximum of 50 requests. When the maximum amount of requests is reached for one user, the earliest one is removed from and the latest one is added to the Redis list. The entire list will be sent to the ML application-API for each request past the point of 20 stored entries in the list. Following each request is a response holding 1 of 3 possible outcomes which are "OK", "block" or "verify". The "verify" parameter will return a message to the user that he/she needs to be verified. However no actual verification process is implemented in the current state of the GRA, and this is only an HTTP response with a body indicating that the user has been flagged.

5

Results

When testing the program, different bash scripts were generated to simulate different kinds of users accessing the program. As the purpose of this application is to distinguish between a normal user and a spammer or bot, it is imperative to assess the different behavior patterns of the aforementioned groups.

When testing for a normal user of an outside application, we have assumed that a user would not surpass 35 requests per minute. Applying a sliding-window algorithm for a five-minute period, we have considered 110 requests to be a reasonable value. Given that 110 requests are smaller than 35×5, a normal user is expected to make bursts of requests when reloading a page as an example. However, the user is not expected to perform 35 requests consistently each minute as already cached data does not require further requests.

The number of requests within a real application would need to be changed depending on user behavior. The previously mentioned values have been set to test the rate-limiting algorithm in an environment expecting a user to make a request every 1 to 5 seconds. A spammer or bot would have a different behavioral pattern, where a much larger amount of requests would be made in short time intervals.

In order to test these cases successfully, there are 2 executable bash scripts, and four input files containing HTTP requests. Both bash scripts are designed to read the input files and place the values in an array. A for-loop is then used to send a random request from the array at each iteration of the loop. The difference between the scripts is that the script designed for spammers does not contain a sleep variable. For a normal user, this is set to pause a random time between 1 and 5 seconds between sending the HTTP request to the GRA Proxy. The possible outcomes from repeatedly sending requests to the GRA Proxy are:

- Being blocked by simple rate-limiting
- Being by the ML application
- Being required to verify that you are not a malicious user by the ML application
- Being allowed to send requests uninterrupted

5.1 Testing for a normal user

As previously mentioned, the tests entailed to mimic a normal user running the program were designed with sleep for periods ranging from 1 to 5 seconds. The requests made during this time period held the same user id, session id, and IP address as normal users would have interchangeable values in these fields. The test data used can be seen in figure A.1

When running the script using this text file, the number of requests sent from the shell was between 17 and 23 every minute, which did not result in blocking further requests from the user. This script has been tested for approximately 10 minutes without any "block" or "verify" action from either the rate limiter or the ML application, proving to be a desirable result.

5.2 Testing for a malicious user

The first input file for this test contained the same user id, session id, IP address, and URL path and can be seen in figure A.2. The second file tested contained the same user id, but different session ids, IP addresses, and URL paths and is shown in figure A.3

When trying these tests without the ML application, removing the sleep interval was tested first. When testing this scenario, the user reached 35 requests and was blocked almost instantaneously for 1 minute, and then blocked again for the consecutive minute. Then, a time of 0.2 seconds was set between every request, resulting in the user being blocked after roughly 2 seconds regardless of various session IDs and IP addresses being used.

Following this, the ML application was applied on top of the `RateLimiter`. The first test, in which every request contained the same IP, session, and URL path ended in the user being asked to verify itself after 23 requests. In the second test, variability was added to the URL path, making it pick a random path from the following:

```
/users
/users/1
/users/2
/send/1
/send/2
/send/3
```

This test was designed to mimic a slightly smarter bot, which would search for users, and send messages to them repeatedly. The test resulted in the user being required to verify itself after 25 requests. The result of this test is that the ML application is able to catch a user before reaching the 35 requests 1-minute limit. This proves a successful connection and integration between the 2 projects.

5.3 Testing the performance of the GRA

This specific test, much like that for the bot scenario, ran the tests excluding the sleep timer between requests. The input file, as can be seen in figure A.4, contained 45 unique requests, and to further test both the capacity and speed of the system, 2 terminals ran this test simultaneously. Additionally, the 1-minute request limit was increased to 100. Each terminal executed 10000 requests in total.

