INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Distributed Mail Transfer Agent

JOÃO PEDRO DE SÁ CARDOSO DOS SANTOS
Outubro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de
Engenharia do Porto

[Distributed Mail Transfer Agent]

João Pedro de Sá Cardoso dos Santos

**A dissertation submitted in partial fulfillment of
the requirements for the degree of Master of Science,
Specialisation Area of Computer Systems**

Supervisor: Dr. Jorge Manuel Neves Coelho
Co-Supervisor: Dr. Ivo Pereira

Porto, October 13, 2020

Abstract

Technological advances have provided society with the means to easily communicate through several channels, starting off in radio and television stations, moving on through E-mail and SMS, and nowadays targeting Internet surfing through channels such as Google Ads and Webpush notifications. Digital marketing has flooded these channels for product promotion and customer engaging purposes in order to provide the customers with the best the organizations have to offer.

E-goi is a web platform whose main objective is to facilitate digital marketing to all its customers, ranging from SMB to Corporate/Enterprise, and aid them to strengthen their relationships with its customers through digital communication. The platform's most widely used channel is E-mail which is responsible for about fifteen million deliveries per day. The e-mail delivery system currently employed by E-goi is functional and fault-tolerant to a certain degree, however, it has several flaws, such as its monolithic architecture, which is responsible for high hardware usage and lack of layer centralization, and the lack of deliverability related functionalities.

This thesis aims to analyze and improve the E-goi's e-mail delivery system architecture, which represents a critical system and of most importance and value for the product and the company. Business analysis tools will be used in this analysis to prove the value created for the company and its product, aiming at maintenance and infrastructure cost reduction as well as the increment in functionalities, both of which comprise valid points for creating business value.

The project main objectives comprise an extensive analysis of the currently employed solution and the context to which it belongs to, followed up by a comparative discussion of currently existent competitors and technologies which may be of aid in the development of a new solution. Moving on, the solution's functional and non-functional requirements gathering will take place. These requirements will dictate how the solution shall be developed. A thorough analysis of the project's value will follow, discussing which solution will bring the most value to E-goi as a product and organization. Upon deciding on the best solution, its design will be developed based on the previously gathered requirements and the best software design patterns, and will support the implementation phase which follows. Once implemented, the solution will need to surpass several defined tests and hypothesis which will ensure its performance and robustness. Finally, the conclusion will summarize all the project results and define future work for the newly created solution.

Keywords: Distributed, Scalable, Parallel, E-mail, Delivery

Resumo

O avanço tecnológico forneceu à sociedade a facilidade de comunicação através dos demais canais, começando em rádios e televisões, passando pelo E-mail e SMS, atingindo, hoje em dia, a própria navegação na Internet através dos mais diversos canais como o *Google Ads* e notificações *Webpush*. Todos estes canais de comunicação são hoje em dia usados como base da promoção, o marketing digital invadiu estes canais de maneira a conseguir alcançar os mais diversos tipos de clientes e lhes proporcionar o melhor que as organizações têm para oferecer.

A E-goi é uma plataforma web que pretende facilitar o marketing digital a todos os seus clientes, desde a PME à *Enterprise*, e ajudá-los a fortalecer as relações com os seus clientes através de comunicação digital. O canal mais usado da plataforma é o E-mail, totalizando, hoje em dia, cerca de quinze milhões de entregas por dia. O sistema de envio de e-mails usado hoje em dia pelo produto E-goi é funcional e tolerante a falhas até um certo nível, no entanto, apresenta diversas lacunas tanto na arquitetura monolítica do mesmo, responsável por um uso de hardware elevado e falta de centralização de camadas, como em funcionalidades ligadas à entregabilidade.

O presente projeto visa a análise e melhoria da arquitetura do sistema de envio de e-mails da plataforma E-goi, um sistema crítico e de alta importância e valor para a empresa. Ao longo desta análise, serão usadas ferramentas de análise de negócio para provar o valor criado para a organização e para o produto com vista à redução de custos de manutenção e infraestrutura bem como o aumento de funcionalidades, ambos pontos válidos na adição de valor organizacional.

Os objetivos do projeto passarão por uma análise extensiva da solução presente e do contexto em que a mesma se insere, passando a uma comparação com soluções concorrentes e tecnologias, existentes no mercado de hoje em dia, que possam ajudar no desenvolvimento de uma nova solução. Seguir-se-á um levantamento dos requisitos, tanto funcionais como não-funcionais do sistema que ditarão os moldes sobre os quais o novo sistema deverá ser desenvolvido. Após isto, dar-se-á uma extensa análise do valor do projecto e da solução que mais valor adicionará à E-goi, quer como produto e como organização. De seguida efectuar-se-á o *Design* da solução com base nos requisitos definidos e nas melhores práticas de engenharia informática, *design* este que servirá de base à implementação que se dará de seguida e será provada através da elaboração de diversos testes que garantirão a performance, robustez e validade do sistema criado. Finalmente seguir-se-á a conclusão que visa resumir os resultados do projecto e definir trabalho futuro para a solução criada.

Acknowledgement

First of all, I would like to thank my parents that provided me with everything they could in order to accomplish this thesis and much more. A special thanks to those who made fun of me every day until the thesis submission but also endured the hardships of listening to my constant blabbering about it. The next acknowledgement goes to my second family, my friends who supported along the journey which culminated in this project. I would also like to thank my supervisors, Jorge Coelho and Ivo Pereira, for all the dedication, revisioning, advices, experience sharing and overall support they provided me with along the development of this project. Thanks to E-goi for providing me with the opportunity to develop this project and supplying me with knowledge and infrastructure to develop and deploy it. A special thanks to Pedro Pinto, a co-worker and friend, who often made himself available to discuss design and implementation details, and offer me his knowledge and advice regarding the thesis context. Last but not least, I would like to thank ISEP, for all the knowledge it supplied me along the Bachelor and Master degrees, without with I would be unable to develop this thesis.

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem and Motivation . . . 2
- 1.3 Objectives . . . 3
- 1.4 Methodology . . . 5
- 1.5 Document Structure . . . 5

2 Context 7
- 2.1 E-goi Platform . . . 7
  - 2.1.1 Basic Functionality . . . 7
  - 2.1.2 Subscriber Data, Reports and Reactivity . . . 8
  - 2.1.3 Integrations . . . 8
  - 2.1.4 Automation . . . 9
  - 2.1.5 E-mail Channel . . . 9
  - 2.1.6 Slingshot . . . 11
- 2.2 Technological Context . . . 11
  - 2.2.1 Domain Name System . . . 11
  - 2.2.2 Simple Mail Transfer Protocol . . . 12
  - 2.2.3 Email Authentication . . . 14
    - DomainKeys Identified Mail (DKIM) . . . 14
    - Authenticated Received Chain (ARC) . . . 15
  - 2.2.4 Deliverability . . . 15

3 State Of The Art 17
- 3.1 Mail Transfer Agent Solutions . . . 17
  - 3.1.1 Postfix . . . 17
  - 3.1.2 Qmail . . . 18
  - 3.1.3 PowerMTA . . . 19
  - 3.1.4 MailerQ . . . 21
- 3.2 MTA Technologies . . . 23
  - 3.2.1 Message Brokers . . . 23
    - ActiveMQ . . . 24
    - RabbitMQ . . . 26
    - Comparative Analysis . . . 27
  - 3.2.2 Compression Algorithms . . . 28
    - Comparative Analysis . . . 28

4 Analysis 31

4.1 Requirements Engineering . . . 31
4.1.1 Functional Requirements . . . 32
4.1.2 Non-Functional Requirements . . . 34
4.2 Value Analysis . . . 35
4.2.1 Innovation Process . . . 35
4.2.2 New Concept Development . . . 36
Opportunity Identification . . . 36
Opportunity Analysis . . . 36
Idea Generation . . . 38
Idea Selection . . . 38
Concept Definition . . . 43
4.2.3 Business Value . . . 43
Customer Value and Perceived Value . . . 43
Value Proposition . . . 44
Business Model Canvas . . . 44
Porter's Value Chain . . . 45

5 Design **49**
5.1 Architecture . . . 49
5.1.1 Components . . . 49
5.1.2 Deployment . . . 50
5.1.3 Queueing Structure and Model . . . 52
Message per campaign, queue per delivery group . . . 52
Message per delivery, queue per campaign . . . 53
Message per delivery, queue per delivery group . . . 54
Model Comparison . . . 54
5.2 Use Cases . . . 55
5.2.1 UC1 - Message Delivery . . . 55
5.2.2 UC2 - Scheduled Delivery . . . 57
5.2.3 UC3 - Message Priority . . . 57
5.2.4 UC4 - DKIM Signature . . . 57
5.2.5 UC5 - CC and BCC Support . . . 58
5.2.6 UC6 - Delivery Logging . . . 59
5.2.7 UC7 - Delivery Status Notifications . . . 60
5.2.8 UC8 - Group Delivery Processes . . . 61
5.2.9 UC9 - Delivery Policies . . . 61
5.2.10 UC10 - Stop/Pause/Resume Deliveries . . . 61
5.2.11 UC11 - Submission Reports . . . 62
5.2.12 UC12 - DKIM Key Management (Extra) . . . 62

6 Implementation **65**
6.1 Technological Environment . . . 66
6.2 Abstraction Layers . . . 67
6.3 Implementation Details . . . 69
6.3.1 UC1 - Message Delivery . . . 70
SMTP API . . . 70
SMTP Client . . . 71
6.3.2 UC2 - Scheduled Delivery . . . 72
6.3.3 UC3 - Message Priority . . . 72

6.3.4 UC4 - DKIM Signature 72
6.3.5 UC5 - CC and BCC Support 73
6.3.6 UC6 - Delivery Logging 73
6.3.7 UC7 - Delivery Status Notifications 74
6.3.8 UC8 - Group Delivery Processes 74
6.3.9 UC9 - Delivery Policies 74
6.3.10 UC10 - Stop/Pause/Resume Deliveries 75
6.3.11 UC11 - Submission Reports 75
6.3.12 UC12 - DKIM Key Management (Extra) 76

7 Experimentation and Evaluation 77
7.1 Hypothesis Enunciation 77
7.2 Hypothesis 1 and 2 77
7.2.1 Methodology 77
7.2.2 Results 78
7.3 Hypothesis 3 80
7.3.1 Methodology 80
7.3.2 Results 80
7.4 Hypothesis 4 80
7.4.1 Methodology 80
7.4.2 Results 81
7.5 Hypothesis 5 83
7.5.1 Methodology 83
7.5.2 Results 83
7.6 Hypothesis 6 85
7.6.1 Methodology 85
7.6.2 Results 85

8 Conclusions 87
8.1 Achievements 87
8.2 Future Work 88

A Compression Results Raw Data 91

List of Figures

2.1 E-goi Delivery Workflow Deployment Model 10
2.2 Single Deliveries Delivery Workflow Deployment Model 11
2.3 SMTP session sequence diagram 13

3.1 Postfix Delivery Workflow 18
3.2 Qmail Delivery Workflow 18
3.3 PowerMTA Architecture 20
3.4 MailerQ Queue Architecture 22
3.5 Compression time for the best algorithm-level pairs (lower is better). 29
3.6 Compression ratio for the best algorithm-level pairs (higher is better). 29
3.7 Compression rating for the best algorithm-level pairs (higher is better). . . . 30

4.1 SWOT Analysis 37
4.2 Analytic Hierarchy Process Hierarchy 39
4.3 Business Model Canvas 45
4.4 Porter's Value Chain 46

5.1 Solution's Components Model 49
5.2 Solution's minimal deployment 50
5.3 Solution's multi-instance deployment 51
5.4 Solution's high-availability deployment 52
5.5 Solution's Delivery Sequence Diagram 55
5.6 Solution's Delivery Logging handling Sequence Diagram 59
5.7 Solution's Delivery Status Notification handling sequence diagram 60

6.1 Compression algorithms abstract layer 68
6.2 Message broker connection abstract layer 68
6.3 Message broker producer abstract layer 68
6.4 Message broker consumer abstract layer 69
6.5 Delivery logs handler representation 70

7.1 Memory usage test case results (lower is better). 81
7.2 Memory usage test case results without Artemis (lower is better). 82
7.3 *Sonarqube* code quality analysis graph 86
7.4 *Sonarqube* code duplication analysis graph 86

List of Tables

3.1 MailerQ Pricing Options . . . 23

4.1 Criteria comparison and relative priority matrix. . . . 39
4.2 Priority Matrix for the "Maintainability" criteria. . . . 40
4.3 Priority Matrix for the "Hardware Recycling" criteria. . . . 40
4.4 Priority Matrix for the "Financial Investment" criteria. . . . 41
4.5 Criteria relative priority matrix. . . . 41
4.6 Solutions' relative priority matrix. . . . 41
4.7 Solutions' global weight matrix. . . . 42
4.8 Consistency formula matrix. . . . 42
4.9 Randomness Index Table. . . . 42

6.1 Python native and third-party libraries used in the solution's development. . 67

7.1 *Jmeter* load testing scenarios. . . . 79
7.2 *Jmeter* load testing scenarios results. . . . 79
7.3 Global memory usage comparative table. . . . 82
7.4 Single delivery speed test results. . . . 84

A.1 LZ4 level 2 compression factors results. . . . 91
A.2 Gzip level 3 factors results. . . . 91
A.3 BZ2 level 9 factors results. . . . 91
A.4 LZMA compression factors results. . . . 92
A.5 LZO level 1 compression factors results. . . . 92
A.6 Brotli level 3 compression factors results. . . . 92
A.7 ZSTD level 4 compression factors results. . . . 92
A.8 Overall compression rating results. . . . 92

Chapter 1

Introduction

1.1 Context

The 4Ps of the Marketing Mix concept [1], consist in four variables which define marketing: Price, Product, Promotion and Place. From these variables, marketing can be defined as putting the right product, in the right place, at the right price and at the right time. Digital marketing arose from the need to improve one of these variables, Promotion.

Promotion is the advertisement of a product, in order to sell it to a customer, through the use of various means of advertising [2]. With the evolution of the technology, several other communication (and consequently, advertisement) channels were created, such as AM/FM radio and Television, which allowed organizations to get closer to their customers by having advertisements directly in their homes (radio and television) and vehicles (radio).

The digital age came along and with it many more communication channels emerged, from SMS to the social networks (i.e. Facebook, Instagram, etc), all the way through E-mail, Push and Webpush notifications. All these channels are nowadays used for advertisement and customer interaction channels and this is where the E-goi platform joins in. E-goi is an omni-channel digital marketing SaaS web platform which allows its users to create, design and send communications to their customers through several channels: E-mail, Voice, SMS, SmartSMS, Push, Webpush and, the latest at the time of this writing, Ads. Besides the previously mentioned capabilities, E-goi is also able to generate reports of the effectiveness of the communications sent and automate sendings and other actions based on flows defined by the user as well as integrating with any software you might have and much more.

E-mail is, at the time of this writing, the channel with most usage within the E-goi platform featuring about 13 million sendings per day, yet there are several challenges to be faced when it comes to sending massive quantities of emails. E-mail providers such as Gmail or Outlook/Hotmail monitor from where and where to e-mails are sent in order to prevent spam or other malpractices from reaching its customers mailboxes. Email deliverability is a digital marketing term which represents the ability to deliver emails to the intended mailbox [3]. Given the monitoring, previously mentioned, made by e-mail providers, care is required in order to increase and maintain the deliverability of the e-mail sending systems, otherwise, these systems could be throttled or even blocked by the e-mail providers which negatively impacts E-goi upcoming email sends, and consequently its customers. There are several ways to improve the deliverability of e-mails, such as implementing e-mail authentications such as DKIM [4] and ARC [5] and controlling the outbound e-mail flow.

Several entities are approached when dealing with e-mails, usually, one creates and sends an e-mail message using a Mail User Agent. The created e-mail message is sent to a Mail

Transfer Agent which will receive it, analyze its recipient and, if possible, deliver it to a Mail Delivery Agent. It is possible that once a MTA receives an e-mail message, it is not able to send it to the responsible MDA and so it sends it to another MTA, which will deliver it to a MDA or re-send it to another MTA, and so on, this process is called e-mail relaying and often occurs due to inter-network boundary systems or different recipient domain name resolutions. The MDA is responsible for managing mailboxes and delivering received e-mails into the correct mailbox, as well as retrieving them for a Mail Retrieval Agent which occurs when one accesses its mailbox. Usually, the MRA and MUA are implemented in the same application such as Thunderbird [6] or Gmail Webmail [7] as these allow one to both create and send e-mail messages.

According to RFC5321 [8], a MTA is a software with the capabilities of both a SMTP server and SMTP client, which means it is able to both send and receive e-mail messages. In order to send its customers e-mails, E-goi uses an architecture based on the Postfix MTA [9] as its e-mail sending system, it is a sound approach and features a good performance, yet, it has several integration problems, decentralized logging, a monolithic structure, high hardware requirements and lacks a great amount of functionality.

1.2 Problem and Motivation

The motivation for this thesis rests on E-goi's e-mail channel sending system architecture, which currently provides a low amount of functionality, high hardware resource usage, decentralized logging data and overall structure, as well as a laborious setup which is very time-consuming when horizontally scaling. The current architecture is based on the Postfix MTA [9], a very well known general-purpose open-source MTA initially released in 1998 under the IBM Public License [10] which has since received incremental patches and stable releases.

E-goi's current e-mail channel sending architecture is based on text files generated at the moment of the campaign's processing copied over SSH [11], into the assigned sending server (or servers), which contain all the required parameters for the sending to take place. In order to send the emails, the Postfix MTA is used. The previously mentioned files are parsed and e-mail messages are built and sent over SMTP into local Postfix instances which will then sign these messages with the DKIM and ARC authentication mechanisms using milters [12] such as Openarc [13] and Opendkim [14], and relay these messages into the internet and eventually, the designed recipient's mailbox.

The current architecture is based on the replication (horizontal scaling) of a monolith containing several postfix instances bounded to different IP addresses, one instance of each milter (Openarc and Opendkim) which are shared by the several postfix instances and several processes which process the text files create and send the e-mail messages into the local postfix instances. It was implemented due to its ease of implementation and to be able to hit the market fast enough with a minimum viable product, yet, it is not suited to the E-goi requirements due to several reasons. First of all, the monolithic architecture does not allow for hardware specialization, each instance of the monolith requires well-performing disks to improve postfix speed as this uses the disk to write and read the messages to send (the in-disk queue concept is approached in 3.1.2). Furthermore, Postfix spawns several processes in order to send and receive e-mail messages, which requires several CPU cores to allow parallelism. It would be ideal for the disk requirements to (for example) be separated from the memory/CPU core requirements, which would allow the hardware to be specialized

for the several parts of the system. This specialization reduces the overall system cost and allows for investment in the best of each component.

The text file generation method, although simple and allowing bulk processing on the sending server, scatters this transactional data through all the sending servers and the bulk processing is not thread-safe, which does not allow parallelism, as the processes are not synchronized to be able to access the same text file without repeating sends, when sending these messages into the local Postfix instance. Furthermore, deploying a postfix instance and having it relaying e-mail messages onto the designed recipient instead of sending the e-mail directly onto the recipient represents a performance loss.

In order to digitally sign the messages with DKIM and ARC authentication, a milter is used, each postfix instance will, for each e-mail message, send it through the Opendkim and Openarc milters which will modify the e-mail message by adding headers which represent the digital signature for each of the authentication methods, these headers should later be validated by the receiving MTAs. The forwarding of these messages to the milters occurs via TCP which causes latency and since several postfix instances share only one instance of a milter, which may cause a bottleneck where performance is lost due to a milter overload. Besides this, the milters load all DKIM/ARC private keys into memory on startup, which requires a large amount of RAM independent of the keys which are going to be used, which increases the more keys there are. Along the milters process runtime, the allocated memory increases (due to overload or memory leaks) and so, in order to maintain the server stability, a monitoring service is responsible for restarting these processes once a certain memory threshold is breached. This restart prevents memory overflow, but it also stops several deliveries as the postfix instances are unable to communicate with the milter shared instances while they're being restarted.

Lastly, the overall system lacks several features and functionality which would provide improved overall e-mail deliverability, ease of management and setup, log indexing, data visualization, stronger and easier integration and lower hardware requirements, and consequently, lower hardware costs. It is not possible to control the outbound flow of e-mail messages which damages deliverability of the sending servers. Logging data is not centralized and is logged in text files which makes searching and indexing harder, besides this, this decentralization only allows keeping logs from the last 3 days of sendings, which sometimes is not enough, due to disk usage limits, for irrefutability of the status for a certain delivery when this is required by some entity. Deliverability dashboards with regards to the latest sendings and ISP responses to these sendings are impossible in a near-realtime time window which further damages e-mail deliverability. Integrating through SSH connections and text files requires to much processing on the client side when compared to simple HTTP requests. Stopping, pausing and resuming a sending is not possible, which prevents the E-goi team from taking administrative action when unwanted campaigns are being sent through their system or when customers want to stop a sending due to a mistake.

1.3 Objectives

The main objective of this thesis is to, through the best practices of software engineering, create a solution which best solves the problems presented in the Motivation section. It is of overwhelming importance to fully understand the concept of what the solution must do, how it should interact with the internet world and other systems and how it will be managed which will improve the final design of the new solution. Furthermore, the current solution must be

studied in detail to identify its flaws and address these issues in the new solution. Provided this, a definition of solutions for each problem, be these architectural or technological, should be researched and tested which will lead to the design of a new architecture that resolves the identified flaws and problems and is easily scalable and integrable within the E-goi platform. Having defined the architecture, its implementation is to be detailed in smaller and easier tasks, thus abiding to the "Divide and Conquer" principle, which will makeup a planned road map to be meticulously followed.

Post implementation, a battery of tests will be executed in order to ensure the robustness, performance and good behavior of the newly created solution for several use cases and provided the solution passed the tests, the next step will be to integrate the system in a staging environment, to test said integration with the E-goi platform, evaluate behavior and identify possible problems or erroneous behaviors outside production environment where these could cause greater damage. Taking into account the large deployment of the current architecture already present in the E-goi platform, a migration plan will need to be developed in order to migrate from the current solution to the new one through incremental stages which will provide a safer and more controlled solution release onto production environment.

Such a system is crucial to the well-being of the company since it represents the last software layer before the e-mail messages are effectively delivered, and so, it should, above all, thrive to provide high-availability and fault-tolerant behaviors as well as abstract and well performing integration interfaces to be able to integrate with different systems and technologies, which will improve its value to external entities if such is of interest and slow the integrating systems the lowest amount of time possible.

As previously mentioned, in section 1.1, e-mail providers usually monitor incoming e-mail messages sources for spam or improper behavior and have reputation systems which grade the sources of incoming e-mails, such as IP addresses, IP address ranges and sending server domain. Deliverability practices dictate that lower reputation sources should throttle e-mail deliveries in order to build reputation in the recipient's e-mail provider systems [3, 15]. The new solution should provide e-mail deliverability improvements by supporting e-mail message DKIM authentication, taking into account that several sender domains will be used, as well as providing outbound e-mail message flow control which will allow for message throttling by recipient domain for each IP address used when sending.

Since the current solution's setup and configuration management is very laborious and time-consuming, the new solution will make it easier by providing deployment and setup tools which should ease the deployment of new instances when horizontally scaling as well as configuration changes over several instances. In order to provide administrative actions on sendings already taking place, the new system will allow identifying messages or groups of messages and defining actions to be taken against these messages (or groups of messages) such as pausing their delivery, resuming it, or even discarding it.

Through the centralization of deliveries logging, the new solution should provide an easily searchable delivery log with greater capability for historical data keeping which will help the deliverability team in its work when analyzing the overall deliveries statistics thus providing previously not available insights which may improve the system and, consequently, the deliverability of future e-mail messages. Furthermore, this historical data warehousing will be the base for building dashboards and extra monitoring tools to further improve the system and provide extra visibility into one of the most used channels in the E-goi platform.

The hardware requirements for the current system are, very high and unspecialized, as the monolith makes use of all the hardware (*i.e.* CPU, RAM, Disk), the centralization of several key layers of the new solution will allow the system to specialize the hardware for each layer and lower these requirements. Furthermore, since the current solution e-mail message signing holds several memory and performance bottleneck problems, the resolution of these problems will also be a key objective which will also contribute to lowering the overall hardware requirements in the signature layer. Improving the current delivery performance should also be an objective as it will also reduce the hardware requirements, using less hardware to do the same, or even more, work.

1.4 Methodology

In order to fulfill the objectives defined in section 1.3, and consequently, create a new solution, the thesis will follow a development methodology based on a set of guidelines, defined in *"Design science research in information systems"* [16], which are specifically suited for the development of information systems.

Starting off with the definition and an in-depth study of the context of the thesis, in order to best comprehend the objective of the current solution and the need for a new one, through a motivation and objective determination, fulfills guideline 2, *"Relevance Of The Problem"*, while the guideline 6, *"Design as a Search Process"*, is abided, through the building of a thorough state of the art by comparing several existing partial solutions and technologies which may be of aid in the development of a new solution.

Moving on to an analysis of the requirements for the new solution, providing a business value analysis based on rigorous decision making techniques, such as the AHP method [17] and the Porter's Value Chain [18] , as well as building a robust test suite for the evaluation of the developed solution will comply with guideline 5, *"Research Rigo"*.

Once the business value has been validated, and based on the decisions made in the analysis process, the design of a new solution will be developed following the best software engineering patterns and practices which will be documented as artifacts through the use of diagrams, in UML language [19], while abiding to the guideline 1, *"Design as Artifact"*.

The implementation will take place respecting the described design and applying the best practices of software engineering. Such implementation will be thoroughly tested and evaluated in order to assert if it complies with all the non-functional requirements and solves all the problems related to the previous solution, thus fulfilling guideline 3, *"Design Evaluation"*.

1.5 Document Structure

The present document is divided amongst several chapters which confine several sections, subsections and, at times, sub-subsections which provide an incremental specification for the theme being discussed in each. The chapters are as follow:

Introduction: This chapter provides a summed up context, by lightly approaching marketing and digital marketing, the E-goi company and platform, the e-mail channel, a little about how it generally works and the entities involved and lastly, a definition of Mail Transfer Agent is provided. Post contextualizing, comes the definition of the motivation for this thesis and objectives for the solution are determined. Finally, a methodology for the development of the thesis is presented.

Context: In this chapter several concepts, entities and technologies will be explained in order to provide the reader with required knowledge and understanding for the next chapters and decisions presented.

State of the Art: This chapter confines a descriptive presentation of several possible partial solutions and technologies which may help in the development of a solution and places them side by side in order to provide a comparative analysis and discussion of all of them.

Analysis: This chapter is where the requirements engineering will be presented, the thesis will be defined through the New Concept Development Model [20] and a discussion will be held in order to decide, based on several decision making techniques and the state of the art, the best solution for the thesis. Lastly, a business value analysis is provided in order to prove the need and viability for the development of the project as requested.

Design: In this chapter, several architectures are developed and studied in order to find the one which best solves the problems of the previous architecture and which is more adaptable to the development of new functionalities and, of course, requires less maintenance.

Implementation: This chapter features most of the technical details through which the implementation of the project was possible and the methodology used in the development of such.

Experimentation and Evaluation: In this chapter, through definition several experiments and evaluation of their results, the implemented solution will be proven as successful or not in solving the problems which motivated the thesis.

Conclusions: In this chapter, conclusions relative to the status of the thesis are presented, as well as problems that delayed some developments and future work which may be done to further improve the created solution.

Chapter 2

Context

2.1 E-goi Platform

E-goi is portuguese company founded in 2000 by Miguel Gonçalves which is, at the time of this writing, overcoming the 100 employees barrier and whose mission is to create effective digital marketing communications, based on a fun and intuitive usability, accessible to all who wish to achieve the best results be it a micro-blogger or the giant multinational.

The following sections will grasp the concept of the E-goi platform and explain several functionalities it provides for its customers and which are of relevance to the work being presented in order to contextualize the thesis as a product for the company.

2.1.1 Basic Functionality

E-goi is a very complex platform in terms of functionalities offered, as previously mentioned in section 1.1, the platform is a hub for digital marketing, it allows designing and sending of campaigns for several channels of communication as well as marketing automation and integration with several platforms.

In order to accomplish this, one must first acquire contacts to which to send. This can be done by simply creating a list of contacts and importing a file with contact information such as e-mail address, phone number, webpush token, first name, last name and date of birth. Yet, not everyone has a well-established business and customers, and so, E-goi provides tools to create forms which can be published in a webpage in order to acquire contact information directly into a list in the platform. The created list can later be modified by adding new parameters or entries and segmented for contact grouping based on list parameters.

Campaigns are the content which will be sent via a chosen channel of communication and so, they are created for a specific channel such as E-mail, SMS, amongst others. E-goi provides several built-in campaign editors, which are front-end applications that allow the customer to easily create complex and content-rich campaigns from scratch, or based on templates created by the E-goi team, through drag-and-drop functionalities and HTML/CSS raw editing. It's worth mentioning that E-goi campaigns may feature dynamic content, which allows changing the content of the campaign based on the subscriber or group of subscriber it is being sent to.