During this test, a redis-node within the cluster would occasionally crash. During this time period, the node would first reconnect and update itself with the rest of the cluster, in which on average five requests were either lost due to a connection error to that specific node. The most likely reason for the node crash is memory overload when the node writes its data to the main memory. In this state, a node is vulnerable to high amounts of traffic as it needs to handle backup-tasks simultaneously. However, the number of requests lost in total is less than 1% of the total stream, deeming it a smaller error.

The system is designed to make each running program single-threaded and scale horizontally by adding more nodes to the Redis cluster and more instances of each running program if traffic requires it. With these results, it is possible to conclude that the system is able to handle at least 20000 requests sent to it as fast as 2 bash terminals can run a loop. It is also possible to conclude the hard requirement for scaling the GRA is the number of CPU cores available. Furthermore, this test proves the resilience of the Redis cluster, being able to recover from a crash in a matter of milliseconds.

5.4 Reconnectivity

As mentioned in the design chapter, the application is able to reconnect between its respective parts whenever one part loses connection or goes down. If the Authentication Server goes offline and the GRA Server intends to update its keys, it has been verified that the server is able to reconnect whenever the Authentication Server goes back online. The public key is fetched once during the GRA Server lifetime, when starting, and reused to verify each token. The only exception to this is if the Authentication Server would generate a new RSA key pair, which using best practices is done every 2 months. With this, it is possible to conclude it is very unlikely that an offline Authentication Server would affect a running GRA Server.

The GRA Proxy has been tested on the GRA Server if the connection is lost with successful results. Before implementing cluster mode in Redis, flushing the buffer proved too fast for a single Redis node to handle in the GRA Server. With cluster enabled, this problem was successfully solved.

6

Discussion

This chapter aims to shine a light on some of the different design choices and priorities made throughout the project and how the final design came to be. It also aims to discuss alternative solutions and future improvements.

6.1 Notable priorities and critical decisions

During the 2 weeks prior to the merge phase, a concrete decision was made to prioritize the runtime environment using docker instead of spending any time developing the customer application. Since a conscious decision had been made to not implement the customer application some aspects of the GRA Proxy itself have not been implemented in relation to it. Most notably, there are no data types or files holding information about customers inside of the GRA Proxy and therefore no limitations on what outside apps are allowed to send a HTTP request to it. This decision was made because the team assumed this would be easiest to implement in parallel with the customer application back end.

During the final stages of the implementation phase, it was decided that the rate-limiting algorithm needed rebuilding and optimization. This decision arose from the code being difficult to read and too nested. As a result of this, there were difficulties adding new functionality. The reason for this was that it was one of the first things implemented in the project during the DAT067 course project, but very fundamental to the entirety of the GRA. This change was performed in preparation for the merge phase as the data stored in the rate-limiting algorithm did not have the correct format to communicate with the ML application.

When stress-testing the product it became clear that one instance of Redis was not enough to keep up with a single instance of the GRA Server. Therefore the implementation of a Redis cluster was made. This allowed for more accurate tests and a product that would act the same in a development environment as in production. However, since the plan is to deploy the GRA using ECS the Redis cluster would have been an AWS service and not something to be implemented as a part of the docker compose. If this project would have been redone, it might have been assumed that the AWS service of a Redis cluster would suffice and that testing it locally is an unnecessary step.

6.2 Future improvements

This section attempts to describe what would be required for the GRA to be a usable SaaS in production. Some of the aspects are not necessary, but very desirable.

6.2.1 customer application

The customer application would be a necessary component of the SaaS if it would be desirable to scale the product to more than internal usage. The purpose of the customer application is to be the link between the GRA and the customer. The customer would specify their DNS or IP address to their front-end application as well as their back-end APIs. Using this information the customer application back-end would communicate it to the GRA Proxy. The customer would in turn replace their API calls with calls to the GRA Proxy. For example the API:

`https://examplebackend.com/api/table?id=123`

would be replaced by:

`https://GRA.com/examplebackend/api/table?id=123`

In the planning phase, the customer application was planned to be built using Node.js and Vue but could be built using any web-based framework. The communication to the customer application would be done by HTTP or HTTPS, meaning it has no technical constraints in relation to the GRA.

6.2.1.1 Statistics and customization

A desirable addition of functionality to the customer application is the gathering of data on a customer application as well as a user-friendly display of the data. This could be the average number of requests per minute or the number of spammers blocked over a time span. Furthermore, a customer should be able to specify the behavior of the GRA Server based on this information. This could be the number of requests allowed per minute, to use the ML, or what ML model is most desirable for their application. What type of statistics, metrics, or customization required or desirable is hard to specify in its totality and would have to be a continuous task evolving in unison with the GRA customers.

6.2.2 Security and authentication

There are some security aspects which would need improvement. Firstly, as mentioned in "Notable priorities and critical decisions" there are no limitations on what outside applications are allowed to access the GRA Proxy. There would also be a need to ensure authentication between the customer application and the GRA Proxy to ensure both that no corrupt data enters the system and that the data is not leaked. Furthermore, there is no authentication between the GRA Server and the ML application, introducing the same problem as mentioned above. This issue could be resolved using the already implemented authentication server. Since it's implemented following the OAuth 2.0 protocol, any OAuth client library could be used to interact with the authentication server.

Another issue regarding security is that both the client secret and the RSA key pair should be updated on a regular basis. The server has the functionality to update both its keys and the secret but no active solution for sharing the client secret with the GRA Proxy in a safe way. The main issue with shared secrets is that they should not travel across the internet. This is something that could be updated manually but preferably should have an automated solution.

6.2.3 Deploying the service

The deployment of the GRA is an absolutely necessary part of it becoming a viable SaaS. A lot of preparatory work has been done for this, using Docker Compose to package each program into its own runtime environment. The next step would be to upload the docker images to a cloud-based service that allows auto-scaling based on traffic.

6.2.4 Improvements to GRA Server

In its current state, the GRA Server has four responsibilities consisting of simple rate limiting, acting as a client towards the ML application, acting as a server towards GRA Proxy, and acting as a client towards the Authentication Server authorizing tokens. These responsibilities are distributed between `GraServer` and `RateLimiter` where `GraServer` is closely tied to `TokenAuthorizer` and `RateLimiter` is closely tied to `MachineLearningClient`. This makes sense in the current flow of GRA but does not allow for adding functionality common to all Classes. A prime example of this is the `publish` method, broadcasting a decision made by either `RateLimiter` or the ML application to all GRA Servers. Furthermore, it limits the ability to customize GRA based on customer needs. The solution to this would be to implement a flow control Class that owns the instance of each responsibility-defining Class.

6.2.5 Testing

The GRA has been tested by using .txt-files as input data to a `cURL` script and logging the results in the respective containers. A large improvement to this would be a more standardized approach such as unit tests. This would allow for a simpler, more automatized testing environment when applying changes to the code base. Furthermore, the performance of the GRA has not been tested to its fullest extent. In the Results chapter, a test spamming the system with 20000 requests was described. However, the upper performance limit of the GRA with the available CPU cores has not been tested.

7
Conclusion

As mentioned in the introduction the report, the original problem was reconstructing a proof of concept to a SaaS. This chapter aims to connect all aspects of the report, and conclude how this problem was solved.

Firstly, the major achievement of this project was going from a proof of concept to a nearly achieved SaaS, which is horizontally scalable. The initial starting point was an untested, unreliable project which would not have fulfilled any particular purpose in a production environment. The final result is a group of applications, each single-threaded and fully asynchronous, with functional docker images together being able to perform the duty of a rate limiter with reliable results in various situations.