Being an omni-channel platform, E-goi provides several channels of communication that allow the customer to deliver different types of content to its subscribers which may be more appropriate in different contexts. Taking the retailing market as an example, the e-mail channel may be the best approach if there is the need to deliver a weekly catalog of new products

when compared to SMS, as the first provides a much more visual approach. Yet, SMS may be best-suited when delivering a notification to a subscriber about a specific last-hour discount or warning of such. Once the communication channel is chosen and the campaign is created, the customers may send it through the platform, accompanying the development of the operation within an interface which provides detailed progress information.

Once the campaign delivery has finished, a basic reactivity report is generated and updated over time which, depending on the channel, will provide different data about interactions the subscribers triggered with the campaign. An e-mail channel basic report, for example, will provide data such as the opening rate, demonstrating how many subscribers opened the e-mail, the soft and hard bounce rate, meaning the rate of subscribers to which delivery failed due to temporary or permanent errors, respectively, and the click rate, the number of clicks in links in the campaign.

2.1.2 Subscriber Data, Reports and Reactivity

Reactivity is the term which comprises the interactions that a subscriber may trigger for a specific campaign, these usually comprise the delivery of the campaign, the campaign opening or visualization, a click on a campaign link, a subscription removal, and others which depend on the channel's capabilities.

Mentioned in the section 2.1.1 is E-goi's capability of generating reactivity reports for its customers as to provide results of campaigns which have been sent, this is due to the fact that from the moment a campaign is being sent, the application saves every reactivity interaction towards the campaign for each subscriber to which this has been sent. Basic reports are generated, and updated as more data is gathered, from the moment a campaign is sent which allows the customers to check which of their campaigns were best received by their subscribers, and adapt the next campaigns to the audience preferences, thus generating more interest.

Basic reports, are, nonetheless, basic, and some customers require a more in-depth analysis of its campaigns, in order to fulfill this need, E-foi provides its customers with advanced reports, allowing them to create report templates to output any parameters of its choosing, grouping the results by any statistical value or reactivity interaction and applying these reports to entire list of contacts or segments of said lists. These templates can then be manually executed, or a recurrence may be defined for them, so that, periodically, the report is executed and the results are automatically delivered to the customer.

2.1.3 Integrations

Being a SaaS, E-goi provides itself in several ways to its customers, a back office allows them to visually, simply and interactively make use of it, yet this is not suited to all customers and in order to bridge this gap E-goi also provides a web API, E-commerce store plugins and a web tracking system designated as *Track&Engage*.

The web API is provided in two versions, version 2 is the oldest and does not reflect the web API software patterns nowadays implemented, such as REST [21], which was the main reason for the creation of version 3, which reflects the REST architectural style and is currently being actively developed, documented and distributed as SDKs to support easier integration through several programming languages.

In order to provide direct integration with E-commerce platforms such as Wordpress [22] and Magento [23], plugins for these platforms are developed which allows owners of such to easily integrate their E-commerce business with the E-goi platform and communicate with their customers using E-goi seamlessly.

Track&Engage is the web tracking tool provided by E-goi to allow its customers, owners of a website, to deploy a system capable of tracking all the actions performed by the visitors of said website. All these actions are saved and processed for later report generation and provide a founding layer for a fully automated experience based on E-goi which will be referred in the next section.

2.1.4 Automation

In order to further facilitate and improve its customers experience, E-goi provides several automation capabilities such as recurrent imports which allow a user to define a location (*i.e.* SFTP server or HTTP request) from where to download a file, a time period for this to occur and a list to where to import the contacts and so, respecting the defined time period, the system will download the file and import its contents into the defined list. This functionality is especially useful for customers who want to integrate new subscribers into E-goi without actively developing an integration through the web API.

Similarly to recurrent imports, recurrent campaigns are possible, for example, through an RSS feed, when a new article is published on the RSS, E-goi will download the HTML code from the RSS link and send it to the configured list of contacts automatically. It is also possible to configure an URL, a recurrence period (*i.e* daily, weekly, monthly) and a list of contacts, based on which E-goi will recurrently retrieve HTML from the configured URL and automatically send it to the configured list of contacts.

Yet, when it comes to automation nothing compares to *Autobots*, these are logical flows which a customer may create in order to automate almost any E-goi related action, these are composed by triggers and actions and are configured for a list of subscribers. Triggers define a condition for a subscriber to start following the defined flow, these can be a time period, a new subscriber in the list (*i.e.* through a form submit or import operation), a Track&Engage event for a subscriber in the list and so many more. Actions represent the flow, such as condition, which allow the evaluation of a condition and changing the flow according to the result of said evaluation, send campaign, which allow sending a campaign (through the several available channels) to the subscriber which is currently going through the autobot flow and tag, which allows adding a tag to the subscriber in the said list and many more.

Summing up and providing a real world example of what is designated as marketing automation, it is possible for an E-goi customer to set up Track&Engage on its E-commmerce website, and through an autobot, trigger the sending of a Webpush campaign to a visitor currently navigating the said website offering a discount in the next purchase.

2.1.5 E-mail Channel

E-goi's most used channel is, as mentioned in section 1.1, e-mail, totalling a sum of about 15 million deliveries per day, a number which is expected to grow as the business improves and more customers join E-goi. In order to send an e-mail campaign in E-goi, one must configure a list of subscriber e-mail contacts to which the campaign will be sent, a sender

name and the campaign to be sent which can be designed using one of the available editors provided by E-goi.

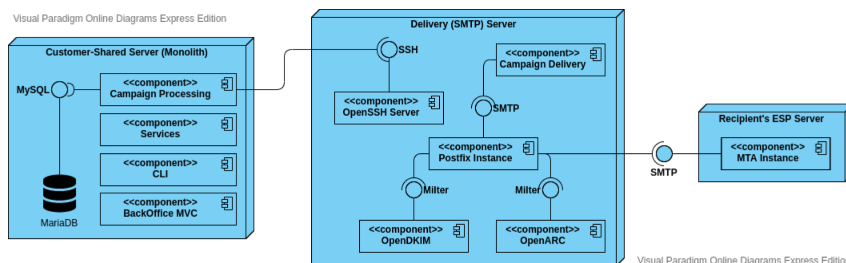

Figure 2.1: E-goi's e-mail delivery architecture deployment model.

Described in the figure 2.1 is E-goi's current email sending architecture deployment model where it is made clear the overall workflow of the campaign sending process, the protocols used to communicate between the different entities and the complexity of the customer-shared monolith server as well as of the sending server. It is important to notice that in each sending server, there are several postfix instances, not just one as represented in the diagram and that these sending servers are distributed amongst three different data-centers in different countries.

There are two phases of the campaign sending, the first one represents the processing of the campaign, which is generated in servers which are shared by several E-goi customers, this is where the campaign's dynamic content will be processed, tracking links will be generated and eventually, the text files required for the sending to take place will be generated and copied into the sending server. The second phase is the campaign delivery and this one takes place in the sending server which builds the e-mail messages and sends them into local postfix instances which will handle the rest of the delivery such as SMTP communication with the recipient's domain provider mail transfer agent, error handling, per recipient domain delivery thresholds and several performance improvements.

Campaigns can be scheduled to be delivered at a certain moment in time, however, what really happens is that the campaign processing (previously mentioned phase 1) is scheduled and when that moment is reached, the campaign processing starts. This is because the current solution is unable to schedule the delivery of a campaign, but only the processing of it.

At a certain moment in time, due to the *autobots* growing usage, E-goi began to have an increasing amount of campaigns targeting only one recipient, and albeit the system was performant enough to process fewer amount of campaigns for higher amount of subscribers, it was not performant enough to process a higher number of campaigns for a single subscriber. In order to fix this limitation, E-goi created an alternative delivery workflow for e-mail campaigns which targeted a single recipient.

This alternative delivery workflow is described in figure 2.2 and comprised some improvements on the processing layer, but the change which brought the most performance improvements was removing the SSH connection and campaign delivery processing from the overall workflow. The processing which was performed in the delivery server was moved onto the campaign processing, and instead of using an SSH connection to copy a file which would still be picked up for further processing and only then originate deliveries, the campaign processing connects via SMTP to one of the postfix instances present in the deliver container and delivers the email message for consequent relaying. Albeit this solution removed some

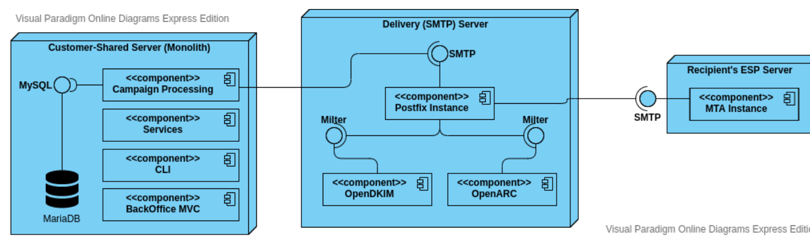

Figure 2.2: E-goi's e-mail delivery architecture deployment model for single deliveries.

abstraction from the delivery system, it provided the overall systems with a much larger performance and it was enough to solve the problem in dealing with large amounts of single delivery campaigns.

The thesis being developed will target the refactoring of the architecture and codebase from the SSH/SMTP connection to the delivery server and onwards.

2.1.6 Slingshot

Slingshot is the code name for E-goi's transactional delivery system which was created to fullfil a market need for customers which simply wanted to deliver e-mail messages without the need to create contact lists or follow the usual platform workflow. Providing an SMTP server entry-point to make the integration with a production ready project seamlessly easy and a REST API for other integration requirements, Slingshot became a fast and reliable transactional message delivery system and a valuable asset for E-goi.

Slingshot was developed as a completely separate system from the E-goi platform, even the technologies used were different, yet, the delivery system was the same as the E-goi platform. The transactional system uses the same communication patterns presented in section 2.1.5 by copying generated text files onto the delivery server for consequent delivery and so, it becomes prone to the same problems as the E-goi platform.

The Slingshot product has many more features and advantages and is being actively maintained and developed but what's most important to retain is that it uses the same delivery system as the E-goi platform, and as such, a successful results for this thesis will have to provide a seamless solution for two different products developed under different technologies with different architectures and purposes.

2.2 Technological Context

In the following sections, several technologies, and its relation to email delivery, will be presented in order to provide the reader with more detailed information for better understanding of the consequent chapters and decisions detailed throughout the document.

2.2.1 Domain Name System

Described by RFC-1035 [24], DNS is a worldwide, hierarchical and decentralized naming system which provides translation of more easily memorized domain names to numerical IP addresses required for locating and identifying computers in the internet. According to the RFC, the system is composed by two entities, resolvers and name servers. The resolver

is an entity which receives queries for information associated with a domain name from a user, thus allowing a user to, in example, ask for the host address of a domain name or mail information. The name servers are databases of domain names and their respective information, also known as records, and are consulted by the resolver entity. The resolver may consult several name servers yet, it should hide this complexity from the user. Resolvers are very common nowadays and implemented in almost every operative system as DNS has become a foundation for the internet as we know it.

For a single domain name, there may be multiple DNS records and these records may be of several types, such as A, NS, MX, CNAME, TXT, amongst others. These record types define different types of information for the domain name to which they are associated. The A record, is a record which translates into the host's IP address and an MX record translates into the domain name of the host where a mail transfer agent, for that domain name, lies. The CNAME record is no more than an alias to a domain name.

In order for an e-mail message to be sent, and according to the RFC-5321 [8], the resolution of the recipient's domain is of extreme importance, as the e-mail will be sent to the mail service located at the IP address of the recipient's domain name MX record, or, if non-existent, A record, or a CNAME record which, in turn, resolves into any of them. If the domain name is not resolvable than the e-mail sending will immediately fail, thus proving the importance of the domain name system for the e-mail channel.

2.2.2 Simple Mail Transfer Protocol

SMTP is the protocol which allows the transferring of e-mail messages from one computer to another, usually designated as client and server. In order to send an e-mail, a SMTP client opens a connection to a SMTP server and a SMTP session is began. Over the course of this session, the SMTP client and server will exchange data, such as the sending server hostname, the recipient name and the message data, in order for the SMTP server to be able to deliver the e-mail message to the intended recipient. The data exchange is achieved through the use of SMTP commands sent by the SMTP client and interpreted by the SMTP server which answers them through the use of codes and descriptive messages which will represent success or failure of the last received command.

Figure 2.3 represents an error free SMTP session through a sequence diagram where it is possible to visualize the complete communication between two entities exchanging an e-mail. These entities are represented as MUA or MTA, in this case, these are the SMTP client, and the MTA, which is the SMTP server. As per the RFC-5321, a SMTP client should begin a SMTP session by issuing the *HELO*, or *EHLO* if ESMTP is supported [8], command followed by its hostname to which the server will respond with a 250 response code, meaning the client may proceed to the next command. The *MAIL FROM* command is then executed, specifying the sender's address (and optionally its name), and the *RCPT TO* command follows it, by defining the recipient's address, both are followed by a 250 response code if no errors occur. It is now time for the *DATA* command, which the e-mail message is defined. The *DATA* command is executed and the SMTP server answers with a 354 response code, which means the client should proceed to send the message line by line, terminating it with a line containing only one dot and no other character. Once a line containing only one dot and no other character is read by the server, this answers with a 250 response code meaning the transaction of the previous e-mail message has been terminated and the e-mail is sent, from where the client will terminate the session by sending the *QUIT* command, receiving a 221 response code from the server and closing the connection.

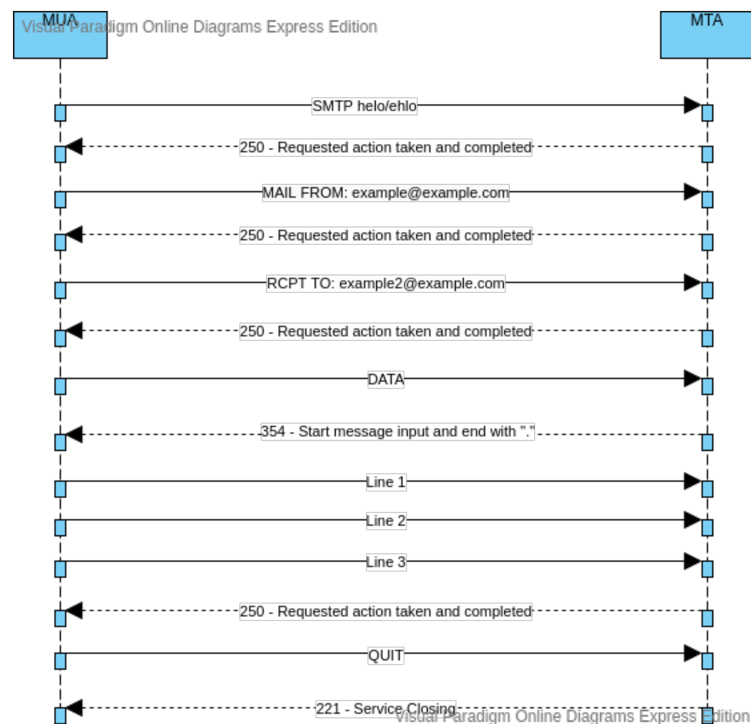

Figure 2.3: SMTP session sequence diagram.

SMTP responses (or replies) are composed by an integer code and a descriptive message whether they are error or success messages, and according to RFC-5321, there are four values for the first digit of the response code:

- 2yz - When the requested command was successfully completed.
- 3yz - When the requested command was accepted but the requested action is being held in advance for pending information. This information should be specified by the SMTP client through the use of more commands (*i.e.* the *DATA* command).
- 4yz - When the command was not accepted and the requested action was not executed. The error condition is temporary nonetheless, and the action may be requested again.
- 5yz - When the command was not accepted and the requested action was not executed. The error condition is permanent and the SMTP client should no retry the same command sequence.

It is also documented, in the RFC-5321, that the second digit of the reply codes is relative to the category in which the response fits. These are as follow:

- x0z Syntax - For syntax errors, such as when syntactically correct commands that do not fit any functional category and unimplemented or superfluous commands.
- x1z Information - For requests for information such as status or help requests.
- x2z Connections - For transmission channel related requests.
- x3z Unspecified.
- x4z Unspecified.

- x5z Mail System - For indicating the status of the receiver mail-system with regards to the request transfer or action.

These reply codes and eventually, the descriptive messages that follow them, allow the SMTP client to best understand its future behavior and provide insights on failed deliveries for deliverability experts and SMTP systems administrators.

E-mail relaying occurs when a SMTP server receives a message whose recipient domain is not resolvable (DNS) to itself, and so, the MTA will accept the message and now act as a SMTP client, by relaying the received e-mail message to the SMTP server to which the recipient's domain resolves to (which will now be different). E-mail relaying is very common yet, in order to conform to RFC-5321, a MTA does not need to be able to relay e-mail messages, it can simply deny the relaying. Nonetheless, if a message is accepted for relaying, and according to the RFC-5321, and the SMTP server later finds out that this message cannot be delivered, it must construct an undeliverable mail notification message and send it to the originator of the e-mail message, informing it of this fact.

2.2.3 Email Authentication

Due to the nature of email delivery and transferring, the messages may travel through several entities (MTAs and other systems) which are able, and also required, to modify them, for example, to add a "Received" header (see [8]) so that following systems know the message's path. Unfortunately, as these entities may modify messages as part of a required behavior, others may seek to change its contents for malicious reasons and such acts should be prevented.

Email authentication mechanism have been developed over the years in order to provide further irrefutability and resiliency to the messages delivered. Along this section we'll be discussing the DKIM and ARC email authentication methods in order to provide a stronger context and the importance of such methods for this thesis.

DomainKeys Identified Mail (DKIM)

As per RFC-6376 [25], *"DomainKeys Identified Mail (DKIM) permits a person, role, or organization that owns the signing domain to claim some responsibility for a message by associating the domain with the message."*. The DKIM authentication was created in order to irrefutably identify if an e-mail message came from the sender present in its headers. The need for this extra assurance comes from the fact that e-mail message's may be tinkered with as they travel through mail transfer agents until reaching its intended recipient inbox.

In order to use DKIM, one needs to configure a TXT DNS record for its sender domain containing the public key which pairs with the private key for the domain and the selector for that key as more can be added. From there, each message to be delivered can be authenticated with DKIM and will be validated by receiving mail transfer agents.

Authenticating an e-mail message with DKIM comprises adding a "DKIM-Signature" header to that same message before attempting to deliver it, the header value is computed and different for each message and is generated based on the private key for that sender's domain. When received by other MTAs, this signature will be validated, and if failed the delivery may be completely rejected or delivered into the spam folder. The signature is validated together with the public key present in the TXT record for the sender's domain.

Authenticated Received Chain (ARC)

ARC represents another email message authentication mechanism just like DKIM, but while DKIM provides irrefutability of the sender's entity, ARC authentication's purpose is to identify all the entities through which an email message has been relayed before being delivered to its intended recipient's inbox.

According to RFC-8617 [**RFC8617**], *"The Authenticated Received Chain (ARC) protocol provides an authenticated "chain of custody" for a message, allowing each entity that handles the message to see what entities handled it before and what the message's authentication assessment was at each step in the handling."*. As such, ARC is not only useful to provide information about which entities handled the message, but also about what message authentication results were obtained in the assessment of each of these entities.

The ARC signature, much like DKIM, is created by adding headers to the message before trying to deliver it, however, in the case of ARC authentication, a set of headers, designated as *ARC Set*, is appended to the message, they are the following:

- **ARC-Authentication-Results**: Contains the message authentication assessment proof as evaluated by the signing entity.
- **ARC-Message-Signature**: Containing a value which identifies the signing entity.
- **ARC-Seal**: Provides a value that allows receiving MTAs to verify the integrity of the previous headers and so improve irrefutability.

The requirements to sign a message are mostly the same, however, the private key must belong to the domain being used by the mail transfer agent. After generating the ARC Set this must be appended to the message so that the authentication is valid and receiving systems are able to validate it.

2.2.4 Deliverability

Deliverability is a concept which defines the chances of an e-mail delivery reaching the recipient's inbox [3], it was created due to the fact that several e-mail providers analyze traffic and through several data analysis based on the sending IP address, sender domain, message content and other factors decide whether an e-mail should be flagged as spam or unwanted, thus removing it from the recipient's inbox and placing it onto the spam folder. This behavior is unwanted by communication agencies and organizations which seek the e-mail marketing business in order to improve the communication channel with its customers.

E-mail providers and many e-mail receiving systems implement measures to prevent message delivery to the recipient's inbox such as [15]:

Sender reputation: A reputation assigned to each e-mail sending entity (*i.e.* sender domain, sending IP address) which is raised or lowered base on several criteria such as the volume of email that a sender broadcasts, the number of delivery failures that they generate as a result of rejects and/or unknown users, and the number of spam complaint notifications that they receive.

Spam Filtering: This is achieved through the analysis of e-mail content implementing several techniques such as Bayesian Filtering where particular words and sentences have particular probabilities of occurring in spam email and in legitimate email, these filters learn these patterns in order to filter out spam e-mails. Other techniques include fingerprinting,

in which a hash is generated from a combination of headers from an e-mail message and following messages are analyzed to spot duplicate deliveries and heuristic filtering which sends received messages through a set of pre-defined rules which assign a probability of the message being spam, resulting in a Spam Score.

Blacklist Operators: These entities host blacklists which contain entries of behaviors in the e-marketing world which are considered spam-like. Many systems consult these blacklists in order to prevent spam behavior. There are several types of blacklists, these are:

- Real-Time Blacklist (RBL) - Contains IP addresses from which spam behavior has been detected.
- Domain Name Server Black List (DNSBL) - Contains Domain names from which spam behavior was detected.
- Spam URL Real-Time Blocklists (SURBL) - Contain URL's which have been flagged for spam behavior.

Most of times, when an IP address or domain name enters a blacklist, all its traffic is rejected by the systems who check that blacklist, and in order to remove the blacklisted entry one must submit a request form explaining the incident to the blacklist entity which will than evaluate it and decide whether or not to remove it.

In order to improve deliverability, e-mail sending systems can [3] [15]:

- Improve data collection through implementation of positive opt-in instead of passive opt-in, this refers to not pre-checking e-mail marketing subscription boxes in forms. E-mail address double entry to prevent mistakes. Sending a validation e-mail (also known as double opt-in) which requires the customer for additional confirmation.
- Authenticate sent e-mails with SPF and DKIM signatures (and other email authentication methods).
- Monitor sending domains reputation through the systems provided by Email Service Providers.
- IP Address and Sending Management through the throttling of e-mails delivered per unit of time and prioritizing the delivery for e-mails which are less likely to complain or bounce back due to delivery errors.
- Maintain good list hygiene by preventing invalid, harvested or spam-trap e-mail addresses from reaching the delivery process and removing addresses that bounced from the lists.

Chapter 3

State Of The Art

3.1 Mail Transfer Agent Solutions

RFC-5321 [8] defines SMTP servers and clients as mail transfer agents due to the fact that they provide a mail transport service, thus, software which is able to act as both an SMTP server and client by providing the same transport constitutes an MTA. In order to provide this transport, an MTA faces several responsibilities such as queueing and scheduling the delivery of e-mail messages as well as delivering (SMTP client) and receiving (SMTP server) e-mails through the SMTP protocol, managing connections to other systems (*i.e.* MUAs, MTAs and MDAs), deferrals and undeliverable mail notifications (also knows as "bounces") processing and delivery status tracking.

Postfix [9] and Qmail [26] are examples of MTAs, but so are MailerQ [27] and PowerMTA [28], what differs from these is that the first two are general-purpose MTAs, very commonly found in corporate e-mail server environments, where a simple e-mail transport service is required, whereas the following two, are enterprise-grade, scalable mail transfer agents, appropriate for high-volume e-mail delivering, based on distributed architectures and holders of several e-mail deliverability improvement capabilities and management utilities for large deployments which are extremely helpful for massive email senders such as E-goi which want to provide its customers with the best practices in email deliverability and consequently, make their campaigns reach their subscribers mailboxes.

3.1.1 Postfix

The Postfix MTA [9] is a free and open-source mail transfer agent initially developed by Wietse Venema and released in 1998, and currently maintained by Google Inc.. It was built as an alternative to Sendmail [29] and it is the MTA in use in E-goi's current e-mail delivery system. Besides the basic responsibilities of an MTA, Postfix supports delivering several messages within the same SMTP session using cache mechanisms, retrying a delivery which has temporarily failed, DKIM and ARC authentication through the use of OpenDKIM [14] and OpenARC [13] milters [12], TLS encryption and authentication support, delivery logging and delivery status notifications and per-destination delivery throttling.

According to the documentation [30], the postfix system uses several components to receive, process and deliver an e-mail message, using queues, represented by filesystem directories, to temporarily store the messages. Figure 3.1 represents the overall architecture and most of the components involved into the system. An e-mail message can be received via a local invocation of the *sendmail* component or via SMTP or QMQP [31] through the *smtpd* and the *qmqpd* components respectively, from here the e-mail messages received are treated in

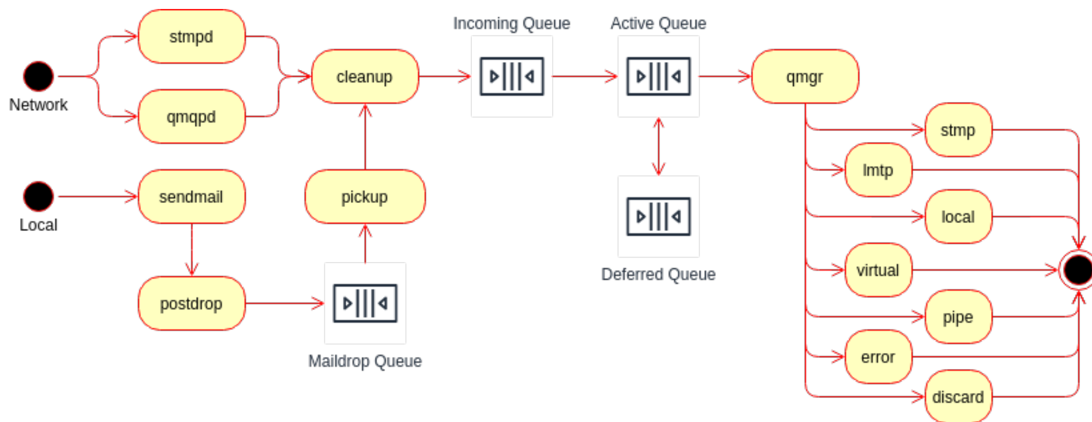

Figure 3.1: Postfix Delivery Workflow Diagram [30].

order to protect the postfix system from harm and sent into the *cleanup* component which validates and cleans the e-mail message before inserting it into the incoming queue and notifying the *qmgr* component of their arrival. The *qmgr* component will pickup a message from the incoming queue and then contact a delivery agent, such as *smtp*, *lmtp*, *local*, *virtual* and *pipe* depending on the means of transport to be used for the delivery. The *error* and *discard* delivery agents simply generate a bounce or discard the given e-mail message. While this delivery is occurring, the message is moved into the active queue, and, if a temporary error occurs (the message has been deferred), it is moved into the deferred queue where it will wait for a defined period until a retry is made on the delivery, this behavior occurs only for a defined amount of times.

3.1.2 Qmail

Qmail [26] was created by Daniel J. Bernstein as a more secure alternative to the Sendmail MTA [29] and released in 1998, being later in 2007 donated to the public domain by its author. Qmail became well-known due to its architecture based on high security principles and its simplicity whether in architecture and setup as presented in [32] and features a modular architecture based on several components with well-defined responsibilities which are present in figure 3.2 which describes the workflow of a delivery through the Qmail MTA system. It distinguished itself from its competitors due to its ease of setup, optimal performance, especially on mailing-lists deliveries, automatic bounce handling and, of course, the superb security.

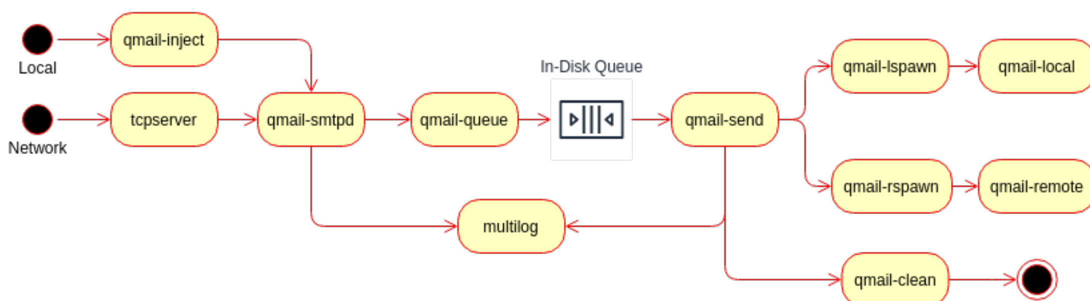

Figure 3.2: Qmail Delivery Workflow Diagram [32].

Each of the presented components are different programs which are executed under its own individual system user, this provides security by design as it is harder to exploit any of the other programs when access is achieved to one of them. Additionally, since each program has a very well defined responsibility, they represent small pieces of code and therefore have a smaller attack surface while also providing easier maintenance due the lower amount of code involved. These components and its responsibilities are as follow:

- *tcpserver* - Responsible for receiving all incoming traffic and passing it to the *qmail-smtpd* component. This is the only part vulnerable to remote attacks.
- *qmail-smtpd* - Responsible for receiving mail messages from the *tcpserver* and passing it to the *qmail-queue* component.
- *qmail-queue* - Responsible for queueing mail messages received from the *qmail-smtpd* component into the on-disk queue.
- *qmail-send* - Responsible for removing mail messages from the on-disk queue and invoking the correct component, *qmail-lspawn* (local) or *qmail-rspawn* (remote), based on whether the message is to be delivered to a local or remote user respectively.
- *qmail-lspawn* and *qmail-rspawn* - Responsible for delivering the given message and invoking as many time as required the *qmail-local* or *qmail-remote* components to deliver the given message.
- *qmail-clean* - Responsible for removing delivered messages from the on-disk queue.
- *multilog* - Responsible for logging the required events from *qmail-smtpd* and *qmail-send*.

E-mail messages arrive to the Qmail system over the network, though the *tcpserver* component, or local invocation of the *qmail-inject* component, an utility which pre-processes e-mail messages, and post pre-processing are forwarded into the *qmail-smtpd* component which will deposit these messages into the outgoing in-disk (filesystem) queue for delivery. The *qmail-send* component will then retrieve messages from the queue and invoke the correct *qmail-lspawn* or *qmail-rspawn* component based on the requirements for the message. Upon invocation these components will invoke as many times as required the *qmail-local* or *qmail-remote* in order to deliver the message for every recipient and notify the *qmail-send* component of the delivery, which will invoke the *qmail-clean* component to remove these messages from the outgoing queue.

3.1.3 PowerMTA

PowerMTA [28] is an enterprise-grade and highly reliable mail transfer agent created by Sparkpost for mass e-mail delivery which provides superior delivery performance, simple setup and real-time deliverability statistics as well as an unrivaled smaller hardware footprint. Being an enterprise-grade and mass delivery solution, PowerMTA provides several extra features in order to improve deliverability and manageability of the system, such as:

- Deliverability Reports - Users are able to generate deliverability reports in order to analyze the detailed statistics of the latest deliveries per VirtualMTA.
- Message Segmentation and Classification - Individual e-mail messages can be classified and categorized in segments to allow for tracking and rule application to these groups.

- Management Console - A web based software that centralizes configuration, monitoring and reporting management of the whole PowerMTA ecosystem such as IP-based reporting, saved reports, configurable session timeouts and advanced reporting filters.
- Authentication Support - PowerMTA supports the latest email authentication mechanism such as DKIM, SPF and DANE.
- IP Rate Limiting / Delivery Policies - IP based rate limiting is designed to allow for additional control and throttling the number of attempted recipients on a per time period basis separately for each IP address and sending domain/VirtualMTA.
- Real-Time Delivery Modifications - Through the real-time monitoring of SMTP responses received for each delivery, PowerMTA analyzes the sender reputation providing immediate notifications of delivery issues and the ability to stop or adjust the delivery settings based on these responses.
- Better Sending Practices - Deliverability features are provided out of the box in order to allow high volume email senders to follow the best email sending practices.
- Scheduled Deliveries - Deliveries can be scheduled to send at a specific time.

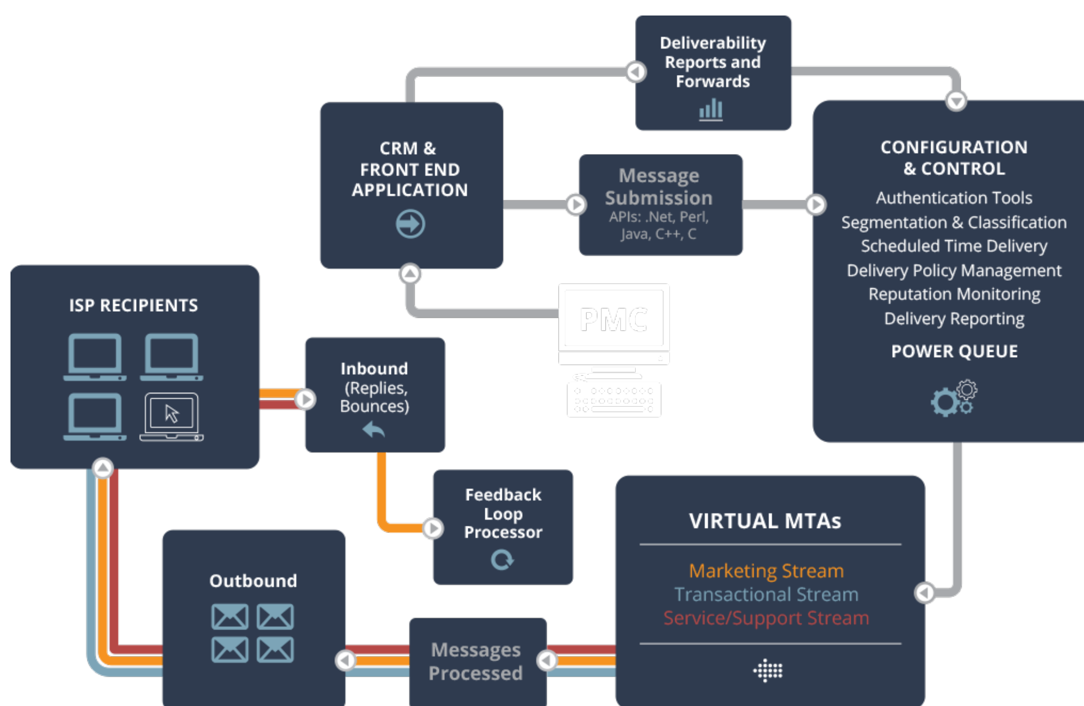

Figure 3.3: PowerMTA Architecture.
https://www.sparkpost.com/wp-content/themes/jolteon/images/graphics/powermta-process.png

Figure 3.3 provides a representation of PowerMTA overall architecture. Email messages are submitted via an API, which allows for easy integration with several systems, and are queued into the *Power Queue*. Virtual MTAs are entities which retrieve email messages from the queue, process and deliver them to the recipient's ISPs, these can be created to deliver different segments or types of messages. The delivery of these messages may result in replies or bounce messages that are redirected by PowerMTA into the *Feedback Loop Processor*

which processes these messages for configured patterns and takes configured actions based on them.

Since it's acquisition by Sparkpost, PowerMTA is provided as a SaaS solution and no longer as an on-premises solution. Their pricing [33] depends on the number of emails sends per month, the highest price-defined plan stands on 525 dollars per month for 1 million email sends per month, this price increases by 0.55 dollars per extra thousand of emails. Taking into account E-goi's current email volume of 15 million per day, described in the section 1.1, and so, 450 millions per month, this would greatly increase the required investment. An enterprise plan exists which provides dedicated account manager, deliverability reporting, a greater SLA and no limit in the number of dedicated IP addresses, yet the pricing is undefined and only discussable upon contact.

3.1.4 MailerQ

MailerQ [27] is a high-performance mail transfer agent developed by Copernica [34] which provides its users with a fast, flexible and cutting edge on-premises MTA system compliant with the latest email deliverability best practices. Built in order to deliver massive amounts of emails and provide visibility over those deliveries results, MailerQ features a central management console with real-time insights in deliveries, errors, attempts, current SMTP connections and overall queues status and configuration management which allows deliverability experts to manage every single aspect of the system.

Besides the mentioned central management console and high performance, MailerQ provides the following features [35]:

- Feedback Processing - User defined response patterns allow bounce classification, process feedback loops and adapt MTA behavior based on SMTP server responses which will improve email deliverability and overall sender reputation.
- Message Segmentation - Messages may be segmented by delivering IP address, domain or message tags which are a many-to-many labels that can be attached to messages for later identification.
- Sending Policies - Email delivery throttling, allows the specification by number of connections and delivery rates for specific combinations of sending server IP address and target recipient's domain.
- Authentication and Security - MailerQ supports DKIM, SPF, DMARC, TLS and ARC mechanisms for email authentication.
- Flood Patterns - Temporarily pause deliveries and then start sending again on a reduced capacity to prevent target server overload.
- Delivery Interception - MailerQ allows temporarily pausing and permanently failing deliveries with forced errors.
- IP Address Warm-Up - Email throttle schedules allow you to create a schedule for a specific number of days describing what email throttles to use on which day. This schedule can then be applied to any combination of source IP and target domain.
- Deliveries Rerouting - Rewrite rules can be used to dynamically reroute messages.

- DKIM Keys Management - For every key, you can specify a signing domain, selector, additional headers to include, sender domain to match on, and the signature type (DKIM or ARC).
- IP Address Grouping - IP Pools allow you to group a set of sending IP addresses under one name (a pool). This allows for management or preview of a whole group at once. IP pools can also be used when injecting emails for sending, instead of specifying individual IP addresses.

MailerQ relies on RabbitMQ [36] message system to manage its queueing structure which is briefly described in figure 3.4, according to the documentation, messages may be injected over REST API, STMP, directly into RabbitMQ, a spool directory or a MailerQ command line utility, from here, the messages, if valid, are placed into the outbox queue. MailerQ will pick messages from this queue and attempt the delivery. If this attempt fails (because the receiving server rejected it with a temporary error, see 2.2.2), MailerQ modifies the message to include info about this first soft error, and places it back into the outbox queue to retry the delivery. The message is also sent to the retry queue to notify external entities that a re-attempt for the message will happen. MailerQ will, after a while, pick up the message from the outbox queue again for a second attempt. This time the delivery succeeds and the message is modified once more to also contain info about this correct delivery, and the message is published to both the success queue and the generic results queue.

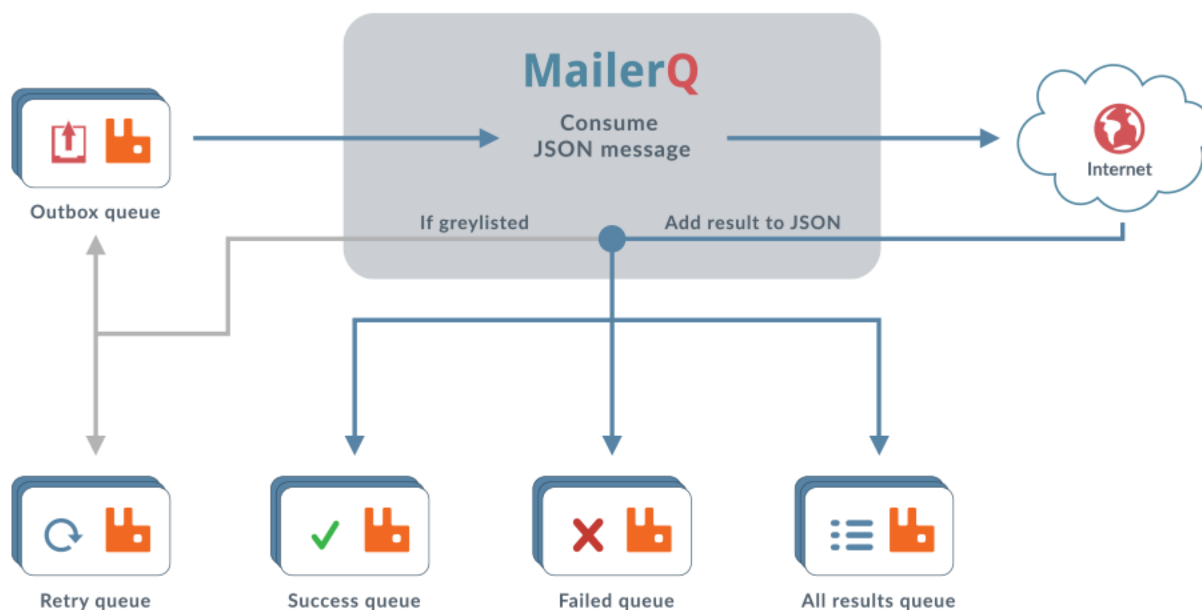

Figure 3.4: MailerQ Queue Architecture.
https://www.mailerq.com/documentation/5.8/Images/mailerq-json-results.png

MailerQ, at the time of this writing, offered two options of perpetual licenses [37], Regular and Enterprise, whose features are described in table 3.1:

Taking into account the E-goi's per day e-mail sending volume of 15 million sends, mentioned in the 1.1 section, the Regular plan option is not a valid option as it would allow at most 240 thousand emails per day, and so, way below the 15 million minimum limit. Furthermore, E-goi's current deployment features sending servers in three different countries, Portugal,

Table 3.1: MailerQ Pricing Options

Features	**Regular**	**Enterprise**
Emails per hour	100000	Unlimited
Sending Domains	10	Unlimited
Management Console	Yes	Yes
Full Feature Set	Yes	Yes
Per Instance Price (one time fee)	10000 Euros	25000 Euros
Support Price (per year)	2000 Euros	5000 Euros

France and Canada, this deployment would require three MailerQ instances which would increase the initial investment by a factor of three, resulting in an initial one-time investment of 75 thousand euros without support, which would create an yearly expense of 5 thousand euros.

3.2 MTA Technologies

Along section 3.1 several currently available MTA solutions and architectures were explored and delved into in order to provide a general knowledge of market available partial solutions and what components a mail transfer agent architecture usually contains and how they cooperate in order to achieve their purpose.

In this section, currently available technologies helpful in the development of an MTA will be approached and explored.

3.2.1 Message Brokers

From the several architectures studied in section 3.1, it can be concluded that all of them possess a queueing system of some sort, whether it be an in-memory queue, an in-disk queue, or a message broker system, all the studied MTAs use queues in order to temporarily hold email deliveries along their lifespan. These queues provide MTAs with increased message reception throughput as receiving messages are simply placed in the queue, by entities usually defined as *producers*, which removes the extra processing of building and delivering the message from the reception process. Later the queues messages are processed, by *consumers* which are the entities that retrieve messages from queues and process them. In the case of email deliveries, the consumers processing of a message is the actual delivery of the same.

These queues are implemented in several manners, Postfix and Qmail use in-disk queues, filesystem directories for each queue and write the received messages as files in those directories, MailerQ uses a message broker system, which is a system which allows the management of queues and placing and retrieving messages on and from those queues and many other functionalities such as message prioritization and scheduling. Some systems use in-memory queues, which provides them with unparalleled performance in the operations in those queues, accepting, however, that all messages may be lost on a disaster use case.

An in-memory queue would not be suited for such an environment, as in the event of a disaster (*i.e.* power failure, and consequent forceful server shutdown or simple application crash), the messages in that queue would be irreversibly lost. An in-disk queue would resolve

this scenario, as the messages would still be written in the disk (provided no data corruption errors occurred) upon the power and server restart, yet as the data is written on the disk, this prevents sharing of these queues amongst several servers, and although this could be solved through shared-filesystem technologies, these require an expensive network setup for the best performance and throughput.

An in-memory queue would not be suited for such an environment, as in the event of a disaster (*i.e.* power failure, and consequent server shutdown), the messages in that queue would be irreversibly lost. An in-disk queue would resolve this scenario, as the messages would still be written in the disk (provided no data corruption errors occurred) upon the power and server restart, yet as the data is written on the disk, this prevents sharing of these queues amongst several servers, and although this could be solved through shared-filesystem technologies, these require an expensive network setup for the best performance and throughput.

A message broker provides the best resolution to the previously mentioned problems, as it allows for communication between several applications through messages which are sent over the network to this system that acts as a intermediary between the applications or systems that need to communicate with each other.

When referring to message based communication systems concepts such as *producers* and *consumers* often arise. Producers, are the entities which create messages, these messages are then, in the case of message brokers, placed in a queue for delivery. Consumers are the entities which retrieve the messages, in the case of message brokers, from a queue, and process them. The act of processing a message is also known as *consumption*. Upon consumption of a message, the consumers, usually, inform the broker about this consumption, through what is called *acknowledgement*, meaning that said message can has been successfully processed and may be discarded.

Most message brokers provide two patterns of message communication, which will be referred to as *Anycast* and *Multicast*. Anycast, defines a point-to-point communication, a single message is produced by a producer to a queue, and will be delivered, by the message broker, to one and only consumer assigned to that queue (which can be many). Multicast refers to a broadcast communication pattern, where a single message is produced by a producer to a queue, and will be delivered to all the consumers currently assigned to that same queue. The names of the communication patterns may vary from message broker to message broker, but usually represent the same pattern.

There are currently several available options for message brokers which are, to this day, being supported and maintained, providing clustering and replication capabilities which are capable fault-tolerant and high-availability behaviors. These options will be compared and discussed in the subsections below in order to find which one could provide a partial solution for this thesis problem.

ActiveMQ

ActiveMQ [38] is an open-source message broker system developed by the Apache Software Foundation [39] which provides a safe and reliable message exchange for inter-application integration and communication over several network protocols such as Openwire, STOMP, AMQP and several more thus offering the power and flexibility to support any messaging use-case which makes it an all-round well performing candidate.

According to the documentation [40], ActiveMQ provides the following features:

- Priority Queues - It is possible to create priority queues which prioritize more important messages, this is controlled by a per-message attribute whose values ranges from 0 to 9.
- Message Scheduling - It is possible to schedule messages to be delivered to the queue after a defined amount of milliseconds, redelivered to a queue every defined amount of milliseconds and for a defined amount of times which can be infinite.
- Message Expiration - Messages can scheduled to expire after a defined amount of milliseconds in queue.
- Advisory Messages - ActiveMQ places messages on an advisory queue when certain events occur in the system such as client disconnects, message expirations, messages being delivered to queues with no consumers and others.
- BLOB Messages - The BlobMessage API allows delivering huge logical files to consumers.
- Async Sends - ActiveMQ supports sending messages to the queues asynchronously in order to greatly increase producing performance.
- Optimized Acknowledgement - Support for transactional acknowledgment (acknowledging several messages in one command) which can greatly increase consuming performance.
- Mirrored Queues - Mirrored queues represent a mechanism that allows a consumer to receive messages from queue without actually consuming them, this is especially useful for monitoring message flow.
- Plugin System - The plugin system allows the user to add plugins developed by the community to their ActiveMQ instance for extra behavior and functionality.
- Message Persistence - Messages can be persistent or non-persistent, persistent messages are able to survive a system crash event as they are persisted on disk while non-persistent messages are persisted in-memory and so are lost on crash events yet, producing non-persistent messages is much faster.
- Message Ordering - ActiveMQ provides is able to deliver messages to several consumers in the order that they were queued.

Clustering and replication [41] are, as previously mentioned, important topics for the wanted solution, ActiveMQ provides several kinds of clustering which answer to different needs, these are as follow:

Broker Clusters: Several brokers are distributed along different servers in the same network and client connections are configured using the failover protocol with static or dynamic discovery. When one of these brokers fail, client connections will attempt connection to another broker configured in the failover protocol. However, messages in the failed broker are not available to any consumer until this broker is recovered and so, albeit a broker high-availability is provided (depending on the number of brokers defined), message replication is not provided and so message availability is still a problem. Another problem with this architecture is that at any moment, one of the brokers might be receiving messages from

producers, but have no consumers connected, an so messages will find themselves stuck in a queue with no consumers for an undefined amount of time.

Networks of Brokers: This architecture is similar to the *Broker Clusters* architecure but resolves the issue of queues with no consumers by having brokers forward messages from queues in this situation to brokers from the cluster which have consumers connected to those same queues. Brokers use auto discovery in order to connect between themselves and form a network of brokers.

Master Slave: The MasterSlave model comprises messages replication to a slave broker, such that if in the event of a hardware, file system or data centre failure, an immediate failover is provided by the slave with no message loss. The problem in this architecture is that in order to provide message replication, it requires a high-availability shared filesystem, a shared JDBC database or a zookeeper [42] server. All these options allow to run as many slaves as are required and have automatic recovery of old master, yet, each of them have its cons:

- Shared File System Master Slave - Requires a high-availability shared file system.
- JDBC Master Slave - Requires a shared JDBC database and is slower since it is unable use ActiveMQ's high performance journal.
- Replicated LevelDB Store - Requires a ZooKeeper server.

Replicated Message Stores: Replicated Message Stores require a highly-available shared file system and a second ActiveMQ instance to be automatically started once the first one crashes using the shared file system.

Artemis [43] is the codename for the HornetQ project after its donation to the Apache Software Foundation and will replace the ActiveMQ project (currently in version 5.18) in version 6 [44] as a merge between both systems is currently occurring. Artemis will feature message replication over the network which will provide the required fault-tolerant and high-availability for both brokers and messages.

RabbitMQ

RabbitMQ [36] is an open-source high-performance message broker solution developed by Pivotal [45] which can be deployed on-premises or cloud environment to meet high availability and scalability requirements.

The documentation [46] highlights the following features:

- Plugin System - A plugin system allows expansion of functionality through user installed plugins.
- Message Persistence - Messages can be persistent or non-persistent, persistent messages are able to survive a system crash events while non-persistent cannot but are faster to produce.
- Flexible Routing - Messages are routed through exchanges before arriving at queues and patterns can be defined for automatic routing of a message to several queue in example.
- Multi-protocol - Supports several network protocols such as STOMMP, AMQP 0.9, MQTT and more.

- Tracing - RabbitMQ provides tracing mechanisms no analyze system misbehaviors.
- Commercial Consulting and Support - Commercial support and consulting is available for a fee.
- Message Priority - Priorities can be defined for messages so they are consumed before lower priority messages.
- Message Expiration - Messages can scheduled to expire after a defined amount of time in queue.

When it comes to clustering and replication, RabbitMQ has a huge support. Clusters of nodes offer redundancy and can tolerate failure of a single node and all definitions (*i.e.* exchanges, bindings, users, amongst others) are replicated across the entire cluster yet queues behave differently, by only existing in the nodes where they were created but can be configured to replicate between nodes through the use of Mirrored Queues. Mirrored queues contents are replicated across a defined number of cluster nodes, and when a node fails, the replicas will undergo a new master election, where the key reliability criteria is the existence of a replica eligible for promotion. Consumers connected to the failed node will have to recover by reconnecting to the cluster, while consumers connected to other cluster nodes will be re-registered by RabbitMQ to the newly appointed master replica.

Comparative Analysis

Both ActiveMQ [38] and RabbitMQ [36] provide a robust and reliable message broker system yet, there are several disadvantages to each. Starting off with ActiveMQ, it provides an evolving solution, currently in a process of fusion with Artemis [43] in order to provide even greater functionality. It provides a large set of functionalities such as message priority and message scheduling, both important for a Mail Transfer Agent system and possesses clustering capabilities. Yet, the clustering provided, at the time of this writing, by ActiveMQ requires a performance decreasing approach, with the centralization into a database, or a shared network filesystem which is somewhat financially expensive especially taking into account situations of high-traffic. The latter of these options is the only that may provide high-availability and message centralization amongst all the nodes of a cluster, which makes it the best option for a highly-reliable system.

RabbitMQ on the other hand provides network based clustering and message synchronization around the cluster nodes, through mirrored queues, yet, it does not allow for native message scheduling which might be a fundamental functionality. Furthermore, RabbitMQ provides support and consulting which may be useful for achieving the best performance and quickly understanding and recovering from disaster recovery scenarios.

It should however be noted that ActiveMQ Artemis [43] provides network synchronization of messages across nodes, and this feature will be implemented on the ActiveMQ 6 version which will then provide both the required functionalities of message scheduling and network synchronization of messages between all nodes of a cluster.

3.2.2 Compression Algorithms

E-mail messages, are mostly large bodies of text content (attachments included) that may reach several tens of MiB, which is one of the reasons why many MTAs, such as Postfix and Qmail opt by using in-disk queues, as it would be dangerous to use in-memory structures to

host such large amounts of data, possibly making the system susceptible to memory overflow vulnerabilities during heavier loads. Furthermore, the impact of large amounts of data is not only observed on volatile memory, but also on the network layer, since the larger the files, the more bandwidth is required to send the data over the network.

Compression technology is nowadays used in order to transform large amounts of data into smaller amounts through the use of extra processing power with different types of compression algorithms. In order to improve the forwarding of messages through the MTA system, one could use compression techonology to decrease their size and optimize the overall system performance albeit the usage of more processing power. It should be noted that only lossless data compression algorithms are suitable for the task at hand, as no data losses can occur.

Along this section we'll study the performance of the following lossless data compression algorithms:

- Lz4 [47]
- Lempel–Ziv–Welch (also known as Gzip) [48]
- Bz2 [49]
- Lzma [50]
- Lzo [51]
- Brotli [52]
- Zstd [53]

Comparative Analysis

In order to find out the best compression algorithm for use in this project, a study was conducted in which two equally important criteria were defined:

- Compression Time - The time it took for the algorithm to compress a given data.
- Data Compression Ratio - A measurement of the relative reduction in size of data. It is expressed through the division of the uncompressed size by the compressed size.

Many data compression algorithms feature several levels of compression, usually, by increasing the level, the algorithms exchange compression speed by compression ratio, thus achieving better compression ratios but at the cost of slower compression times. In order to evaluate time progression through several different data sizes, the tests were executed for six different sizes of data expressed in megabytes: 5, 10, 15, 20, 25 and 30 Megabytes. In order to generate this data, an automated process was used in which a website's HTML was continually downloaded and appended to a file until the data content was greater or equal to the required size after which the content, if exceeding, would be trimmed to the required size.

In order to ensure reliable test results, each run, for each algorithm (and each of its levels) and for each data size, was executed 100 times and the data being compressed was kept constant for each of these runs. A statistical median was then applied to each of the 100-time executions in order to obtain the most correct value.

Since we are looking for the best compression ratio in the least amount of compression time possible, a rating variable was defined as the division of the compression ratio by the compression time, so that it is possible to rank all the algorithms and its levels. This rating was calculated for each of the algorithms (and its levels) previously referred which allowed the selection of the best algorithm-level pairs to be shown in the results.

The results for the best algorithm-level pair regarding the compression time are described in figure 3.5 and the same results for the compression ratio are described in 3.6. Raw data for the defined factors for each of the best algorithm-level pairs is also available as tables at appendix A.

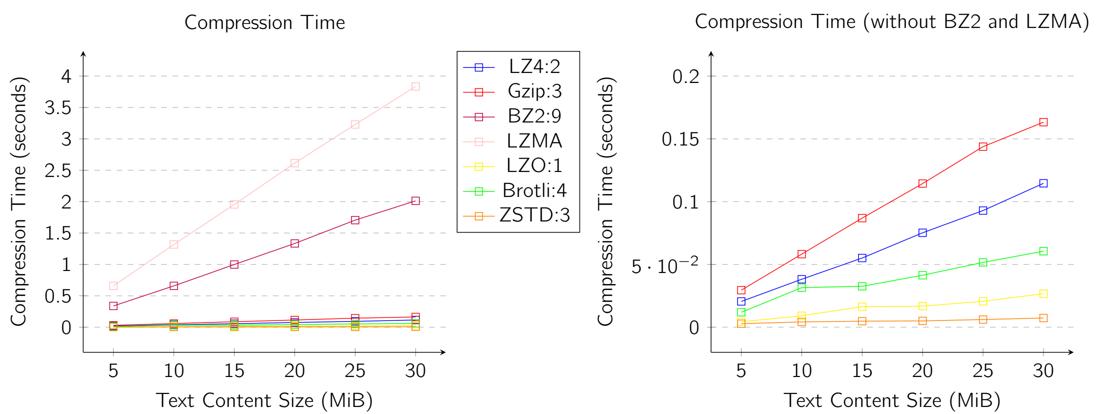

Figure 3.5: Compression time for the best algorithm-level pairs (lower is better).

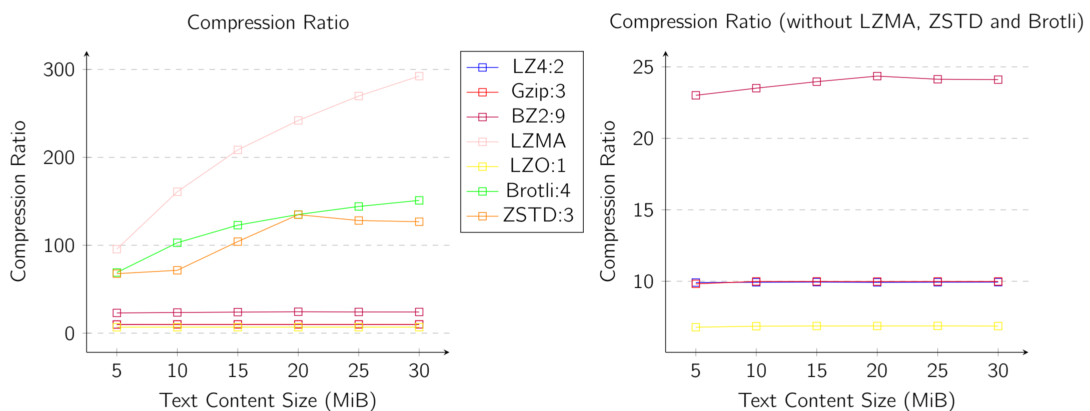

Figure 3.6: Compression ratio for the best algorithm-level pairs (higher is better).