Secondly, the code base at large follows Java conventions in terms of Class dependencies, meaningful and consistent methods and variable naming. Furthermore, each function performs a relatively small and specific task. What is accomplished by doing this is making every function as adaptable as possible, allowing easier implementation of new functionality. Key variables in each Class have been identified and purposely set as global variables to adjust easier with the implementation of a customer application. Above Classes, each application has been implemented to function independently, communicating only with HTTP and WebSockets, allowing a single application to be replaced by another filling the same purpose. For example, the Authentication Server could be changed for another similar service if outsourcing security would be desirable. We conclude that the code base and architecture of the GRA would be a valuable asset for any party interested in developing a similar product.

Finally, it is worth expressing that this project could have been achieved with a lot less complexity if the number of requests are kept to a fairly small amount. This SaaS is designed to handle traffic from a large number of sources and not from a single application or even several. The number of requests kept in mind while developing, specified by the product owner was 2 million per minute, which is very uncommon for a single application. The GRA is a service that is estimated to manage this with enough CPU cores available. Hence, even though the project did not reach its fully intended scope as a degree project, the quality of the completed parts compensates for this according to us.

Bibliography

[1] J. D'Hoinne, R. Kaur, J. Watts, and A. Hils. (2022) Cloudflare named a leader in the gartner® magic quadrant™ for web application and api protection (waap). [Online; accessed 25-May-2023]. [Online]. Available: https://www.cloudflare.com/lp/gartner-magic-quadrant-waap-2022/

[2] E. Berzelius, L. Bagiu, V. Hagenbo, L. Rosin, E. Näslund, and C. von Schenck, *Gateway Request Analyzer*, 2022.

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. (2001) Principles behind the agile manifesto. [Online; accessed 24-May-2023]. [Online]. Available: https://web.archive.org/web/20100615235054/http://agilemanifesto.org/principles.html

[4] H. S. Sverrisdottir, H. T. Ingason, and H. I. Jonasson, "The role of the product owner in scrum-comparison between theory and practices," *Procedia-Social and Behavioral Sciences*, vol. 119, pp. 257–267, 2014.

[5] J. Ponge, *Vert. x in Action: Asynchronous and Reactive Java*. Manning Publications, 2020.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc2616: Hypertext transfer protocol–http/1.1," 1999.

[7] H. Zimmermann, "Osi reference model-the iso model of architecture for open systems interconnection," *IEEE Transactions on communications*, vol. 28, no. 4, pp. 425–432, 1980.

[8] W3TECHS. (2023) Usage statistics of default protocol https for websites. [Online; accessed 25-May-2023]. [Online]. Available: https://w3techs.com/technologies/details/ce-httpsdefault

[9] OAuth0. (2023) Json web tokens. [Online; accessed 25-May-2023]. [Online]. Available: https://auth0.com/docs/secure/tokens/json-web-tokens

[10] ——. (2023) What is oauth 2.0? [Online; accessed 25-May-2023]. [Online]. Available: https://auth0.com/docs/secure/tokens/json-web-tokens

[11] D. Hardt. (2012) The oauth 2.0 authorization framework. [Online; accessed 25-May-2023]. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6749

[12] Redis. (2023) Introduction to redis. [Online; accessed 19-June-2023]. [Online]. Available: https://redis.io/docs/about/

[13] S. Tarkoma, *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012.

[14] R. contributors. (2023) Scaling with redis cluster. [Online; accessed 25-May-2023]. [Online]. Available: https://redis.io/docs/management/scaling/

[15] Docker. (2023) What is a container? [Online; accessed 25-May-2023]. [Online]. Available: https://docs.docker.com/get-started/
[16] S. Grunert. (2019) Demystifying containers - part i: Kernel space. [Online; accessed 25-May-2023]. [Online]. Available: https://medium.com/@saschagrunert/demystifying-containers-part-i-kernel-space-2c53d6979504
[17] Docker. (2023) Docker compose overview. [Online; accessed 25-May-2023]. [Online]. Available: https://docs.docker.com/compose/
[18] S. Chacon and B. Straub, *Pro git*. Springer Nature, 2014.
[19] D. Tools. (2023) Unix time stamp. [Online; accessed 27-May-2023]. [Online]. Available: https://www.unixtimestamp.com