The results for the rating variable are shown in figure 3.7 where it is possible to observe that ZSTD at compression level 3 grants a clear advantage from any other compression algorithm for all contemplated content sizes. It should however be noted that the rating variable used, assumes that the compression time and the compression ratio are equally important variables, which may not be accurate for all use cases.

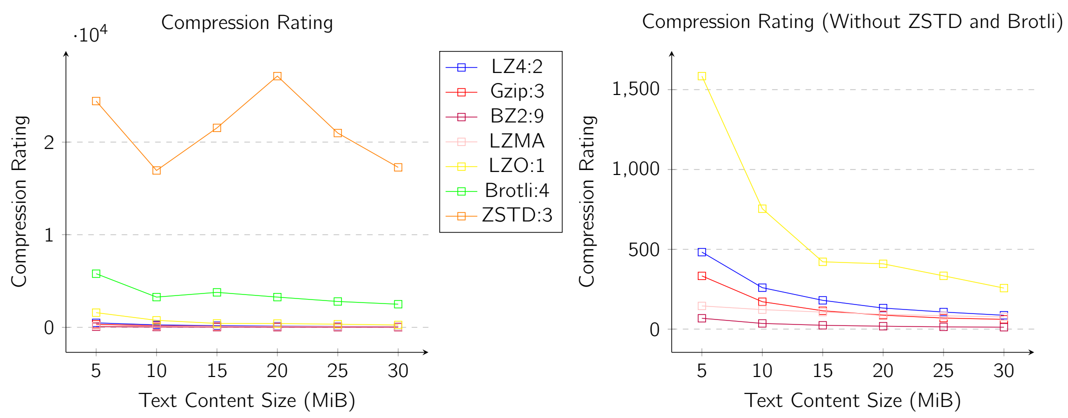

Figure 3.7: Compression rating for the best algorithm-level pairs (higher is better).

Chapter 4

Analysis

4.1 Requirements Engineering

According to Suzanne and James Robertson [54], *"Requirements are what the software product, or hardware product, or service, or whatever you intend to build is meant to do and to be."* which defines a software solution as the fulfillment of its requirements. Requirements engineering is one the most important parts in the development of software as these will define which functionalities set will the product provide, what are its properties and qualities and in what molds the solution will be developed.

Requirements can be Functional or Non-Functional. Suzanne and James [54] defined functional requirements as *"(. . .) the actions the product must carry out to satisfy the fundamental reasons for its existence."*, a definition which translates a functional requirement as a function that the product being designed must be able to do. Non-Functional requirements are as important as the functional requirements, albeit not providing functionality, Suzanne and James [54] accentuate that *"Nonfunctional properties may be the difference between an accepted, well-liked product and an unused one."*, and define them as *"(. . .) the character, or the way that functions behave."* in which functions translate into the functional requirements, thus defining the non-functional requirements as the molds in which the functional requirements are to be implemented.

> *"You use nonfunctional requirements to specify response times, or accuracy limits on calculations (. . .) when you need your product to have a particular appearance, or be used by nonreaders, or adhere to laws applicable to your kind of business."*[54]

Guidelines were created as an attempt to standardize requirements engineering description and classification. In their book, Robert Grady and Deborah Caswell [55] introduced the FURPS model, as a standard for software engineering requirements classification as a whole defending that *"(. . .) adding a new function might improve functionality but decrease performance, usability and/or reliability."*, the model defines five variables which are important for the definition of a requirement, Functionality, Usability, Reliability, Performance and Supportability. FURPS was later modified into FURPS+ in which the '+' sign defines an extra set of variables which do not fit the previously mentioned one. The meaning of each of these variables is the following[55]:

- Functionality - The feature sets and capabilities present in the product.
- Usability - Aesthetics, training materials, wizards, user documents and other human factors.

- Reliability - Properties of the system related to recoverability, predictability, accuracy and fault-tolerance.
- Performance - Attributes such as speed, availability, throughput, response times and recovery time.
- Supportability - Testability, maintainability, compatibility, configurability and extensibility.

The description of the functional requirements will be developed as Use Cases, which were first introduced by Ivar Jacobson who accentuates that *"Use cases make it clear what a system is going to do and, by intentional omission, what it is not going to do."* [56], these describe a functional requirement of a system, *"(. . .) enabling the envisioning, scope management and incremental development of systems of any type and size."* in order to facilitate the communication between the customer requirements and the developing team implementation, thus providing the customer with an implementation which best fulfills its needs. The S.M.A.R.T. principle was first introduced by George Doran in his article *"There's a SMART way to write management's goals and objectives"* [57] which defined five variables to describe management goals and objectives that can be applied to User Cases description. Use Cases should, therefore, be[57]:

- Specific, by targeting a specific are of improvement.
- Measurable, so that it is possible to evaluate or quantify progress.
- Assignable, meaning it should always be specified who will implement the goal.
- Realistic, as it should state what results can be achieved given the available resources.
- Time-Related, by specifying when it is expected that the results will be achieved.

4.1.1 Functional Requirements

After identifying the problems with the current solution, it is time to define the functional requirements for the new solution, these will be enunciated as Use Cases in the brief format as previously mentioned in section 4.1.

The use case definition will always feature an actor, which defines the external entity which will be interacting with the system in the use case, along this section, there will be mentions to the following actors:

- Administrator - The entity which is responsible for the deployment of the solution.
- User - The entity which is integrating with the solution in order to deliver emails.

Use Case 1: Message delivery

As a user of the system, I want to be able to request the delivery of an e-mail to a defined recipient. In order for this use case to succeed, the user must, at least, provide an e-mail address as recipient, an e-mail address as sender and a version of the e-mail body and its content-type as well as the IP pool from which the e-mails should be sent. Additionally, the user may be able to provide custom e-mail headers in key-value pairs which should be inserted in the delivered e-mail message, as well as tags which are alfa-numeric strings which should be associated with the message in order to allow indexing delivery results and statistics by these tags. The system will provide the user with a unique identifier that identifies the submitted delivery in the system.

Use Case 2: Scheduled delivery

As a user of the system, I want to be able to schedule an e-mail to be delivered to a defined recipient. In order for this use case to succeed, the user must provide the number of seconds by which the delivery should be delayed when performing Use Case 1.

Use Case 3: Message Priority

As a user of the system, I want to be specify a priority for a delivery request. In order for this use case to succeed, the user must provide a priority value from 0 to 9, in which the bigger the value the higher the priority, when performing Use Case 1.

Use Case 4: DKIM Signature

As an administrator of the system I want to be able to configure the delivery processes to digitally sign the messages to be delivered with a valid DKIM signature. In order for this use case to succeed the solution needs to be able to sign the e-mail messages with a DKIM signature and allow the administrator to configure locations from where the DKIM keys corresponding to the message sender name should be retrieved from as well as other required parameters for a DKIM signature such as the headers to sign, the canonicalization process to use and the selector.

Use Case 5: CC and BCC Support

As a user of the system I want to be able to deliver messages to CC and BCC addresses. The addresses under CC should receive the message and information about the other addresses which received it (excluding BCC addresses) while the BCC addresses should receive the message as if it was only directed to them. In order for this use case to succeed the use must provide the CC and BCC addresses as well as their names (optional) when performing Use Case 1.

Use Case 6: Message Delivery Logs

As an administrator of the system, I want to be able to consult logs of every delivery attempt for a certain message. In order for this use case to succeed the solution must be able to log information about each delivery to a data source (*i.e.* file or database table) which can be consulted by the administrator.

Use Case 7: Delivery Notifications

As a user of the system, I want to be able to receive notifications about the events for a certain delivery. For this use case to succeed, the user must provide a web-hook URL and eventual authentication mechanism (*i.e.* user and password credential) when performing Use Case 1.

Use Case 8: Group Delivery Processes

As an administrator of the system, I want to be able to group delivery processes in pools, so that a group of delivery processes will only deliver emails from a certain pool. In order for this use case to succeed, the user must provide the system with a pool name and add delivery processes to that pool.

Use Case 9: Delivery Policies

As an administrator of the system, I want to be able to define delivery policies, which represent a maximum amount of concurrent deliveries which may be occurring per recipient's

domain. In order for this use case to succeed, the user must configure a policy by providing the system with the recipient domain for which the policy applies and the maximum amount of concurrent deliveries. It should be possible to define a global configuration per delivery process as well.

Use Case 10: Delivery Stop/Pause/Resume

As an administrator of the system, I want to be able to stop, pause or resume deliveries that have already been submitted for delivery but have not yet been delivered. In order for this use case to succeed, the user must provide the system with data about the deliveries it wants to stop, this data can be a delivery unique identifier or a message tag.

Use Case 11: Submission Reports

As an administrator of the system, I want to be able to obtain reports about submissions and deliveries performed by the system. It should be possible to retrieve these reports by submission, delivery SMTP Client, submission tags or delivery groups. Available data for the report should contain the following:

- Submission time - Time of submission.
- Submission interface - How the submission was created.
- Transactions - The transactions which occurred for the submission and data about them.
- Failed Transactions - The transactions which were unsuccessful.
- Successful Transactions - The transactions which were successful.

4.1.2 Non-Functional Requirements

The non-functional requirements are as important as the functional requirements and will be described through the FURPS+ rationale, as discussed in 4.1.

Functionality:

- TLS Encryption must be used for deliveries whenever possible.
- SMTP Response Handling must be able to distinguish temporary from permanent errors and, respectively, retry the delivery or fail it.

Usability: No usability related requirements were made.

Reliability:

- The solution must provide fault-tolerant behaviors when it comes to network failures and central system crashes.
- The solution must provide high-availability and redundancy deployment options.
- The solution must provide delivery results for each attempted delivery for irrefutability purposes.

Performance:

- The solution must not reduce the old solution's delivery submission ratio of 5 deliveries per second.

- The solution must be scalable and provide increased performance through horizontal scaling.

Supportability:

- The solution must be hosted on CentOS operating system servers.
- The solution must support the ActiveMQ message broker as queueing system.

Others ('+'):

- The solution must reduce current hardware and infrastructure costs.

4.2 Value Analysis

In order to design and implement any new product or functionality, this one should first be the target of a value analysis that covers both the company level, the need for it, its cost and benefits (*i.e.* when it comes to human resources, development time and infrastructure costs), and at the market level, the market need for the new product, its desire for it. Along this section, the whole value analysis and innovation process will be approached and carefully described.

4.2.1 Innovation Process

Innovation is the process through which companies and other entities add value to both themselves and their product(s) [58], which may be accomplished in several ways such as price competition, improvement of existing functionalities, creation of new functionalities and even the development of new products, and according to Peter Koen [20], that process can be divided into three sequential stages:

- Fuzzy Front End
- New Product Development
- Commercialization

The Fuzzy Front End method was first introduced by Reinertsen and Smith [59], and represents the first stage of the innovation process whose objective is the breading of a concept based on a defined opportunity. In order to accomplish that concept, Fuzzy Front End defines a sequence of steps which should be taken, starting on the identification of existing opportunities, moving on to proving the value of said opportunities, generating ideas based on the proven opportunities which add value to them, selecting the best idea and finally, define the concept which is to be implemented in the next phase (New Product Development).

The New Product Development stage defines the process of converting the created concept into an actual product, this also encompasses market analysis, definition of client channels as well as testing the product and, eventually, making minor changes.

Lastly, once the product has been created, tested and approved, it is ready to hit the market and so the Commercialization stage begins, where the main objective is to sell the resulting product.

The sequential iteration of these stages defines the life cycle of an idea, since the moment of its creation until it becomes a product or functionality and, eventually, is commercialized. It is expected, that ideas that survive through to the commercialization stage are ideas whose

value has been effectively proven and that will add value to the product or entity responsible for it.

4.2.2 New Concept Development

Peter Koen, accentuates that *"Attention is increasingly being focused on the front-end activities that precede this formal and structured process in order to increase the value, amount, and success probability of high-profit concepts entering product development and commercialization."* [20]. This implies that a well-defined Fuzzy Front End stage will allow greater confidence when moving through the next stages or prevent the development of ideas whose final product will fail due to a weaker analysis in the Fuzzy Front End stage.

New Concept Development is a model created, in 2002, by Peter Koen [20] in order to fulfill the lack of a standardized language, terminology and best practices for the Fuzzy Front End stage, properties which existed for the New Product Development and Commercialization stages. The NCD model is composed by three key components:

Engine - Represents the *"(. . .) leadership, culture, and business strategy of the organization that drives the five key elements that are controllable by the corporation."* [20].

Five Key Elements - The five controllable key elements of the FFE: Opportunity Identification, Opportunity Analysis, Idea Generation and Enrichment, Idea Selection and Concept Definition.

Influencing Factors - Represents factors that are external to the corporation, an so, relatively out of its control, yet influencing towards the innovation process, such as laws, government policies, customer, competition and the enabling science.

In the following sub sections, the NCD model will be applied to the current problem statement in order to understand if the defined opportunity adds value to both the E-goi organization and its product.

Opportunity Identification

Innovation opportunities may arise in a variety of ways, such to fulfill and unfulfilled market need, to provide a better solution for a market need, or a more competitive option.

In section 1.2, several problems with a current system were identified and represent an opportunity to improve the E-goi platform in several possible ways, through the refactoring of the e-mail sending system thus providing better performance, reliability, more functionality and better data visualization into an area which is of extreme importance for the business as accentuated in section 1.1.

An opportunity definition by itself does not add value to an organization, it must be analysed and its value must be proven through conclusions of said analysis.

Opportunity Analysis

The opportunity analysis stage encompasses the identification of the strengths and weaknesses of an opportunity, which may be revealed through a SWOT analysis. A SWOT analysis features the Strengths, Weaknesses, Opportunities and Threats behind an idea. While the strengths emphasize the reasons why the idea adds value to the organization,

the weaknesses reveal factors which will cause value loss such as development time. Opportunities reveal influencing factors, external to the organization, which may add value to the idea, while threats encompass external influencing factors which may damage the idea's value such as governmental policies, laws and market competition.

For the defined opportunity of refactoring E-goi's email sending system, a SWOT analysis was generated in order to analyse its value for the E-goi platform and organization as well as if the benefits of the implementation of the opportunity outweigh the sacrifices, figure 4.1 represents said analysis.

Figure 4.1: SWOT Analysis.

Regarding the Strengths identified, the implementation of the opportunity will provide the product with much greater delivery control, and as such, improve the overall e-mail deliverability for most e-goi customers, furthermore, it will reduce hardware usage and so lower infrastructure maintenance costs, provide easier setup and scalability which in turn will make responsible human resources free for other tasks and provide greater insights on the overall system and its deliveries as well as a better integration. It should also be noted that in-house developed solutions allow for custom behavior adding in order to quickly answer market needs or specification changes. Nonetheless, there are visible weaknesses, such as the initial development/implementation cost, the periodic maintenance costs and difficulties which may occur when migrating from the old system into the new.

Opportunities feature the increase in the overall platform usage and especially the e-mail channel usage while threats are made up by current and possibly new market competition as

well as the implementation of laws which in some way limit digital and/or e-mail marketing behaviors.

Summarizing, it is possible to verify that the SWOT analysis results are complacent with the success of the opportunity and opportunity's value addition to the company's product, both internally, as this will reduce maintenance costs both on human resources (setup time) and infrastructure costs, as externally, since it ill provide better service for E-goi customers through the improvements on e-mail deliverability. Such advantages outweigh the predicted implementation costs and so it possible to move into the next stage of the NCD, Idea Generation.

Idea Generation

Through the analysis of the opportunity, its value and reasoning as well as the clear advantages in its implementation were proven. The next NCD stage features a brainstorming process from which solutions that resolve all of the problems identified in section 1.2 and fulfill the functional and non-functional requirements enunciated along the section 4.1 should be generated for later selection. After a thorough internal analysis, based on enunciated the state of the art (see chapter 3), E-goi's technological and financial capabilities and the requirements engineering, three solution approaches were generated:

- A PowerMTA based implementation.
- A MailerQ based implementation.
- The development of a new distributed MTA solution using available technologies.

Although quite different in implementation details, all the enunciated solutions fulfill the defined requirements and as such, a decision must be taken on the most viable solution and which stands better chances of success. Due to the importance of such a decision, a thorough analysis based on several key criteria and decision making methods will be provided in the next section.

Idea Selection

AHP (Analytic Hierarchy Process) is a structured and mathematical decision making method created in order to provide an effective decision making rationale based on several configurable key-criteria and several possible solutions [17]. Along this section an approach to the method will be made by using it to chose the most viable and correct solution from the ones generated in the previous section 4.2.2.

The first step in the AHP method is the **Hierarchy Division**, according to which an enumeration of the possible solution and the respective evaluation criteria should be arranged in a hierarchical form such as the one presented in figure 4.2 which is applied to the current problem statement.

The main objective for the current problem statement is to refactor E-goi's email sending system, and may be achieved through the implementation of one of the solutions previously generated in section 4.2.2, these are:

- X: A PowerMTA based implementation.
- Y: A MailerQ based implementation.

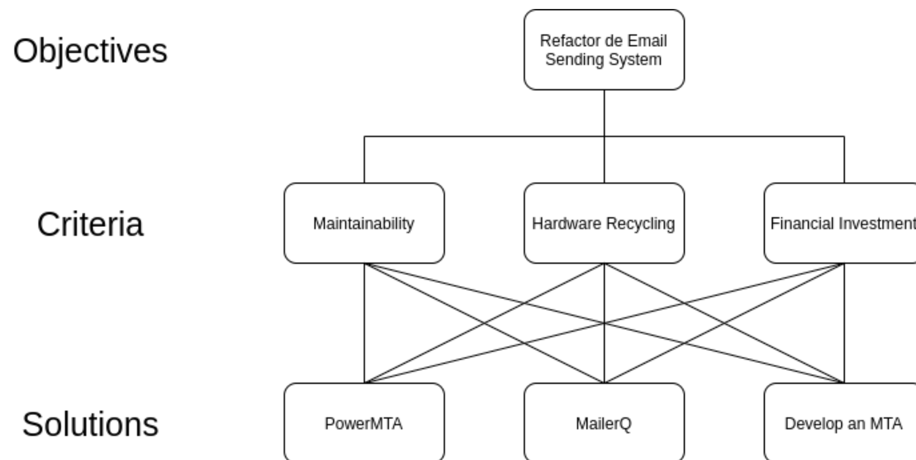

Figure 4.2: AHP Hierarchy for the current problem statement.

Table 4.1: Criteria comparison and relative priority matrix.

Criteria	A	B	C	Relative Priority
A	1.00	3.00	3.00	0.59
B	0.33	1.00	2.00	0.25
C	0.33	0.50	1.00	0.16

- Z: The development of a new distributed MTA solution using available technologies.

In order to evaluate these solutions several comparison criteria must be defined which will allow the evaluation of the solutions based on themselves and, consequently, the election of the most viable and correct solution. The criteria defined were the following:

- A: Financial Investment, which represents how ease it is to maintain, scale and modify the solution in order to react to growth or market needs and achieve competitive advantage.
- B: Hardware Recycling, that represents the reuse of E-goi's current, on-premises hardware infrastructure.
- C: Financial Investment, the investment required in order to develop and implement the solution.

Provided the definition of the hierarchy structure through the enumeration of the main objective, the key evaluation criteria and the possible solutions, it is possible to move into the next AHP step, **priority definition**. The second step comprises the definition of the priority for each of the defined evaluation criteria. Through the use of the Satty scale [60], the defined criteria were prioritized and the matrix and relative priority values obtained are described in table 4.1.

The priority matrix (see table 4.1) provides insights in the comparison of the criteria importance for the current problem statement between themselves and so it is possible to see that the "Maintainability" criteria is three times as important as the "hardware Recycling" or "Financial Investment" criteria, which is fairly accurate since E-goi requires a solution which is malleable to the constant requirements changing and evolution of e-mail deliverability in

Table 4.2: Priority Matrix for the "Maintainability" criteria.

Solutions	**X**	**Y**	**Z**	**Relative Priority**
X	1.00	0.33	0.25	0.12
Y	3.00	1.00	0.50	0.32
Z	4.00	2.00	1.00	0.56

Table 4.3: Priority Matrix for the "Hardware Recycling" criteria.

Solutions	**X**	**Y**	**Z**	**Relative Priority**
X	1.00	0.11	0.11	0.05
Y	9.00	1.00	1.00	0.47
Z	9.00	1.00	1.00	0.47

order to achieve competitive advantage over its competitors. Furthermore, it is possible to evaluate that the "Hardware Recycling" criteria is twice as important as the "Financial Investment", which represents the fact that by removing the necessity for On-Premisses hardware for the email sending system would require the selling of this hardware, and as such, financial value would be lost as this would not be resold at purchase price and the financial investment on the final solution (SaaS solution) would also include hardware costs after all thus resulting in a higher investment price.

Through the priority matrix manipulation, it is possible to calculate its normalized matrix through which it is possible to obtain a relative priority value for each of the defined criteria which relates to the last column of table 4.1 where is possible to observe that the "Maintainability" is the most important criteria, followed by the "Hardware Recycling" and lastly, the "Financial Investment".

Once the evaluation criteria have been prioritized, comes the prioritization of the solutions based on those same criteria. The process requires the building of a matrix for each criteria (A, B and C). Each matrix represents the comparison of each of the solutions (X, Y and Z) between themselves based on one criteria. The matrices were built and the results are described in tables 4.2, 4.3 and 4.4.

According to the values in table 4.2, it is possible to claim that, when it comes to "Maintainability", solution Z provides the best option as it represents an in-house developed solution and so allows for much easier and direct maintainability (through source-code modification) as well as behavior customization which is not possible in the X and Y solutions seen that they are third-party applications. Nonetheless, they provide behavior configurability functionalities which are able to fulfill this criteria to a certain degree, just not as fully as solution Z.

Solutions Y and Z fulfill, as proven in 4.3, the "Hardware Recycling" criteria equally as both of them provide an on-premises deployment option and so they obtain the same relative priority when it comes the this criteria. Solution X, became a SaaS solution since its acquisition by Sparkpost as previously mentioned in section 3.1.3 and so does not allow an on-premises solution which prevents it fulfilling this criteria.

Table 4.4: Priority Matrix for the "Financial Investment" criteria.

Solutions	X	Y	Z	Relative Priority
X	1.00	0.50	0.33	0.16
Y	2.00	1.00	0.50	0.30
Z	3.00	2.00	1.00	0.54

Table 4.5: Criteria relative priority matrix.

Criteria	Relative Priority
A	0.59
B	0.25
C	0.16

Fulfilling the "Financial Investment" criteria, comes solution Z, as can be observed in figure 4.4, which provides the lowest expected financial investment of all the solutions since both solution X and Y require a high monetary investment especially due to E-goi's email sending system multi-datacenter architecture, high number of required IP addresses and, of course, high sending volume.

Arranging the relative priorities of both the criteria and the solutions in own matrices, we obtain the matrices described, respectively, by the tables 4.5 and 4.6.

The multiplication of these matrices provides the global weight for each of the solutions based on the defined criteria as show in table 4.7.

The matrix represented in table 4.7 defines the global weight for each of the solutions which were being evaluated, and from it, it is possible to conclude that solution Z is the most viable and correct, yet, this assumption is only possible after the AHP method has been validated, which comprises the last step in the AHP method, **Consistency Validation**. The assurance of this consistency can be proven through the use of AHP consistency index, CI, equation:

$$CI = \frac{(\lambda max - n)}{(n - 1)}$$

in which $n = 3$ and λmax is given by the largest eigenvalue of the matrix that results from the consistency calculation and is presented in table 4.8.

Table 4.6: Solutions' relative priority matrix.

—	A	B	C
X	0.12	0.05	0.16
Y	0.32	0.47	0.30
Z	0.56	0.47	0.54

Table 4.7: Solutions' global weight matrix.

Solution	Weight
X	0.11
Y	0.36
Z	0.53

Table 4.8: Consistency formula matrix.

X	1.82
Y	0.77
Z	0.48

The matrix eigenvalue is calculated through the average of the division of the consistency matrix values (see table 4.8) by the values of the criteria relative priority matrix (see table 4.5):

$$\lambda max = \frac{1.82/0.59 + 0.77/0.25 + 0.48/0.16}{3} = 3.05$$

Applying λmax to the consistency index, CI, formula we obtain the following result:

$$CI = \frac{(\lambda max - n)}{(n-1)} = \frac{(3.05-3)}{(3-1)} = 0.03$$

Provided the consistency index, CI, it is possible to calculate the consistency ratio, CR, represented by the following formula:

$$CR = \frac{CI}{RI}$$

where RI represents the randomness index which is provided by table 4.9 through the value of n.

Since n = 3, the RC value is the following:

$$CR = \frac{0.03}{0.58} = 0.05$$

Table 4.9: Randomness Index Table.

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IR	0	0	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

Given that RC < 0.1, the value's consistency has been proven and the AHP method can be considered valid.

Since the AHP method has been validated regarding the consistency of its results, it is possible to consider them as valid. The results for the solutions provided in section 4.2.2 were the following:

- X: 0.11
- Y: 0.36
- Z: 0.53

Taking into account that solution features the highest result, the implementation of an on-house MTA solution is considered the most valid and so it is possible to move into the next phases of the development.

Concept Definition

The concept definition comprises the last stage of the NCD model, and defines that a concept for the solution should be provided in order to finish the model. The concept for the recently identified solution consists of a distributed, scalable and fault-tolerant e-mail delivering system which should provide ease of integration, setup and maintenance and seamless addition of new features.

4.2.3 Business Value

The contents of this section feature several business value analysis according to different models.

Customer Value and Perceived Value

In order to fully understand the value of a product, it must also be analysed from the customers perspective and so concepts such as Customer Value and Perceived Value should be defined to provide this extra understanding.

The value of a product is defined by the importance of this product for the customer, the needs it fulfills and of how much the customer is willing to pay for it, and since most organization's income is provided by its customers, it is of extreme importance that such organizations thrive to improve its products values.

Customer value is a definition which refers to the value that a customer feels towards a given product according to its personal perception. A product provided by an organization or entity may have different values for different customers. These values are often influenced by the customer's experience and interaction with the product or entity, in example, a customer whose experience was great is expected to come back for more experiences while customers which faced difficulties of any kind and were not able to correctly make use of the product will probably resume the product's customer value to that bad experience, thus not re-attempting the usage of the product [61].

Customer Perceived Value is defined by the difference between the benefits and the sacrifices perceived by the customer [62]. It is possible to frame this concept in the current problem statement, if we consider the benefits to be an improved e-mail deliverability and high-probability to reach the subscribers inbox, a high-performing system capable of delivering

campaigns for lists of hundreds of thousands of subscribers in the smallest amount of time possible as well as a reliable system which provides fault-tolerant behaviors and allows the E-goi organization to better scale and add value to its product.

Value Proposition

A value proposition aims at providing the reasons by which a customer should acquire a product or service and which benefits will he gain in case of said acquisition.

Within the scope of this project, the value proposition is a simple, reliable, performing and distributed Mail Transfer Agent solution with e-mail deliverability best practices implemented by design. The solution will not only provide a scalable, stable and management-easy email sending environment as well as insights on all deliveries and extra functionalities.

Business Model Canvas

The Business Model Canvas was first introduced by Alexander Osterwalder in 2004 [63] and comprises an artifact which represents an organization's global business logic through the description of several components which are arranged in blocks for a visual representation (see figure 4.3), these are:

- Key Partners - The entities which provide some sort of behavior required by the organization (*i.e.* delivery companies).
- Key Activities - The processes that must be completed in order for the customer to be served.
- Key Resources - The tangible and intangible assets which are used everyday.
- Value Propositions - The reasons why the customers will want to buy the organization's product and what benefits will they acquire from it.
- Customer Relationships - What kind of relationships does the organization seek with its clients, short-term? Long-term?
- Channels - How will the organization reach its clients, through a web platform? Physical Stores?
- Customer Segments - Description of customer segments who will be interested in the organization's product.
- Cost Structure - The main expenses required by the organization, examples are human-resources wages and advertising.
- Revenue Streams - The sources of revenue for the organization.

The canvas business model was used within the scope of this project to provide greater understanding on the business logic and is detailed in figure 4.3 where it is possible to observe several customer segments which are provided by E-goi, from small companies to big communication and marketing agencies with whom customer technical and accounting support long-term relationships are created through channels such as E-goi web platform, webinars, audiobooks and web blog posts.

Furthermore, on the left side of the canvas model, it is described how the E-goi organization achieves its goals, through key activities such as web development, customer support, customer accounting and product selling, which require key resources such as technical knowledge from several areas and web servers, and with the help of partners which are trained by the E-goi partnership program to sell the product in foreign countries, email service providers and SMS providers in example.

E-goi's main financial costs are represented in the bottom part of the business model canvas and are human-resources wages, hardware, development utilities and office supplies and its revenue streams are the payment of customers product plans which are dependent of the sending volume and feature-set.

Figure 4.3: Business Model Canvas.

Porter's Value Chain

First introduced by Michael Porter in 1985, the Value Chain concept provides a representation of how the several activities of an organization interact in order to create value for its customers. The proposed value chain is a general-purpose strategic management tool which can be used by any organization who seeks to examine all its activities and how they interact with each other, this is especially useful to evaluate where it is possible to increase value or reduce costs, or both. The main objective behind the value chain is to divide the organization key activities by categorizing them into smaller components and understanding how they work together instead of attempting to evaluate the organization as a whole similar to the "Divide and Conquer" principle.

In figure 4.4 lies the visual representation of the value chain as provided by Porter, from which it is observable that the main activities of a company are divided into two types, Primary Activities, that refer to the activities related to the creation of the organization's product(s) as well as sales, maintenance and support of the same, and Support Activities, which relate to the activities that feed into the primary activities by providing them with the required resources to allow them to produce.

The primary activities of Porter's Value Chain are subdivided into the following types:

- Inbound Logistics - The activities responsible for the reception, storage and inventory control as well as product inputs.

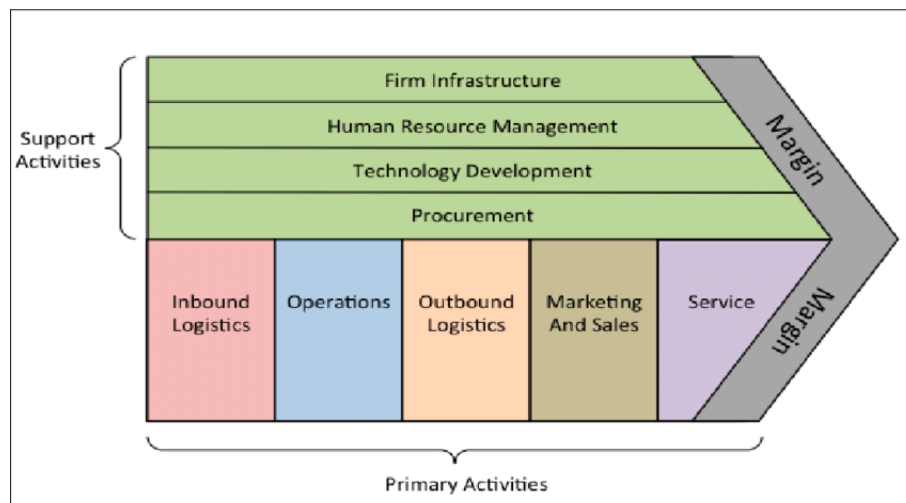

Figure 4.4: Porter's Value Chain [64].

- Operations - The activities related to the conversion of the feedstock into tangible products.
- Outbound Logistics - The activities which comprise the retrieval, storage and product distribution to its customers.
- Marketing and Sales - The activities related to the selling and promotion of the organization's products.
- Services - All the product support activities.

The secondary activities of the value chain are subdivided into the following:

- Firm Infrastructure - Activities whose main objective is to keep the daily operations running smoothly.
- Human Resources Management - Activities related to the organization human resources such as the hiring of new employees as well as the maintenance of its contracts conditions. These are important activities and may generate value through the motivation of the employees.
- Technology Development - Activities related to the development of the technologies of the organization's products such as version updates which may improve performance and add new functionalities, thus providing extra value to the product.
- Procurement - Activities related to acquisitions which are required by the organization in order to provide its maintenance and that of its products such as software licenses for employees and even the products.

The margin illustrated in Porter's value chain represents the profit margin, which is the difference between the value that an organization is able to create and the cost of creating that value. So the greater the value an organization is able to create, and the lower its costs, the higher will its profits be.

Within the scope of this project, it is possible to describe its main activities and in which primary and support activities of the value chain do they fit thus allowing us to determine

how and where will it generate value for the organization. The following primary activities were identified:

- Operations - The development of the product itself.
- Outbound Logistics - The created product will be integrated into the E-goi platform and as so, distributed through it.
- Services - The maintenance of the systems in which the product will be deployed and the product's configuration and management.

As for the secondary activities the following were specified:

- Technology Development - The maintenance of the product itself and that of the software it requires (software updates).
- Procurement - The provisioning of IP addresses and hardware for the future scaling of the system (if/when required).

Through these activities it'll be possible to create value for the company, this value, will be generated through the refactoring of a system critical to E-goi, its email sending system, which features several problems (as discussed in section 1.2) such as a time-consuming setup and a high hardware resource usage which represent costs for the organization. The new solution, will mitigate problems and so, reduce these costs, thus creating value for the organization.

Chapter 5

Design

5.1 Architecture

Given the mass delivery and scalability requirements, the solution's architecture should be based on a distributed architecture as there will be a high volume of e-mail messages to deliver per day (see section 1.1) through several defined IP addresses and a high-throughput is required in order to prevent undesirably larger delays between the customer's action of sending a campaign and the campaign's delivery starting. A distributed architecture features the distribution of the solution's required processing through several layers which allows physical distribution and specific layer scaling through parallelism [65].

Along this section, we will be discussing the solution's architecture, from its components and their integration to its deployment in several environments and the design of each of the components and its responsibilities.

5.1.1 Components

As the solution will have to comply with the growth of a mass e-mail delivery business, it should be scalable in all its components, in order to allow a higher degree of hardware specialization for each of them and to satisfy each layer's growth independently as well as the requirements for geographical distribution and horizontal scaling defined in chapter 4 to a full extent. In figure 5.1 we present the components model which identifies the set of components representative of the solution, these are, as with many mail transfer agents, the SMTP server and the SMTP client.

Figure 5.1: Solution's Components Model.

The SMTP server is responsible for receiving e-mail delivery requests and queueing them for delivery in the broker system, it represents the gateway for the whole system and will be exposed for integration through a REST API. The SMTP client component is the layer responsible for retrieving messages from the broker system and delivering them to the intended recipient, this layer must be highly configurable in order to allow for a diverse range of limit configuration regarding the deliveries (*i.e.* maximum number of concurrent deliveries per recipient domain, delivery rate per recipient domain, delivery IP address and hostname to use in the SMTP transactions, etc). The communication between these components is assured through a message broker which will allows the two components two communicate with each other by sending messages to each other with the broker acting as a moderator.

These layers provide integration interfaces which allow the different components to communicate between themselves and external systems to integrate with the solution. The SMTP server component is exposed via a REST [21] API interface, that will centralize the receiving of requests for the several required functionalities, especially e-mail delivery transactions. This component will provide a simple, easy and scalable interface for any external systems such as E-goi to integrate with. Albeit this interface is a REST API, a later improvement could be the addition of another interface to this component in the form of an SMTP Server, which would allow MUA's, MTA's and other systems to communicate directly with the system via a protocol they already implement (SMTP).

Responsible for delivering e-mail messages to the intended e-mail providers, the SMTP client will also deal with digitally signing e-mail messages with DKIM authentication prior to its delivery, delivering messages to the intended email service provider and dealing with all the errors that may arise from those deliveries (*i.e.* temporary and permanent SMTP errors) as well as making the necessary efforts to retry deliveries which temporarily failed and, last but not least, enforcing the configured delivery policies. This component represents a critical component and must be able to deal with all errors that may arise from its operation as to prevent delivery duplication or missed deliveries, this could mean networking issues, broker system unavailability, etc.

The integration of these components will be achieved through the use of a message broker (see section 3.2.1) which will be responsible for managing the required mail queues and the concurrent access to them as well as providing a base for features such as message priority and message scheduling which will be critical for correct implementation of some of the use cases presented in chapter 4.

5.1.2 Deployment

Deployment is the act of installing the solution into its hardware infrastructure, and by taking into account the requirements for geographic distribution and horizontal scalability defined, a distributed architecture was designed which grants a greater degree of deployment versatility to the solution. There are several deployment options available and they will be discussed along this section, ranging from simple one server minimal deployments to highly scalable, distributed and available deployments over several geographically distributed servers.

Although the solution's architecture allows it to be distributed, this may not be required for minimal deployments as some customers may be looking for a simple and lightweight MTA to deliver their company's emails and receive any others. Figure 5.2 represents the deployment model for a minimal deployment, a single server hosts the message broker instance and one instance of the solutions components, the SMTP server and SMTP client. This minimal deployment minimizes the infrastructure requirements but provides no high-availability neither high-throughput although one can increase the number of instances allocated to each component (or only one of them) in order to increase its throughput.

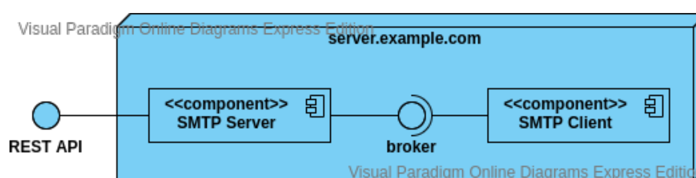

Figure 5.2: Solution's minimal deployment.

On the other hand, a more advanced customer may wish to specialize its deployment to better suit the hardware to the solution's components needs, or maybe the customer already has a broker system in his company and wishes to reuse it in its deployment. Figure 5.3 shows a deployment suitable for these customer needs, the broker can be physically separated from the solution's components which allows to allocate better servers for this layer, and cheaper ones for the other layers, or reuse an already owned broker system.

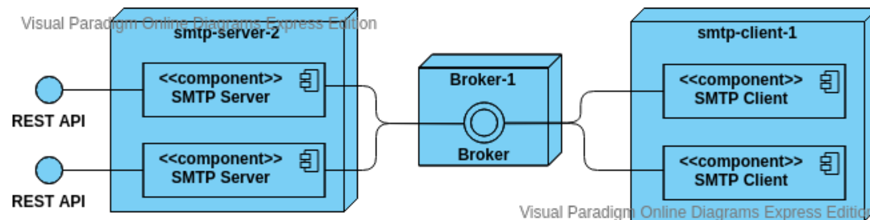

Figure 5.3: Solution's multi-instance deployment.

Furthermore, being able to physically separate the solution's components allows for hardware specialization for these, in example, the SMTP server REST API will be target of larger numbers of incoming connections than that of the SMTP Client component, which may require increasing the operating system port ranges to larger values to be able to accept more connections and/or the use of multi-core CPUs to allow parallelism in the request processing which will further help in this endeavor, yet, they will mostly not require access to the disk as the e-mail messages will be sent into the message broker system which resides in another server and so, there's no need for the servers holding this component to feature expensive and high-performing disks which helps to reduce costs by specializing the hardware for the component it will be supporting.

As per the current E-goi deployment, the SMTP client component will be target of the most replication behavior as these will be distributed geographically in different data-centers and with different configurations. When it comes to hardware they should mostly require CPU cores and RAM depending on the delivery activity of each SMTP Client (*i.e.* number of SMTP Client instances and e-mail messages content size).

Given that the e-mail channel represents their most used channel and the number of deliveries per day, it is safe to assume that the delivery of all the messages requested and the high-availability of the delivery system is critical for the company's reliability, and so, in order to fulfill this need, it is possible to guarantee an increasing degree of high-availability by deploying the solution as seen in figure 5.4 which represents a smaller scale deployment of a highly-available solution deployment.

The message broker system is a core component in the architecture as it provides the communication between the SMTP server and client components which means it represents a single point of failure that must be dealt with. Fortunately, most of these message broker systems feature clustering capabilities which provide high-availability deployments in the service through message replication along several nodes (see section 3.2.1), a behavior which can be observed in figure 5.4.

The application components must both be able to detect failures in communications with the broker system and failover to different nodes of its cluster if such are available. When it comes to high-availability on the solution itself, this is provided through the replication of components. The solution also allows the spawning of several instances of each component in each server it is deployed as can be seen in figure 5.4 which can be used to support a

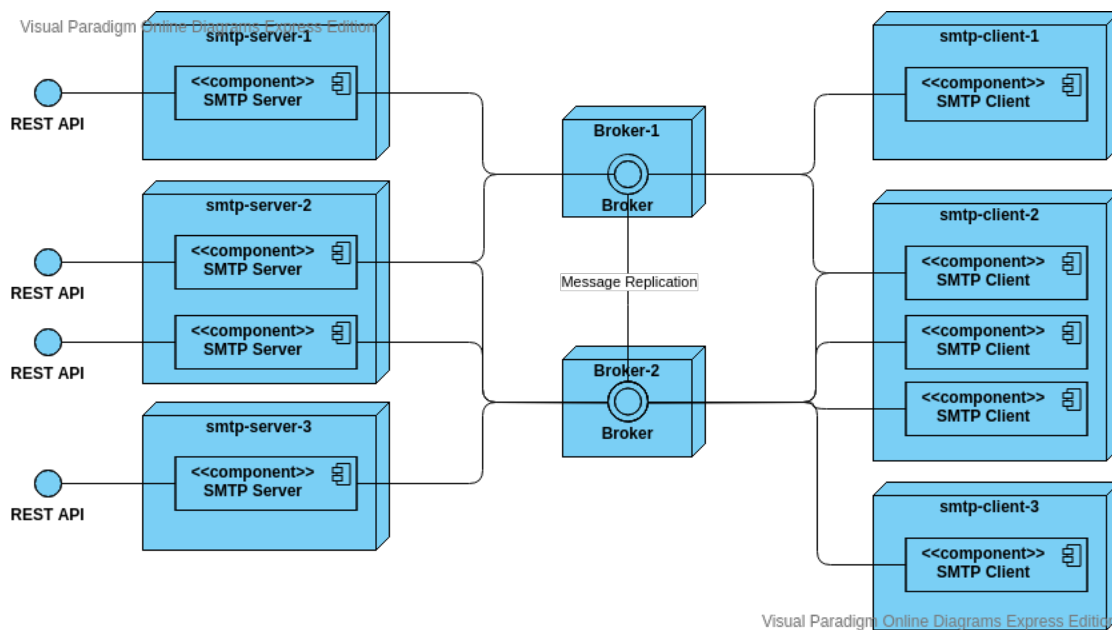

Figure 5.4: Solution's high-availability deployment.

higher throughput whether of delivery requests or message deliveries. A network traffic load balancer can be placed before the REST API instances in order to grant high-availability for the incoming communications with the several instances of the SMTP Server component while SMTP Client component may be replicated with the same configurations to a degree that losing one of them, due to any disaster, won't even be noticed in the deliveries traffic.

5.1.3 Queueing Structure and Model

In order to move on to the use case design we must first decide how will the message broker be used by the solution in order to fulfill the functional and non-functional requirements imposed in chapter 4, this comprises the definition of the concepts that both the queues and the messages placed in them will represent.

Three approaches were designed in order to define this queueing model, each with its pros and cons, and they will be presented and discussed in the following subsections.

Message per campaign, queue per delivery group

This model presumes that each queue, represents the delivery group from which the email messages are to be delivered (see Use Case 9 in 4.1) and that each message is a campaign to be sent via the E-goi platform. The broker message content would then feature:

- The list of recipients to which to deliver the campaign.
- The campaign's text and HTML version.
- Extra headers to be appended to the e-mail message.
- Custom data per recipient (in order to support the dynamic campaign data feature present in the E-goi platform).

This model would provide the solution with knowledge to concepts of the E-goi platform and reduce the number of messages in queue, thus removing strain and processing from the broker system. Furthermore, if messages were to carry data from which customer does the campaign belong to, delivery throttling schedules could be based on E-goi customers or vary from campaign in campaign and since this processing would be implemented in the SMTP client's side, it would provide better performance for larger campaigns as the SMTP client could sort the recipient's by domain and deliver several messages in the same SMTP session.

Yet, although there are clear advantages, the concept of "campaign" is not part of the Mail Transfer Agent domain and this would remove abstraction from the solution. This model would also largely increase the processing logic under the SMTP client component when it comes to acknowledging a message consumption, delivery re-attempt on temporary failed deliveries. In the case of disaster events (*i.e.* if a crash event were to occur while a campaign was being delivered, the message would not be acknowledged and would be delivered to another SMTP client for delivery, which would have no way of knowing which recipients had already received that message and consequently cause the redelivery of already delivered e-mail messages).

Message per delivery, queue per campaign

Presuming that each queue represents a campaign to be sent via the E-goi platform and that the messages in placed in it would represent the atomic deliveries (one for each recipient) of that campaign, the broker message content would then feature:

- The recipient to which to deliver the message.
- Extra headers to be appended to the e-mail message.

The content of the message would be stored in a centralized environment together with the information about the dynamic content, which SMTP client instances would be able to query in order to retrieve it and start the delivery for all the recipients waiting delivery in queue.

This option would still be able to keep knowledge about the E-goi system and allow behavior customization through E-goi platform domain concepts, and most important, it would solve most of the previous option disadvantages, since if a disaster were to occur, only the broker messages which were currently being processed would be in an unknown state and be redelivered to another SMTP client instance, which would greatly reduce the number of e-mail messages that could be delivered to a recipient which had already received them.

Nonetheless, this option has its disadvantages, once again, we are introducing the "campaign" concept from the E-goi product into the Mail Transfer Agent environment which reduces its abstraction degrading its market value to other products which do not share this "campaign" concept. Furthermore, implementing the delivery groups required by Use Case 9 (see section 4.1) is not contemplated in this queueing model which would require this logic to be passed on to the solution side, thus increasing the amount of work required to implement the final solution. Last but not least, this implementation would also create the most queues when compared to the others, these would require a cleaning routine of some sort for those whose usefulness has expired. The amount of messages created would also be largely increased when compared to the first model.

Message per delivery, queue per delivery group

This model presumes that each queue is a delivery group, thus fulfilling Use Case 9 (see section 4.1) immediately, and that each message message in a queue is an e-mail message to be delivered to a single recipient. This message's content would feature:

- The recipient to which to deliver the campaign.
- A representation of the email message ready to be loaded and delivered by the SMTP client.

The clear advantages of this model is that it makes no assumptions about the integrating business logic and so its a very abstract solution which allows the easier addition of new features via extra logic on the SMTP server or SMTP client components. In example, tags could be added to the message which would allow the identification of messages or groups of messages on the SMTP Client component (since tags are an abstract concept, they would be able to represent a variety of concepts from external businesses or systems) and behavior such as delivery policies could be defined at a decreased granularity level (message level and not campaign) based on this tagging functionality. Furthermore, if delivery temporary errors ocurred, this model would allow for the SMTP client to simply send the message back into the broker system to be re-delivered after a configurable timeout until a defined number of retries for a delivery had been breached. The model is also better than the previous when it comes to disaster events as a crash could at most cause a message to be redelivered and not a larger amount as in the first Model. The abstract concept in this model, also provides a much greater business value, which is that an abstract solution, may be able to integrate with any system, and so, the solution could become a distinct sellable product which would add even more value to the organization.

The downside of this model is that the number of messages largely increases which may require a more robust broker system and that the abstraction created decreases the knowledge about the integrating system and so does not allow for custom behaviors based on that system's logic or domain although these may be created as described previously (the tagging functionality example).

Model Comparison

All the three models are possible to implement and have good advantages, yet, due to the clear advantages such as greater abstraction which improves the maintainability of the solution, as well as the extra value added by the fact that an abstract solution can be sold as a distinct product and the low number of disadvantages, the "Message per delivery, queue per delivery group" model was the chosen for the solution's implementation. Figure 5.5 represents the delivery workflow for an e-mail message through a sequence diagram according to this queueing model. In the represented flow, a temporary error is given at the first delivery attempt to which the SMTP client responds by requeueing the message in the broker system with a scheduled delay. After this delay, the message is once again retrieved from the queue by the consumer, which may not be the same as the previous, and the delivery attempt is successful, thus finalizing the delivery process for that message.

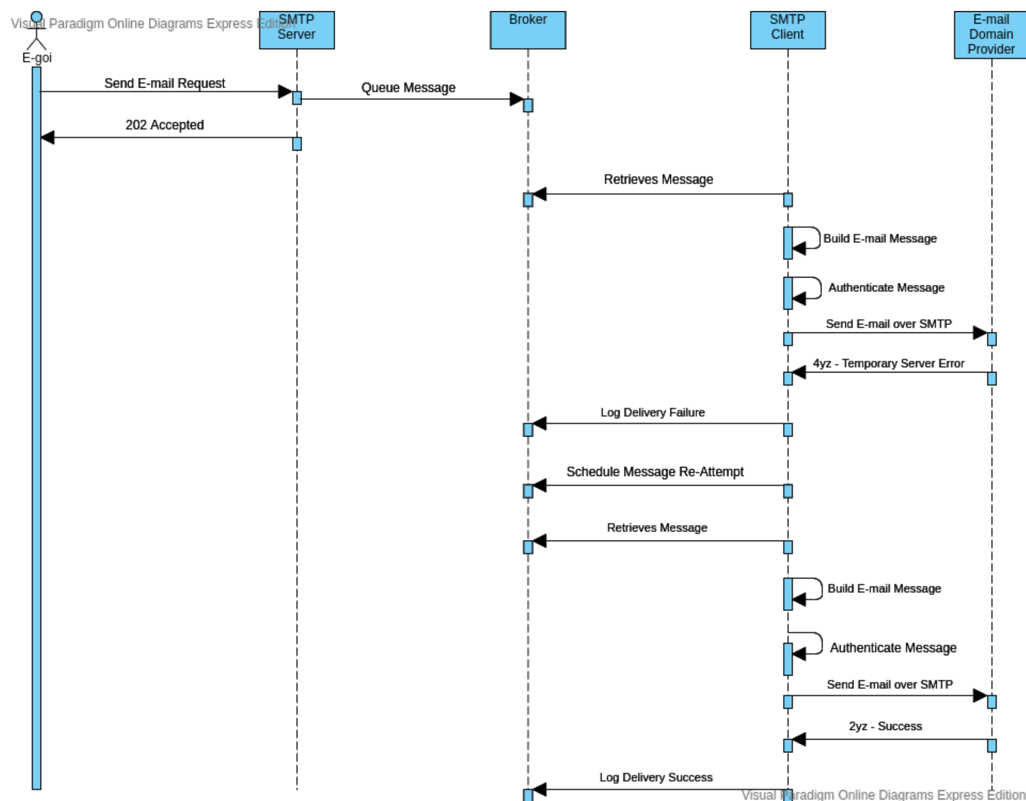

Figure 5.5: Solution's Delivery Sequence Diagram.

5.2 Use Cases

With the solution's architecture defined and provided that it supports the high-availability and scalability non-functional requirements, it is possible to move on to a decreased granularity level and discuss the use cases to implement, their design and inner workings. Along this section we will be approaching each use case in order to define which components will be involved in it and how they will implement it.

Subsections containing the "Extra" word on their title may be found throughout this section and represent further functionalities which have been added to the product albeit they do not comprise requirements.

5.2.1 UC1 - Message Delivery

The message delivery is, by far, the most complex use case as it represents the main objective of the entire solution. All the components will be involved and there are high performance requirements defined for its implementation.

An email message delivery requires, at least, three parameters, the recipient address, the sender address, and the content to be delivered, of which several versions may be provided (*i.e.* plain text content and HTML text content). Provided that these minimal requirements reach the SMTP server component, the message delivery request is considered valid and it becomes possible to create a representation of the email message with all its parameters, compress it with a compression algorithm and load it into the respective queue in the message broker and move on to the SMTP client component and address to its delivery. In order

to provide better performance for environments with higher amounts of submissions, and since it is possible, in an SMTP session, to sequentially submit several messages, the SMTP server component should be able to support submissions of several submissions and not only one.

Several optional parameters are also allowed by the SMTP server:

- Extra headers which must be added to the email message before delivery.
- Tags which is a collection of alphanumeric strings that may be used to more easily identify a message as per the integrating system's domain. These should be placed in the message which is delivered to the message broker.

Upon retrieval of the message from the message broker, the SMTP Client will decompress it and load the email message representation into memory, resolve the MX record belonging the recipient's domain (if non-existent, falling back to the A record is a must), open an SMTP session with the respective MTA at that location and deliver the e-mail message.

In this process, several errors may arise, temporary errors and permanent errors. Temporary errors, are those which occur due to a temporary unavailability be it of the receiving MTA or due to a network failure, in which case a delivery should be retried at a later time. In these cases the message should be resent to the message broker and delayed by a configurable amount of time before another attempt at delivery is retried. A maximum amount of delivery attempts per message must also be configurable in order to prevent messages from staying in the queue indefinitely. Possible temporary errors are as follow:

- No MX record or A record exist for the recipient's domain.
- The SMTP transaction results in a status code between 400 and 500.
- The receiving MTA is not available (includes network failures).

Permanent errors are the result of errors in the delivery process in which case a re-delivery at a later time will not change the result of the delivery. In these cases, the broker message should be acknowledged and the delivery should not be retried again. The following are considered permanent errors:

- The SMTP transaction results in a status code larger than 499.
- The maximum amount of delivery attempts for the current message has been breached.

It should be noted that the delivery of a message is achieved through an SMTP transaction which, as described in section 2.2.2, represents a set of commands and replies that are exchanged between two MTA systems. Each of these commands is sent over the network and, as anything sent over it, they are subject to the latency between the both MTAs, as such, if the latency between the two MTAs were of 20ms, the SMTP transaction, without TLS encryption and the lines after the DATA command which are variable, would last for at least $11 * 20 = 220ms$ of exclusively roundtrip time. This represents the minimum amount of time that the actual delivery would take (excluding the time taken to establish the connection to the recipient's MTA, the TLS encryption and the content delivery whose time length depends on the content's size) which is not acceptable for a model which is only able to handle one SMTP transaction per SMTP client instance as this would require scaling the SMTP client instances to huge amounts for a large deployment to feature a great delivery performance something which would increase the required resources and consequently, the amount of hardware required and as such, the infrastructure cost.

In order to solve this performance problem while maintaining hardware requirements, each SMTP client instance should be able to retrieve messages from the broker and process them asynchronously (*i.e.* using a thread for each delivery), which would allow for several deliveries to occur in parallel and since the process is highly IO bound (due to network latency) the CPU usage would be kept low. This thread pool should have a limit of threads which represents the maximum number of parallel deliveries that may occur, a value which should match the message prefetch size defined in the message broker subscription. The broker message must, nonetheless, only be acknowledged once the delivery of the email message has been completed.

5.2.2 UC2 - Scheduled Delivery

The delivery schedule is easily implemented by appealing to the message broker functionalities. All the message brokers discussed in section 3.2.1 implement some sort of scheduling mechanism which allows to either delay the delivery of a message to a consumer by a specified amount of seconds, or to schedule a message to be delivered to a queue at an exact moment.

As such, this use case should be implemented by adding optional "delay" and "schedule" parameters in the delivery request (SMTP server component) when performing Use Case 1. These parameters represent, respectively, the delay in seconds by which the delivery should be delayed, and the exact time at which the message should be delivered to the queue. If both are present in the delivery request, the "schedule" parameter should be ignored.

5.2.3 UC3 - Message Priority

Message priority is also a mostly common message broker functionality and should be implemented based on such as it requires much less logic on the solution's codebase.

Similarly to Use Case 2, a "priority" optional parameter should be added to the delivery request (SMTP server component) when performing Use Case 1. This parameter must be an integer value ranging from 0 to 9 (inclusively) which defines the priority, the higher, the most prioritized the delivery of said message will be relatively to others in the queue.

5.2.4 UC4 - DKIM Signature

DKIM represents a digital signature method which provides irrefutability that a message was delivered from the domain which is originated and that its contents have not been modified while in transit. DKIM signing will result in the addition of a "DKIM-Signature" header to the message containing the signature as its value before being delivered. In order to sign a message using the DKIM authentication several parameters are required:

- The sender's domain which is present in the message to be signed.
- The DKIM private key to be used in the signature.
- The DKIM selector associated with the private key.
- The canonicalization process to use, possible values are: "simple/simple", "simple/relaxed".
- The list of message headers to be signed.

The component responsible for the message signing will be the SMTP client as it represents a process which must be done for each delivered message, if all the parameters required are present. However, for some messages this might not be possible and so it should be possible to configure the SMTP client instances to deliver the messages even if these are not signed.

The signing process requires a private key which is different for each sender domain. Although a small business or a customer implementing a minimal deployment of the solution may only deliver messages from a small number of distinct sender domains, when approaching a large scale deployment such as E-goi's current deployment, this number may easily scale to the thousands or tens of thousands of sender domains. As such, the SMTP client instances should be capable of retrieving the private key from a configurable location (filesystem directory or URL) in order to better suit the integrating system's needs.

It must also be noted that retrieving a private key for each delivery in a large scale deployment may decrease the system's performance greatly, especially if the retrieval is depending on network access (*i.e.* retrieving from a URL), in order to solve this, the solution should implement an in-memory caching mechanism to cache these private keys for faster access, the maximum amount of keys to be cached should be configurable and the cache replacement mechanism should follow a least recently used fashion.

5.2.5 UC5 - CC and BCC Support

CC and BCC support can be implemented on either of the components, the SMTP client could implement logic to deliver the same message to CC and BCC addresses after it retrieved it from the broker system. In a similar manner, upon receiving a request for delivery of a message with CC or BCC addresses, the SMTP server could expand this message into one message for each address and set the CC header to all the addresses except the one being addressed to thus delivering more messages to the broker.

If this logic were to be placed into the SMTP client, we would be adding a variation to the delivery workflow, something which would have to be taken into account when debugging or testing and we would also be adding an extra responsibility to this component which already features several (*i.e.* email delivery, DKIM signature, delivery logging and delivery notification triggering, etc). Furthermore, the SMTP client represents the most critical component as it is the one which actually delivers the intended messages to the subscribers, keeping it simple must be a concern in order decrease the amount of responsibilities and so, the amount of distinct errors which may arise from the delivery process and that need to be dealt with.

On the other hand, implementing this logic on the SMTP server would add extra processing to this layer, thus decreasing its performance which has high throughput requirements (see chapter 4). Nonetheless, it would allow us to keep the SMTP client delivery workflow constant for every message, as well as, making every broker message atomic, as one message in the broker would represent one and only one delivery to be made, while the opposite would mean that one broker message could represent more that one delivery as this message would be expanded into several by the SMTP client component when into the delivery workflow.

Given that the implementation will be applied to the SMTP server component as it allows the solution to be more transparent in its inner-workings and more maintainable since a variation on a critical component has been avoided keeping its workflow constant for every message it consumes. Summing up, when the SMTP server component receives a message which contains CC or BCC addresses, it is to expand this message into all the combinations

of messages that its delivery would originate and load them into the broker system as atomic messages which represent a delivery to a single recipient address.

5.2.6 UC6 - Delivery Logging

The envisioned solution is required to log information about all the delivery events or transactions (*i.e.* failures or successes) into a persistent datastore. This functionality responsibility falls under the domain of the SMTP client, as it is the only component with enough information to provide its implementation. As such, at the end of each delivery, the SMTP client will send messages with the required information to a specialized queue in the broker system (as can be observed in figure 5.5), which will then be processed by another consumer into a configurable persistent data store (*i.e.* file, database, etc). The supported data stores should support SQL proficient databases. It is important that a rotation strategy is configurable for these stores (*i.e.* daily, weekly, monthly or yearly rotation).

It is, however, important that the delivery events logging messages are not lost as they are the guarantee of an event for a certain transaction at a certain time which may be used for irrefutability purposes or to discern if a message was or not delivered when a disaster, on the SMTP Client component, has ocurred while delivering a message and one wishes to prevent message delivery duplication from occurring. Taking this into account, the solution must be able to ensure that these messages are not lost due to broker unavailability or networking issues. Figure 5.6 details the design for the delivery logging production and consumption.

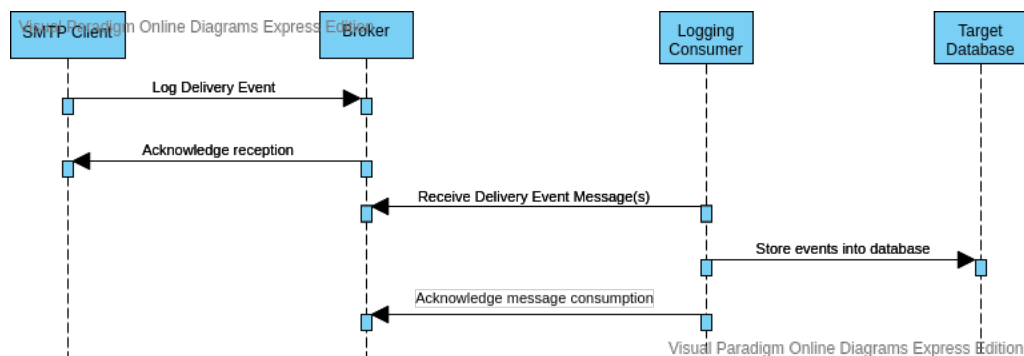

Figure 5.6: Solution's Delivery Logging handling sequence diagram.

The data to be logged should be complete, indexable and well structured in order for log handler abstractions to be created, as such, the following parameters must be logged:

- Submission Id: A unique identifier of the submission for which this transaction belongs to.
- Delivery Status: The status of the delivery, possible values are "delivered", "deferred" and "undeliverable".
- Status Reason: The reason for the status assigned to this delivery.
- Delivery Time: The time at which the delivery was completed (whether it failed or not).
- SMTP Code: The SMTP transaction response code.
- SMTP Message: The SMTP transaction response message.

- Delivery length: The time it took for the delivery process to complete (from the moment of the reception of the broker message to the delivery completion).
- Message headers: The email message headers as a JSON string.
- SMTPClient data: Data describing the configurations of the SMTP client which delivered the message as a JSON string.

All this data should then be passed to an abstract handler entity which can be extended in order to allow for several implementations of delivery logging data unto several data stores thus improving the solution's maintainability. In order to increase consuming performance the consumer implementation should retrieve several messages from the broker and deliver them as a batch to the handler implementation so that it may write it in batches which increases writing and, consequently, message consuming performance.

5.2.7 UC7 - Delivery Status Notifications

Delivery status notifications are configurable per message, by providing an URL, and an eventual authentication method (*i.e.* user and password) if required, along with the message to deliver. The workflow is similar to the delivery events logging and should be ready to face communication problems with the target URLs as some may be temporarily or permanently unavailable. In figure 5.7 is possible to observe a use case where the first attempt to deliver a status notification fails with an HTTP 5xx status code, which represents an unavailability of some sort on the target URL. The workflow re-delivers the message to the broker with a configurable delay so that this delivery will be attempted once more after this delay. There should, nonetheless, be limits to the amount of attempts for each status notification in order to prevent stacking of notifications which will never be delivered.

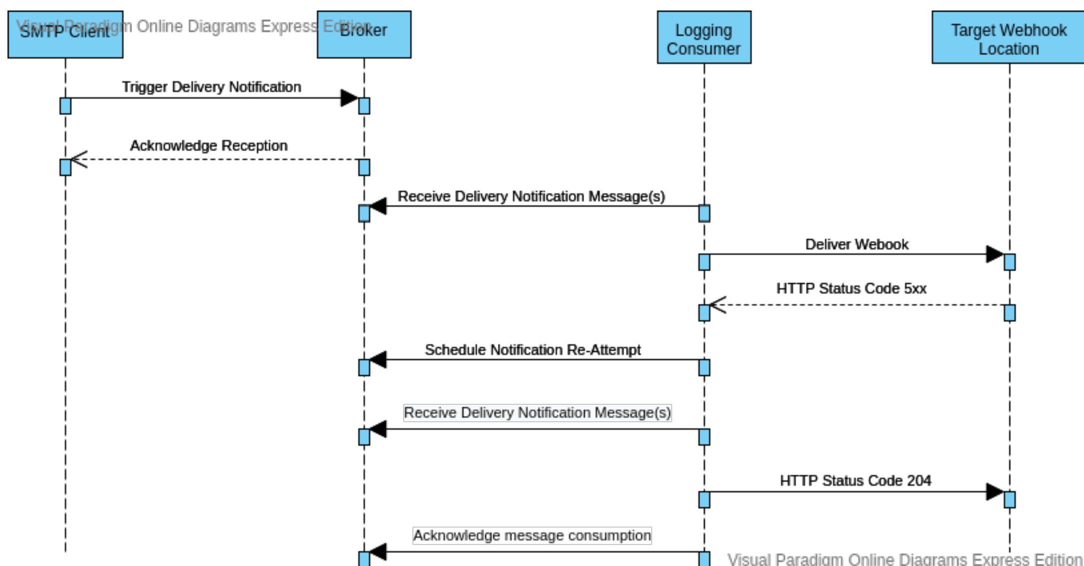

Figure 5.7: Solution's Delivery Status Notification handling sequence diagram.

Due to the high number of notifications which may be generated from a large scale deployment, the implementation must, such as in use case 1 (see 5.2.1), make use of parallelism in order to improve the performance of notification delivery per process while providing the

configuration of a maximum number of parallel notifications in order to allow adjusting the throughput to the target system's capabilities.

5.2.8 UC8 - Group Delivery Processes

Delivery processes are represented by instances of the SMTP client component in the solution's architecture, and each instance may be configured independently from any other. To fulfill this use case, the solution must be able to allow a "pool" configuration parameter per SMTP Client instance which represents the delivery process group to which that instance belongs to, this pool will define the queue in the broker system from where messages shall be retrieved.

When submitting a message for delivery, via realization of Use Case 1, the user must be able to specify the pool from which the message is to be delivered, and the SMTP server will submit the message to the broker queue which represents that pool, if any SMTP client instance(s) are configured to that pool, they will retrieve this message and deliver it, else the message will remain indefinitely waiting for an SMTP client to be configured for that pool.

If a pool parameter is not present when realizing Use Case 1, the message should be sent into the "default" pool, which is the main pool of the solution, to which all configured SMTP client component instances will fallback to in the case they are not configured to any specific queue.

5.2.9 UC9 - Delivery Policies

Delivery policies may represent the limitation of the delivery flow in several manners although only one is part of the requirements, which is a limitation of the amount of parallel deliveries per recipient domain.

While designing the delivery process (see subsection 5.2.1) it was defined that the deliveries would be dispatched asynchronously and that there would be a limit to the maximum amount of parallel deliveries. In order to implement this use case, the SMTP client should receive a configuration property which contains a mapping of the recipient's domain to the maximum amount of parallel deliveries for that domain. The SMTP client will then deliver accordingly to that configuration falling back to the default value if the domain is not present in the configuration map.

5.2.10 UC10 - Stop/Pause/Resume Deliveries

Stopping, pausing and resuming deliveries in the current architecture and delivery workflow is nothing but the act of evaluating a condition prior to the delivery of the message. The SMTP Client component must be capable of receiving notifications during runtime containing a condition and an action (discard or pause), it will keep this data in-memory for fastest access and before each delivery, it will check if it triggers any of the conditions, if so, it will trigger the action associated with it. In order to allow a further integration with integrating business, the conditions should allow evaluating tag's values as well as email message headers' values, which will allow stopping messages through integrating business' logic (*i.e.* if a tag represents a campaign in the E-goi platform, it is possible to stop all deliveries associated with that campaign as they will all contain the same tag) or through delivery logic (*i.e.* stop all deliveries from a certain sender address).

When stopping a delivery, this should be treated as if the delivery was undeliverable, the delivery must not be made and must be logged as "undeliverable" and the reason should be "stopped". When a delivery is paused, the delivery is not be made, the broker message is to be sent back into the broker, with a small delay of at least 5 seconds in order to prevent system clogging with the same message, the delivery is to be logged as "deferred" and the reason should be "paused", these deferred must not count towards the maximum delivery attempts. When a delivery is resumed, the configuration to "pause" is simply deleted from the in-memory structure in the SMTP client component which allows all the paused deliveries to be resumed as soon as they return to the SMTP client.

5.2.11 UC11 - Submission Reports

Submission reports comprise information about created submissions and the transactions which were performed for each of them. Most of the required information is present in the persistent data store, however, the submission details are not being logged and are required parameters. As such, submission logging is required to be implemented.

These submission logs are required to be stored into a persistent data store for each successful submission, however performing this storing synchronously for each submission would gravely impact the performance of the submission interface. In order to maintain the submission interface performance, these logs will be temporarily stored into the message broker, on a 'submission-logs' queue for later consumption and storage into a data store, much alike UC6 (see section 5.2.6).

A specialized consumer will then be able to retrieve a batch of messages from this queue and persist them onto a datastore which will be queried by the SMTP REST API in order to compile results for the requested submission reports. The data to be logged for each submission should contain the following parameters:

- Submission Id: A unique identifier of the submission for which this transaction belongs to.
- Delivery Status: The status of the submission, possible values are "delivered" and "undeliverable".
- Submission Time: The time at which the submission was completed.
- Submission interface: The interface used to perform the submission (currently SMTP REST API).
- Submission length: The time it took for the submission processing to complete.
- Source: The IP address from where the submission request was performed.

The submission and delivery reports should be requested through the SMTP REST API which will then access the persistent datastore where the delivery and transaction logs are stored and retrieve the requested information. This functionality can be used to retrieve data for each submission, through its submission unique identifier or through other parameters such as submission tags, delivery group or delivery SMTP client instance.

5.2.12 UC12 - DKIM Key Management (Extra)

DKIM keys represent a very important resource to most deliveries, especially for large deployments such as it is E-goi's use case, where every improvement to deliverability is of major

importance as it provides a better service for all its customers. In order to create a safe and centralized environment for these keys to be kept, the REST API should provide methods of managing these keys, such as creating new keys, retrieving created keys and updating them. These keys data, will then be stored in the datastore allocated to the solution's instance and SMTP clients can be configured to retrieve the DKIM keys via the REST API (see section 5.2.4).

In order to create a DKIM key, a private-public key pair must be provided, along with the sender domain which they represent. As an alternative, the sender domain alone may be provided and the private-public key pair will automatically be generated by the solution and persisted into the database. An additional parameter should be supported in the latter use case indicating the number of bits of the key to be generated, which, if not present in the request, should default to 1024 bits, the recommended value by the DKIM specification [25].

The REST API will then make these keys available to any system which requires them, in example, the SMTP clients, which as discussed in section 5.2.4, may be configured to retrieve the DKIM keys via URL thus achieving the centralization of the DKIM keys as well as its management. This achievement, not only does it reduce disk usage by preventing duplication of the DKIM keys through all the solution's instances, as it allows these keys to be easily updated, something which may prove beneficial for security purposes such as refreshing older DKIM keys which may have already been compromised or creating a periodic rotation strategy.

Chapter 6

Implementation

Along this chapter we describe the implementation of the solution based on the state of the art analysis (see chapter 3), the requirements defined in chapter 4 and according to the design developed in the previous chapter 5.

Subsections containing the "Extra" word on their title may be found throughout this section and represent further functionalities which have been added to the product albeit they do not comprise requirements.

Implemented as a multiprocessing application, based on the producer-consumer model for message exchanging between its processes through the use of a message broker, the solution's components (SMTP server and client defined in chapter 5) and its interfaces (REST API) are represented as operating system processes. A single parent process, the master process, is spawned at application start and is responsible for spawning and monitoring a set of configured child processes which may be any of the following:

- SMTP API: The process which hosts an instance of the SMTP Server component REST API interface on a configurable network interface and port.
- SMTP Client: The process which represents an instance of the SMTP client component responsible for delivering the email messages to the intended recipients.
- Delivery Logger: The process responsible for persisting the delivery logs onto a configurable data store.
- Delivery Notifier: The process responsible for sending the delivery notifications to their intended target.

All the above processes represent entities of which several instances may be created in order increase its performance, thus, it is possible to increase the REST API interface throughput by configuring the master process to spawn one more SMTP API children and have the traffic be load-balanced between the two instances through the use of a reverse proxy or load balancer, which will technically double the throughput of said component as long as hardware resources are available. The configuration of the set of children to be spawned is totally up to the entity which configures the application and so one may deploy instances which only spawn SMTP API or SMTP Client instances which grants the solution a huge versatility.

It is worth mentioning that several deployments of the solution may be made in different servers independent of their location, using different message broker instances or the same, in order to horizontally scale the system as a whole and provide better availability or performance to the integrating systems.

The application was logically divided into two type of modules, the shared modules and the components modules, the first relates to codebase which is used globally through the application and is shared by all the components modules. Logics such as message broker communication, DNS resolution, email message building, DKIM signing and signature validation, and data compression all belong in the shared modules while logics such as email delivery, request receiving by the REST API and delivery logging make up the components modules and are only used by each of those modules as it is their responsibility and inherent logic.

6.1 Technological Environment

We now present the technologies used in the implementation such as the programming language, the message broker, libraries and frameworks.

The message broker used was Artemis, as discussed in section 3.2.1, since it implements all of the required functionalities such as message priority, message scheduling and provides clustering capabilities for high-availability deployments. Furthermore, E-goi requested that ActiveMQ was used, as it is the current Message Broker technology being used by the product, and it was agreed that, since Artemis and ActiveMQ are currently being merged in order to become ActiveMQ 6, Artemis would satisfy that requirement. In order to communicate with the message broker the application uses the STOMP network protocol.

The whole solution's codebase was implemented using the Python programming language under version 3.6 although efforts were made to keep it compatible with versions 3.7 and 3.8. Several native and third-party libraries were used to ease and aid in the development of the overall solution's codebase, these, along with its main purpose and usage, are listed in table 6.1.

These libraries greatly reduced the development cost as most of them solved complex problems while providing easy to use interfaces and maintaining their main purpose clear and concise.

In order to automate the deployment of the developed code into a staging, and eventually production, environment, a CI/CD pipeline was created using the *Jenkins* automation server. Besides the deployment of new code changes into a staging environment, several tests and validations were integrated into the pipeline in order to ensure better code correctness and quality. The pipeline, comprises the following steps:

1. Most recent changes git checkout.
2. Unit tests execution.
3. *Pylint* code quality analysis.
4. *Sonarqube* Code Quality Analysis.
5. Deployment of changes into staging environment.
6. *Jmeter* Load Testing against staging environment.
7. Deployment of changes into production environment.

If at any time, any of these tests or validations fails, the pipeline fails, consequently not allowing the changes to proceed into production environment until the detected errors are

Table 6.1: Python native and third-party libraries used in the solution's development.

Type	Library	Purpose
Native	*email*	Email/SMTP message building and managing logic.
Native	*smtplib*	Email delivery over SMTP protocol logic integrated with the *email* library.
Native	*multiprocessing*	All the multiprocessing, parallelism and concurrency related logic.
Third-party	*falcon*	Micro-framework which provides REST and HTTP abstractions which ease the development of REST APIs. It was used in the development of the REST API interface for the SMTP server component.
Third-party	*dkimpy*	Responsible for all the logic DKIM digital signing layer.
Third-party	*dnspython*	DNS resolution and caching layer global to each application instance.
Third-party	*stomp.py*	All message broker related communications which are achieved over STOMP network protocol.
Third-party	*sqlalchemy*	Used communicate with SQL fluent data stores such as MariaDB and SQLite.
Third-party	*pylint*	For code quality analysis.

fixed. The pipeline attempts to ensure that only the most correct and valid codebase changes are allowed into production environment thus protecting this environment from faulty code logic or implementation.

6.2 Abstraction Layers

Abstraction layers were created at certain variation points in order to allow for an easier to maintain application, the identified variation points as well as their justification and implementation will be discussed along this section.

Compression algorithms as more and better algorithms may be developed with technology advancement along time that, if implemented, could improve the solution's overall performance and so, it represents a layer which should be easily maintainable and exchangeable. As it is possible to observe in figure 6.1, an abstract *CompressionAlgorithm* parent class was created in order to define a blueprint for compatible implementations for other algorithms, thus allowing expansion of the supported algorithms through the creation of a new child class.

Message broker entities such as Producers, consumers and connections are a core component of the solution's codebase, yet, there are several brokers available, and several libraries to communicate with them which employ different exposed APIs which make it hard to create a single broker abstraction layer. Nonetheless, the broker represents a critical dependency for the solution and so it should be easily maintainable and exchangeable. Consequently, three abstraction layers were created to represent the message broker layer: Consumer (see figure 6.4), Producer (figure 6.3) and Connection (figure 6.2).

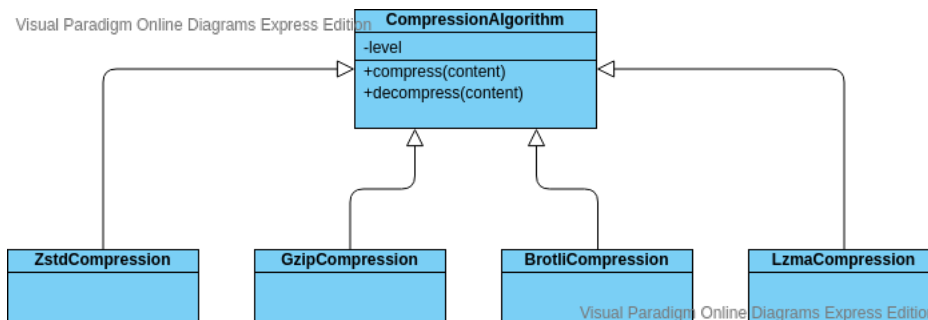

Figure 6.1: Compression algorithms abstract layer.

With the goal of centralizing message broker connection objects into a single point, a static abstract factory class was created which can be easily extended in order to implement connection object creation for several other message broker platforms or different libraries. This class has a single method which can be called to retrieve an alive connection to the message broker system. Every component in the solution uses this method when a connection object is required.

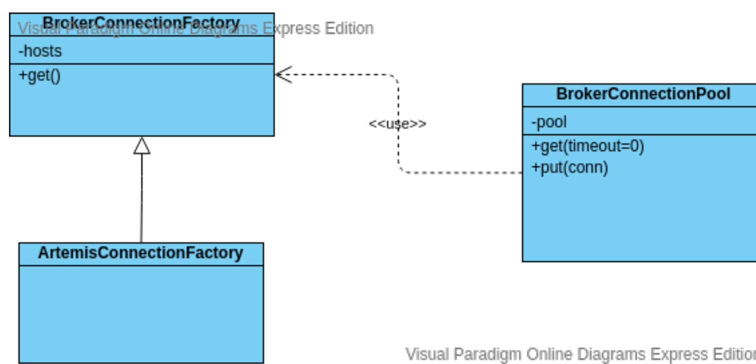

Figure 6.2: Message broker connection abstract layer.

The message producer is where all the logic for sending messages to different queues, with a configured delay, schedule or priority is comprised. It should also be noted that this entity has support for transactions which greatly reduces the delivery time of large amounts of messages to the broker by reducing the roundtrip time thus improving the solution's performance.

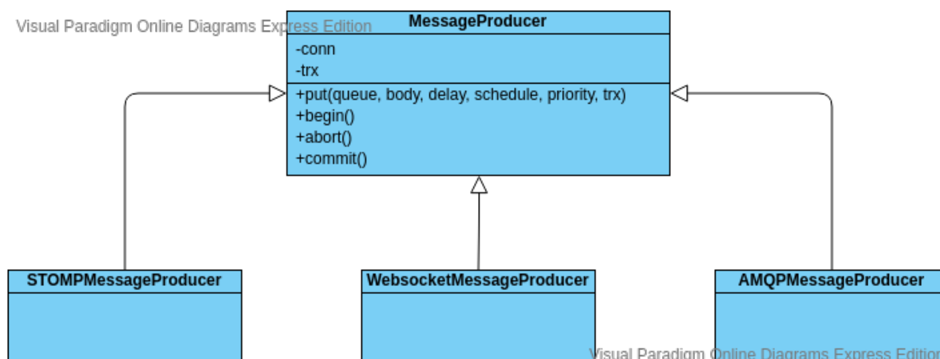

Figure 6.3: Message broker producer abstract layer.

The message consumer holds all the logic behind retrieving messages from the broker system

and an abstraction is very much required due to the fact that there are synchronous and asynchronous implementations of consumer layers amongst the several libraries available, yet the concepts, albeit achieving the same functionalities, are not compatible. The synchronous model, does not make assumptions on the implementation and provides a method which upon call retrieves a message from the message broker, leaving the application to do what it must and later acknowledge the message consumption by calling another method. As for the asynchronous model, a listener is usually created through the implementation of several methods from an abstract class which is later associated with a connection and subscription. These methods are then called asynchronously, usually by a thread that is created upon the association of the listener with the connection, as messages are received from the broker, these listener methods usually represent the reaction to events which happen along the connection lifespan (*i.e.* message reception, a disconnection from the message broker, the sending of a message, etc).

The abstract layer was designed based on a synchronous model, as it makes less assumptions about the implementation and encapsulates less logic thus providing a more controllable API to the components modules. The library used to communicate with the broker, *stomp.py* (see previous section), provides an asynchronous model implementation of consumers, and in order to make it compatible with the synchronous abstraction, a listener was created which receives every message and places it in a thread-safe shared-memory queue, from where the items can be retrieved synchronously by calling the *get* method.

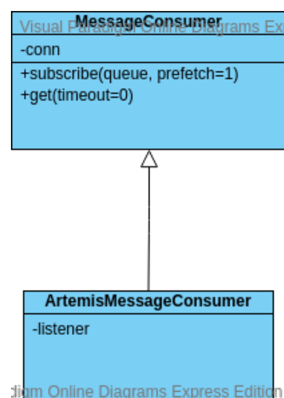

Figure 6.4: Message Broker consumer abstract layer.

Both the message producer and message consumer require a connection to be instantiated which allows using the same connection to consume and produce if such is required, thus minimizing the number of connections which may be used by the solution.

Delivery Logs Handlers were created as an abstract layer to provide support for multiple data stores, whether they'd be SQL based or NOSQL based or anything else that may eventually be developed. As shown in figure 6.5, an abstract class *DeliveryLogHandler* may be extended through the implementation of the *handle* method in children classes in order to support different behaviors for the persistence of the delivery logs in the target data store.

6.3 Implementation Details

Along this section, noteworthy implementation details, if there are any, will be presented for each of the defined use cases (see section 4.1) in order to provide further understanding of the solution's inner workings.

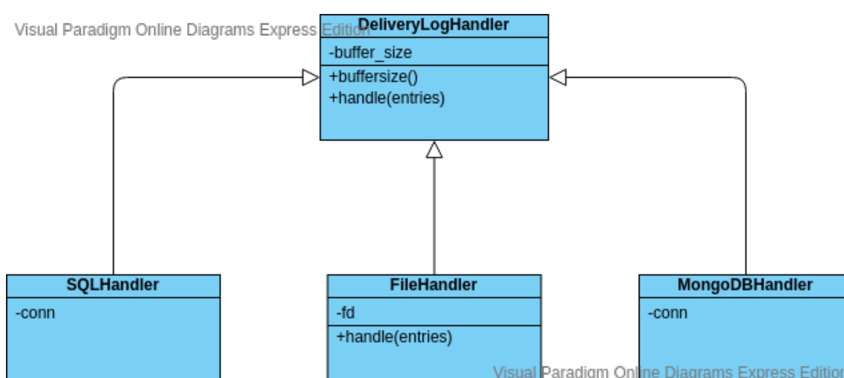

Figure 6.5: Delivery logs handler representation.

6.3.1 UC1 - Message Delivery

Since this represents a complex use case whose logic spreads over different components, its implementation details will be approached in subsections which refer to each of the components.

SMTP API

Maintaining balance and control of the amount of connections to an external system should be addressed by the application such that leaving unused connections open or opening an excessive number of connections does not occur. Many of these external systems implement protections against this to prevent malfunctions, anyway this should be controlled by the application which is making use of them. In order to prevent this, a connection pool mechanism was created. This mechanism is based on an in-memory thread-safe queue where message broker connection objects may be placed into and retrieved from, thus providing the re-usage of the connection objects along the SMTP API interface requests. It is possible, in order to prevent an excess of connections to the message broker, to define a maximum amount of connections for this pool, which allows controlling the number of connections an SMTP API instance is able to open to the broker. Furthermore, being able to reuse the message broker connection objects provides improved the performance in environments where the network roundtrip time is higher and in cases where the SMTP server is under high load as the connection is already connected to the message broker and so, no time is lost connecting to it.

The API implementation features a middleware layer which retrieves a connection from the pool for each submission request and when the request processing ends, it places the connection back into the pool for re-usage.

Message submission related logic was implemented as a single REST resource designated as *submission*, and following the REST design, in order to submit a message for delivery (or create a submission) the POST method should be used. Providing the recipient and sender email addresses and a text content for the message to be delivered are the minimum requirements for a successful submission. An array of submissions may be provided in order to create several submissions through a single API request, thus mimicking the ability to submit several messages over the same SMTP session in the API.

SMTP Client

As mentioned in section 5.2.1, the SMTP client's workflow for a single delivery is, succinctly, to retrieve a message from the message broker, decompress it, resolve the MX record (if non-existent, falling back to the A record) belonging the recipient's domain, open an SMTP session with the respective MTA at that location, deliver the e-mail message and lastly, if this is message is "deferred", requeue the message. This process is highly dependent on the network communication which is prone to fluctuations in latency and eventual failures and so the solution should thrive, as much as possible, to improve its performance on this layer.

The first noteworthy improvement is the time-to-live based in-memory caching of all the DNS resolutions which allows reducing the number of DNS lookups along an entire application instance as this was implemented globally to the entire application instance since several SMTP client instances may be spawned in the same application instance. Furthermore, the cache entries TTL can be configured by application instance and the failed lookups are not cached in order to prevent the constant messaging deferral due to not being able to resolve the recipient's domain address MX or A records.

Moving on to the message delivery layer, a performance bottleneck was mentioned in section 5.2.1 highlighting the impact of network latency in SMTP transactions between MTA that have a high latency between them. A solution was proposed, based on multi-threading, and was implemented using a thread pool mechanism. This mechanism is natively implemented in Python's *multiprocessing* module and provides a thread pool object which allows submissions of "work" to this object. "Work" is just a term used to define an executable function. Once submitted, this work will be queued until a thread from the pool is able to execute it. A "future" object is returned upon submission which allows tracking the status of this submission and know if the "work" associated with it has already been executed. The SMTP client uses this thread pool mechanism to deliver several messages concurrently, and as these deliveries complete, the SMTP client acknowledges the message consumption to the message broker which prevents the loss of broker messages (or deliveries) when an SMTP client instance crashes due to some disaster.

There is, however, something else which may be implemented in order to achieve better delivery performance without incurring into a multi-threaded implementation, which is, once again, a connection pool mechanism. Several MTAs are capable of reusing the same SMTP connection in order to deliver more messages which reduces the overall delivery time, and this can be achieved through connection pooling by the recipient's address domain. This mechanism was implemented based on the same principle as the message broker connection pool yet a queue is created for each recipient domain as it represents a connection to a different external system (a different MTA). It is possible to limit the amount of connections to fit inside the pool per domain in order to respect the recipient's MTA connections limits, if more threads than available connections are delivering to this domain, the threads will wait until a connection is available.

All the layers mentioned above were improved in order to provide a greater delivery performance, leaving a last one remaining, the requeueing of the message, if its delivery status is "deferred", where a safety perspective is most important than a performance one. When a message is deferred, it must be resent to the same queue in the message broker for a later attempt. This action represents a single point of failure before the message acknowledgement is sent to the message broker and after the corresponding email message was delivered

which, in case of unexpected errors, such as network failures, could cause the delivery workflow to fail without acknowledging a delivery attempt. In order to prevent this, a separate parallel layer was created to ensure these message producing events are delivered to the message broker. This layer, denominated Async-Producer, was achieved through the implementation of an instance global separate parallel process which provides a single in-memory queue where messages can be placed by other processes (SMTP Client or Server instances) to be delivered to a certain queue in the message broker, thus eliminating the possibility of errors occurring in the SMTP client delivery process after the message was delivered and before it was acknowledged (except for the acknowledgement itself). Besides improving the safety of the solution, this layer also provides extra performance to the layers which make use of it, as the messages are sent into memory and not to the network (which is much more prone to failures), as well as providing an instance global entity for asynchronous delivery of messages to the broker which may be very helpful in implementing other use cases. The async producer was implemented so that it is able to deal with all broker message delivery failures and so, ensure that no messages are lost and all are delivered correctly to the message broker.

6.3.2 UC2 - Scheduled Delivery

Scheduled deliveries can be scheduled via two parameters, 'schedule' and 'delay'. The first one, sets the delivery to only be sent for delivery at a certain moment in time, defined by a unix timestamp. The second one, sets the delivery to be sent for delivery after a specified amount of seconds after the submission.

This use case was implemented easily thanks to the native scheduler present in the *Artemis* message broker, which provides both the functionalities previously mentioned (delay and schedule). As such, the SMTP REST API provides two per submission fields ('delay' and 'schedule') which when received, are validated and associated with the broker message, coming into effect when the messages are delivered to the broker.

6.3.3 UC3 - Message Priority

The message priority is a per submission setting provided by the SMTP REST API, which allows setting a priority for this submission in the queue which it has been submitted to, consequently allowing submissions with higher priority to be sent to SMTP Clients faster than others with lower priority.

The *Artemis* message broker natively provides this functionality per message broker message which eased the development of this use case. When a 'priority' field is set for a submission, the SMTP REST API validates and appends this setting to the broker message prior to its delivery to the message broker.

6.3.4 UC4 - DKIM Signature

The DKIM signature itself was implemented by using the *dkimpy* third-party library, which is pretty fast and does not represent a bottleneck to the SMTP client delivery workflow. However, the retrieval of the DKIM keys should support both network and filesystem based retrieval methods (see section 5.2.4) in order to provide centralization capabilities for large scale deployments, and this can become a bottleneck. If the network is unavailable, the private key can't be retrieved and so the message cannot be delivered. If the network

latency is high, the key retrieval will take more time than usual. But even if a filesystem based deployment is employed, if disk access is slow for any reason, the delivery workflow performance will be affected. In order to prevent this, an approach was taken to implement an application instance global in-memory caching mechanism which is able to cache the least recently used private keys in order to provide the fastest access as possible and minimizing access to both disk and network. The number of maximum entries in the cache is, of course, configurable which allows being conservative in the memory usage.

In order to support the required file retrieval method mentioned in section 5.2.4, the application instance can be configured to retrieve the key from a file system location or through an HTTP GET request to a specified URL. The configuration property is a string which can begin by "http://", "https://" or "file://", followed by the URL or filesystem location, in which a "domain" string can be used and will, during runtime, be replaced by the sender domain. This implementation provides a simple and straightforward implementation to ease the DKIM signing configuration along both small and large scale deployment of the solution.

6.3.5 UC5 - CC and BCC Support

As described in the design section 5.2.5, the CC and BCC support implementation will maintain the message broker's messages atomicity by creating all the combinations of e-mail messages to be delivered when CC or BCC fields are present in the submission request. As such, when the SMTP REST API receives a submission request of one message with two CC addresses, it creates three messages in the broker, each of them with the corresponding CC/BCC addresses, and attached to them, the same email object string representation.

On the SMTP Client side, when a message is received, it is validated for CC addresses, and if such are present, and after the message string representation is uncompressed, it's CC/BCC email message header is set with the addresses and names present in the retrieved broker message.

This implementation adds a very small logic variation to the SMTP Client in order to maintain message atomicity on the broker side as one broker message represents one and only one message to be delivered. Albeit a variation is added to the SMTP client, there is no performance loss as it simply sets a header on the email message before delivering it, and since the logic added is really simple, it does not represent something which might affect the delivery process debugging at any time.

6.3.6 UC6 - Delivery Logging

The delivery logs represent a set of data regarding all the deliveries made by the solution which is to be persisted into a data store for accounting purposes. It is important to note that these logs may be the only information left to aid in asserting whether a message was delivered or not, which can be useful when errors, such as failures in the broker message acknowledgments, occur and it is imperative that the corresponding email message is not redelivered or lost.

The only moment where enough information is available to fulfill the required data for the delivery logs is after the email message is delivered, as only then will we know if the delivery was successful or not. In order to implement this use case, the Async-Producer (see 6.3.1) was once again used, as it maintains the delivery performance and ensures this message will be delivered to the message broker even if network is unavailable for a long period of time.

After delivered to the broker these messages are the consumed by another process, the delivery logger, which stores them into a data store using the configured log handler (see previous section 6.2). It is only possible to spawn one instance (per application instance) of this delivery logger as more would cause concurrency in the target data store and since this is a highly IO bound process, it would only slow both instances down. It is however possible to increment the buffer size to allow the process to consume more messages at once and write them in larger transactions into the target data store which often increases the overall writing performance.

6.3.7 UC7 - Delivery Status Notifications

Delivery status notifications are webhook notifications which should be sent as close to the delivery moment as possible since they represent that a delivery event occurred at a certain moment in time and this may be useful for the integrating application to show a percentage of completion for a set of deliveries to its customers and alike features.

Similarly to UC6 (see 6.3.6), the moment past email message delivery is the moment at which more information is available regarding the delivery, as such, this is also where the delivery status notifications will be sent to the broker. And as in the previous UC, the Async Producer will be, once again, reused to fulfill the delivery of this message assynchronously without adding points of failure to the SMTP client process and without performance losses.

However, as in UC1 (see 6.3.1), the delivery of webhooks is a highly network IO bound process and as such, it would not be scalable to deliver the webhooks one by one and this is where parallelism will aid. Although several instances of the delivery status notifications process could be spawned, using a threadpool similar to what was implemented in UC1, will reduce the resource usage (due to the lower resource consumption that threads provide relative to processes) and it will be possible to deliver several webhooks in parallel thus thus improving the performance for each instance of the delivery status notifications process. The number of threads in the threadpool is configurable in order to allow tuning the performance relatively to the webhook target system if such represents a concern.

6.3.8 UC8 - Group Delivery Processes

The delivery process grouping is fairly simple as it uses basic message broker logic. Each SMTP Client instance is able to subscribe to a single message broker queue from where it retrieves messages to deliver. This is a configuration which must be set for each SMTP client in the solution's configuration file.

When creating a submission via the SMTP REST API, it is possible to set, for each submission, a 'pool' field which represents the name of the queue to which the message will be delivered. If none is specified the delivery will be sent into a 'default' queue. If a pool is specified which does not yet exists, the queue will be automatically created.

6.3.9 UC9 - Delivery Policies

The delivery policies, as discussed in section 5.2.9, represent a set of manners in which the delivery flow of an SMTP client instance may be limited in order to fullfil deliverability requirements or delivery requirements of MTAs of specific recipient domains. Whatever the use case, the only required limitation was the number of parallel deliveries per recipient domain, which means that for certain configured recipient domains, the SMTP client must

only deliver at most X messages in parallel. This implementation will allow keeping under control the number of opened connections to a recipient's domain MTA, which represents one of many deliverability constraints usually employed by receiving MTAs in order to control the inbound flow of messages and protect against SPAM attempts.

Fortunately, an SMTP connection pool was implemented in section 6.3.1 to provide better delivery performance, and this structure, already groups connections by recipient domain. As such, in order to implement this requirement, it is only required that the administrator of the system is able to define a mapping, per SMTP client instance, of the recipient domain to the limit value which will limit the number of connections which may at most be in the connection pool for that domain. As mentioned in UC1, parallel delivery threads wait for a connection to be available in the pool before proceeding to the delivery, together with the imposed limitations, this provides the fulfillment of this requirement's implementation.

6.3.10 UC10 - Stop/Pause/Resume Deliveries

Artemis message broker as most, provides its clients with two different message delivery methods to consumers, *ANYCAST* or *MULTICAST*. *ANYCAST* delivers each message to a single subscribed consumer, thus allowing message consumption to scale (faster consumption) with the consumers number. *MULTICAST* delivers each message to all subscribed consumers thus allowing all consumers to receive the same message. Each behavior is configurable per queue, and useful for different use cases.

UC10 implementation was achieved through the usage of message broker queues *MULTICAST* behavior. Every application instance with SMTP Client instances will automatically spawn a consumer for a 'notifications' *MULTICAST* queue which will be listening for messages from this queue.

It is possible, through the SMTP REST API, to stop, pause or resume the delivery of created submissions. When requested, the SMTP REST API sends a message into the 'notifications' queue which is then consumed by the subscribed consumers. Each consumer upon receiving a message updates its instance in-memory pre-delivery settings to stop, pause or resume the delivery of a respective submission. These settings are also flushed into disk in order to endure instance restarts or unexpected crashes.

Simple logic was then added to SMTP Client in order to validate for each message to deliver (prior to its delivery processing) if the related submission is paused or stopped, which if so will take the according action of rescheduling the delivery or discarding it respectively.

6.3.11 UC11 - Submission Reports

The submission reports required the implementation submission logging into a persistent data store, which followed an implementation much alike UC6 (see section 6.3.6). As discussed in section 5.2.11, the impact of submission logging in the SMTP REST API interface must be as low as possible. In order to accomplish it, the submission logging data will be first delivered into the message broker to a specialized queue. A negligible performance loss for this implementation was achieved by sending the submission logs broker message together with the submission messages.

With the messages being temporarily stored in the broker, a consumer was developed by abstracting the UC6 consumer to a configurable datastore and broker queue, thus completing the submissions logging functionality.

In order to retrieve the reports, it is possible to query the SMTP REST API, which is able to retrieve data from the persistent data store, by configured query parameters which identify the required data.

6.3.12 UC12 - DKIM Key Management (Extra)

DKIM key management was implemented in order to fulfill a need for centralization of the DKIM keys which will both save disk usage (by not being replicated through several file systems) and by providing ease of management of DKIM keys. Through the addition of the *dkim-key* REST resource to the REST API, implementing the GET, POST, PATCH and DELETE methods, the use case implementation was achieved.

As mentioned in section 5.2.12, if no key is provided in the request for key generation, the application must be capable of generating one. This is achieved by generating an RSA-256 public/private key pair which is stored together with the sender domain, if one for that domain already exists, the create request is rejected.

It is possible to retrieve the details of a DKIM key through a GET request to the *dkim-key* resource, however, SMTP clients searching for a DKIM private key to sign email messages assume the response body is only the private key in plain text. In order to keep the SMTP clients configuration simple, this resource allows filtering the required parameters through a query parameter named *fields* and if the *Accept* HTTP header is set to *text/plain*, the response will fulfill the requirements for the SMTP clients to successfully interpret the response body and retrieve the DKIM key.

Chapter 7

Experimentation and Evaluation

7.1 Hypothesis Enunciation

As previously mentioned in section 1.2, the current solution's biggest problems are the high hardware requirements, an outdated and weakly performant integration interface (SSH protocol text file copying) and the low amount of functionality regarding e-mail deliverability. To evaluate whether the developed solution fulfils the defined requirements (provides a worthy improvement when compared to the currently deployed solution), the following hypothesis have been defined:

1. Improve the single delivery message submission time (REST API) to at most 40 milliseconds while not under load.
2. Improve the submission interface parallel request handling to at least 20 single delivery submissions per second.
3. Improve the number of new functionalities by a factor of 3.
4. Lower Hardware Resource Requirements by 25%.
5. Improve the current delivery speed per single delivery.
6. Maintain the best code quality according to *Sonarqube* and *Pylint* code analyzers (Rate 'A' in Sonarqube).

The proof of these hypothesis will grants us a measure of how capable the developed solution is of solving the current's solution problems and bottlenecks, and consequently, if it represents an improvement when faced with the previous. the following sections will present the methodologies and magnitudes used to measure and evaluate the hypothesis defined along this section, as well as the technologies involved in it.

7.2 Hypothesis 1 and 2

Since hypothesis 1 and 2 are tightly related, as both of them assert the solution from a performance and load resiliency point of view, they will be evaluated using the same methodology.

7.2.1 Methodology

Hypothesis 1 and 2 intend to evaluate the developed solution from a performance and reliability stance. The first hypothesis, defines a time limit of 40ms for the submission

of a single delivery in the system while not under load, thus defining high-performance requirements for the submission action. The second hypothesis defines an amount of parallel submission requests the system should be able to handle per unit of time, which will provide a view of the system's reliability while under load. The submission moment, refers to the moment when the delivery has been queued for delivery and is safely residing within the MTA system. In the current solution, this moment refers to the moment in which the REST API answers with a 202 HTTP Status Code to the message submission request.

Apache Jmeter, is a stress and load automated testing utility which is capable of simulating several parallel requests per second as well as recording all the requests' responses status, errors and times which represent valuable data in proving these hypothesis. As such, a set of tests will be configured in *Apache Jmeter* according to the hypothesis definitions and the test results will be thoroughly analyzed. Variables such as the ratio of successful and failed requests due to system overload as well as the standard deviation from the request time limit define by hypothesis 1 are of utmost importance.

7.2.2 Results

As defined in section 7.2.1, hypothesis 1 and 2 would be proven by gathering data from the solution's REST API interface through the use of the *Apache Jmeter* tool. This tool provides several test scenario configuration parameters, yet only the following were used:

- Virtual Users: The maximum number of parallel active virtual users which may be sending requests to the REST API.
- Ramp Up Length: The length throughout which the active virtual users count should be incremented until the maximum.
- Test Length: The length for which the test should run after the maximum count of virtual users has been reached.

Adding to the definition of these parameters were the REST API submission resource URL and the the respective body content for a single delivery submission as well as a maximum timeout of 1 second per request. As previously mentioned (see section 6.1), a load testing scenario was integrated into a continuous delivery pipeline and is executed for each change to the codebase. The results for these tests are published for each pipeline execution and provide individual and trend reports.

It should be noted that these tests were executed against a staging environment composed by a single container, with the following hardware specifications:

- CPU: 8x Intel(R) Xeon(R) CPU D-1541 @ 2.10GH
- RAM: 4GiB DIMM DDR4 Synchronous 2667 MHz
- Disk: Samsung SSD 970 PRO 512GiB
- Filesystem: ZFS RAIDZ-2 (4 disk array)

As per the application itself, this was deployed with 8 instances of the SMTP REST API load-balanced by an Nginx reverse proxy which distributed the requests evenly through all the API instances. The Artemis broker instance resided in the same container and application access to it was achieved via localhost connection in order to discard any network latency between the application and the broker system. It should be noted that between each

scenario testing, both the application and broker instance were restarted and all the queues were purged so that the environment conditions are as equal as possible for all test runs.

Table 7.1: *Jmeter* load testing scenarios.

Scenario	**Virtual Users**	**Ramp Up Length (s)**	**Test Length (s)**
#1	20	30s	1m
#2	20	30s	3m
#3	30	30s	3m
#4	40	30s	3m
#5	50	30s	3m

Table 7.1 presents the definition of the *Jmeter* test scenarios which were used to test the hypothesis. Scenario 1 relates to the test scenario, which has been integrated into the continuous delivery pipeline, which was created in order to fullfil the previously defined hypothesis environment variables (see section 7.2.1). Scenarios 2, 3, 4 and 5, in which the test length was increased to 3 minutes and the virtual users parameter is increased until 50, were created in order to further test the solution's limits based on the staging deployment.

Table 7.2: *Jmeter* load testing scenarios results.

Parameters/Scenario	**#1**	**#2**	**#3**	**#4**	**#5**
Total Requests Executed	30616	76824	82879	85222	84026
Avg. Throughput (req/s)	340.18	365.83	394.66	405.82	400.12
Error Requests Percent (%)	0.24%	0.32%	0.36%	0.27%	0.36%
Min. Response Time (ms)	14ms	13ms	14ms	14ms	14ms
Avg. Response Time (ms)	48ms	50ms	70ms	90ms	115ms
90% Response Time (ms)	79ms	79ms	115ms	156ms	197ms

The results obtained for all scenarios, are presented in table 7.2. From the results of scenario 1, it is clear that both hypothesis 1 as 2 are successfully proved. The minimum request time, which matches the response time for the first requests, and thus, the moment at which the system was not under load, is well below 40ms at 14ms. Furthermore, with an error percentage of 0.24% and an average throughput of 340 requests per second, the solution's staging deployment is more than capable of handling the kind of traffic defined in hypothesis 2. It should also be noted, that the average response time while under load of 48ms is very much close to the time limit defined for a "not under load" environment which further proves hypothesis 1.

From observing all scenarios, it is possible to conclude that the test length parameter does not have much impact on the numbers as a scenario for 1 minute (scenario 1) and another for 3 minutes (scenario 2) achieved closely the same results, thus proving that the solution is able to withstand long length high load scenarios. Scenarios 3, 4 and 5 only increase the number of virtual users and the application is able to deliver the same performance, maintaining the throughput close to 400 requests per second albeit increasing the average response time. However, as a scalable application, more SMTP REST API instances and

more containers may be added in order to achieve a lower load per instance and so improve the response time of each of them.

7.3 Hypothesis 3

7.3.1 Methodology

Hypothesis 3 evaluates the developed solution from a functionality stance, providing insight on the improvements in functionality when compared to the current solution, which has been deemed of great importance in section 1.2. It is however fairly simple to evaluate as the enumeration of the new functionalities and their impact in the E-goi platform will suffice.

7.3.2 Results

Besides creating a distributed, scalable and highly-available architecture, a requirement for the solution was to increase the number of functionalities available, which is the core of hypothesis 3. Comparing the new solution with the previous, the following five added functionalities were identified:

- Submission Delivery Notifications.
- Submission Priority.
- Submission Scheduling.
- Centralized submission and transactions logs.
- DKIM key retrieval over HTTP.

All these functionalities are of high value for the solution as they provide simple resolutions to impossible problems in the old solution. Delivery notifications provide E-goi with real-time information about the deliveries which have been processed. Submission priorities provide a way to order deliveries, which may be used as a sales advantage or to prioritize deliveries for internal alternative systems such as Slingshot (see section 2.1.6). Submission scheduling allows the scheduling of a delivery to a certain moment in time or delay it for a certain amount of seconds, which may be useful to implement delivery throttling for an E-goi campaign or to process a scheduled campaign without sending it (see section 2.1.5).

Further functionalities, such as the centralization of submission and transaction logs into an indexable database provide the deliverability team with historical raw data for traffic analysis, report generation and insights on the submissions and delivery transactions. The DKIM key retrieval over HTTP allows the centralization of the keys into a centralized location, something that in the old solution could not provide as it required disk access to all DKIM keys.

7.4 Hypothesis 4

7.4.1 Methodology

One of the biggest problems with the current solutions lies within the high hardware requirements associated with its deployment and the fourth hypothesis intends to evaluate this by stating that the new solution must be able to lower this variable by 25%. Since the new

system relies on a distributed architecture while the current one relies on a monolithic architecture, the effects of this change should be taken into account when analyzing the values for this hypothesis, as they may not support this hypothesis for small scale deployments but do so for large scale deployments.

Taking into account the current solution's minimal deployment of a single container allocated with 1 CPU core, 4GiB of RAM and 8 postfix instances to be used for deliveries, this hypothesis will be tested by analyzing the resource usage of both the solutions while under two moments:

1. Upon container boot, without any deliveries having occurred.
2. Upon container boot, during and after 30 deliveries of the same message to 3 different recipient domains have occurred.

It should be noted that the analysis of the second moment, should feature data gathered from the duration of the 30 deliveries as well as immediately after in order to provide resource usage data during the deliveries which may be higher for any of the systems while under load and lower after load has passed and memory cleanup processes have been applied.

7.4.2 Results

Figure 7.1 shows a progression of the global memory usage throughout the length of the test for both solutions. As the plot shows, the developed solution has a higher memory consumption than the postfix based one, however, it should be noted that the first, is a distributed system, as such, capable of distributed deployments and layer specialization. The developed solution's higher memory usage is mainly affected by the message broker memory requirements, yet, the queueing layer is able to be centralized in order to save resources in larger deployments (as it is E-goi's use case), something which was impossible in the previous solution.

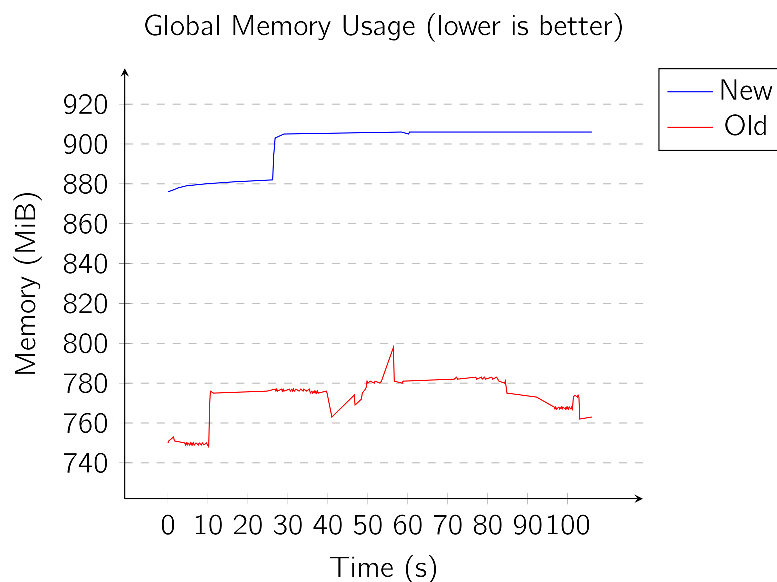

Figure 7.1: Memory usage test case results (lower is better).

To this end, throughout the test, the Artemis message broker used a mean of 650MiB worth of memory, if we were to subtract this memory usage to the memory usage for the developed solution, the plot in figure 7.1 would become something alike the one in figure 7.2.

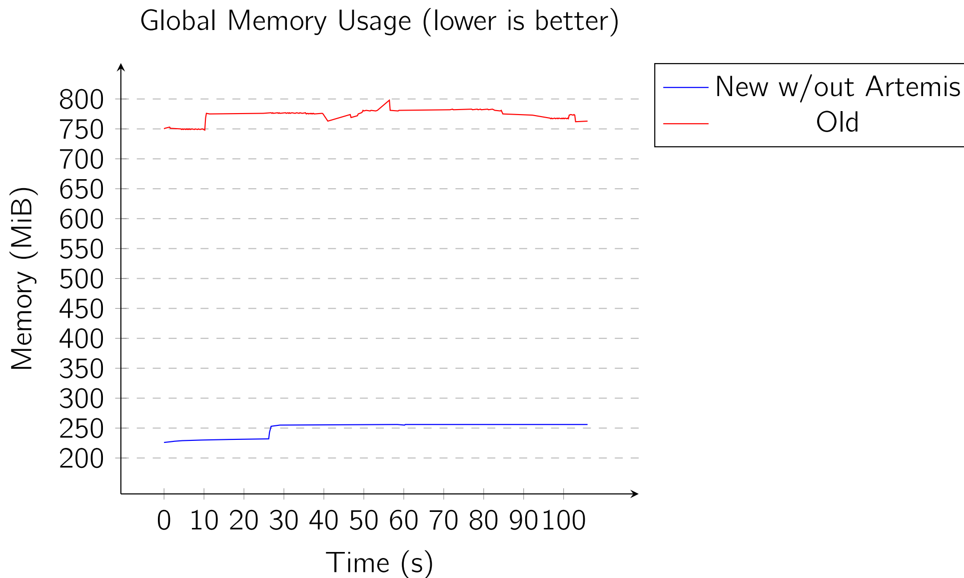

Figure 7.2: Memory usage test case results without Artemis (lower is better).

It is clear to see the reduction of memory consumption achieved by the developed solution in two comparable deployments of both solutions. Table 7.3 shows a comparison of the memory usage test results for each of the solutions. Comparing both the old and new (without Artemis as the layer will be centralized) solutions it is possible to highlight a 71% reduction in the global memory usage. Furthermore, throughout the test, the old solution memory usage increased by 50MiB while the new solution increased by only 40MiB from the baseline.

Table 7.3: Global memory usage comparative table.

Statistic/Solution	Old (MiB)	New (MiB)	New w/out Artemis (MiB)
Mean	773	899	249
Median	775	905	225
Minimum	748	876	98
Maximum	798	906	148

A reduction of 71% in memory usage in the developed solution is more than enough to fulfill the hypothesis defined. Taking into account that, for each group of eight postfix instances, a reduction of 71% comprises 550MiB worth of memory and that E-goi currently has 57 (in Portugal only) groups of such, it is expectable to have, at least, 31.350GiB worth of memory to use for the Artemis message broker instances before exceeding the baseline memory usage for the old solution.

Albeit the hypothesis objective was only to reduce the memory usage, this test case also revealed a reduction in the CPU usage. While the old solution used a mean of 60% of one CPU core throughout the test length, the new solution was able to reduce this usage

by 45%, achieving a remarkable 22% usage of one CPU core which also provides further hardware cost reductions.

7.5 Hypothesis 5

7.5.1 Methodology

The single delivery speed is the magnitude proposed by the fifth hypothesis and represents time it takes to deliver a message from the moment the message delivery request is sent to the system until the first message delivery attempt is made. The magnitude represents the time from the moment the message enters the MTA system and the time this system logs its first delivery attempt. This hypothesis intends to evaluate the improvement in the overall system workflow which technically represents the immediate performance improvement (if such is achieved) for the E-goi platform upon changing to the new system.

The hypothesis will be proven through the statistical analysis of the delivery time of the same message, to the same recipient via freshly booted instances of each of the solutions (both current and new). An amount of at least 50 tests will be made to ensure that caching mechanisms and other performance improvement mechanisms implemented in the solutions are used and the statistical analysis of these tests will output the results which will prove, or not, the hypothesis. Furthermore, the message recipient used for testing will belong to an e-goi mail server so that network latency and fluctuations are as much disregardful as possible between each test.

7.5.2 Results

There are several moments in the solution's workflow in which performance improvements can be expected and these moments should be separated from each other if possible along the testing of this hypothesis. Hypothesis 5 purpose is to evaluate the performance at each of these moments and demonstrate whether there were, or not, improvements in the delivery workflow. The moments where performance can be expected are the following:

- Message Submission - The submission of the message.
- Message Processing - The processing of the message on the SMTP client side, which includes DKIM/ARC authentication.
- Message Delivery - The delivery of the message, which includes the DNS resolution and recipient mail server connection and message transfer.

Postfix provides logging of the length, in seconds, for several moment of the delivery of a message along the mail transfer agent. These moments, according to the documentation, are the following:

- Time before queue manager (including message transmission and milters time)
- Time in queue manager
- Connection setup time (including DNS, HELO and TLS)
- Message transmission time

Having executed the defined delivery test fifty times as previously discussed, all the values were gathered from the postfix instance logs and the statistical median for each of them was calculated, resulting in the following time lengths per moment:

- Time before queue manager: 0.110
- Time in queue manager: 0.014
- Connection setup time: 0.012
- Message transmission time: 0.140

Postfix delivery moments, however, do not correspond to the delivery moments defined for the test, and so must be adapted. It is clearly possible to associate the "Time before queue manager" moment as the message processing stage, as this is where the DKIM and ARC authentication and more processing occurs. The "Connection setup" and "Message transmission" stages correspond to the defined message delivery moment and so they will be summed in order to adapt to the scenario. The submission time was measured as the time it took for the SMTP transaction to complete between E-goi and the postfix instance which would eventually deliver the message and the statistical median obtained was of 102ms. It would however be interesting to compare how much more performant is the SMTP submission relative to the SSH submission, and so, these results will be presented further ahead.

Regarding the solution's results, their data was gathered from the application's logs, which was adapted in order to record the length for each of the required moments. As such, the length of the submission corresponds to the length of the POST requests required to create the submission. The message processing stage corresponds to all the processing the SMTP client component does before delivering the message and the delivery length is the time it took to deliver the message, including resolving MX records and connecting to the recipient mail transfer agent.

Table 7.4: Single delivery speed test results.

Parameter/Solution	**Old - SSH (ms)**	**Old - SMTP (ms)**	**New (ms)**
Message Submission	3124	102	17
Message Processing	97	110	3
Message Delivery	153	152	28

Table 7.4 holds the results for the defined testing scenario for each of the solutions, the old and the new one, and its respective values for each of the defined moments. It should be noted that, in order to provide a sense of magnitude and richer results, the old solution has been tested via both possible submission protocols, SSH and SMTP. From this table, it is possible to see that the SSH submission is much slower than the SMTP submission, however, a submission of a larger amount of messages (*i.e.* thousands or tens of thousands) would take much more than 3 seconds and that is why the SMTP submission is used for only single delivery submissions. However, it is clear that the new solution is capable of much higher performance in all of the moments defined for single delivery submissions, thus proving hypothesis 5 true.

7.6 Hypothesis 6

7.6.1 Methodology

Sonarqube and *Pylint* (supports only *Python*) are well respected tools which perform automated code reviewing, detecting bugs, code smells and security flaws within static code. Hypothesis 6 was defined in order to defend a code quality baseline for the solution's codebase. Integrated into the solution's continuous delivery pipeline is the *Pylint* code review which generates a report that is then fed into the *Sonarqube* code analysis which produces the following parameters:

- Bugs: Errors which cause the code not to properly work whether generally or in specific use cases (*i.e.* logic flaws, syntax errors, unused variables or code chunks, use of uninitialized variables, etc).
- Vulnerabilities: Well-known exploited vulnerabilities (*i.e.* hard-coded sensitive data, SQL injection flaws, wrong variable access modifiers, etc)
- Code Smells: Bloating or dispensable code, object oriented programming abuses and coding style flaws.
- Security Hotspots: Wellknown code chunks which have lead to security vulnerabilities (*i.e.* Weak cryptography hashing, regular expression DOS, etc).
- Duplicated Blocks: The percentage of duplicated blocks of code along the codebase.

Since this analysis is integrated into the continuous delivery pipeline, it's results are updated for each modification that is made to the codebase and if a certain threshold is breached this analysis is capable of failing the pipeline in order to prevent the code from moving onto staging environment while the errors are not fixed.

The results for this analysis provide valuable insights into the quality of the codebase implementation and assure that the quality is kept constant as changes occur to it through time.

7.6.2 Results

Defined in order to provide a code quality baseline for the solution's codebase, hypothesis 6 results regarding the latest stages of the codebase development, are described in figures 7.3 and 7.4, which show the code quality review results over time.

Figure 7.3 show the historical data for bugs, code smells and vulnerabilities as per Sonarqube analysis, where it is possible to see that both bugs and vulnerabilities were practically non-existent along the codebase development, however, code smells were clearly more present along it, being fixed throughout the development.

Figure 7.4 provides a visualization of the duplication of code lines or blocks along the development of the codebase (with a total of about two thousand lines of code), which has been reduced to 0% after an initial increase to about 4%, something which was later fixed, bringing this value back to 0%.

As such, hypothesis 6 is proven as *Sonarqube* rated the code quality with an 'A' and the metrics evaluated are mostly on their best possible values.

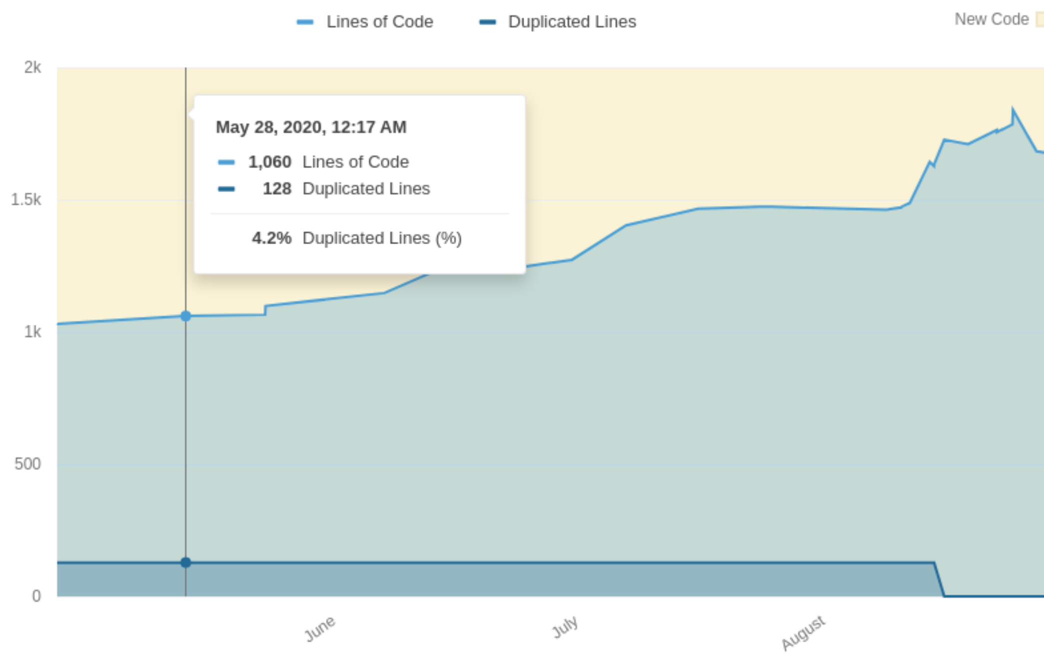

Figure 7.3: *Sonarqube* code quality analysis graph.

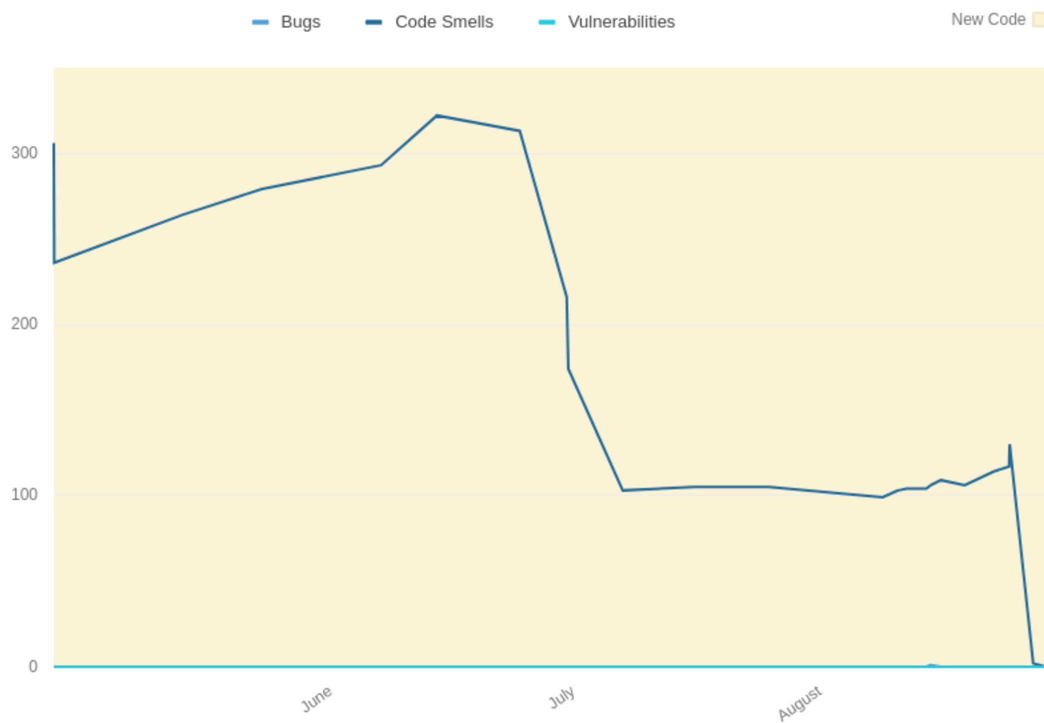

Figure 7.4: *Sonarqube* code duplication analysis graph.

Chapter 8

Conclusions

The overall development of this project, followed a set of guidelines, presented in section 1.4. These guidelines comprised an in-depth study of the current solution in order to best understand its purpose, its functionalities and its flaws and, from there, identify the requirements for a better solution.

We proceeded with an analysis of existing complete or partial solutions followed by a detailed search of technologies, development patterns and methodologies which may abide the defined requirements and solve the identified problems and flaws. Then we proceeded with the analysis of all the data gathered and its conjunction with the requirements defined, the analysis of the problem and solution as a whole, the strict definition of the solution's functional and non-functional requirements, and the evaluation of the solution's business value and the justification for its implementation.

We developed the design of a solution from a higher grained point of view, through a distributed, scalable and malleable architecture, to a lower grained one, through the design of each use case required for the solution taking into account performance, safety and scalability requirements. The solution's implementation, abiding to the designed artifacts and to good software development practices and patterns, achieved a final result which succeeded in the stage of testing and validation.

Along the following sections we summarize the development of this thesis, its main achievements and potential future work.

8.1 Achievements

The first objectives in the project, were the in-depth study of the encountered solution, in order to understand its purpose, which needs it suppressed and which it didn't and to enumerate its flaws and problems which, according to section 1.2, are the following:

1. Low amount of functionality.
2. High hardware resource usage.
3. Decentralized logging data.
4. Laborious setup per instance.
5. Monolithic architecture.
6. Milter (OpenDKIM and OpenARC) SPOF and TCP forwarding.
7. Milter high memory usage.

8. Transactional data scattering.
9. Local postfix instance mail relaying.
10. SSH protocol integration interface.

All these flaws were addressed in the developed solution. The logging data was persisted in a centralized SQL proficient datastore. Each delivery process, is now responsible for signing the messages it delivers with DKIM signature which eliminates the milter SPOF and TCP forwarding problems. Furthermore, the DKIM/ARC private keys required for deliveries are loaded into memory using an LRU cache, whose size is defined by the user in the configuration file, thus resolving the high memory requirements for DKIM/ARC signatures. The email message delivery is performed directly with the recipient's mail server (no mail relaying) which also eliminates the need for an ARC signature. A REST API was developed as the system's entry point which provides a much simpler and easy to integrate with integration interface. The new solution features a malleable, scalable and distributed architecture which allows for several deployment options, which are discussed in depth at section 5.1.2. A simple and single JSON formatted configuration file is responsible for the overall solution's instance configuration, providing much easier deployments of new instances when horizontally scaling. All the transactional data is now stored in a centralized message broker layer preventing transactional data scattering. The amount of functionalities has been increased by a factor of five and the hardware usage per instance has been lowered by 33% (see sections 7.3.1 and 7.4.2).

Having achieved a scalable solution which solves all the problems identified in the previous solution it came the time to thoroughly test it, proving its resiliency to errors, its throughput and overall capacity and finding its limitations as well as assuring its codebase quality. Load testing scenarios were created and triggered against a deployment of the application in order to understand the throughput of the developed integration interface and evaluate its performance. The hardware usage under these situations was carefully monitored and compared with the one of the previous solution as to deem the best one. The overall transaction speed was evaluated as to find which solution could finish a single delivery faster. And the codebase was continually assessed along its development in order to ensure the best quality scores within the available tools.

The developed solution achieved all its main objectives, producing a valuable product as output which surpassed all the tests performed to it and so it was deemed ready and successful for a first trial of integration scenarios with the E-goi platform.

8.2 Future Work

Albeit having achieved all it proposed to, there are several additions which would be worth the development to further improve the solution. In order to, according to RFC5321, become a fully fledged mail transfer agent, the solution would require an SMTP server interface, capable of receiving submissions over SMTP protocol. Not only would this interface make the solution compliant with RFC5321, it would also allow the reception of delivery status notifications (DSN), which are usually delivered by other mail transfer agents with whom previous deliveries were performed to inform us about the status of said deliveries.

Additional delivery policy configurations such as being able to define a specific delivery rate (never deliver more than X messages per unit of time for a specific recipient domain) per

SMTP client instance would provide an interesting alternative or complement to the currently implemented maximum number of concurrent connections per recipient domain.

The instance configuration and deployment of new instances has been greatly eased by implementing a simple configuration syntax which can be edited by other applications, however, remote access is still required to the configuration file in order to change its parameters. A valuable addition would be the storage of these configurations into a centralized database which would allow new instances to be configured from the REST API as soon as they're registered in a master instance. A web user interface could then be implemented to further ease the configuration and management of the whole deployment.

An interesting extra feature would be time-based deliveries, these deliveries would only be delivered at certain time intervals of the day (*i.e.* between 6PM and 8PM). This is especially useful for users who are not looking for their campaign to be delivered in a single day, but are more worried about the time of the day at which this is delivered to its subscribers.

With the SMTP protocol based SMTP server interface developed, one would be able to start receiving DSN and other feedback from several MTAs to which deliveries had occurred. This feedback usually provides information about failed deliveries, and so, it represents an event for a delivery which is capable of changing the status of the related submission. Adding logic capable of recognizing this feedback and automatically processing it, updating the submission status and eventually (if configured) delivering status notifications back to the submitter would represent a very useful feature.

There are several features which can be added, yet, care must be taken in order to keep the solution as abstract from any business logic as possible, as this abstraction makes the solution useful for a much larger market share, which increases its selling revenue.

Bibliography

[1] William D Perreault and E Jerome McCarthy. *Basic marketing: A global managerial approach*. McGraw-Hill/Irwin, 2002.

[2] Mohammed Nuseir and Hilda Madanat. "4Ps: A Strategy to Secure Customers' Loyalty via Customer Satisfaction. International Journal of Marketing Studies; Vol. 7, No. 4; 2015 ISSN 1918-719X E-ISSN 1918-7203 Published by Canadian Center of Science and Education". In: *International Journal of Marketing Studies* 7 (July 2015), pp. 78–87. doi: 10.5539/ijms.v7n4p78.

[3] Dave Chaffey. *Total e-mail marketing*. Routledge, 2006.

[4] *DomainKeys Identified Mail (DKIM)*. (Accessed on 02/21/2020). url: http://dkim.org/.

[5] *ARC Specification for Email*. (Accessed on 02/21/2020). url: http://arc-spec.org/.

[6] Mozilla Foundation. *Thunderbird — Make Email Easier. — Thunderbird*. (Accessed on 02/21/2020). 2004. url: https://www.thunderbird.net/en-US/.

[7] Google LLC. *Gmail - Email from Google*. (Accessed on 02/21/2020). Apr. 2004. url: https://www.google.com/gmail/about/.

[8] J. Klensin. "Simple Mail Transfer Protocol". In: (Oct. 2008), p. 12. url: https://tools.ietf.org/html/rfc5321.

[9] Wietse Zweitze Venema. *The Postfix Home Page*. (Accessed on 02/21/2020). Dec. 1998. url: http://www.postfix.org/.

[10] *IBM Public License Version 1.0 (IPL-1.0) | Open Source Initiative*. (Accessed on 02/21/2020). url: https://opensource.org/licenses/IPL-1.0.

[11] *SSH Protocol – Secure Remote Login and File Transfer*. url: https://www.ssh.com/ssh/protocol.

[12] *Postfix before-queue Milter support*. (Accessed on 02/21/2020). url: http://www.postfix.org/MILTER_README.html.

[13] The Trusted Domain Project. *GitHub - trusteddomainproject/OpenARC: Open source ARC implementation*. (Accessed on 02/21/2020). url: https://github.com/trusteddomainproject/OpenARC/.

[14] The Trusted Domain Project. *GitHub - trusteddomainproject/OpenDKIM*. (Accessed on 02/21/2020). url: https://github.com/trusteddomainproject/OpenDKIM/.

[15] Deliverability Hub of the Email Marketing Council. *Email Deliverability - Whitepaper_53cf9dddc9c03.pdf*. (Accessed on 02/23/2020). Jan. 2010. url: https://dma.org.uk/uploads/Email%20Deliverability%20-%20Whitepaper_53cf9dddc9c03.pdf.

[16] Alan Hevner and Samir Chatterjee. "Design science research in information systems". In: *Design research in information systems*. Springer, 2010, pp. 9–22.

[17] T.L. Saaty. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. Advanced book program. McGraw-Hill, 1980. isbn: 9780070543713. url: https://books.google.pt/books?id=Xxi7AAAAIAAJ.

[18] Michael E. Porter. "Competitive Advantage: Creating and Sustaining Superior Performance". In: 1985.

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. "Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)". In: *J. Database Manag.* 10 (Jan. 1999).

[20] Peter A. Koen et al. "1 Fuzzy Front End : Effective Methods , Tools , and Techniques". In: 2002.

[21] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. (Accessed on 02/23/2020). 2000. url: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[22] Automattic. *WordPress.com: Create a Free Website or Blog*. (Accessed on 02/23/2020). url: https://wordpress.com/.

[23] *eCommerce Platforms | Best eCommerce Software for Selling Online | Magento*. (Accessed on 02/23/2020). url: https://magento.com/.

[24] P. Mockapetris. *https://www.ietf.org/rfc/rfc1035.txt*. (Accessed on 02/23/2020). Nov. 1987. url: https://www.ietf.org/rfc/rfc1035.txt.

[25] Murray S. Kucherawy Dave Crocker Tony Hansen. "DomainKeys Identified Mail (DKIM) Signatures". In: (Sept. 2011), p. 14. url: https://tools.ietf.org/html/rfc6376.

[26] Daniel J. Bernstein. *qmail: the Internet's MTA of choice*. (Accessed on 02/21/2020). 1983. url: https://cr.yp.to/qmail.html.

[27] Copernica. *A high performance Mail Transfer Agent (MTA) | MailerQ*. (Accessed on 02/21/2020). 2000. url: https://www.mailerq.com/.

[28] Sparkpost. *PowerMTA - SparkPost*. (Accessed on 02/21/2020). url: https://www.sparkpost.com/powermta/.

[29] Eric Allman. *Sendmail Sentrion Open Source - Open Source Email Server | Proofpoint*. (Accessed on 02/21/2020). 1983. url: https://www.proofpoint.com/us/products/open-source-email-solution.

[30] *Postfix Documentation*. (Accessed on 02/21/2020). url: http://www.postfix.org/documentation.html.

[31] D. J. Bernstein. *draft-bernstein-qmtp-01 - Quick Mail Transfer Protocol (QMTP)*. (Accessed on 02/21/2020). url: https://tools.ietf.org/html/draft-bernstein-qmtp-01.

[32] Hafiz Suliman Munawar, Ralph Johnson, and Raja Afandi. "The Security Architecture of qmail". In: (Jan. 2004).

[33] Sparkpost. *Email Delivery Service Pricing | SparkPost*. (Accessed on 02/21/2020). url: https://www.sparkpost.com/pricing/.

[34] Copernica. *Powerful tool for email marketing and email automation.* (Accessed on 02/21/2020). url: https://www.copernica.com/en/home.

[35] Copernica. *Features | MailerQ*. (Accessed on 02/21/2020). 2000. url: https://www.mailerq.com/features.

[36] Pivotal. *Messaging that just works — RabbitMQ*. (Accessed on 02/21/2020). url: https://www.rabbitmq.com/.

[37] Copernica. *Pricing | MailerQ*. (Accessed on 02/21/2020). 2000. url: https://www.mailerq.com/pricing.

[38] Apache Software Foundation. *ActiveMQ*. (Accessed on 02/21/2020). url: https://activemq.apache.org/.

[39] Apache Software Foundation. *Welcome to The Apache Software Foundation!* (Accessed on 02/21/2020). url: https://www.apache.org/.

[40] Apache Software Foundation. *ActiveMQ Documentation*. (Accessed on 02/21/2020). url: https://activemq.apache.org/features.

[41] Apache Software Foundation. *ActiveMQ Clustering*. (Accessed on 02/21/2020). url: https://activemq.apache.org/clustering.

[42] Apache Software Foundation. *Apache ZooKeeper*. (Accessed on 02/21/2020). url: https://zookeeper.apache.org/.

[43] Apache Software Foundation. *ActiveMQ Artemis*. (Accessed on 02/21/2020). url: https://activemq.apache.org/components/artemis/.

[44] Apache Software Foundation. *ActiveMQ Artemis from HornetQ*. (Accessed on 02/21/2020). url: https://activemq.apache.org/how-does-activemq-compare-to-artemis.

[45] Pivotal. *Leading companies build and run their most important applications with Pivotal. | Pivotal*. (Accessed on 02/21/2020). url: https://pivotal.io/.

[46] Pivotal. *Documentation: Table of Contents — RabbitMQ*. (Accessed on 02/21/2020). url: https://www.rabbitmq.com/documentation.html.

[47] Yann Collet. *GitHub - lz4/lz4: Extremely Fast Compression algorithm*. (Accessed on 02/23/2020). url: https://github.com/lz4/lz4.

[48] J. Ziv and A. Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE Transactions on Information Theory* 24.5 (Sept. 1978), pp. 530–536. issn: 1557-9654. doi: 10.1109/TIT.1978.1055934.

[49] Julian Seward. *bzip2*. (Accessed on 02/23/2020). 1996. url: http://www.bzip.org/.

[50] Augusto Horita et al. "Lempel-Ziv-Markov Chain Algorithm Modeling using Models of Computation and ForSyDe". In: Oct. 2019, pp. 152–155. doi: 10.3384/ecp19162017.

[51] Markus Oberhumer. *oberhumer.com: LZO real-time data compression library*. (Accessed on 02/23/2020). url: http://www.oberhumer.com/opensource/lzo/.

[52] Google LLC. *GitHub - google/brotli: Brotli compression format*. (Accessed on 02/23/2020). url: https://github.com/google/brotli.

[53] Yann Collet. *Zstandard - Real-time data compression algorithm*. (Accessed on 02/23/2020). url: https://facebook.github.io/zstd/.

[54] S. Robertson and J. Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Pearson Education, 2012. isbn: 9780132942843. url: https://books.google.pt/books?id=yE91LgrpaHsC.

[55] R.B. Grady and D.L. Caswell. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, 1987. isbn: 9780138218447. url: https://books.google.pt/books?id=o4hGAAAAYAAJ.

[56] Ivar Jacobson, Ian Spence, and Brian Kerr. "Use-Case 2.0". In: *Queue* 14.1 (2016), pp. 94–123.

[57] George T Doran. "There's a SMART way to write management's goals and objectives". In: *Management review* 70.11 (1981), pp. 35–36.

[58] Michael Jacobides, Thorbjørn Knudsen, and Mie Augier. "Benefiting from Innovation: Value Creation, Value Appropriation and the Role of Industry Architectures". In: *Research Policy* 35 (June 2006), pp. 1200–1221. doi: 10.1016/j.respol.2006.09.005.

[59] P.G. Smith and D.G. Reinertsen. *Developing Products in Half the Time*. Industrial Engineering Series. Van Nostrand Reinhold, 1995. isbn: 9780442020644. url: https://books.google.pt/books?id=q0GwQgAACAAJ.

[60] Thomas Saaty. "Decision making with the Analytic Hierarchy Process". In: *Int. J. Services Sciences Int. J. Services Sciences* 1 (Jan. 2008), pp. 83–98. doi: 10.1504/IJSSCI.2008.017590.

[61] Andreas Eggert and Wolfgang Ulaga. "Customer perceived value: a substitute for satisfaction in business markets?" In: *Journal of Business & industrial marketing* (2002).

[62] Philip Kotler et al. "Creation customer value satisfaction and loyalty". In: *Marketing management* 13 (2009), pp. 120–125.

[63] Alexander Osterwalder. "The Business Model Ontology – A Proposition in a Design Science Approach". In: (Jan. 2004).

[64] Jinhyo Yun. "Editorial Open Innovation in Value Chain for Sustainability of firms". In: *Sustainability* 9 (May 2017), p. 811. doi: 10.3390/su9050811.

[65] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012. isbn: 9780124159938. url: https://books.google.pt/books?id=zpaHa5cjLwwC.

Appendix A

Compression Results Raw Data

Below lie tables with raw data regarding the results of the best level for each compression algorithm discussed in section 3.2.2. It is possible to observe the increase of compression time along with the size of the content being compressed which was expected as with the increase of the compression ratio. The decompression time is also present for curiosity purposes as it always represents a very small fraction of time when compared to the compression time.

Table A.1: LZ4 level 2 compression factors results.

—	**5 MiB**	**10 MiB**	**15 MiB**	**20 MiB**	**25 MiB**	**30 MiB**
Compression Time	0.0205	0.0382	0.0551	0.0752	0.0929	0.1146
Compression Ratio	9.9113	9.9282	9.9417	9.9236	9.9336	9.9397
Decompression Time	0.0024	0.0036	0.0054	0.0064	0.008	0.0098

Table A.2: Gzip level 3 factors results.

—	**5 MiB**	**10 MiB**	**15 MiB**	**20 MiB**	**25 MiB**	**30 MiB**
Compression Time	0.0295	0.0582	0.0869	0.1144	0.1438	0.1633
Compression Ratio	9.8194	9.9881	9.9885	9.9799	9.9864	9.986
Decompression Time	0.0168	0.0288	0.0449	0.0622	0.0719	0.0882

Table A.3: BZ2 level 9 factors results.

—	**5 MiB**	**10 MiB**	**15 MiB**	**20 MiB**	**25 MiB**	**30 MiB**
Compression Time	0.3385	0.6584	0.9984	1.3341	1.7056	2.0127
Compression Ratio	23.0077	23.5077	23.9631	24.3529	24.132	24.1048
Decompression Time	0.0676	0.133	0.2085	0.2752	0.3314	0.3941

Table A.4: LZMA compression factors results.

—	5 MiB	10 MiB	15 MiB	20 MiB	25 MiB	30 MiB
Compression Time	0.6583	1.3173	1.9553	2.614	3.2295	3.8346
Compression Ratio	95.6186	160.8537	208.4307	241.9692	269.7601	292.429
Decompression Time	0.0098	0.015	0.0234	0.0303	0.0347	0.0438

Table A.5: LZO level 1 compression factors results.

—	5 MiB	10 MiB	15 MiB	20 MiB	25 MiB	30 MiB
Compression Time	0.0043	0.0091	0.0163	0.0168	0.0206	0.0267
Compression Ratio	6.7858	6.8566	6.8706	6.8723	6.8772	6.8659
Decompression Time	0.0036	0.008	0.0108	0.0136	0.0176	0.0204

Table A.6: Brotli level 3 compression factors results.

—	5 MiB	10 MiB	15 MiB	20 MiB	25 MiB	30 MiB
Compression Time	0.0119	0.0315	0.0326	0.0414	0.0516	0.0605
Compression Ratio	69.0163	102.8439	122.8874	134.9746	144.1257	151.0259
Decompression Time	0.0039	0.0062	0.0087	0.0183	0.0227	0.0253

Table A.7: ZSTD level 4 compression factors results.

—	5 MiB	10 MiB	15 MiB	20 MiB	25 MiB	30 MiB
Compression Time	0.0028	0.0042	0.0048	0.005	0.0061	0.0073
Compression Ratio	67.6839	71.5101	104.1891	134.7858	128.1265	126.7645
Decompression Time	0.0016	0.0037	0.0024	0.0049	0.0084	0.0109

Table A.8: Overall compression rating results.

—	5 MiB	10 MiB	15 MiB	20 MiB	25 MiB	30 MiB
LZ4	482.3201	259.6407	180.2946	131.9095	106.9613	86.7399
GZIP	333.3343	171.6449	115.0065	87.2209	69.4251	61.1558
BZ2	67.9655	35.7057	24.0012	18.2536	14.1488	11.9761
LZMA	145.2618	122.1083	106.5973	92.5654	83.5289	76.2614
LZO	1585.0499	754.2969	421.6636	408.9975	333.8016	257.128
BROTLI	5790.6284	3265.1349	3774.1874	3263.3071	2791.9602	2495.4106
ZSTD	24435.1436	16938.1782	21535.948	27129.1035	20979.8839	17272.9441