A

Appendix 1

```
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/login
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/users
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/chat/839
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/chat/576
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/chat/098
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/chat/354
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/send
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/amazingcatfact
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/somedogfacts
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/gardening
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/beerenthusiast
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/login
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/gardening
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/send
-H 'userId: normal' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/login
```

Figure A.1: Test data emulating a normal user

```
-H 'userId: spammerboi' -H 'session: catnet' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://10.0.10.17:7890/172.20.0.40:8081/send
```

Figure A.2: Test data emulating a malicious user, using the same search query, user, IP address and session

```
-H 'userId: spammer' -H 'session: catspam1' -H 'ip_address: 1.3.2.2' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/login
-H 'userId: spammer' -H 'session: catspam1' -H 'ip_address: 1.3.2.2' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/users
-H 'userId: spammer' -H 'session: catspam1' -H 'ip_address: 1.3.2.2' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/839
-H 'userId: spammer' -H 'session: catspam1' -H 'ip_address: 1.3.2.2' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/576
-H 'userId: spammer' -H 'session: catspam1' -H 'ip_address: 1.3.2.2' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/098
-H 'userId: spammer' -H 'session: catspam1' -H 'ip_address: 1.3.2.2' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/354
-H 'userId: spammer' -H 'session: catspam2' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/send
-H 'userId: spammer' -H 'session: catspam2' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/amazingcatfact
-H 'userId: spammer' -H 'session: catspam2' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/somedogfacts
-H 'userId: spammer' -H 'session: catspam2' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/gardening
-H 'userId: spammer' -H 'session: catspam2' -H 'ip_address: 1.3.1.9' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/beerenthusiast
-H 'userId: spammer' -H 'session: catspam3' -H 'ip_address: 1.3.3.1' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/login
-H 'userId: spammer' -H 'session: catspam3' -H 'ip_address: 1.3.3.1' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/gardening
-H 'userId: spammer' -H 'session: catspam3' -H 'ip_address: 1.3.3.1' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/send
-H 'userId: spammer' -H 'session: catspam3' -H 'ip_address: 1.3.3.1' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/login
```

Figure A.3: Test data emulating a malicious user, using different search queries and different sessions & IP addresses

```
-H 'userId: abc122' -H 'session: abcder' -H 'ip_address: 1.2.3.5' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/login
-H 'userId: tpr859' -H 'session: abcytf' -H 'ip_address: 1.2.3.1' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/users
-H 'userId: pog123' -H 'session: hpeder' -H 'ip_address: 1.2.3.2' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/839
-H 'userId: igr345' -H 'session: jdport' -H 'ip_address: 1.2.3.3' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/576
-H 'userId: mr.POG' -H 'session: kluirt' -H 'ip_address: 1.2.3.4' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/098
-H 'userId: dwu312' -H 'session: cmyrte' -H 'ip_address: 1.2.3.6' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/chat/354
-H 'userId: blt343' -H 'session: swpert' -H 'ip_address: 1.2.3.7' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/send
-H 'userId: psft35' -H 'session: zwedrt' -H 'ip_address: 1.2.3.8' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/amazingcatfact
-H 'userId: wsr231' -H 'session: bmtrpt' -H 'ip_address: 1.2.3.0' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/somedogfacts
-H 'userId: mlk890' -H 'session: dutrof' -H 'ip_address: 1.2.4.5' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/gardening
-H 'userId: jug764' -H 'session: clgtpr' -H 'ip_address: 1.2.5.5' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/beerenthusiast
-H 'userId: ikt709' -H 'session: eu7gfd' -H 'ip_address: 1.2.6.5' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/login
-H 'userId: bogor1' -H 'session: zeprtt' -H 'ip_address: 1.2.7.5' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/gardening
-H 'userId: berzan' -H 'session: ccffgg' -H 'ip_address: 1.2.8.5' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/send
-H 'userId: slydog' -H 'session: wdefte' -H 'ip_address: 1.2.9.5' -H 'content-type: application/json' -X GET http://172.30.112.1:7890/172.20.0.40:8081/login
```

Figure A.4: Test data emulating randomized requests, used for testing system capacity

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY