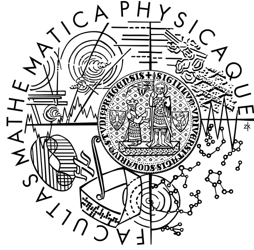


**FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University**

MASTER THESIS

Horizontal scalability for e-mail delivery in Mailtrain

Department of Distributed and Dependable Systems

Author: Bc. Erik Kučák

Supervisor: prof. RNDr. Tomáš Bureš, Ph.D.

Study programme: Software Systems

Study branch: Dependable Systems

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature, and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I wish to thank my supervisor prof. RNDr. Tomáš Bureš, Ph.D. He has generously given his time, talents, and advice to assist me in the production of this thesis. I would also like to thank all my colleagues and friends, who supported my work and studies. Last but not least, I thank my loving family, without them nothing would be possible.

Název práce: Horizontální škálovatelnost pro doručování e-mailů v Mailtrainu

Autor: Bc. Erik Kučák

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: prof. RNDr. Tomáš Bureš, Ph.D.

E-mail vedoucího: bures@d3s.mff.cuni.cz

Abstrakt: Mailtrain je samoobslužná aplikace s otevřeným zdrojovým kódem postavená na Node.js, která poskytuje vlastnosti, jako je správa seznamů odběratelů, segmentace seznamů, vlastní pole, šablony e-mailů, spouštěné a RSS kampaně atp. Jedná z hlavních nedostatků Mailtrainu je neschopnost horizontálně škálovat, což má za následek výkonnostní limit při doručování kampaní velmi velkým seznamem adresátů. Hlavním cílem této práce je rozšířit Mailtrain tak, aby dovoloval doručovat kampaně (včetně příloh, propojených obrázků a sledování uživatelů) distribuovaným a horizontálně škálovatelným způsobem. Práce by měla obsahovat návrh rozšíření, jeho implementaci a vyhodnocení výkonu pro srovnání rozšíření se stávajícím výkonem Mailtrainu.

Klíčová slova: Horizontální škálovatelnost, distribuovaný systém, Node.js, doručování e-mailů

Title: Horizontal scalability for e-mail delivery in Mailtrain

Author: Bc. Erik Kučák

Department: Department of Distributed and Dependable Systems

Supervisor: prof. RNDr. Tomáš Bureš, Ph.D.

Supervisor's e-mail: bures@d3s.mff.cuni.cz

Abstract: Mailtrain is a self-hosted open-source newsletter application built on Node.js which provides features such as subscriber lists management, list segmentation, custom fields, e-mail templates, triggered and RSS campaigns, etc. One of the main shortcomings of Mailtrain is the inability to scale horizontally, which results in performance limits when delivering campaigns to very large mailing lists. The main goal of this work is to extend Mailtrain to allow it to handle the delivery of campaigns (including attachments, linked images, and user tracking) in distributed and horizontally scalable manner. The thesis should include the design of the extension, its implementation, and performance evaluation to compare the extension with the existing performance of Mailtrain.

Keywords: Horizontal scalability, distributed system, Node.js, e-mail delivery

Contents

1 Introduction 5

1.1 Motivation . . . 5
1.2 Goals . . . 5
1.3 Contents . . . 6
1.4 Sources . . . 6

2 Mailtrain overview 7

2.1 Database model . . . 7
2.1.1 List . . . 8
2.1.2 Campaign . . . 8
2.1.3 Message . . . 10
2.1.4 File . . . 11
2.2 Database schema . . . 11
2.3 Features . . . 12
2.3.1 Multiple users . . . 12
2.3.2 Hierarchical namespaces . . . 13
2.3.3 Subscriber lists management . . . 13
2.3.4 Custom fields . . . 13
2.3.5 List segmentation . . . 13
2.3.6 Automation . . . 13
2.3.7 Campaign e-mail templates . . . 14
2.3.8 Custom reports . . . 14
2.3.9 Builtin Zone-MTA . . . 14
2.4 Architecture . . . 14
2.4.1 Process tree . . . 16
2.4.2 Services . . . 17
2.4.3 Sender . . . 18
2.4.3.1 SenderMaster life cycle . . . 20
2.4.3.2 SenderWorker life cycle . . . 21
2.4.4 PUBLIC server . . . 22
2.5 Sending e-mail use cases . . . 24
2.6 Summary of goals . . . 25
2.7 Source code overview . . . 25
2.8 Deployment requirements . . . 26

3 Performance analysis 28

3.1 Overall Sender analysis . . . 28
3.2 Sender components analysis . . . 30
3.2.1 SenderMaster analysis . . . 30
3.2.2 SenderWorker and SMTP server analysis . . . 31
3.3 Summary of Sender bottlenecks . . . 32
3.4 Analysis conclusion . . . 33

4 New architectural design **34**
4.1 New database system . . . 35
4.1.1 Integrating MongoDB into Mailtrain . . . 35
4.1.2 Ensuring data consistency . . . 38
4.2 Distributed Sender . . . 39
4.2.1 Synchronizer . . . 41
4.2.2 Scheduler . . . 43
4.2.3 DataCollector . . . 44
4.2.4 SenderWorker . . . 44
4.2.4.1 Life cycle . . . 44
4.2.4.2 Worker synchronization . . . 46
4.2.4.3 State diagram . . . 47
4.2.4.4 WorkerSynchronizer . . . 48
4.2.4.5 PlatformSolver . . . 49
4.2.4.6 Correctness testing . . . 49
4.3 HAPUBLIC server . . . 50
4.4 Modes . . . 52
4.4.1 Centralized . . . 52
4.4.2 Distributed . . . 52
4.5 Source code overview . . . 52
4.5.1 The current version . . . 52
4.5.2 Deleted subtree . . . 54
4.5.3 Updated subtree . . . 54
4.6 Deployment requirements . . . 56
4.6.1 Centralized mode . . . 56
4.6.2 Distributed mode . . . 56

5 Evaluation **57**
5.1 Test data . . . 57
5.2 Test environment . . . 57
5.3 The old version . . . 58
5.3.1 Sender . . . 58
5.4 The new version . . . 59
5.4.1 Centralized Sender . . . 59
5.4.2 Distributed Sender . . . 60
5.4.2.1 Unsynchronized workers . . . 60
5.4.2.2 Synchronized workers . . . 61
5.5 Comparison of all variants . . . 62
5.5.1 Evenness of the distribution of messages . . . 64
5.5.2 HAPUBLIC server . . . 65
5.5.2.1 Simulated version . . . 66
5.5.2.2 Real version . . . 66
5.6 Evaluation summary . . . 67

6 Conclusion **69**
6.1 Future work . . . 69

List of Figures **73**

List of Abbreviations **74**

A Attachment **75**

1. Introduction

1.1 Motivation

Quality marketing is one of the most critical aspects of doing business. E-mail marketing is an ideal way to get messages straight to the people that matter most to building business. It could be either an e-commerce business trying to generate more leads by e-mailing corresponding customers or a blogger sending the latest content to related readers.

Mailtrain is one of the systems for this purpose. It supports popular features such as subscriber lists management, list segmentation, custom fields, custom e-mail templates, triggered and RSS campaigns, etc. All of these features are popular today and help achieve quality business marketing. It is used by customers who need a fast, simple, and accessible system.

1.2 Goals

In the current version 2, customers with too many extensive campaigns have problems with performance. Mailtrain cannot effectively scale on large data because sending campaigns with too many e-mails simultaneously to subscribers is slow. The current version is parallel but centralized so that it can run only on one computer node. If customers want to solve this performance problem, there is only one possible solution to speed up the system. They have to ensure better and more efficient hardware where Mailtrain will run. This approach is called vertical scalability.

The most significant disadvantages of vertical scalability are that it requires a huge amount of financial investment, greater risk of hardware failure causing bigger outages, and limited upgradeability in the future [22].

Another significant problem is the availability of services. Some of them require non-stop correct running, such as sending campaigns, additional sending linked images when a subscriber opens received an e-mail, and handling subscriber events (opening received an e-mail, clicking on some e-mail link, etc.). Such services are called critical services, and a feature of services is called high availability. Since Mailtrain currently does not support high availability, it may cause significant problems. The first problem is losing critical data due to unprocessed subscriber events or incorrect e-mail rendering when Mailtrain is not running. The second problem is the loss of subscribers due to late e-mail receiving when Mailtrain is too slow to send a massive amount of campaign e-mails simultaneously.

So this thesis's first and primary goal is to make Mailtrain horizontally scalable for sending campaigns with too many e-mails simultaneously and enable this system to run in a distributed system. It will solve the primary problem of customers. The second goal is to ensure the high availability of all critical services that require non-stop running.

1.3 Contents

The present thesis explains all the needed technical background for Mailtrain of the current version, followed by a meticulous analysis of the necessary system components in terms of their performance. Based on the identified issues, a novel architecture will be proposed to address the aforementioned problems. Subsequently, it will be demonstrated that the updated version 3 exhibits notable performance improvements and enhanced fault tolerance.

Below is described each chapter:

- **Chapter 2 - Mailtrain overview** This chapter describes the current version of Mailtrain. It includes all needed database entities, features, architecture, services, source code overview, and deployment requirements. Firstly, knowing what Mailtrain provides and how it works for designing a new and better architecture is crucial.
- **Chapter 3 - Performance analysis** In the third chapter, there is analyzed architecture focusing on performance and trying to find all significant bottlenecks which cause a slowdown while sending too many e-mails simultaneously.
- **Chapter 4 - New architectural design** In the fourth, the most important chapter, there is explained the new architecture of Mailtrain that solves all problems using the knowledge gained in the previous chapter.
- **Chapter 5 - Evaluation** This chapter compares both versions with performance tests on some large data. Then there is pointed out that the new version can scale horizontally and has significantly increased performance.
- **Chapter 6 - Conclusion** In the last chapter, this thesis's total results and what is still essential to do in the future are summarized.

1.4 Sources

The source code is publicly accessible in the GitHub repository and is also attached in the thesis attachment A. The repository can be cloned by typing the command:

```
1 $ git clone https://github.com/Riko196/mailtrain.git
```

2. Mailtrain overview

A comprehensive comprehension of Mailtrain is fundamental in identifying performance bottlenecks and designing a new architecture. This chapter will delve into the database model, encompassing the fundamental database entities and their associated database schema. Additionally, the chapter will present an overview of all supported features, the system's architecture, the source code overview, a summary of all required goals, and a brief description of the deployment requirements needed for using this system.

Notation: throughout the thesis, there is an assumption that the reader has sufficient technical knowledge about terms such as SMTP server, SMTP protocol [17], e-mail format, RSS, etc. Mailtrain is written in Node.js platform [13] using JavaScript language and uses properties of this language [10]. Therefore, this thesis has no explanation for these properties and terms.

2.1 Database model

To ensure a comprehensive comprehension of the supported features and system architecture of Mailtrain, a concise depiction of the underlying database model is presented. This encompasses an elucidation of the fundamental entities and their relationships that are utilized throughout the system. The following is a list of the basic database entities, along with a brief description of their roles and the relationships that exist between them.

- **User** - represents an authorized entity in the system assigned role with given permissions and manages all other entities corresponding to those permissions, such as lists, campaigns, templates, etc. Mailtrain always contains one unique user named `admin` with full permissions over the entire system.

- **Subscriber** - this entity represents e-mail receiver. Each subscriber belongs to at least one list.

- **List** - a group of many subscribers with customizable values fields that can be assigned to any number of campaigns.

- **Campaign** - the most important entity of Mailtrain that represents a group of lists of subscribers that, according to defined rules, receive e-mail defined by some template.

- **Message** - represents a record in the database table from which e-mail is generated for a particular subscriber.

- **File** - represents a record in the particular database table and the file in the file system associated with it, which is then used as a report, a campaign file, a campaign attachment, a campaign e-mail template, or for some other purposes.

- **Channel** - a group of many campaigns that share common characteristics (such as a template, newsletter information, etc.).
- **Template** - represents an e-mail template used by some campaign to render specific e-mail for one particular subscriber.
- **Report** - stored information about some specific event that occurred in Mailtrain.
- **Namespace** - a set of names to identify and refer to entities. A namespace ensures that all of a given set of entities have unique names so that they can be easily identified.
- **Send configuration** - a setting for sending e-mails that contains information such as e-mail header, SMTP server, DKIM signing, throttling, etc. Users can have multiple send configurations and choose which campaign is sent by which send configuration.

Some entities are less needed than others in this thesis because there is no direct work with them. It is essential to describe only those that will be used with more details and technical background.

2.1.1 List

One of the first and basic entities is a list of subscribers used by many other entities. Users can manually create each list in the admin GUI or import an appropriate CSV file.

Figure 2.1 depicts the exact definition of all MariaDB tables related to subscriber and list entities. Where `subscription__i` represents a table of subscribers for an i-th list, `custom_field` represents a dynamically added new field for an i-th list, and `lists` represents basic information about each created list.

2.1.2 Campaign

The second and most important entity is a campaign with which Mailtrain mainly works. It is essential to describe information about what data it contains as a record in the MariaDB database, which types exist, which statuses it has, and which functions it supports.

In a newsletter application, a campaign is typically defined as a specific e-mail or series of e-mails that are sent to a targeted audience with a specific goal in mind.

In Mailtrain, a campaign is created by the user, who designs and develops the content of the email named as an e-mail template, including the subject line, body, images, and calls to action. This e-mail template is then used to generate e-mail specifically for each subscriber. The campaign may be targeted towards a particular segment of the subscriber list or the entire list, depending on the goals of the campaign. Then each campaign contains send configuration with information about the configuration for the SMTP server that sends all these e-mails.

To measure the success of a campaign, Mailtrain provides various metrics, such as open rates, click-through rates, and conversion rates. These metrics can be used to track the effectiveness of the campaign and make adjustments to future campaigns based on the results.

There are four supported types of campaigns supported by Mailtrain:

1. **Regular** - a classic type of campaign where is required manually set up a time when a campaign starts delivering. It is the most commonly used campaign.

2. **RSS** - a type of campaign that quickly automates the entire system which listens on a specific defined RSS feed's URL, and every time a newsletter appears, it creates a new campaign (RSS_ENTRY type) with the same subscribers. It starts delivering automatically without any manual work.

3. **RSS_ENTRY** - a type of campaign not directly visible to a user and can not be manually created. This campaign is created fully automatically by some specific RSS campaign when its listener on the RSS feed's URL receives information about the campaign for sending.

4. **Triggered** - a type of campaign that is also automated and similar to an RSS campaign. The definition of this campaign includes a group of triggers. A trigger is a listener that has assigned some defined event, and it checks whether this event happened. Sending of this type of campaign is started when for some trigger, its event happens (such as five days have passed since the regular campaign `A` has been delivered).

To avoid misunderstanding, it is essential to mention that only Regular, RSS_-ENTRY, and Triggered campaign types can be sent. RSS type represents the listener, which creates RSS_ENTRY when sending a new campaign.

These statuses of a campaign are defined in Mailtrain (in the brackets, it is indicated which campaign types are involved):

1. **Idle** - a campaign has been successfully created, but has not been scheduled yet. (Regular and RSS_ENTRY)

2. **Scheduled** - a campaign is scheduled for sending and is waiting for the next processing. (Regular and RSS_ENTRY)

3. **Pausing** - sending campaign is waiting for stop but is still sending (Regular and RSS_ENTRY).

4. **Paused** - sending of a campaign is paused (Regular and RSS_ENTRY).

5. **Sending** - a campaign is currently sending (Regular and RSS_ENTRY).

6. **Finished** - sending of a campaign has been successfully finished (Regular and RSS_ENTRY).

7. **Inactive** - a campaign is in inactive status. It is not listening to the RSS feed's URL (RSS) or triggers (Triggered).

8. **Active** - a campaign is in active status. It is listening to the RSS feed's URL (RSS) or triggers (Triggered).

Figure 2.1 depicts the exact definition of all MariaDB tables related to the campaign entity.

2.1.3 Message

The third entity that is crucial to describe is a message. Mainly information about what data it contains as a record in the MariaDB database, which types exist, which statuses it has, and which functions it supports.

To avoid misunderstanding, it is crucial to explain the difference between the terms message and e-mail. The term message is a MariaDB table record containing only primary data for generating one e-mail intended for one subscriber. The term e-mail means an entirely generated e-mail with all needed data (subject line, body, addressee, receiver, linked images, attachments, etc.) prepared for sending.

There are five types of messages:

1. **Regular** - a classic type of message used for sending regular and RSS_-ENTRY campaigns.

2. **Triggered** - a type of message used for sending triggered campaigns.

3. **Test** - a type of message used for testing purposes for each campaign type.

4. **Subscription** - a type of message used to send an e-mail to a subscriber when she/he decides to subscribe or unsubscribe herself/himself from a particular campaign.

5. **API_Transactional** - a type of message created manually by a user that is used for a particular purpose.

The message types are further classified based on their frequency of use into two categories: campaign messages and queued messages. The campaign message type comprises the regular type, while the queued type includes the remaining types. This dichotomy arises due to the large volume of campaign messages that share a specific campaign and necessitates a more optimized and aesthetically pleasing procedure for their sending. In contrast, queued messages are less frequently employed, and their sending involves a relatively straightforward and less optimized procedure in the code. Additionally, messages associated with triggered campaigns are classified as queued messages, given their infrequent usage in production, which may lead to confusion.

Each message is always in a precisely defined status. Below are described all defined statuses:

1. **Scheduled** - a message is prepared for generating and sending e-mail.

2. **Unsubscribed** - a message for an unsubscribed subscriber. This status of a message is used in case some subscriber has unsubscribed himself but has not been removed from Mailtrain yet. A message in this status is not sent but automatically ignored during message scheduling.

3. **Sent** - a message has been successfully sent to a particular subscriber.
4. **Failed** - sending of a message failed with an unrecoverable error on the Mailtrain side, and a message will not be tried to send again.
5. **Bounced** - sending of a message failed with an unrecoverable error on SMTP server size, and a message will not be tried to send again (status used only by some special SMTP servers).
6. **Complained** - sending of a message failed with an unrecoverable error on SMTP server size, and a message will not be tried to send again (status used only by some special SMTP servers).

Figure 2.1 depicts the exact definition of MariaDB tables `campaign_messages` and `queued`.

2.1.4 File

Mailtrain also stores a lot of unstructured data, such as reports, campaign files, campaign attachments, campaign e-mail templates, etc. All of them are stored directly in the file system with an appropriate MariaDB table that contains information about them.

It would be useful to describe file entities important in this thesis. Here is the description list:

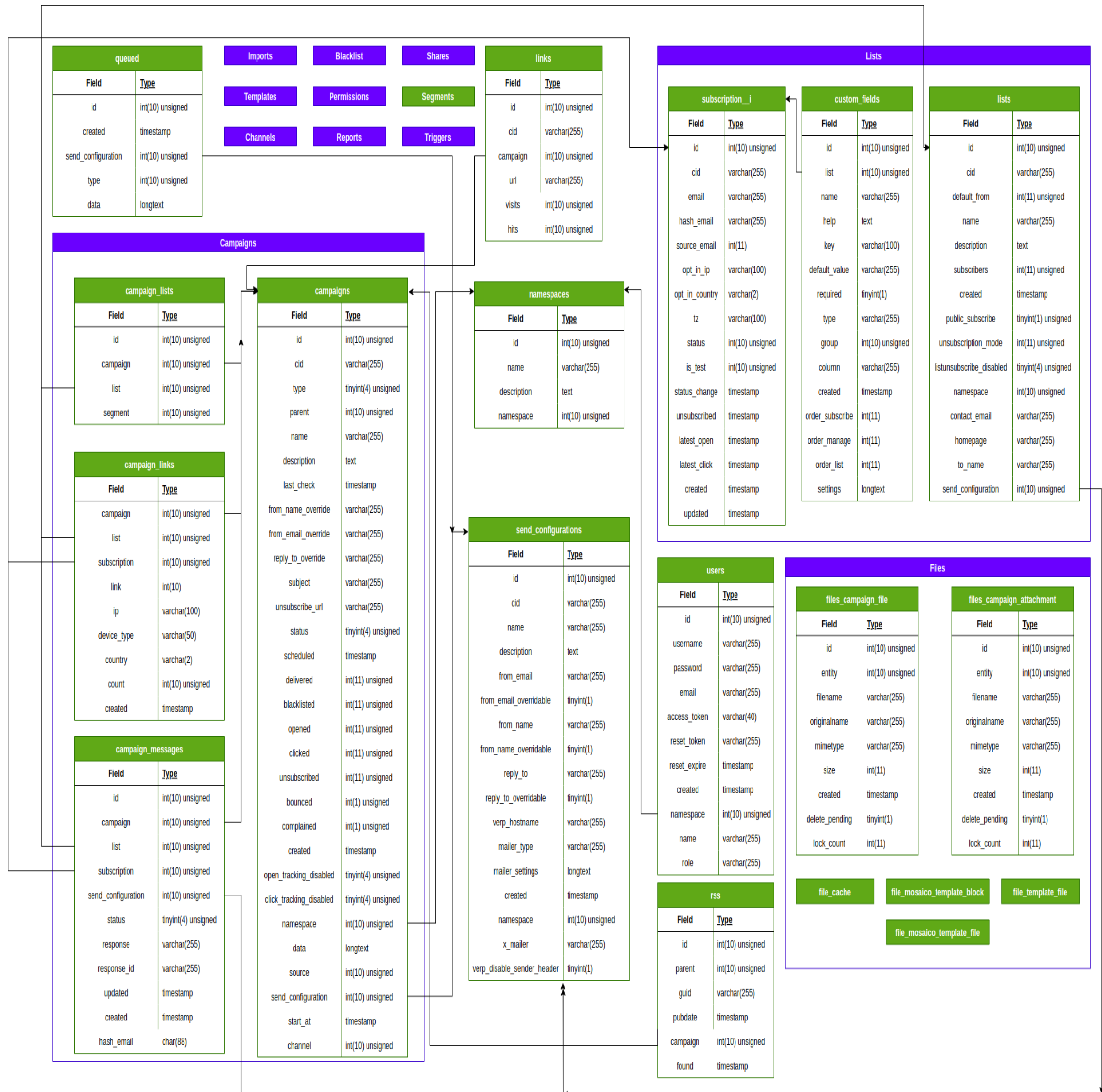
1. **Campaign file** - represents a campaign file that is linked to sending an e-mail and additionally sent when a subscriber reads it. This type of file includes all e-mail-linked images that are requested and rendered after opening an e-mail and all other files that are connected to an e-mail by link and additionally sent by clicking on this link. Table `files_campaign_file` is used for storing information about this entity.
2. **Campaign attachment** - represents a file that is directly attached to sending e-mail and sent together with e-mail. Table `files_campaign_attachment` is used for storing information about this entity.
3. **Campaign e-mail template** - a file type that stores how the e-mail template for some created campaign should look.

Figure 2.1 depicts the exact definition of all MariaDB tables related to described file entities. Other tables related to file entities are not necessary for this thesis, so they do not contain schema.

2.2 Database schema

This section illustrates the precise database schema of the Mailtrain database, derived from the database model expounded upon in the preceding section. The tables are depicted in green, and groups of tables with shared data are highlighted in purple. Only the tables and groups of data that are of significance for this thesis are presented with their respective columns and data types.

Figure 2.1: Database schema



2.3 Features

Prior to delving into the system's architecture, it is imperative to provide a succinct overview of all the features supported by Mailtrain.

2.3.1 Multiple users

Mailtrain supports multiple users with granular user permissions and flexible sharing. Each user can read and make campaigns, lists, channels, etc., and all other entities to which a user has permission. A user can also easily share all entities with another user that a user owns.

Below are described all roles and their permissions:

- **Global Master** - all permissions (default role for `admin`.
- **Campaigns Admin** - under the namespace in which the user is located,

the user has all permissions for managing lists, templates, and campaigns and the permission to send configurations.

- **Campaigns Admin (multiple namespaces)** - has a basic set of permissions to set up campaigns, edit lists, and templates. The particular namespaces to which it has access have to be shared individually.
- **None** - no permissions.

2.3.2 Hierarchical namespaces

Mailtrain utilizes namespaces to provide a hierarchical structure for enterprise-level use cases. A user may create any number of namespaces, which together form a tree with the root node labeled as `root`. Permissions are assigned to individual users over their corresponding subtree. The user `admin` possesses root-level permissions. It is worth noting that each entity must be created by a user and be associated with a specific namespace, excluding the namespace itself.

2.3.3 Subscriber lists management

One of the essential features that Mailtrain supports is subscriber list management. Each user can define lists with particular subscribers. Subscribers can be added to the list by a user with the manual procedure or automatically by importing some CSV file. They can subscribe alone to the list by filling out the subscriber form.

2.3.4 Custom fields

For each created list, a user can customize subscriber fields dynamically after the list has been created. Mailtrain supports many useful types for this field, such as Text, Number, Date, JSON, etc. This whole procedure is also GUI friendly, and all fields can be created by Checkboxes, Radio buttons, Drop down, etc.

2.3.5 List segmentation

Each list that is created can be subdivided into segments that represent a particular subset of the list defined based on a condition. These segments can be used in the campaign definition as a new list instead of defining a new list, resulting in memory conservation. Defining these segments is facilitated through the GUI, which negates the need for the user to be proficient in a particular programming language. Rather, familiarity with the fundamentals of mathematical logic is sufficient for the user to effectively utilize this feature.

2.3.6 Automation

Mailtrain is designed to provide maximum automation without requiring manual intervention within the system. It accommodates automated RSS and triggered campaigns that function automatically upon creation. Users are solely responsible for defining these campaigns, after which they are self-updating and

self-creating. This feature provides users with substantial time savings while minimizing the probability of errors arising from user errors.

2.3.7 Campaign e-mail templates

To create a campaign, it is first necessary to define how the delivered e-mail template should look. By template is during sending made specific e-mail to a particular subscriber. Mailtrain supports a GUI-friendly procedure for defining templates (WYSIWYG editor) without any programming or needing knowledge about programming or markup languages. The final code of the defined template is then automatically generated in HTML. The second procedure for creating a template is MJML-based, whose definition requires knowing MJML markup language.

2.3.8 Custom reports

After sending many different campaigns, there are a lot of interesting data, such as the count of opened e-mails, the count of clicked links, etc. For the user to have an overview of this mass of data, the user can define custom reports. Each report contains some specific campaign and report template. In the report template, it is necessary to select the output format (HTML or CSV), define a piece of code executed on these data (in JavaScript language), output fields, and a rendering template. That defines how computed metrics will be rendered (required only for HTML format). Metrics are generated and available in the selected format when the report is defined and created. These metrics are dynamically refreshed depending on time.

2.3.9 Builtin Zone-MTA

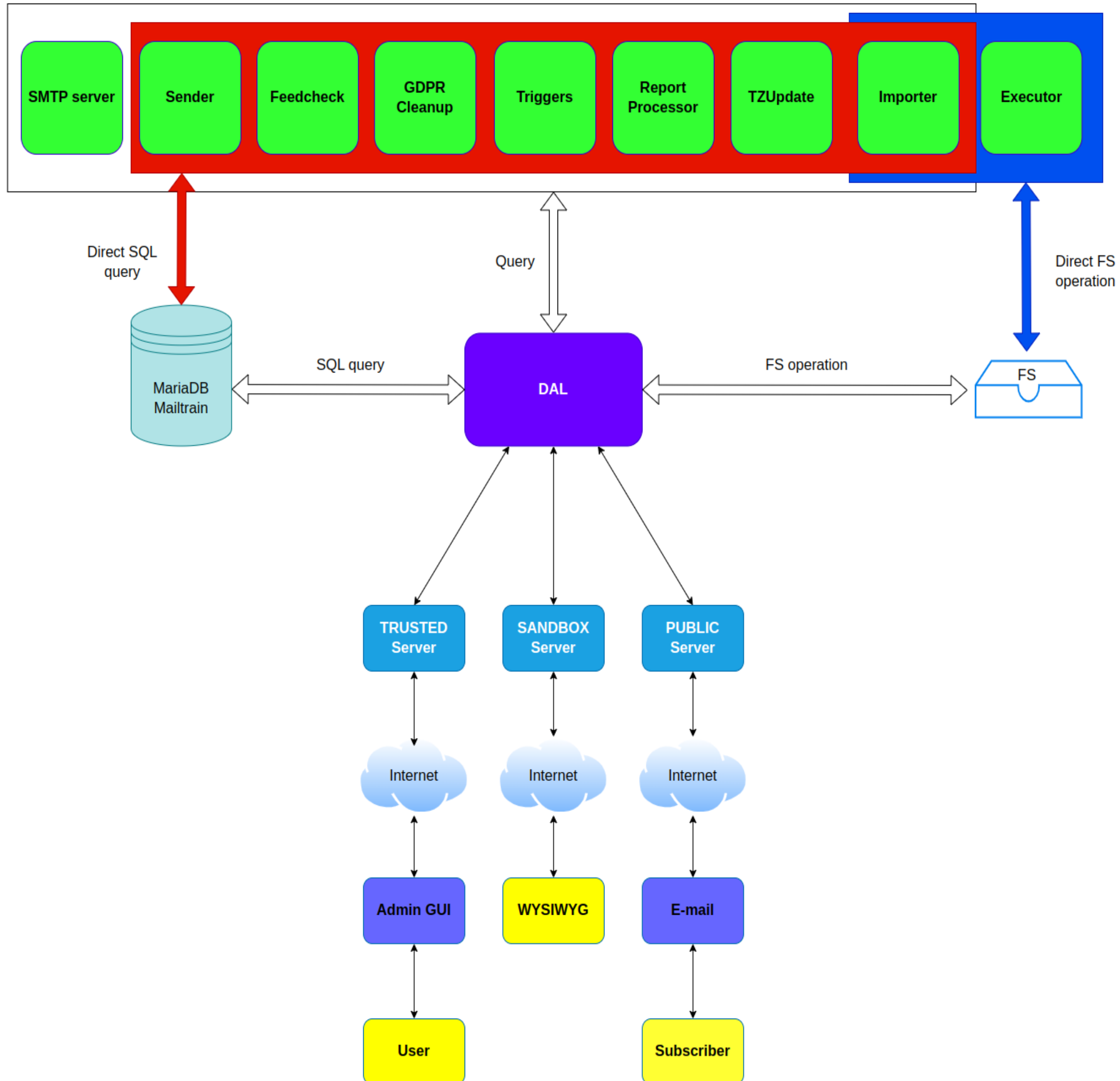
ZoneMTA is a contemporary outbound SMTP relay (MTA/MSA) that offers precise management over the routing of distinct messages. The system allows the routing of messages from trusted senders through high-speed virtual "sending zones," which utilize IP addresses with high reputation, while less trusted senders can be directed through slower virtual "sending zones" or IP addresses with less reputation. Furthermore, ZoneMTA provides features that are often found in commercial software, such as message rewriting, IP warm-up, and an HTTP API for message posting. [25].

2.4 Architecture

Upon providing an overview of all the supported features, the subsequent section aims to provide a detailed account of the architecture of Mailtrain. Initially, the high-level architecture will be discussed, followed by an in-depth explanation of the crucial components associated with sending e-mails and other imperative services that demand high availability.

Figure 2.2: Mailtrain architecture

Services



Firstly, it is necessary to start with the frontend part of Mailtrain. As Figure 2.2 depicts, three participators give input to Mailtrain (user, WYSIWYG editor, and subscriber). Two of them use GUI for interaction. A user interacts with Mailtrain through admin GUI, whereby manages Mailtrain. It is an interactive and responsive GUI, written in React [2]. A subscriber interacts with Mailtrain through a received e-mail.

All of this thesis work deals with the backend part, so the thesis will now primarily focus on it. Mailtrain contains three web servers, which are referred to as **Trusted**, **Sandbox**, and **Public**. They represent separated instances of the Express web application framework [6] within the root process running as asynchronous operations. This separation allows Mailtrain to guarantee security and avoid XSS attacks [14] in the multi-user settings. Each server is also responsible for different types of requests and is made for different types of entities. The function of these three servers is as follows:

- **Trusted** - This is the main server used only by authenticated and authorized users. It provides endpoints through which users can manage lists, subscriber

fields, campaigns, and all other supported entities.

- **Sandbox** - This server is used to host WYSIWYG template editors and is not directly visible to users.
- **Public** - This is a server created for subscribers. It does not require authentication and hosts subscription management forms, files, and archives. It means handling events (opening e-mails, clicking links, etc.), additionally sending linked files in e-mails, and sending archived e-mails. All of these requests are automatically sent by a subscriber through received e-mail.

Mailtrain uses two permanent storages. For structured data, it uses a relational database. During the installation of Mailtrain, there is a choice for installation between MySQL and MariaDB database systems. Throughout the thesis, it is presumed that the MariaDB database system has been installed. The classical file system is used for unstructured data, such as reports, campaign files, campaign attachments, etc.

2.4.1 Process tree

To avoid misunderstanding, it is helpful to mention that in this thesis, the terms 'process' and 'processes' mean terms in the context of operating systems. If it is needed to use the term 'process' in the context of some abstract procedure or life cycle of some component, then it is used term procedure or operation. This restriction also does not apply to the words 'processing' or 'processed' that are always used outside of the operating systems context.

Mailtrain consists of many processes that together create one process tree. The main reason for using more processes is that it is much more maintainable to separate components that execute totally different functionality. The second crucial reason is that the version of Node.js used on the backend side of Mailtrain is single-threaded. All cases where parallelization is needed are solved by forking new processes. Describing components shown in Figure 2.2 from the point of view of processes could be divided into three categories:

1. **Root process** - The main process, launched first when Mailtrain is started. It forks other processes, starts asynchronous operations related to this process, such as all three web servers, and manages the system.
2. **Services** - Forked processes by the root process intended for a special activity. Each service consists of one process forked by the root process, and then some services may fork other auxiliary processes.
3. **Others** - The rest of the processes running separated from Mailtrain and its process tree, represent all services of the operating system used by Mailtrain (file system, MariaDB database system, etc.), or processes running separated from the node where Mailtrain is running (user, WYSIWYG editor, and subscriber).

2.4.2 Services

Mailtrain contains many services responsible for different use cases. All of them will be described briefly, and in the next section, only the Sender will be described more deeply because it is an essential service for us. All services represent the processes forked during the system starting and communicate with each other through the MariaDB database, communicate with the root process through interprocess communication using messaging, or sometimes through HTTPS network communication protocol (for example, Sender and SMTP server). Most of these services work on the principle of a periodic task. A service has set up its period, after which it executes some primary function in an endless loop. These services and their primary purpose are described below:

- **Report Processor** - Service that computes all requested reports.
- **Triggers** - Service responsible for handling all trigger events (an event that starts sending some triggered campaign). If any trigger event occurs, the service prepares a particular trigger campaign for the following handling (creating all messages of the given triggered campaign and preparing them for the Sender, which then sends it).
- **Feedcheck** - Service used for handling RSS campaigns. It always checks the feed URL of all active RSS campaigns to see whether there is some new campaign for sending. If so, the service creates a new RSS_ENTRY campaign and prepares it for Sender.
- **Importer** - Service used for creating new lists by importing files in CSV format. If some user wants to create a list by importing a file, the Importer handles this request and imports all given data stored in a file and creates all needed records in the MariaDB database.
- **Sender** - The most critical and complex service responsible for scheduling, making, and sending all e-mails sent by Mailtrain (campaigns and queued messages).
- **Executor** - Service that represents privileged executor. If Mailtrain is started as root, this process keeps the root privilege to be able to fork workers who compute custom reports and can execute the command `chroot`.
- **GDPR Cleanup** - In Mailtrain, GDPR rules are respected, so there is the GDPR Cleanup service, which is responsible for deleting all subscriber data who unsubscribed himself.
- **TZUpdate** - This service re-calculates timezone offsets once a day. It is needed for sending messages using the subscriber's local time. The best option would be to use the built-in timezone data of MariaDB, but the availability of timezone data is not guaranteed as it is an optional add-on. So instead of keeping a list of timezone offsets in a table, it is possible to use JOIN with the subscription table. The subscription table includes a subscriber's timezone name, and the TZOffset table includes the offset from UTC in minutes.

- **SMTP Server** - Server for sending generated and prepared e-mails through SMTP protocol used by Sender. Mailtrain supports three types of SMTP servers (centralized and serial SMTP server, Zone-MTA mentioned above, and AWS SES cloud e-mail service [1]).

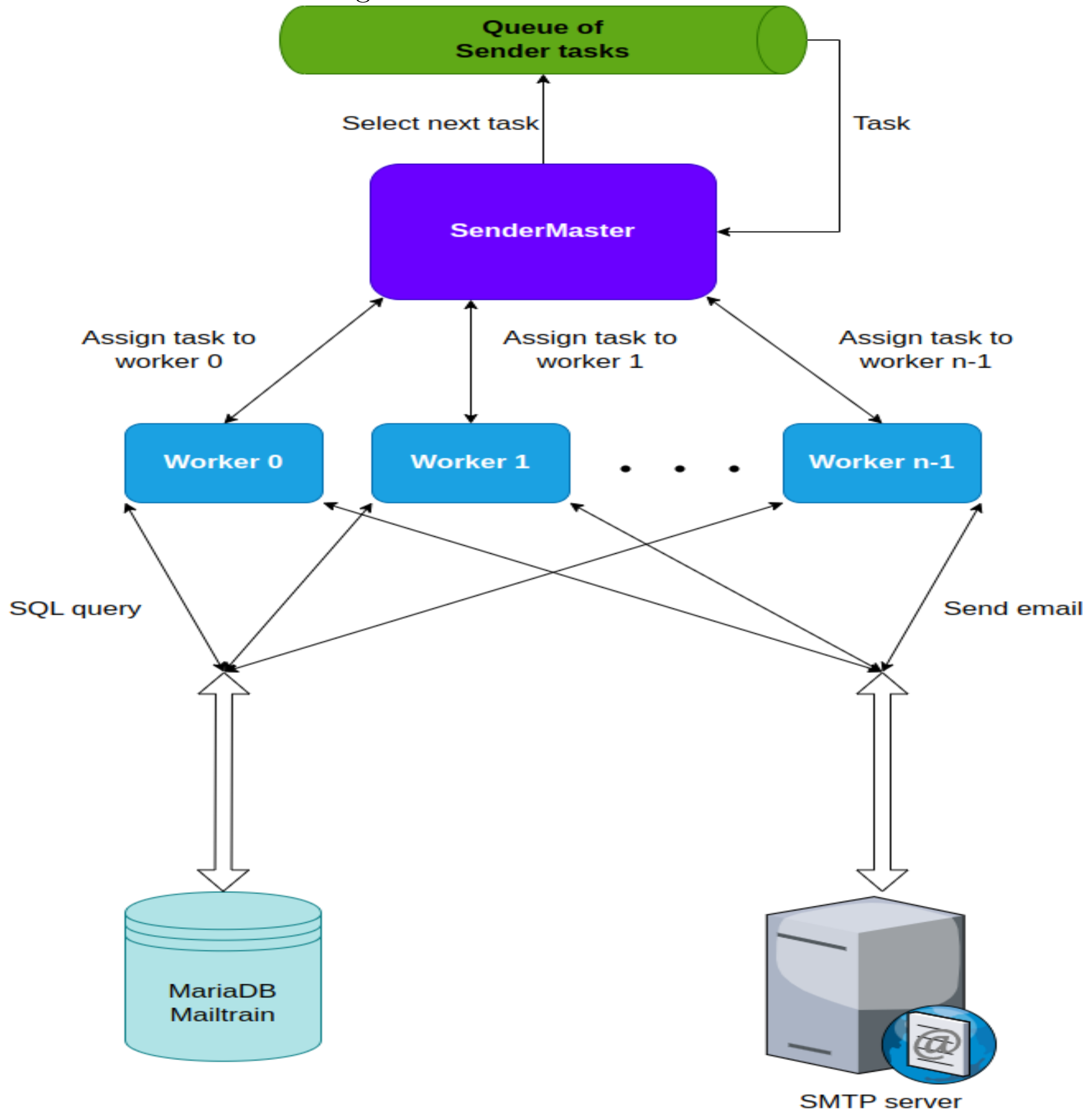
The component that provides access to the MariaDB database and file system is called DAL or the data access layer. It is not a new process. It represents just API used by the root process or services containing all crucial queries to the MariaDB database or file system. As Figure 2.2 depicts, there are also other direct queries into the MariaDB database or file system. DAL mainly contains more general queries used by many other components, but there are also specific queries executed only by one component. In that case, a component executes a direct query without DAL API. Services that execute DAL API queries are in the white group, services that execute direct SQL queries are in the red group, and services that execute direct file system operations are in the blue group. There can also be group intersections. For example, the Importer uses all types of queries, and the Executor uses only direct file system operation. On the other hand, web servers access the data primarily only by DAL API.

In the following section, the Sender service will be explained more deeply.

2.4.3 Sender

As expected, Sender is the most critical and complex service of Mailtrain. It is responsible for scheduling, making, and sending all e-mails to subscribers. This system component causes the most degradation in performance. So it is necessary to describe how the architecture looks and works deeply.

Figure 2.3: Sender architecture



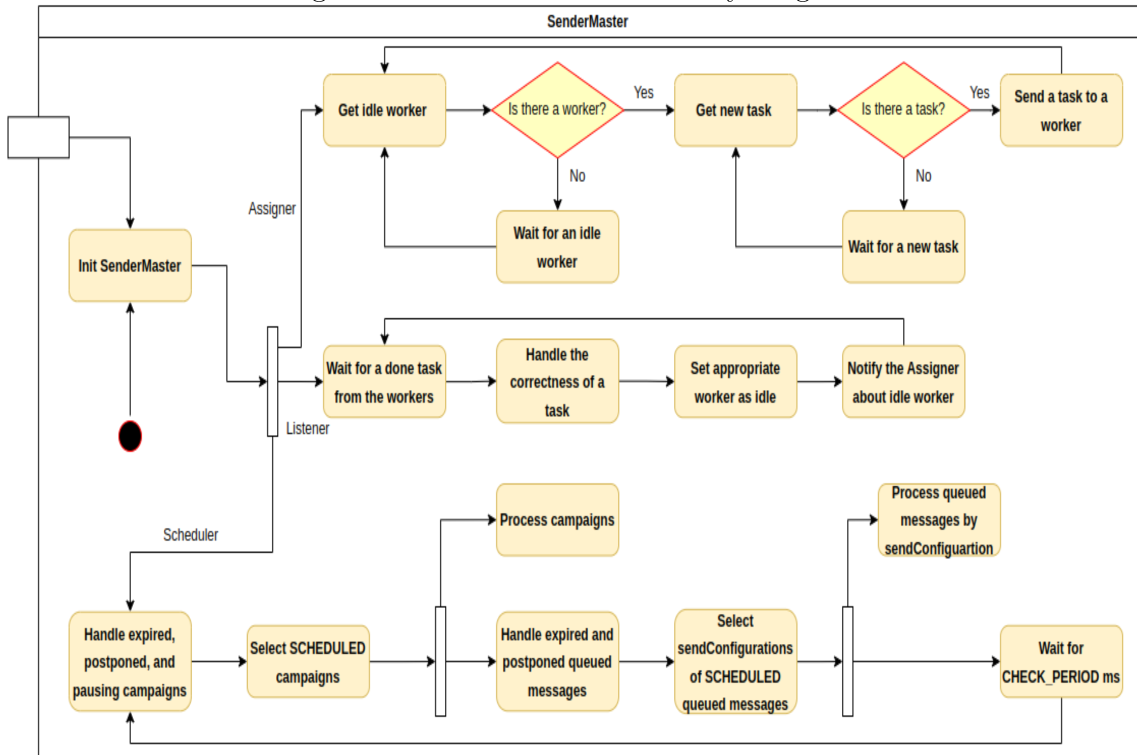
The service architecture starts with the queue of tasks. A task in this context means either some campaign in scheduled status or some queued messages prepared for the scheduling procedure. So this queue component does not represent some uniform data structure, but some abstract structure consisting of records from different MariaDB tables (`campaigns` and `queued`) where scheduled campaigns or queued messages are stored.

It is possible to create a task through admin GUI where a user clicks on the start button (regular campaign), which causes a change of campaign status to scheduled. An RSS campaign is scheduled by the Feedcheck service when it gets new data from the relevant RSS feed's URL, and then it creates a new RSS_-ENTRY campaign which is then set as scheduled just like a regular campaign. The last campaign type is started by service Triggers when a given event occurs. In that case, it creates all messages for this campaign and stores them in `queued` MariaDB table. Less used types of messages (Test, Subscription, API_Transactional) are created at the request of the given service and stored in `queued` table.

2.4.3.1 SenderMaster life cycle

If any of these tasks appear in the queue, then `SenderMaster` is supposed to schedule this task, store all scheduled messages in another queue, find idle `SenderWorker`, evenly assign a chunk of messages related to this task to found `SenderWorker`, and finally handle the response from `SenderWorker` after sending the whole chunk. In case of failed sending, handle the error correctly and reschedule this task if possible. Each task produces scheduled messages, which can be described as another queue containing messages prepared for `SenderWorkers`. If the details are ignored, it can be imagined as one queue of scheduled messages. Figure 2.4 precisely depicts how `SenderMaster` performs this whole procedure.

Figure 2.4: SenderMaster activity diagram



The root process forks the process during the system's start. Immediately after initialization, `SenderMaster` executes three asynchronous operations (Assigner, Listener, and Scheduler). The Assigner waits for an available task and evenly assigns it to idle workers. After task execution, the Listener waits for responses from `SenderWorker`, sets a postponing in case of a failed chunk, and notifies the Assigner about idle workers. The Scheduler in the given period continuously checks the queue of Sender tasks (Figure 2.3) that represents either scheduled campaigns or queued messages prepared for the following handling before assigning them to `SenderWorker`.

To avoid misunderstanding, it is essential to draw attention to the fact that the Scheduler and the Assigner work with tasks but are entirely different types. The Scheduler takes tasks from the queue of Sender tasks (Figure 2.3), and the Assigner takes tasks resulting from the Scheduler. So in this context, a task is meant as a chunk of scheduled messages (campaign or queued) prepared for

`SenderWorker`. One chunk always contains messages from the same campaign or only queued messages.

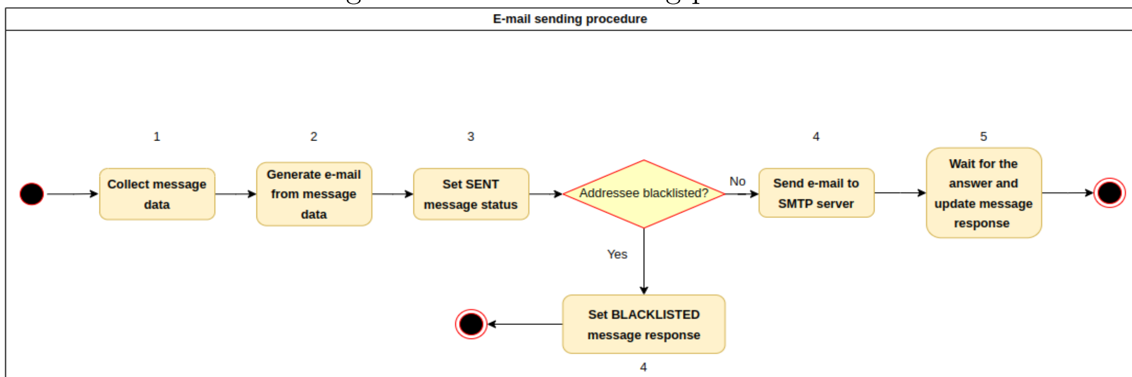
Now it is time to focus on the Scheduler (the most critical and complex subcomponent of `SenderMaster`) that, in the given periodic time, continuously checks the queue of Sender tasks and ensures the scheduling of each task. It starts by checking scheduled campaigns, and if there are some scheduled campaigns, the Scheduler checks whether they are not expired or postponed and move them to the next step. In the next step, the Scheduler selects all scheduled campaigns that have passed the previous step and create for each campaign an asynchronous operation that ensures sending the whole campaign. This asynchronous operation is responsible for creating messages of the given campaign, making and notifying the Assigner about a chunk of messages prepared for `SenderWorker` (new task available), and finally changing campaign status to finished when sending is done or paused when sending campaign is paused. Then the Scheduler continues by the same algorithm but for scheduled queued messages. After execution of one scheduling period, the Scheduler waits for CHECK_PERIOD ms and repeats the period.

2.4.3.2 SenderWorker life cycle

All processes of `SenderWorker` are forked by `SenderMaster` according to defined numbers in the configuration file. After forking, each `SenderWorker` starts in state idle and waits for a task (chunk of campaign or queued messages) from `SenderMaster`, which sends a task to `SenderWorkers` using interprocess communication.

This sending procedure contains the logical steps themselves executed by `SenderWorker` when it sends one e-mail since receiving a message (from a given chunk) from `SenderMaster` until sending a generated e-mail to the SMTP server.

Figure 2.5: E-mail sending procedure



After receiving a task from `SenderMaster`, `SenderWorker` generates an e-mail for each scheduled message intended for a particular subscriber. A message data from the received chunk contains only some of the needed information for generating a particular e-mail, so `SenderWorker` has to execute a couple of SQL queries to receive additional data. Firstly, `SenderWorker` is supposed to receive common data for this chunk (such as campaign data, send configuration, lists, attachments, template, etc.). This initialization is executed once for each chunk (end of step 1).

Then it generates one particular e-mail and has to execute some SQL queries for data related to this specific e-mail (such as subscriber fields, whether a subscriber is blacklisted, etc.). After performing these steps, the e-mail is generated and prepared for sending (end of step 2).

When the e-mail is generated, `SenderWorker` will set the message status in the database as sent (end of step 3). Then `SenderWorker` has to determine whether the addressee is blacklisted. If so, just set a BLACKLISTED response on this message. It can continue with the following message. Otherwise, `SenderWorker` creates transport with the SMTP server if it has not been created yet. This operation is done once for each send configuration during the entire program run and sends the e-mail to the server. It is waiting for its response, and after receiving a response, `SenderWorker` writes the response into the database table where a message is stored. Then continues with the following message in a given chunk. After sending all e-mails for each message, it sends a message using interprocess communication to `SenderMaster` with information about success.

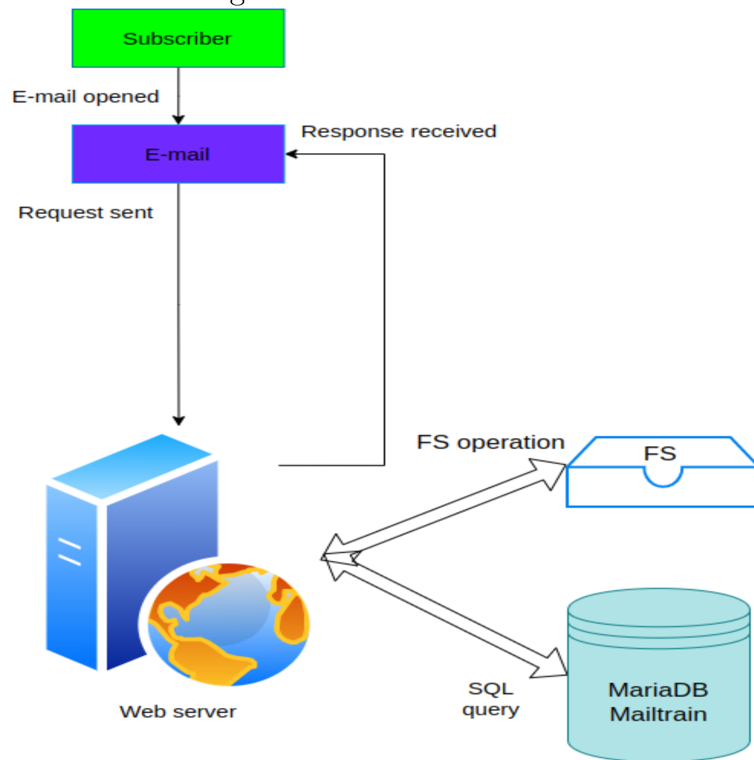
During sending e-mails between steps 3 and 5, sending is located in the critical section (message status is sent, and message response is null). It means that in this section, Mailtrain must not be turned off. Otherwise, some subscribers either will not receive their e-mails or receive them more than once because at the next start `SenderMaster` will not be able to recognize whether an e-mail has been sent to the SMTP server. The response could not be saved, or an e-mail was not even sent to the SMTP server.

The whole procedure of Sender service was described. In the following section, the PUBLIC server will be described.

2.4.4 PUBLIC server

Now it is essential to explain the PUBLIC server more deeply. The PUBLIC server does not require authentication and is used by subscribers for processing requests related to sent e-mails. In Figure 2.6, there is shown the internal architecture of the PUBLIC server.

Figure 2.6: PUBLIC server



There are four types of requests that are processed by the PUBLIC server:

1. **Subscription** - this type of request pertains to the subscription status of a particular subscriber and can be used to either subscribe or unsubscribe them.

2. **Links** - a type of request for processing requests related to links events. When some subscriber opens a received e-mail or clicks on some e-mail's link, this request is automatically sent and processed. This information is then used in making campaign statistics and reports.

3. **Archive** - a type of request that sends all archived e-mails which have been already sent. According to arguments, it generates the whole e-mail for a specific subscriber and sends it again.

4. **Files** - a type of request that sends all campaign files whose link is defined in the e-mail template. These files are requested when some subscriber opens the received e-mail. The requested file is additionally sent and received in the subscriber's e-mail.

In Mailtrain, it is crucial to ensure that the execution of requests of type `Links` and `Files` is highly available. Otherwise, when the PUBLIC server is not running, and some subscriber e-mail is open, it will cause that e-mail to be rendered without linked images, linked files will not be sent on request, and lose any information about subscriber events, which can even cause the loss of a subscriber itself.

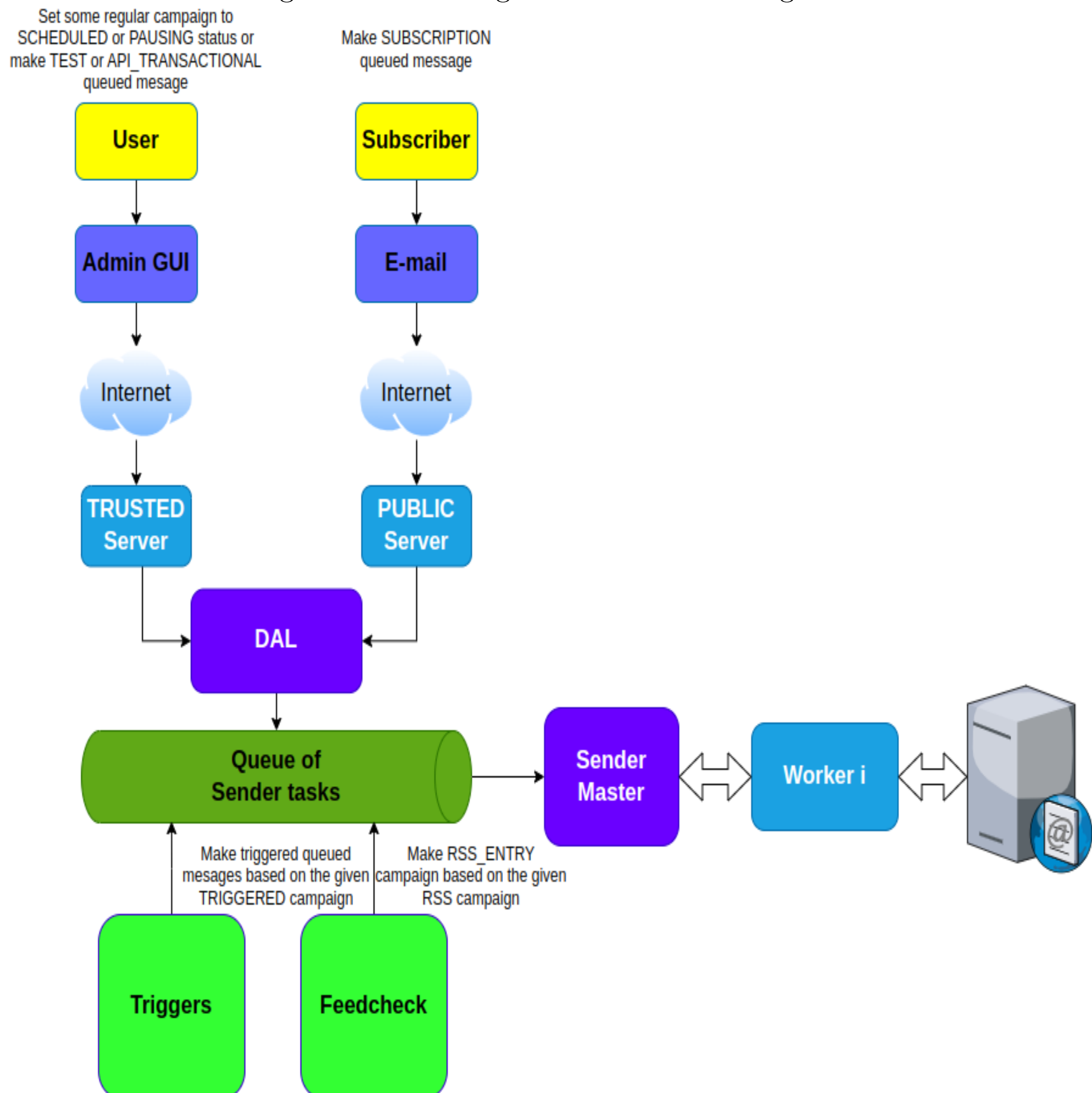
2.5 Sending e-mail use cases

Following a detailed description of the various message types and their corresponding sending components, it is beneficial to provide a comprehensive summary of all potential use cases for each message type. This includes tracing the life cycle of a message from its inception to its sending by the SMTP server. The use case diagram below provides a visual representation of all potential use cases for the various message types.

Figure 2.7 depicts four participators making Sender tasks (user, subscriber, Triggers service, and Feedcheck service). It is either a campaign task (user changed campaign status, Feedcheck created new RSS_ENTRY campaign) or queued task (subscriber created subscription message, a user created API_Transactional message, or Triggers created queued messages for a given triggered campaign).

When some task is located in the Sender queue, then the procedure of sending continues by `SenderMaster` as was described above.

Figure 2.7: Sending e-mails use case diagram



2.6 Summary of goals

After understanding all the necessary technical background about the current version of Mailtrain, it is important to summarize all goals also with needed technical details that are crucial to achieving in the next version:

1. Making Sender service horizontally scalable. It means that after adding more and more sending campaigns simultaneously, Mailtrain can increase the number of `SenderWorkers` without potential limits, which causes faster performance.
2. Ensuring high availability of critical services (Sender and PUBLIC server). It includes ensuring that critical service is running properly and correctly without any restrictions (except performance) in case it runs on at least one node and regardless of whether the rest of the system is currently running.
3. Adding to Mailtrain safe shutdown of the system. In the current version, if running Mailtrain is turned off, then it may cause sending messages will fall into an inconsistent state if there were sending e-mails in the critical section at that time.
4. Precise refactoring of the Sender component. The code is currently unstructured and messy, does not use any recommended standard of JavaScript language, `SenderMaster` contains callback hell, and documentation is of small scale. The goal is to create code that will be much more readable, maintainable, and upgradeable with quality documentation focused on quick understanding.

2.7 Source code overview

In this tree structure, the source code structure of Mailtrain is described. Only essential primary directories and modules used in this thesis are further described. This version is stored in `v2` Git branch.

```
mailtrain/
    client/ - the whole source code of Admin GUI written in React
    docs/ - brief documentation about features and deployment
    locales/ - all supported translations of Admin GUI stored in JSON
    mvis/ - the source code of a visual analytics tool
    server/ - the whole source code of the backend
        config/ - configuration file stored in YAML format
        files/ - directory where all file entities are stored
        lib/ - directory where auxiliary modules for other modules
               are stored
            mailers.js - a module used by SenderWorker for sending
                         e-mails to the SMTP server
            message-sender.js - a module used by SenderWorker for
                                generating e-mails from messages
        models/ - directory where queries for all entities
                  are written, it represents the DAL component
```

```
    protected/ - this directory serves for generated reports
    routes/ - directory where routing functions for all entity
              endpoints are written
        archive.js - a module used by the PUBLIC server where the
                     endpoint for sending archived e-mails is defined
        files.js - a module used by the PUBLIC server where the
                   endpoint for sending campaign files is defined
        links.js - a module used by the PUBLIC server where are
                   all endpoints for handling subscriber events
        subscription.js - a module used by the PUBLIC server where
                          are all subscription endpoints defined
    services/
        workers/ - directory where the ReportProcessor service
                   is written
        executor.js - source code of Executor service
        feedcheck.js - source code of Feedcheck service
        gdpr-cleanup.js - source code of GDPR Cleanup service
        importer.js - source code of Importer service
        postfix-bounce-server.js - test SMTP server source code
        sender-master.js - source code of SenderMaster service
        sender-worker.js - source code of SenderWorker service
        test-server.js - test SMTP server source code
        triggers.js - source code of Triggers service
        tzupdate.js - source code of TZUpdate service
        verp-server.js - test SMTP server source code
    setup/ - config files for the MariaDB database system
    test/ - a couple of backend tests
    app-builder.js - functions for setting up servers
    index.js - the main module that starts Mailtrain
    package.json - records important metadata of the server
                   source code
setup/ - deployment scripts
shared/ - all entity enums shared between the source code of
          the client and the server
zone-mta/ - config files for running the builtin Zone-MTA feature
```

2.8 Deployment requirements

In the last section, each requirement (hardware and software) is written that is necessary for the proper and smooth running of the entire system.

Recommended hardware requirements:

- **CPU** - 2 vCPU
- **RAM** - 4096 MB
- **Storage** - At least 500GB internal data storage
- **Network** - Network connection with a speed of at least 1,000 Mbps

Recommended operating systems:

- **CentOS 7+**
- **Debian 10+**
- **Ubuntu 18.04+**

Mailtrain is designed primarily for the most popular Linux distribution operating systems and is built on Node.js (v14+) and MySQL (v8+) or MariaDB (v10+). For operating systems not mentioned above, deployment and correct running are not guaranteed.

The exact deployment procedure is stored in `mailtrain/docs` directory.

3. Performance analysis

In this chapter, there will be described the base shortcomings and limits that current architecture contains. It means making performance tests on the Sender component as a whole, finding the main bottlenecks of the Sender component, and finding functions and modules in the source code responsible for that. Finally, in the next chapter, the solution for all bottlenecks and limits will be introduced from this analysis.

As was explained in the previous chapter, sending too many campaign e-mails simultaneously takes the most time, especially regular and RSS campaigns which are the most often used in production. So in this performance analysis, one regular campaign with a sufficiently large number of subscribers will be used since the number of sending campaigns does not influence the speed of sending, just the sum of all subscribers from each campaign.

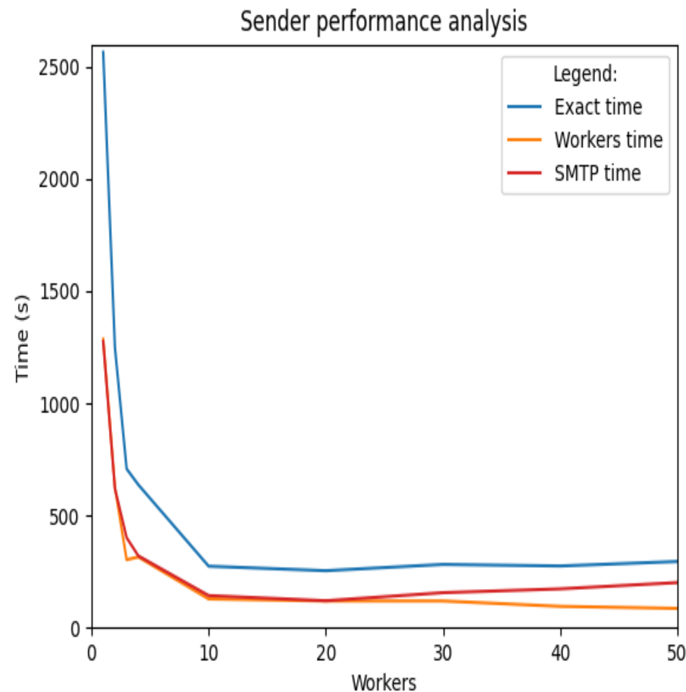
All performance tests have been executed on Intel® Core™ i3-7100U CPU @ 2.40GHz × 4 CPU and one centralized serial SMTP server (`services/test-server.js`) running on the same node.

3.1 Overall Sender analysis

For a better analysis of each bottleneck described above, Figure 3.1 depicts data from the overall Sender analysis where the x-axis indicates the number of `SenderWorkers` and the y-axis indicates the time during which the work was performed measured in seconds. In the graph, three curves represent a type of work (Exact time, Workers' time, and SMTP time).

The exact time is defined between points when the button to start sending the campaign is pressed and when the campaign status of FINISHED is displayed in the admin GUI. Workers' time is the average time per one `SenderWorker`, but it does not include SMTP server time, although after e-mail creation `SenderWorker` waits for the SMTP response. Only the time between receiving a chunk of messages, making e-mails, and sending them to the SMTP server is counted. SMTP time represents a time that takes only sending created e-mails delivered to the server from `SenderWorker`.

Figure 3.1: Sender performance analysis



In each test, 50000 e-mails have been sent for one regular campaign. From measured times, it is evident that the exact time can be divided in half, where the first half takes the SMTP server. It is a little bit more for a bigger number of `SenderWorkers`, but only because the centralized and serial SMTP server was used, and therefore increasing the number of `SenderWorkers` causes faster e-mail making and sending to the SMTP server. The second half takes all `SenderWorkers` working including making and sending e-mails except waiting for responses from the SMTP server. It includes mainly SQL queries gathering data about each chunk of messages, making e-mails, and other little operations.

For measuring the exact time, manual measurement was used through the stopwatch, and for Workers and SMTP time, there was a classical method used to determine how much time it took for function execution. Below, as an example, is a code for SMTP time measuring. The example function is located in the file `mailtrain/server/lib/mailer.js`, and its name is `_sendMail`.

```
{
    const startTime = new Date();
    const response = await transport.sendMailAsync(mail);
    const endTime = new Date();
    smtpTime += endTime - startTime;
}
```

Figure 3.2: SMTP time measuring

So one statement before the function call is stored as the current timestamp `startTime`, and one statement after the call is stored as the current timestamp `endTime`. The time difference is added to the result, whose sum of all time calls represents SMTP time.

After finding out these measured times of overall analysis, the next section

will analyze the performance of each Sender component which will result in more details.

3.2 Sender components analysis

After executing the overall Sender performance analysis, it is crucial to find out which exact functions in each source code module are responsible for slowing down.

Figure 2.7 depicts all possible use cases for sending e-mails. In this performance testing, one regular campaign is used, so the whole scenario starts from the user's point of view when a user starts sending a regular campaign. It is evident that the status from pressing the button until making a task in the Sender queue takes a negligible amount of time since it is just one request from the admin GUI to the TRUSTED server and executing one simple SQL query to the MariaDB database. The performance time of sending procedure is much more questionable from when `SenderMaster` starts to execute its procedure until all e-mails are sent through the SMTP server, and all responses are handled by `SenderMaster` and stored in the MariaDB database.

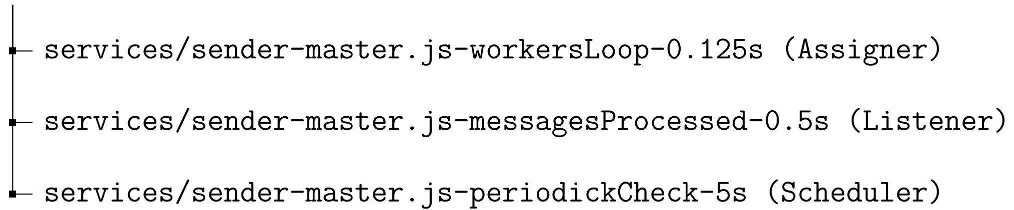
Executing some performance tests and analyzing each responsible function of the given module (component) is essential. From this point, all performance tests will contain just one `SenderWorker` to achieve the most accurate executed time. In the case of asynchronous operations, just like in `SenderMaster`, it is necessary to know precisely which functions represent particular callbacks and count only time when `SenderMaster` is computing and not just waiting for some another event.

The test will contain one regular campaign with 5000 subscribers and be sent by one `SenderWorker`, and test-server SMTP will be used.

The result of this analysis for each critical component will be a call tree, a structure very similar to a call stack. Still, unlike a call stack, a call tree contains all significant function calls that were called during program execution (function calls with a negligible amount of time are omitted). Each path from the root to some leaf in this tree represents one particular executed call stack. Each node of this tree contains the exact name of the function and module where this function is written and a time in seconds representing how much time this function took. Also, the execution time of all its children (called functions) is counted, so the sum of the time of all its children should be equal to node time (if the noise is neglected).

3.2.1 SenderMaster analysis

Firstly, it is essential to analyze `SenderMaster` component and find out how much time takes its functions during program execution. After analyzing this component, next will be necessary to analyze `SenderWorker` and SMTP server components. Then all of these components will be compared and found out main bottlenecks of the Sender.

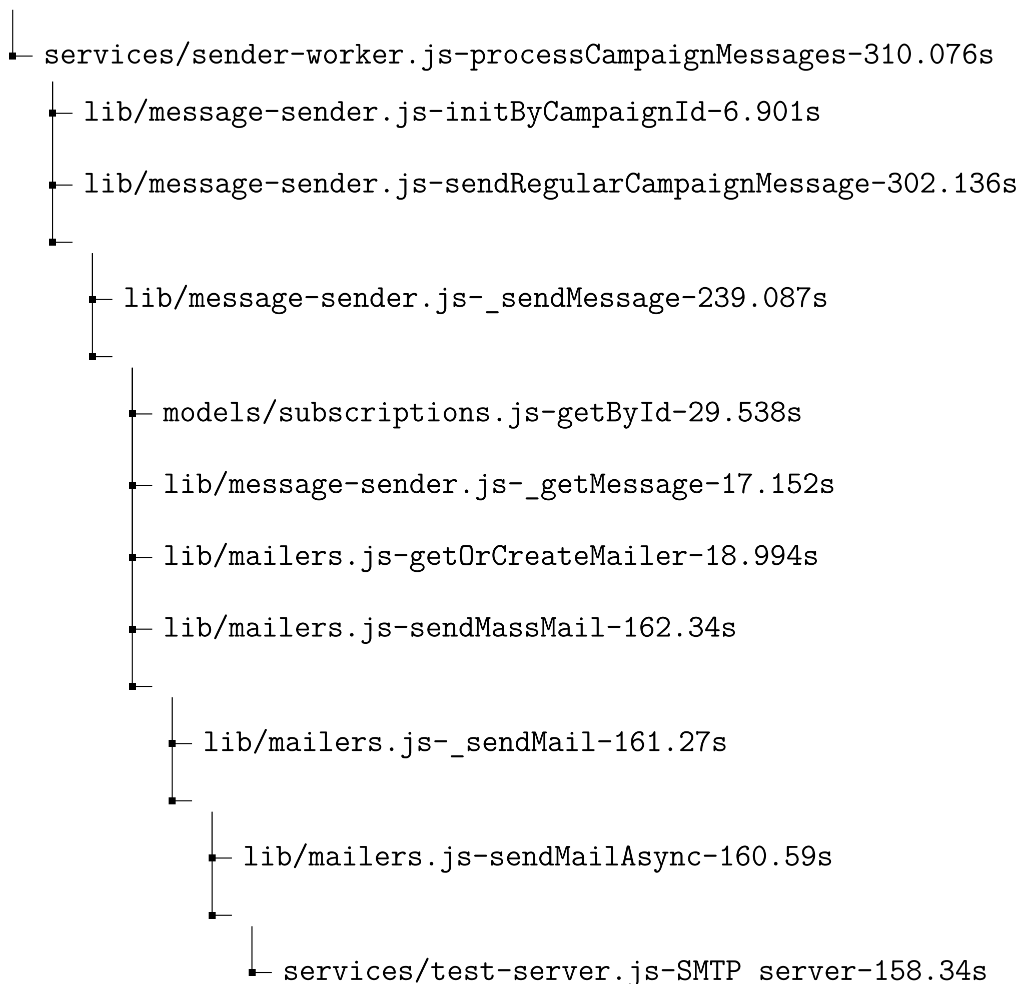


In this analysis, only times are counted when `SenderMaster` is computing something and not waiting for `SenderWorker` response. Because it is such a short time, the whole call tree with all called functions is not shown, just the root functions of particular components (Assigner, Listener, and Scheduler). It looks like that `SenderMaster` does not take much time compared to the exact time.

3.2.2 SenderWorker and SMTP server analysis

The second and most expensive components in terms of time are `SenderWorker` together with the SMTP server. The analyzed call tree below describes the execution time for each significant function call from `SenderWorker` and SMTP server component.

The root directory for this tree is located in `mailtrain/server`.



From the call tree, it is possible to see that `SenderWorker`, also with the SMTP server, takes almost all the time to send a campaign. From the total exact time of 320 seconds, the SMTP server took 158.34 seconds, and `SenderWorker` took

310.076 seconds or 151.736 seconds if the time when `SenderWorker` was waiting for the SMTP server responses is not counted.

3.3 Summary of Sender bottlenecks

There are four main bottlenecks in our current architecture of Sender sorted in descending order of impact:

1. **Inefficient SMTP server** - The first and the most significant bottleneck is the SMTP server. A centralized serial SMTP server for sending e-mails was used. So when `SenderWorker` makes all e-mails for sending and sends them to the SMTP, and the server throughput limit is reached, then all other `SenderWorkers` with made e-mails must wait for SMTP server processing. If this bottleneck wants to be removed, having a high-performance SMTP server with the same performance as the rest of the Sender components is crucial. Only in this case will the SMTP server not be a bottleneck that slows down the Sender processing.

2. **Centralized MariaDB database** - If all functions on the call tree of `SenderWorker` are analyzed in the source code, then it follows that most of the time of `SenderWorker` execution (except waiting for the SMTP server) takes SQL queries when all `SenderWorkers` need to query the same MariaDB database. MariaDB database system has limited performance in parallel computing, so in the next version, it needed to use a database system for the Sender component that can handle large data more efficiently and scale horizontally.

3. **Centralized `SenderWorkers`** - Since the current version of Mailtrain already has a parallel architecture of Sender, which runs only on one node. The limit of the node has been reached, and further adding `SenderWorkers` on one node would only make the performance slower. So in the next version, it is needed to ensure running `SenderWorkers` in a distributed system and reach horizontal scalability for adding `SenderWorkers`.

4. **Centralized `SenderMaster`** - The last bottleneck is in the `SenderMaster` component. The worker scheduling and task assignment are done by one centralized process. Although from the executed analysis, `SenderMaster` does not take too much time compared with `SenderWorkers` and SMTP server, if the next version should be able to scale horizontally on large data, then the Sender scheduling cannot depend on one centralized process. Firstly, there would be a performance problem with too many `SenderWorkers` in scheduling and processing. The second problem would be related to high availability in case of failure since `SenderMaster` runs on one node. So in the next version, `SenderWorkers` scheduling and processing cannot be done by this or other similar centralized strategies.

3.4 Analysis conclusion

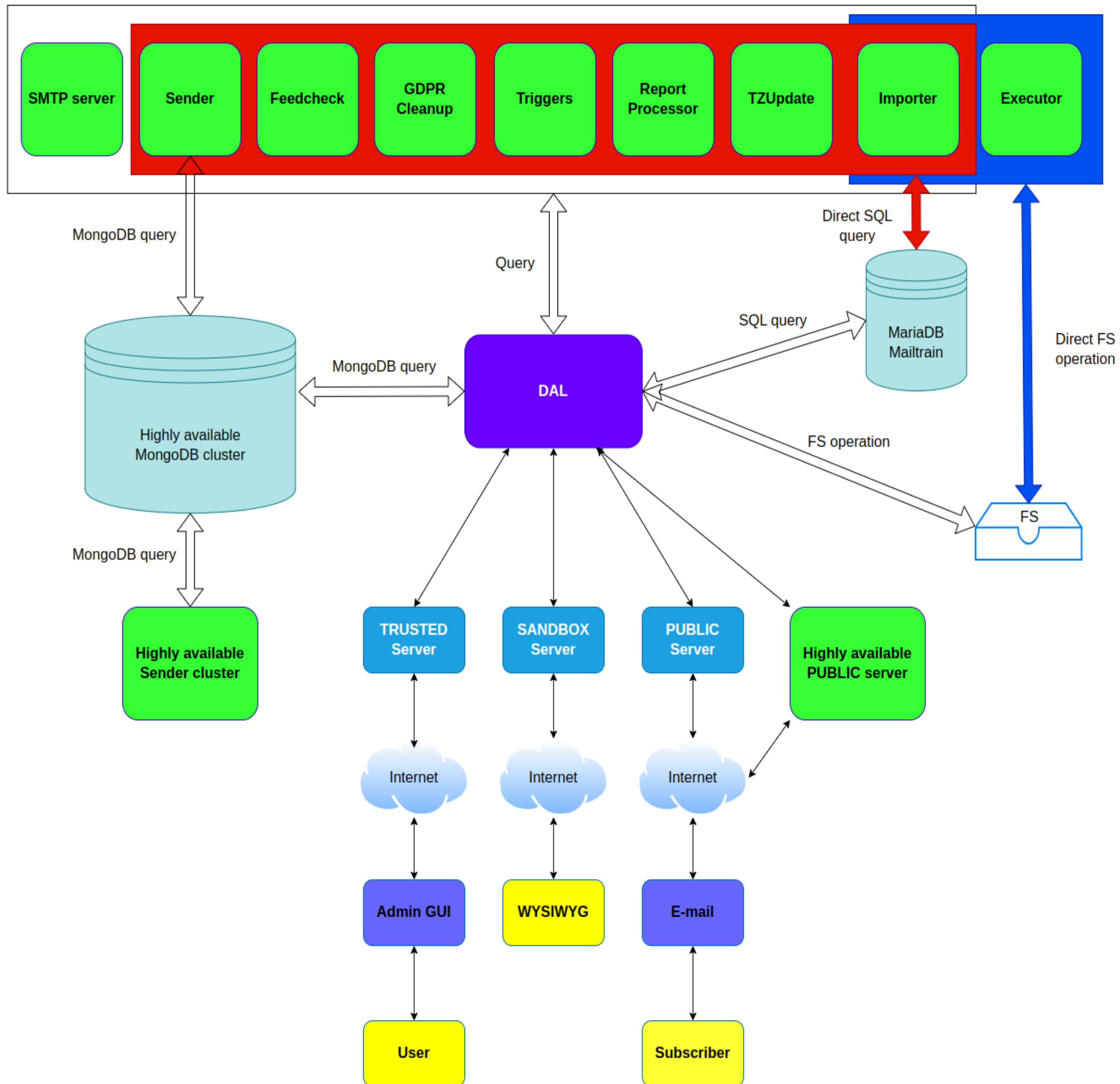
This chapter conducted a thorough analysis of the performance of the Sender component in its entirety, as well as the individual components joined with the source code modules and functions. Through this analysis, the performance bottlenecks hindering the sending of campaigns were identified, and recommendations were put forth to eliminate them in the subsequent version of the system.

Based on the above-mentioned findings, the subsequent chapter will be devoted to designing a novel Mailtrain architecture that not only addresses these bottlenecks but also encompasses all the features outlined in Section 2.6.

4. New architectural design

After describing the Mailtrain overview and execution performance analysis, it is time to introduce the architecture of the new version of Mailtrain. The whole architecture is shown in Figure 4.1.

Figure 4.1: Distributed Mailtrain design



Mailtrain is divided into two sections. The first section is highly available and includes three components (highly available MongoDB cluster, highly available Sender cluster, and highly available PUBLIC or HAPUBLIC server). All of these components can run independently on the rest of Mailtrain. They only depend on the highly available MongoDB cluster. They can also run in a distributed system and scale horizontally. The second section is, as expected, non-highly available, including the rest of Mailtrain, which can still run only on one node.

The HAPUBLIC server has a purpose, just like the PUBLIC server. However, it supports high availability, horizontal scaling, and types of requests `Links` and

`Files` have been moved to it from the PUBLIC server since they belong to the group of critical services.

4.1 New database system

MongoDB [8] has been chosen as the new database system for processing large data of the Sender component and ensuring high availability for both the Sender component and the HAPUBLIC server. The main reasons for this choice are summarized below compared with the current supported database systems MySQL and MariaDB:

MySQL database system [4] has limited options for scalability, and it supports vertical scalability by adding more resources to the existing database server, which does not help since the upper limit has been reached. For horizontal scalability, it offers to add read replicas. Still, it also has many limitations compared to MongoDB (a replica set replicates a group of MongoDB servers that hold the same data, ensuring high availability and disaster recovery). It is limited to five replicas and can be used only for read operations. Since many write operations are in the Sender component, this would not bring too significant an acceleration. There has been added multi-master replication focused on solving this problem with many write operations, but its implementation is more limited than the functionality available in MongoDB.

In the MongoDB database system, the situation is much more comfortable and suitable for horizontal scalability. The limit for replication is much bigger, and it gets better with newer versions. It supports distributed transactions across many replicas and has overall faster execution of queries on large data in most cases.

In addition to replication, MongoDB supports one more highly flexible feature named sharding. When some customer's data starts to overgrow, and one node cannot store it, or there are too many write operations, data is distributed across many servers. A lot of memory and time is saved. This approach can horizontally scale applications regardless of the number of read/write operations. In case of this thesis mainly concerns the number of campaign messages.

In contrast, the MariaDB database system [3] utilizes various engines and components integrated into the MariaDB Server, creating a robust and scalable cluster that also supports sharding, such as the MariaDB MaxScale [15] and MariaDB Spider engine [20]. Nevertheless, its foundation and available services have some limitations that hinder its complete integration into the big data [11] and cloud-based environment of modern agile development.

MongoDB was built from the ground up to support cloud elasticity and any amount of data, starting from small on-device databases to large multi-petabyte clusters. It runs the same way anywhere you want, including within our fully featured and scalable cloud database offering, MongoDB Atlas.

4.1.1 Integrating MongoDB into Mailtrain

Upon the selection of a new database system for Mailtrain, various issues arise that require resolution before its integration. For instance:

- Which data to store in MongoDB and which to keep in MariaDB?
- Which data should be stored temporarily and which permanently?
- How will the schema of the MongoDB database look?
- How to ensure data synchronization and consistency between MariaDB and MongoDB?

It is simpler to start with the HAPUBLIC server where only `Links` and `File` requests are processed. The server should be able to read the latest data for both types of requests at any time. It means accessing the latest data in tables `campaign_links` and `files_campaign_file`.

For the Sender component, many more tables and their SQL queries are needed for sending some specific campaign or queued message. Except for read operations, the Sender should also be able to write processed data into the database (for example, the response from the SMTP server of some sent message).

The first possible solution is to migrate all the needed tables into the MongoDB database and reprogram all their SQL queries. This solution looks straightforward since the Sender and the HAPUBLIC server would be totally independent of the MariaDB database. But there are too many SQL queries for these tables used by different components for computing various metrics and reports or inserting, updating, and deleting records. Executing many of these queries has no problem with performance, and there is much better support of SQL language than NoSQL MongoDB, which does not support JOIN operations. Therefore, migrating all of these tables and reprogramming all their queries from MariaDB into MongoDB does not make sense from a time point of view.

The second solution is to leave all the needed tables in the MariaDB database. When MongoDB needs some data, it will send one big query into the MariaDB database, receive the queried data and send the result back after processing. This solution requires a slight change of SQL queries but has several problems. MariaDB is non-highly available, so MongoDB may not get a response at any time. Another problem is that the results of many big queries can have a significant intersection between them. So it would be a big waste of memory (for example, two sending campaigns can have many familiar subscribers).

Instead, a much better solution is used that considers the advantages and disadvantages of the previously mentioned solutions. Only tables with a large data size will have their own collections in MongoDB. This will solve the problem of wasting a lot of memory. However, so that it is not necessary to rewrite all SQL queries of particular tables into MongoDB queries, there is used a different strategy. All created collections are of one of these two types:

The first type represents something called symmetric replication. It means that the MariaDB table and its corresponding MongoDB collection must always have the same state (each row has exactly one equivalent document and vice versa). So Mailtrain has to maintain that if some write operation is executed on the MariaDB table, then this operation is also automatically and atomically called on the corresponding collection in MongoDB and vice versa. This type of collection is named symmetric replication collection.

The second type of collection represents a processing queue. It means a record is created in a MariaDB table, but processing it is only possible in the corresponding

MongoDB collection. For example, campaign messages are created and stored in the MariaDB table, but sending procedure is done by a service that communicates only with MongoDB. So, in this case, Mailtrain has to maintain that when this record is created, it is sent to a particular MongoDB collection. After successful processing, this record is returned to the MariaDB table with processed data and removed from the MongoDB collection. It follows that Mailtrain will need to have some component for constant synchronization between these MariaDB tables and their corresponding MongoDB collections. This component is named Synchronizer and will be introduced in the following sections. This type of collection is named queued collection.

If some collection is created from its particular table, the collection's schema is created so that all columns represent keys and all rows represent values for a particular key.

Here is a list of all collections created in the MongoDB database that has also its own table in the MariaDB database:

- **blacklist** - a symmetric replication collection used by the Sender to store information about each blacklisted subscriber.
- **campaign_links** - a queued collection used by the HAPUBLIC server for storing information about each campaign link, mainly about how many times subscribers clicked a link.
- **campaign_messages** - a queued collection used by the Sender to store all campaign messages scheduled for sending.
- **files_campaign_file** - a symmetric replication collection used by the HAPUBLIC server where information about each campaign file is stored.
- **links** - a queued collection used by the Sender to store information about each campaign link used in the e-mail template.
- **queued** - a queued collection used by the Sender to store all queued messages scheduled for sending.
- **subscription___i** - a symmetric replication collection used by the Sender to store information about each subscriber from a list i.

There are also some still needed data from different MariaDB tables necessary for making and sending e-mails, such as data about subscriber custom fields, campaign attachments, campaign e-mail templates, send configurations, etc. But all these records take up little memory, so instead of creating corresponding collections in MongoDB, it is more optimal to constantly send data together with newly created tasks (sending campaigns or queued messages). For example, suppose some campaign A is set to SCHEDULED status. In that case, the non-highly available Sender has to collect all needed data from tables that do not have corresponding collections for sending this campaign and send them in one operation together with task data. The newly created collection for sending campaigns is named `tasks`. Collecting all required data is the responsible `DataCollector` and

will be introduced in the following sections. There can still be some intersection data between some tasks, but it will only be negligible.

It is not crucial to store these tables in the non-highly available MariaDB database because if the non-highly available section has failed or is turned off, a user cannot start sending new campaigns. All campaigns that started before failing are sent without the need for interaction with the non-highly available section.

4.1.2 Ensuring data consistency

From the explanation of the previous section, it is pretty evident that problems of high availability, wasting of memory, data migration, and collection schemes are solved. However, the problem of synchronization and data consistency between each table and its collection still remains open.

The situation is easier for queued collections since there is no requirement for constant consistency. The main requirement is that each record from some table has to be sent to a particular collection. In the shortest possible time after processing, the result has to be sent back to the same table and a record updated. A record cannot be lost in both directions. The main idea of this solution is that each record is always first sent and then deleted from a particular table/collection. In case of system failure between sending and deleting, it is done again after the system starts. If a record has been already sent, the thrown exception about a duplicated record is just ignored for the insert operation, a record is updated more times for the update operation, or the delete operation is executed more times. All of these cases do not break data consistency. Deeper details will be explained in the Synchronizer section.

Ensuring data consistency for symmetric replication collections is more complicated because it is crucial to ensure that each SQL operation executed on some table will be immediately and atomically executed also on the particular collection. It means that both databases must have the same data at any time. This restriction is because the Sender always needs the latest data for sending e-mails. Otherwise, if some user updated some subscriber's records, the Sender would still use some old data until synchronization execution. The second problem is durability or what to do in case of system failure during transaction execution. The requirement is that transaction upon MariaDB and MongoDB database is either fully committed or aborted.

The solution is based on TCC [12] protocol with minor changes. Two transactions must be executed as one whole transaction, one for the MariaDB database and one for the MongoDB database. When it is needed to execute some request consisting of some SQL and MongoDB queries, it is always first started MongoDB transaction and immediately after that MariaDB transaction. All of these queries are commonly executed, and after executing all queries, the MariaDB transaction tries to commit changes, and then the MongoDB transaction tries to commit changes. So when some error or system failure occurs during query execution or MariaDB transaction committing, all changes are successfully roll-backed. The only problem happens when a MariaDB transaction is successfully committed and a MongoDB transaction is aborted for reasons such as write conflicts. In this case, the user is appropriately informed that databases are now in an inconsistent state,

and the user has to fix it manually by immediately repeating the transaction. Nevertheless, it is essential to realize that the probability of write conflicts strictly between MariaDB and MongoDB transactions committing is extremely low since the query is executing on databases with the same data.

There is still a problem with durability because resistance to system failures or other external factors is not guaranteed in case failure occurs between MariaDB and MongoDB transactions committing. So this case must be implemented manually. As mentioned above, it is essential to realize that the probability of system failure strictly between MariaDB and MongoDB transactions committing is extremely low. But if it happened, the script for checking and ensuring data consistency would be executed during the next system start.

It is helpful to mention that this solution in this situation is quite sufficient. It is essential to realize that in production, all symmetric replications are primarily read and updated just by the user who created them so the probability that transactions will be successfully aborted and roll-backed because of write conflicts is almost zero. In the next version, there can be considered using the two-phase commit protocol [9] as the solution if it will be necessary.

4.2 Distributed Sender

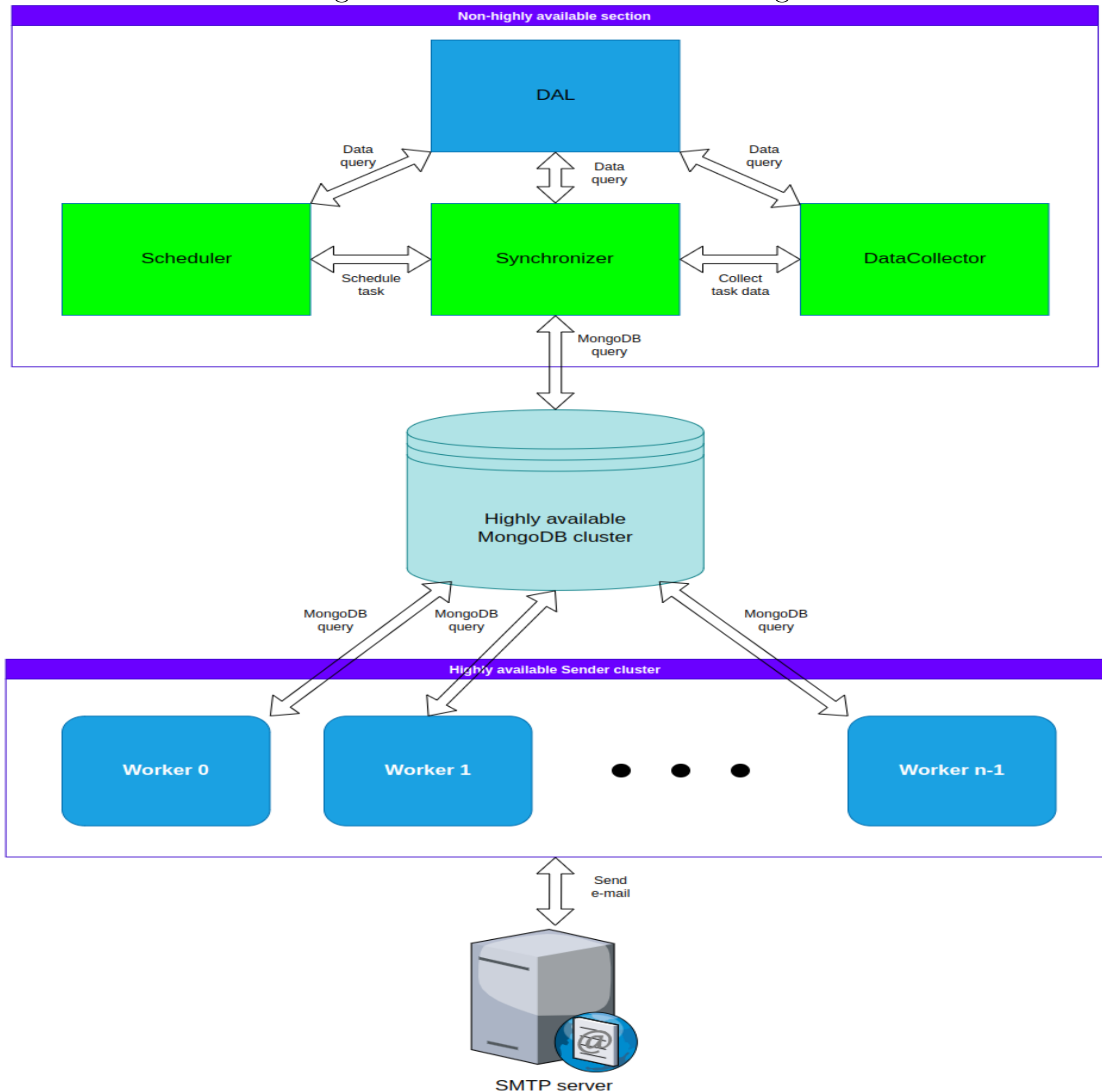
As was described in the previous section, the Sender is in the new version divided into highly available and non-highly available sections. Both sections communicate together only through the MongoDB cluster. The primary purpose of the highly available section is to make and send e-mails with high performance and the possibility of horizontally scaling the number of `SenderWorkers`. On the other hand, the primary purpose of the non-highly available section is scheduling tasks (campaigns and queued messages), handling processed tasks, and synchronizing these two sections. It means sending all the processed data from the MariaDB database into the MongoDB cluster and vice versa.

There are three components in the non-highly available section. The main is called Synchronizer, which ensures synchronization between these two sections. The second is Scheduler, which schedules all campaigns and queued messages, and the last is `DataCollector`, whose purpose is to collect all needed data for making and sending all e-mails for a given task. Synchronizer uses it when sending all needed data to MongoDB cluster in one operation.

The highly available section contains only `SenderWorker` that will be described in more detail in the following sections.

The last component is the SMTP server, which remains unchanged.

Figure 4.2: Distributed Sender design



Compared to the previous version, this architecture has no centralized `SenderMaster` for task assignment. The task assignment is done with a completely different strategy which is not centralized and thus is not dependent on one specific process. Each scheduled message always has also stored a hash of a subscriber's e-mail address in the database. It is a relatively long value, so the first four bytes are cut from it, and even this value (called `hashEmailPiece` in the collection `campaign_messages`) is used for task assignment. During `SenderWorker` initialization, there is an assigned hash range for each according to this statement:

```
const range = {
    from: Math.floor(MAX_RANGE / maxWorkers) * workerId,
    to: Math.floor(MAX_RANGE / maxWorkers) * (workerId + 1)
}

if (workerId === maxWorkers - 1) {
    range.to = MAX_RANGE;
}
```

Figure 4.3: SenderWorker default hash range computing

The value MAX_RANGE represents constant 2^{32} and defines range <0, MAX_RANGE) which `SenderWorkers` evenly divide among themselves according to their `workerId`, `maxWorkers` represents a maximum of `SenderWorkers` which can run (value can be changed in the configuration file before every Mailtrain start), and `workerId` represents the Id of currently initializing `SenderWorker` (values from the set Ids = {0, 1, ..., maxWorkers - 1}). Every `SenderWorker` sends only scheduled messages whose hash value belongs to its hash range.

This hash is computed according to the popular algorithm SHA-512, so the evenness of the distribution of messages among `SenderWorkers` is guaranteed [21] after neglecting a minor deviation.

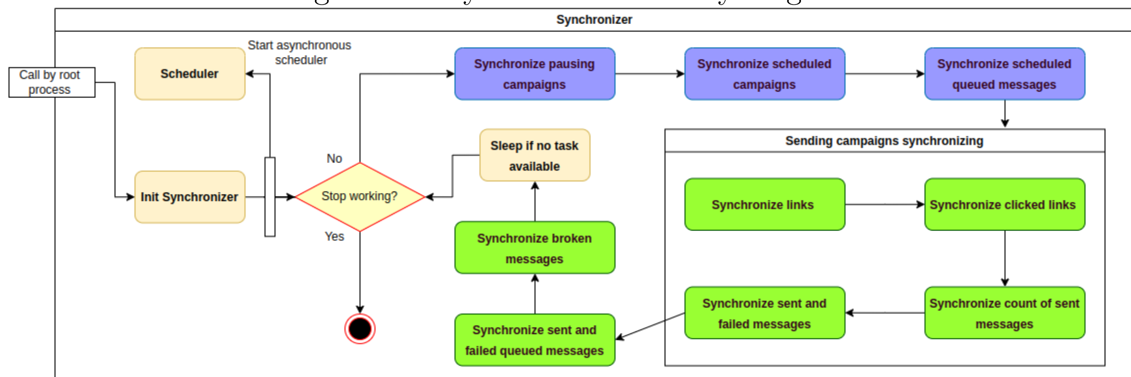
The following sections will describe components of non-highly available and highly available sections in more detail.

4.2.1 Synchronizer

Since Mailtrain is divided into two sections that can run independently of each other, it is obvious there has to be some component for bilateral synchronizing data between these sections. For synchronizing from the non-highly available section into highly available, there are operations for all collections of queued collections (collections of symmetric replication are synchronized directly by the Mailtrain strategy described above). There are operations of pausing campaigns that are sending, starting sending of SCHEDULED campaigns, and the same for queued messages. For synchronizing from the highly available section into non-highly available, there is synchronizing data about sent campaigns and queued messages, setting the FINISHED status for sent campaigns, and synchronizing campaign links. It is helpful to mention that campaign messages contain the most memory in both databases and take the most time in Mailtrain.

The process starts with Mailtrain forking and then with initialization. To ensure a safe shutdown of the system, the Synchronizer during initialization catches program interrupt signals `SIGINT` and `SIGTERM` [16]. Then it sets a callback that changes the variable value according to Synchronizer will know that it has to stop working after the current iteration. The Synchronizer also starts an asynchronous operation that represents the Scheduler (the same component as the Scheduler in the `SenderMaster`) which will be described in the next section. After initialization, the Synchronizer continues in the life cycle.

Figure 4.4: Synchronizer activity diagram



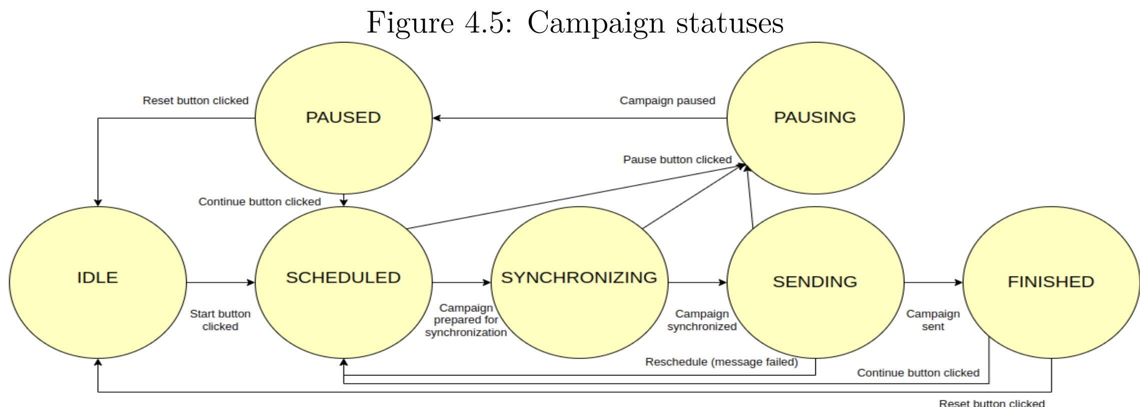
The Synchronizer shares three lists with the Scheduler. It is the list of pausing campaign IDs, scheduled campaign Ids, and scheduled queued messages. These lists represent the only connection through which the Scheduler and the Synchronizer communicate. The Scheduler, always in its check period, inserts new values into these lists, and the Synchronizer picks them and continues with the following processing.

So when the Synchronizer starts its life cycle, it starts with synchronizing operations from the non-highly available section into highly available, where it works with all these shared lists. Firstly, it checks the list of pausing campaigns. If some sending campaign is requested to be paused, Synchronizer will remove the corresponding sending campaign from the MongoDB database.

It is essential to mention that since this synchronizing was added to the sending campaign procedure, a new status for a campaign was created. It is called SYNCHRONIZING and defines the status between SCHEDULED and SENDING statuses. So when the Scheduler finds some scheduled campaign, it changes its status to SYNCHRONIZING. Then it is changed to the SENDING status if all needed data are sent to the MongoDB database, and `SenderWorkers` can start with sending.

The state diagram 4.5 depicts all possible campaign (Regular, RSS_ENTRY) status changes:

Figure 4.5: Campaign statuses



So the Synchronizer picks from the list of scheduled campaigns in SYNCHRONIZING status. It has to collect all the needed data from `DataCollector` for sending procedure and send it to the MongoDB database in one operation. After sending, the campaign status is set to the SENDING value, and `SenderWorkers` can start with sending.

Finally, regarding operations in this direction, it checks the list of scheduled queued messages, collects all needed data from `DataCollector` for sending them, and sends them together with collected data into the MongoDB database.

The second groups of operations have the opposite direction, from the highly available MongoDB database into the non-highly available section. Then it is needed to synchronize all sent campaign messages. Firstly, the Synchronizer synchronizes just the count of sent messages, and even then, it synchronizes one chunk of sent messages. The reason for this strategy is that it represents a trick to speed up the sending procedure. Synchronizing all messages is a much slower operation than synchronizing the count of them. Since it is essential to

give current feedback to a user about sent campaign messages, the Synchronizer first synchronizes the count of sent messages and then the chunk of messages itself. So also, after sending the campaign when it is in FINISHED status, some synchronizing operations of the remaining messages must be done. Then the Synchronizer synchronizes queued messages.

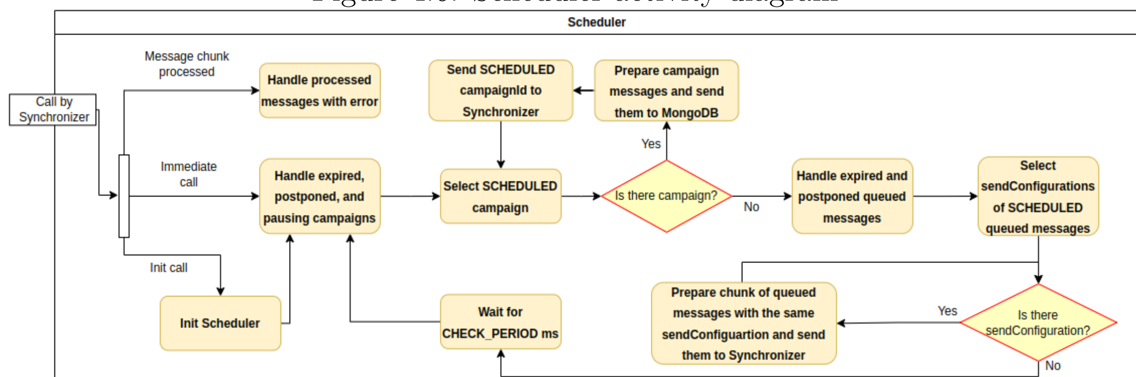
The last operation is removing broken messages. The term broken message represents a message with the status SENT, no response, and is stored in the MongoDB for more than one day. This type of message can occur when a node that is sending this message is turned off in an unexpected way, and the sending procedure is in the critical section. Since it is hard to find out whether a message was sent, this message is not tried to send again. The probability that a message in this state occurs is very low.

After execution of the whole iteration, if there is no available task (shared lists with the Scheduler are empty and the Synchronizer synchronized no data from MongoDB in the last iteration), then the Synchronizer goes to sleep and is woken up by the Scheduler at the nearest start of the check period.

4.2.2 Scheduler

As in the previous version, where `SenderMaster` in set period checked in the MariaDB database whether there are some scheduled campaigns or queued messages, here it is also essential to have a component for the same purpose.

Figure 4.6: Scheduler activity diagram



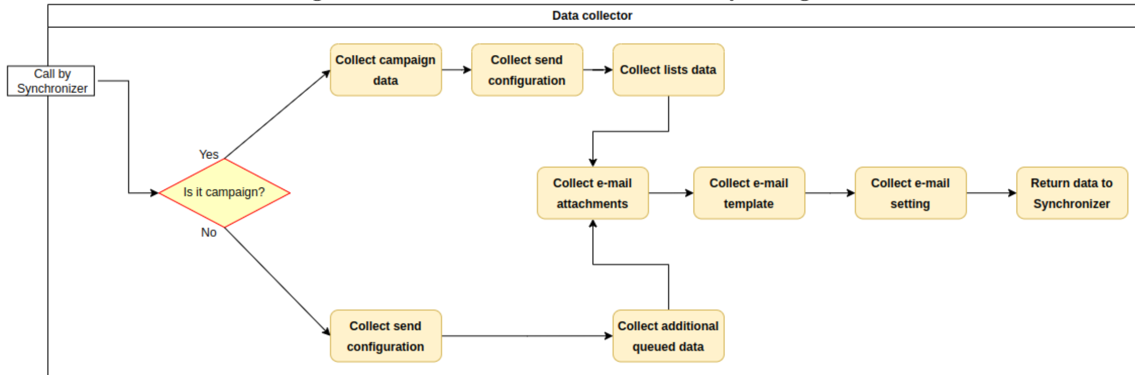
The Scheduler represents an asynchronous operation started by Synchronizer during initialization. Just like `SenderMaster`, campaigns and queued messages are processed in the defined period and sends them to the Synchronizer for the following processing described above.

This component also supports immediate calls. When a user clicks on the Send button to send some campaign, the Scheduler can be in the state when it has just accomplished one period and gone to sleep. In that case, a scheduled campaign must wait for CHECK_PERIOD milliseconds until the Scheduler procedure starts. So for this case, when some campaign is set to SCHEDULED status, the immediate call for the Scheduler is called, so there is no need to wait for the start of the next period if the Scheduler is sleeping right now.

4.2.3 DataCollector

When the Synchronizer needs to collect all data for some campaign or queued message and send them to MongoDB in one operation, it uses `DataCollector`. `DataCollector` represents a stateless component that executes only read queries into the MariaDB database. It does not represent any forked process or asynchronous operation running in the background. `DataCollector` is just a class with methods for data collecting which are called by Synchronizer when it is needed.

Figure 4.7: DataCollector activity diagram



The procedure of collecting data for a campaign or queued message is shown in Figure 4.7.

4.2.4 SenderWorker

Following the description of the components from the non-highly available Sender section, it is now appropriate to shift focus to the most critical component of the highly available Sender section, namely, the `SenderWorker` responsible for making and sending e-mails.

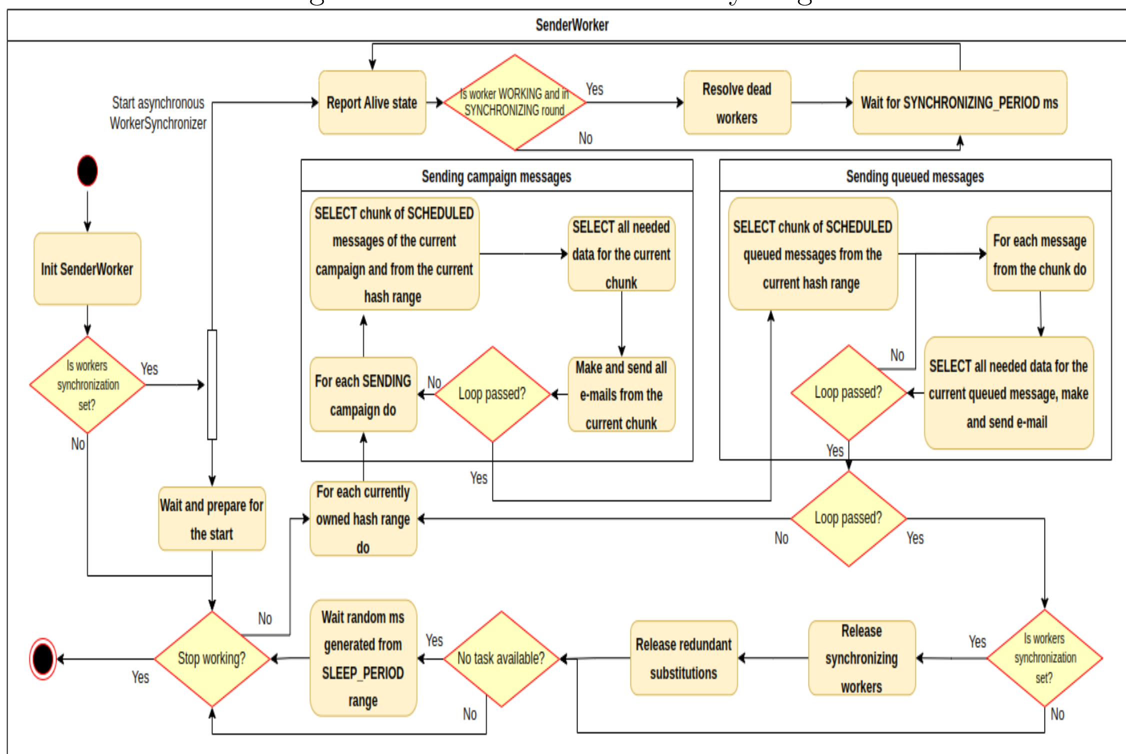
4.2.4.1 Life cycle

The life cycle is preceded by initialization where `SenderWorker` has to set all needed fields for running. It includes the Id, the maximum number of `SenderWorkers`, the hash range, and the state in which it is actually located.

After initialization, it is asked whether worker synchronization is set. Firstly, the life cycle will be explained when worker synchronization is not set. The following section will explain the reason for this synchronization and how the life cycle changes when worker synchronization is set.

Figure 4.10 depicts the whole procedure of `SenderWorker` from initialization to the entire life cycle.

Figure 4.8: SenderWorker activity diagram



Now `SenderWorker` starts the life cycle. Just like the Synchronizer, to ensure a safe shutdown of the system, `SenderWorker` during initialization catches program interrupt signals `SIGINT` and `SIGTERM`. Then it sets a callback that changes the variable value according to `SenderWorker` will know that it has to stop working after the current iteration.

The sending starts with the loop that iterates each currently owned hash range (which will be explained in the following section). Then it starts with campaign messages, where it iterates through each campaign in SENDING status, and for each campaign, it selects a chunk of its messages from the current hash range. The next step includes selecting all needed data from MongoDB collections of this chunk for making and sending e-mails. So it is important to select all subscribers blacklisted. Everything is done in one MongoDB operation. This is the subsequent significant optimization compared to the previous version. In the previous version, `SenderWorker` received a chunk of messages from `SenderMaster` and executed many SQL queries for each message separately which caused unnecessary delay. Then `SenderWorker` iteratively makes e-mails from messages and sends them. The procedure of making and sending e-mail remains the same. Only the source code was rewritten with a focus on readability and extensibility.

If a chunk was sent for each sending campaign, it continues sending queued messages. Here is essential to mention that MongoDB queries are not optimized for queued messages, but it is not a big problem since queued messages are not widely used in production.

If there is no available task right now, it generates a random number of seconds from the SLEEP_PERIOD range (10, 30) and will go to sleep for this time. The reason for using this strategy is that there cannot be any trigger in the non-highly available section that would send information to all `SenderWorkers`

about tasks since it cannot depend on the non-highly available section. On the other hand, it would not be optimal if all `SenderWorkers` sent their requests to the MongoDB cluster constantly without any pause in case there is no task. It would cause overwhelm the entire MongoDB cluster. So with this strategy, there is no centralized trigger, nor MongoDB cluster is overwhelmed since there are pauses between two requests, and the probability that all `SenderWorkers` send their requests at the same time is very low.

4.2.4.2 Worker synchronization

So far, only `SenderWorker` containing one default and unique hash range computed during initialization was described. But in production, it is common for the number of `SenderWorkers` to change quickly depending on currently running nodes and the amount of work available. Either some node fails, or the running platform increases or decreases the number of `SenderWorkers` depending on the amount of work available. In both cases, the Sender has to ensure that all running `SenderWorkers` will temporarily substitute the work of dead `SenderWorkers`.

This procedure is called worker synchronization and will be described right now. If there are currently working and dead `SenderWorkers` and this number can change depending on the time, it is essential to define some states where `SenderWorkers` can be. Each `SenderWorker` can be in one of these defined states:

1. **SYNCHRONIZING** - `SenderWorker` process has started and is waiting for the start of its life cycle.
2. **WORKING** - `SenderWorker` process is executing work.
3. **DEAD** - `SenderWorker` process is dead.

Since every `SenderWorker` must be able to read the state of any other `SenderWorker` at any time, it is crucial to have some place of memory where it would be stored, accessible to any `SenderWorker`, and highly available. For this reason, a new MongoDB collection has been created with the name `sender_workers` and contains every information about each `SenderWorker`. The size of this collection is equaled to the value of maximum `SenderWorkers`.

A document from this collection has this schema:

```
{
  "title": "sender_workers",
  "required": [
     "_id",
     "state",
     "range",
     "lastReport",
     "substitute"
  ],
  "properties": {
    "_id": { "bsonType": "number" },
    "state": { "bsonType": "number" },
    "range": {
        "bsonType": "object",
        "required": [
             "from",
```

```
                "to"
            ],
            "properties": {
                "from": { "bsonType": "number" },
                "to": { "bsonType": "number" }
            }
        },
        "lastReport": { "bsonType": "timestamp" },
        "substitute": { "bsonType": ["null", "number"] }
    }
}
```

Figure 4.9: A document schema for sender_workers collection

Where _id means the Id of `SenderWorker`, state means the current `SenderWorker` state from the set of states defined above, range means its default and unique hash range, `lastReport` contains the timestamp when `SenderWorker` lastly reported alive state, and `substitute` contains the Id of `SenderWorker` that currently substitutes this `SenderWorker` (null if not substituted).

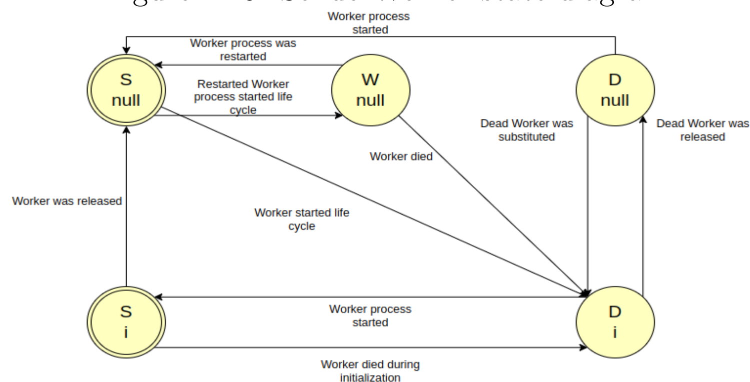
Another property Except for the default hash range, each `SenderWorker` also contains a list of hash ranges of `SenderWorkers`, whose this `SenderWorker` currently substitutes.

4.2.4.3 State diagram

The present section outlines the state diagram for `SenderWorker`, where the nodes symbolize the `SenderWorker` states. The upper character represents the state defined in the preceding section, and the lower character represents the Id of `SenderWorker` that currently substitutes this `SenderWorker`). The edges signify transitions between these states.

Figure 4.10 depicts the state diagram with all its nodes and transitions.

Figure 4.10: SenderWorker state diagram



There are two possible initial states (S, null) and (S, i). During the entire first start, each `SenderWorker` gets the initial state (S, null). Then after each next start, `SenderWorker` can be either in the state (S, null) if it is not substituted or (S, i) if it is substituted by `SenderWorker` i. If `SenderWorker` starts in the state (S, i), it has to wait until `SenderWorker` i will release it.

After this step, `SenderWorker` can move into the state (S, null) and then into the state (W, null). This state means that `SenderWorker` started the life cycle and

is working. From the working state, `SenderWorker` can get to either (S, null) state or (D, i) state. The first transition means that `SenderWorker` was turned off but started fast enough so that other `SenderWorkers` did not mark it as dead. The second transition means that `SenderWorker` stopped to work, and another working `SenderWorker` marked it as dead and substituted it.

If `SenderWorker` is in the (D, i) state, its process is dead and not working. But there are two different states for dead `SenderWorker`. The first is (D, i) and means that it is substituted by `SenderWorker` i. The second means it is not substituted by any other `SenderWorker`. The exact reason for having these two different states will be described in the next section. Still, the main reason is to achieve balanced substitutions among all other working `SenderWorkers`. So sometimes some `SenderWorker` has to release also dead `SenderWorkers` if it substitutes too many and some another working `SenderWorker` substitutes too few.

4.2.4.4 WorkerSynchronizer

This component is part of `SenderWorker` procedure and ensures synchronization among all other `SenderWorkers`. During initialization, `SenderWorker` starts asynchronous operation, which represents `WorkerSynchronizer`. Its primary purpose is to report the alive state and resolve dead `SenderWorkers` periodically.

The asynchronous operation starts with the period it reports its alive state. It means updating field `lastReport` in the collection `sender_workers` to the actual timestamp. Then it waits for SYNCHRONIZING_PERIOD of milliseconds.

After waiting for defined milliseconds, the period repeats again, and if it executes SYNCHRONIZING round (each fifth round), `WorkerSynchronizer` starts resolving dead `SenderWorkers`. If some `SenderWorker` is in the DEAD state, it means that its `lastReport` was not updated for at least five rounds.

Since each `SenderWorker` takes all dead workers and tries to substitute them, it is evident that there can appear some race condition which can cause inconsistent database state in case two working `SenderWorkers` try to substitute some dead `SenderWorker`. To solve this problem, MongoDB transactions are used that support ACID properties [24]. So each query to the `sender_workers` collection has to be executed through a transaction to avoid some inconsistent database state. Suppose two workings `SenderWorkers` want to substitute some dead `SenderWorker`. In that case, they execute transactions where one will be successfully committed and the other will be aborted.

When the number of `SenderWorkers` decreases and increases depending on time, it can reach the state when some working `SenderWorker` substitutes too many dead `SenderWorkers` and some another working `SenderWorker` substitutes too few, which would mean a rapid deterioration of performance in sending e-mails. So it is also crucial to ensure balanced substitutions among all working `SenderWorkers`. To achieve this balanced state, each `SenderWorker` always has to know how many `SenderWorkers` at most it should substitute. This number is called the balance factor and is computed according to this formula:

$$BF = n/w + r - s$$

Where n is the number of non-working (SYNCHRONIZING or DEAD state) `SenderWorkers`, w is the number of working `SenderWorkers`, r is the remainder after

division, and s is the number of non-working `SenderWorkers` that are currently substituted by this `SenderWorker`. The number w cannot be zero. Otherwise, there would be no working `SenderWorkers`, and no one could compute it. If the balance factor equals 0, it means that this `SenderWorker` has balanced substitutions and does not need to change it. If the balance factor is greater than 0, it means that this `SenderWorker` has too few substitutions, and it has to substitute at most BF dead `SenderWorkers`. If the balance factor is less than 0, it means that this `SenderWorker` has too many substitutions, and it has to release exactly $|BF|$ substituted `SenderWorkers`.

During synchronization `SenderWorker` knows whether it has to substitute some dead `SenderWorkers` or release them. Since `WorkerSynchronizer` cannot release substituted `SenderWorkers` directly because it has no information whether there are some currently sending e-mails of these `SenderWorkers`, when `SenderWorker` main loop accomplishes one iteration, it continuously checks the balance factor. It releases all synchronizing `SenderWorkers` waiting to release their hash range and redundant dead `SenderWorkers`.

4.2.4.5 PlatformSolver

Since the highly available Sender cluster can run under different platforms in distributed mode (Slurm, Kubernetes, etc.), `SenderWorker` has to recognize the platform under which it is running and adequately execute the initialization and life cycle. Two environment variable names differ depending on the running platform. It is an environment variable for worker Id and an environment variable for maximum `SenderWorkers`. So this class is used by `SenderWorker` in initialization to read these environment variables appropriately.

4.2.4.6 Correctness testing

Prior to proceeding to the next component, it is imperative to explicate the methodology employed to ascertain the correctness of worker synchronization. It should be noted that the correctness of worker synchronization is pivotal in achieving the correctness of the entire Mailtrain system. Any latent bugs in this component may result in sending redundant e-mails or losing some e-mails.

The verification of the correctness of distributed systems is a complex task that cannot be solely accomplished by means of an automatic unit or end-to-end (E2E) testing. This is because such testing methodologies fail to capture the intricate interactions between multiple concurrent threads or processes, which can give rise to concurrency bugs such as race conditions or deadlocks.

To effectively test the correctness of distributed systems, it is crucial to first identify the cases in which these concurrency bugs can arise. The next step involves determining how to trigger the program state for these cases. Finally, manual testing on the triggered program state is necessary to detect any concurrency bugs that may arise. So the correctness testing was conducted manually on these significant test cases that have been specifically designed to highlight the potential concurrency issues that may arise, namely:

1. **Many `SenderWorkers` try to substitute dead `SenderWorker`** - This test case verifies that if some `SenderWorker` fails and all other working `SenderWorkers`

endeavor to substitute it, only one `SenderWorker` will do so while other transactions either fail due to write conflicts or will not even have time to start.

2. **Multiple `SenderWorkers` awaken** - This test case aims to ensure that if some dead `SenderWorkers` have woken up, they begin working either immediately or in the next synchronization round of their substitute.

3. **`SenderWorker` that substitutes other dead `SenderWorkers` has died** - This test case verifies that if `SenderWorker` (A) that currently substitutes many other dead `SenderWorkers` fails, the `SenderWorker` (B) substituting it will first cancel all substitutions of `SenderWorker` A and then substitute `SenderWorker` (A). So other working `SenderWorkers` can substitute for them.

4. **Network problem** - This test case aims to ensure that if some `SenderWorker` has a network problem and cannot communicate with the MongoDB database, then it automatically calls the initialization method again and has to pass synchronization. So after reconnecting, it starts working with initialized values.

5. **Balanced substitutions** - This test case aims to ensure that if many `SenderWorkers` substitute other dead `SenderWorkers` evenly and some dead `SenderWorkers` start to wake up, then all dead `SenderWorkers` are evenly substituted again among all working `SenderWorkers` so that all substitutions are always balanced.

6. **Deadlock detected** - This test case aims to ensure that if dead and substituted `SenderWorker` wants to start and all other `SenderWorkers` are non-working (they are either dead, synchronizing, or in the working state but with too old last report time), then it has to mark all working `SenderWorkers` as dead, remove all substitutions and start to work.

4.3 HAPUBLIC server

This section describes the last highly available component of Mailtrain, the HAPUBLIC server, created from the PUBLIC server by moving two types of requests (Files, Links) subscribers call through their received e-mail. Since these two types of requests belong to critical services of Mailtrain and require non-stop running, the HAPUBLIC server supports high availability and can ensure non-stop running.

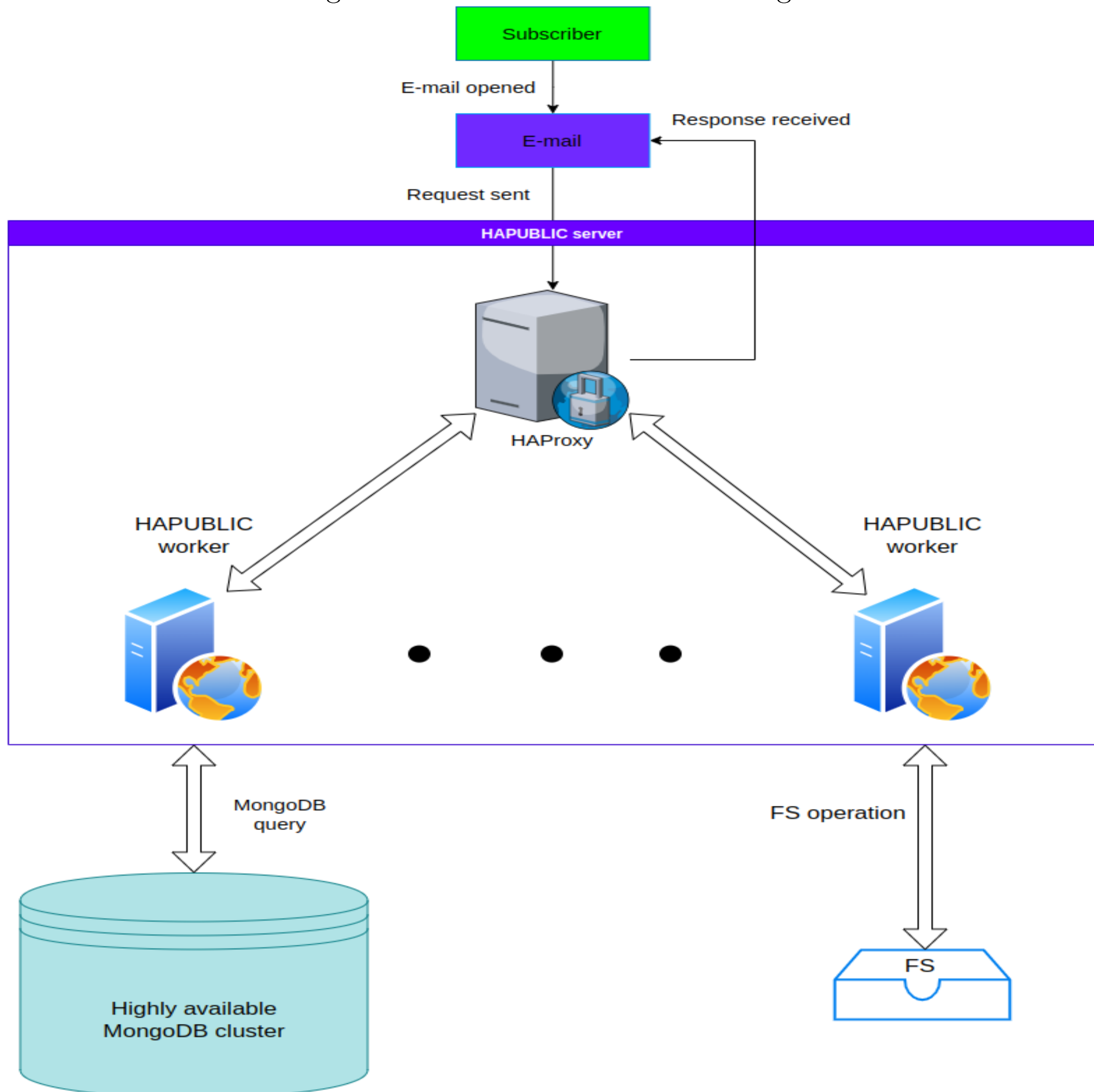
Figure 4.11 depicts the internal architecture of this component. The HAPUBLIC server consists of three parts:

1. **HAProxy** - a reliable and popular open-source load balancer and reverse-proxy server that is used to distribute network traffic across multiple servers in a high availability (HA) environment for TCP and HTTP-based applications [7].

2. **HAPUBLIC worker** - a process representing an independent web server instance of Express framework for processing Files and Links types of requests.

3. **Keepalived** - an open-source tool that is used to provide high availability for Linux-based servers by implementing the Virtual Router Redundancy Protocol (VRRP) protocol [19], it is used as a support service for HAProxy to ensure high availability.

Figure 4.11: HAPUBLIC server design



The HAPUBLIC server starts by starting the HAProxy process and forking all HAPUBLIC workers. The node where HAProxy is currently running is called the master. All subscriber requests are received by HAProxy running on this node, from where they are evenly distributed among HAPUBLIC workers. For this load balancing, HAProxy uses the Round-Robin algorithm [5], which is suitable for this purpose.

When some HAPUBLIC worker receives a request, a request is processed, and a response is sent back to HAProxy. The optimal configuration is to have at least one HAPUBLIC worker on each node. To ensure the independence of the non-highly available section, a HAPUBLIC worker communicates only with the MongoDB cluster and file system, the consequence of which is that all moved requests have

been reprogrammed so that not to query the MariaDB database. Also, all MariaDB tables used by a HAPUBLIC worker (`campaign_links`, `files_campaign_file`) had to be created in the MongoDB database, as was mentioned in Section 4.

On each node, there must also be a Keepalived process that constantly checks the master node to see whether HAProxy is still running and did not fail. In case of the master node failure when HAProxy does not run, Keepalived processes from each node will elect another node where the HAProxy process starts by leader election algorithm [23]. If a new node is elected, HAProxy starts there and becomes a new master node. This ensures the HAPUBLIC server's high availability.

4.4 Modes

It is still essential to ensure the financially accessible and straightforward maintenance of Mailtrain. For this reason, there have been created two modes of Mailtrain. First, centralized is a cheaper mode designed for customers with small data. The second mode is distributed and designed for customers with large data where vertical scalability is unsuitable. The mode is selected during system configuration before starting. The standard procedure is that a customer starts in the centralized mode with small data, and when data is growing, a customer will gradually switch to the distributed mode.

4.4.1 Centralized

The centralized mode defines, as the name suggests, Mailtrain will run on just one node. It follows that in this mode, Mailtrain cannot support high availability, and therefore all critical services do not have ensured non-stop correct running.

4.4.2 Distributed

On the other hand, the distributed mode represents the mode for which this system was designed. Mailtrain can run on more nodes and supports high availability. Therefore, all critical services have ensured non-stop correct running.

4.5 Source code overview

After completing the implementation, it is valuable to look at the source code tree compared to the source code tree of the previous version. This version is stored in `v3` Git branch.

4.5.1 The current version

The current source code tree in this section is also displayed with a brief description for each important directory or module.

```
mailtrain/
    client/ - the whole source code of Admin GUI written in React
    deployment/ - deployment scripts
```

```
docs/ - brief documentation about features and deployment
locales/ - all supported translations of Admin GUI stored in JSON
mvis/ - the source code of a visual analytics tool
server/ - the whole source code of the backend
    config/ - configuration file stored in YAML format
    files/ - directory where all file entities are stored
    lib/ - directory where auxiliary modules for other modules
           are stored
           hapublic/ - directory where the HAPUBLIC server is
                       written
           report-processor/ - directory where ReportProcessor
                              service is written
           sender/
               mail-maker/ - directory where classes for making
                             e-mails are written
               mail-sender/- directory where classes for sending
                             e-mails are written
               sender-worker/ - directory where auxiliary classes
                                for SenderWorker are written
               synchronizer/ - directory where DataCollector
                               and Scheduler is written
               sender.js - functions for forking Sender
    models/ - directory where queries for all entities
              are written, it represents the DAL component
    performance-test/ - directory where is written script for
                        parallel file requests
    protected/ - this directory serves for generated reports
    routes/ - directory where routing functions for all entity
              endpoints are written
    services/
        executor.js - source code of Executor service
        feedcheck.js - source code of Feedcheck service
        gdpr-cleanup.js - source code of GDPR Cleanup service
        importer.js - source code of Importer service
        postfix-bounce-server.js - test SMTP server source code
        report-processor.js - source code of ReportProcessor
                              service
        synchronizer.js - source code of Synchronizer service
        sender-worker.js - source code of SenderWorker service
        test-server.js - test SMTP server source code
        triggers.js - source code of Triggers service
        tzupdate.js - source code of TZUpdate service
        verp-server.js - test SMTP server source code
    setup/ - config files for MongoDB and MariaDB database system
    test/ - a couple of backend tests
    app-builder.js - functions for setting up servers
    index.js - the main module that starts the whole system
    package.json - records important metadata of the server
                   source code
shared/ - all entity enums shared between the source code of
          the client and the server
```

```
zone-mta/ - config files for running the builtin Zone-MTA feature
```

4.5.2 Deleted subtree

In this section, leaves of this source code subtree display just directories or files that have been removed compared to the previous version of Mailtrain.

For a more accurate display of all deleted files, type the command:

```
1  $ git diff --name-only --diff-filter=D v2...v3
```

```
└─ mailtrain/
   ├─ server/
   │  ├─ lib/
   │  │  ├─ mailers.js
   │  │  ├─ message-sender.js
   │  │  └─ senders.js
   │  └─ services/
   │     ├─ workers/
   │     ├─ sender-master.js
   │     └─ sender-worker.js
   └─ setup/
```

4.5.3 Updated subtree

In this section, leaves of this source code subtree display just directories or files that have been inserted or updated on a large scale (red color) or updated (blue color) on a small scale compared to the previous version of Mailtrain.

For a more accurate display of all changed files, type the command:

```
1  $ git diff --name-only  v2...v3
```

For a list of the number of changed lines, type the command:

```
1  $ git diff --numstat  v2...v3
```

For displaying all changes in one particular file, type the command:

```
1  $ git diff v2...v3 -- <file_path>
```

```
└─ mailtrain/
   ├─ client/
   ├─ deployment/
   └─ server/
      ├─ lib/
      │  ├─ hapublic/
      │  ├─ report-processor/
      │  ├─ sender/
      │  ├─ fork.js
      │  ├─ helpers.js
      │  ├─ mongodb.js
      │  └─ urls.js
      ├─ models/
      │  ├─ blacklist.js
      │  ├─ campaign.js
      │  ├─ files.js
      │  ├─ links.js
      │  ├─ lists.js
      │  ├─ queued.js
      │  ├─ send-configurations.js
      │  └─ subscriptions.js
      ├─ routes/
      │  └─ files.js
      ├─ services/
      │  ├─ synchronizer.js
      │  └─ sender-worker.js
      └─ setup/
         ├─ knex/
         │  ├─ seeds/
         │  └─ data-generator.js
         └─ mongodb/
```

4.6 Deployment requirements

In the last section, each requirement (hardware and software) is written that is necessary for the proper and smooth running of the entire system. Since there are two different supported modes, requirements differ according to chosen mode.

The exact deployment procedure for both modes is stored in `mailtrain/deployment` directory.

4.6.1 Centralized mode

For centralized mode, the deployment requirements remain the same as in the previous version, described in Section 2.8. The whole guidance for deploying the new version of Mailtrain in the centralized mode is stored in the thesis attachment A.

4.6.2 Distributed mode

In distributed mode, the situation is much more complicated. The deployment requirements remain the same as in the previous version, described in Section 2.8, and several other requirements will be added.

Additional hardware and software requirements:

- Each node where the MongoDB and HAPUBLIC server will run must be able to connect to all other nodes where all other MongoDB and HAPUBLIC server instances will run. It is important to ensure that network and security systems, including all interfaces and firewalls, allow these connections.
- Each node where the HAPUBLIC worker will run together with the non-highly available section must be able to read and write into one shared file system.
- Installed and configured Slurm Workload Manager [18] on each node. For any other platforms, the deployment procedure is not documented and has to be done by the customer manually.

It is important to note that the deployment procedure for distributed mode is much more complicated than for centralized mode and requires some manual operations.

5. Evaluation

In the final chapter, an evaluation and comparison of the two versions will be conducted with respect to their performance. The principal objective is to compare the two versions of the Sender component, the PUBLIC server with the HAPUBLIC server, and to demonstrate that the updated version is capable of horizontal scaling for large data sets, whereas the previous version is not.

This evaluation should also include all proper combinations of parameters that can influence the speed of the whole procedure of components. The Sender component includes running mode, whether worker synchronization is turned on/off, and the number of `SenderWorkers`. It would also be valuable to demonstrate the evenness of the distribution of messages among `SenderWorkers` for a significant number of `SenderWorkers`.

The PUBLIC server includes the number of sent parallel requests, the size of the requested file, and in the HAPUBLIC server, there is also the number of parallel running servers.

For all of these combinations of parameters is valuable to execute performance tests on some reasonable test data in some distributed system with the setup using full performance and compare it with each other.

5.1 Test data

It is crucial to ensure that testing data will be large enough to reach a state where the different behavior of both versions will be clearly visible. It means that the vertical scaling used in the old version does not bring any acceleration, it even slows down, but horizontal scaling used in the new version does.

For the Sender component, there will be used one big campaign with 1000000 subscribers. Since the number of sending campaigns does not significantly affect the speed of sending procedure, only the sum of subscribers, it is not vital to have more than one campaign for performance testing.

For the PUBLIC and HAPUBLIC servers, there will be used one 1MB image file that represents an image with a standard size used in production.

5.2 Test environment

As a test environment for evaluation of the old and new versions, there is used `parlab` cluster maintained by MFF HPC metacenter.

Here is included a brief specification of the cluster:

The cluster contains one head node and eight worker nodes, and on each worker node are two sockets of Intel Xeon Gold 6130 CPU with 16 cores. So each worker node contains 32 CPU cores, and the whole cluster contains 256 CPU cores. The cluster contains 128GB RAM. All nodes are interconnected by InfiniBand FDR (56 Gb/s) for high-performance messaging using MPI. Moreover, they are interconnected by 10 GbE for all other traffic. The front-end server is connected by 10 GbE to the external world.

All nodes share one disk array connected to the head node with 2x FC16. The head node is connected to worker nodes with 3x 10GE. This disk array uses auto-tiering, where on tier 0, there are SSDs with a capacity of 1.1TB, and on tier 3, there are HDDs with a capacity of 260TB.

One of these worker nodes is used for running the non-highly available section of Mailtrain, and it is also used for running one MongoDB server. All other worker nodes, including the previous one, are used for running the Sender cluster and HAPUBLIC server.

Since there is no affordable SMTP server with the same performance as the cluster, there is no SMTP server used in the whole testing. When `SenderWorker` calls the function to the SMTP server, only a constant successful response is returned instead of the calling function.

5.3 The old version

Firstly, it is essential to analyze the old version of Mailtrain. It is crucial to set up the cluster so that it uses the maximum number of resources for one node. Therefore both comparisons will bring the best possible performance times with regard to the limits of the cluster.

The PUBLIC server will be tested together with the HAPUBLIC server in the last section.

5.3.1 Sender

The first measurement graph will be described for the Sender component. The x-axis indicates the number of `SenderWorkers`, and the y-axis indicates the total amount of time measured in seconds needed for sending the whole campaign.

Each measured time represents the exact time, just like in Chapter 3. The exact time consists of initialization time and sending time. The initialization in the old version defines the time between points when the start button is clicked until `SenderMaster` prepares the campaign for sending. The sending time in the old version defines the time between points when `SenderWorkers` start making and sending e-mails until the campaign is sent.

The initialization time was always 50 seconds for each test, and the rest belonged to sending time, hence the work of `SenderWorkers`.

Figure 5.1: Old Sender performance analysis

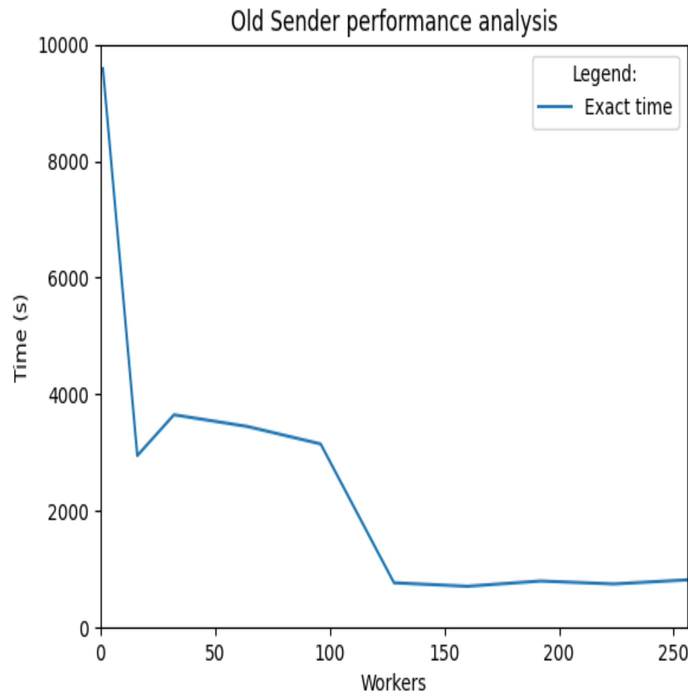


Figure 5.1 depicts the measurement graph of time for sending the campaign with one million e-mails.

The best run for this version of the Sender component was for 160 number of `SenderWorkers` with a duration of 710 seconds.

5.4 The new version

In the new version of Mailtrain, it is essential to analyze both modes and for the distributed mode to analyze cases when worker synchronization is turned on/off. This will show not only overall performance times for both modes but it will also respond to questions such as what a significant acceleration will only use the new algorithm without adding new nodes bring, what a significant acceleration will adding more nodes bring, and what percentage of the time takes worker synchronization.

5.4.1 Centralized Sender

The graph definition remains the same as in the previous measurement. The exact time definition now has new points definition according to the new version of Sender. The initialization defines the time between points when the start button is clicked until `Synchronizer` prepares the campaign for sending and sends all needed data to the MongoDB database. The sending time defines the time between points when `SenderWorkers` start making and sending e-mails until the campaign is sent but not fully synchronized with the MariaDB database.

The initialization time was always 110 seconds since now all campaign messages had to be first created in the MariaDB database and then synchronized with

the MongoDB database. The rest belonged to sending time, hence the work of `SenderWorkers`.

Figure 5.2: Centralized Sender performance analysis

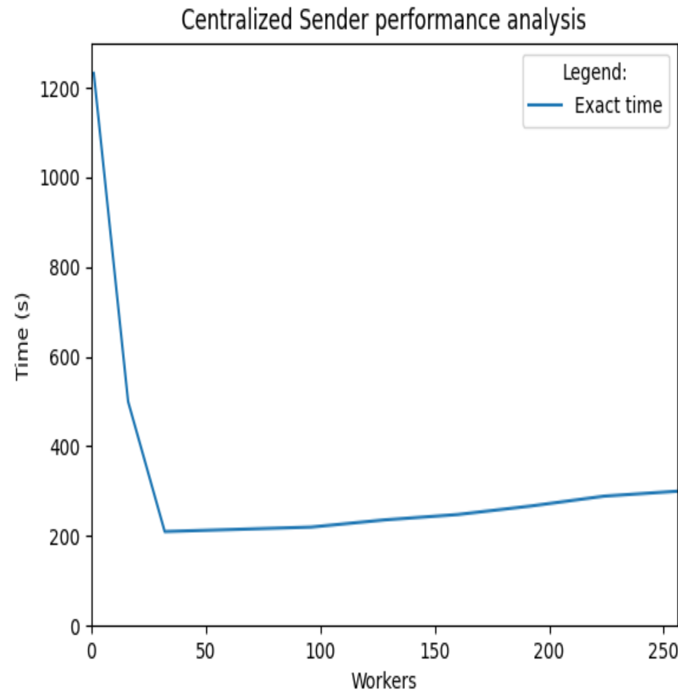


Figure 5.2 depicts the measurement graph of time for sending the campaign with one million e-mails in the centralized mode.

The minimum time for this version of the Sender component was for 32 number of `SenderWorkers` with a duration of 210 seconds which represents approximately 3.4x acceleration compared to the old version of Mailtrain. If it is considered only the sending time, it represents 6.6x acceleration. It is also evident that the new version uses resources of the one node much more efficiently since the exact times are much shorter. Also, after exceeding the limit of the number of CPU cores for each `SenderWorker`, the exact times are only increasing.

5.4.2 Distributed Sender

Now, it is time to analyze performance for the distributed mode where all cluster worker nodes are used with maximum use of resources. The first measurement will be executed with unsynchronized `SenderWorkers`. Then the second measurement with turned-on synchronization will be performed, resulting in a total increase in time needed for worker synchronization.

5.4.2.1 Unsynchronized workers

Just like in the previous measurement of the centralized mode, the graph definition and the exact time definition remain the same.

Figure 5.3: Distributed unsynchronized Sender performance analysis

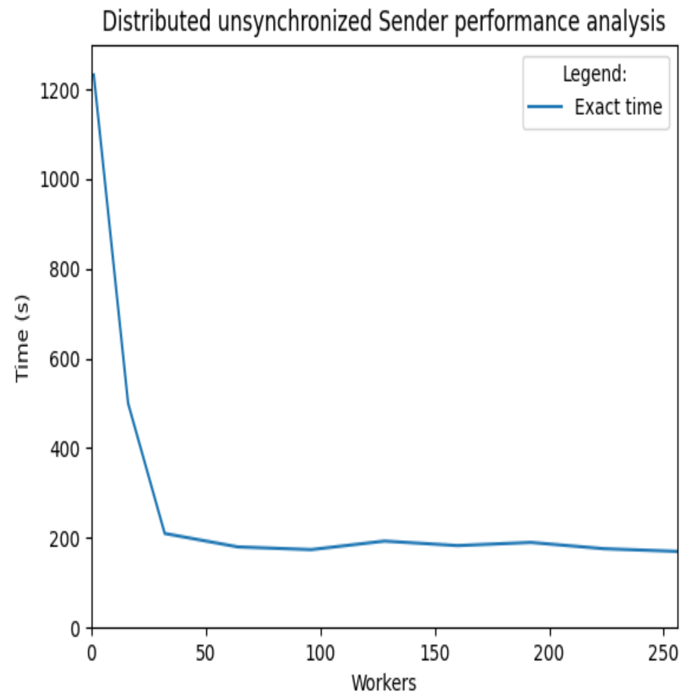


Figure 5.3 depicts the measurement graph of time for sending the campaign with one million e-mails in the distributed mode of unsynchronized `SenderWorker`.

It is evident from the graph that increasing the number of `SenderWorkers` does not decrease the exact time so much. The main reason for this is that the shared disk array reached the maximum bandwidth, and adding more nodes with other CPU cores cannot bring more significant acceleration.

The minimum time for this version of the Sender component was for 256 number of `SenderWorkers` with a duration of 170 seconds which represents approximately 4.2x acceleration compared to the old version of Mailtrain. If it is considered only the sending time, it represents 11x acceleration. This acceleration is the maximum of all measurements. Compared to the measurement of the centralized mode, the acceleration for the exact time is approximately 1.23x, and only for the sending time, it is approximately 1.7x. It is valuable to mention that, in this case, the minimum time is measured in the maximum number of workers so increasing the number of `SenderWorkers` does not increase the exact time for a significant number of `SenderWorkers` as in the centralized mode measurement.

5.4.2.2 Synchronized workers

So far, worker synchronization was always turned off. Now it is time to compare the previous Figure 5.3 with the example where worker synchronization is turned on and shows what kind of system slowdown it will bring.

Figure 5.4: Distributed synchronized Sender performance analysis

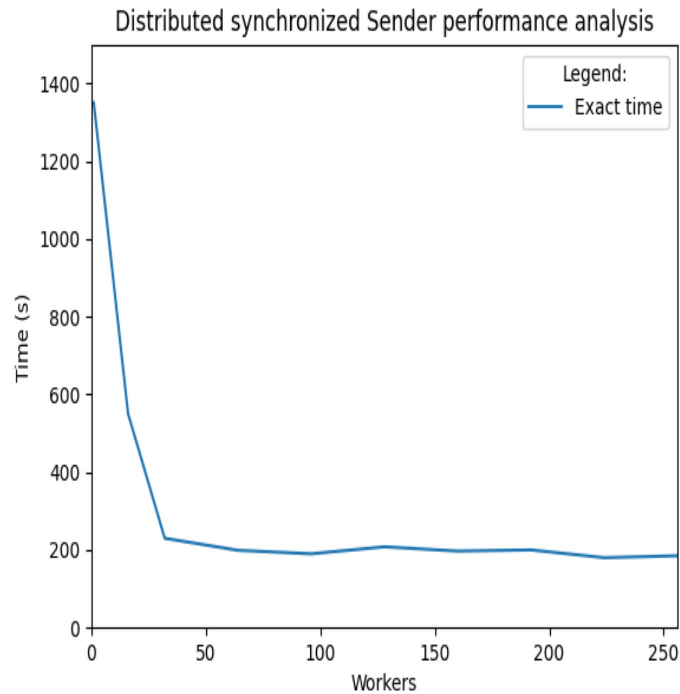
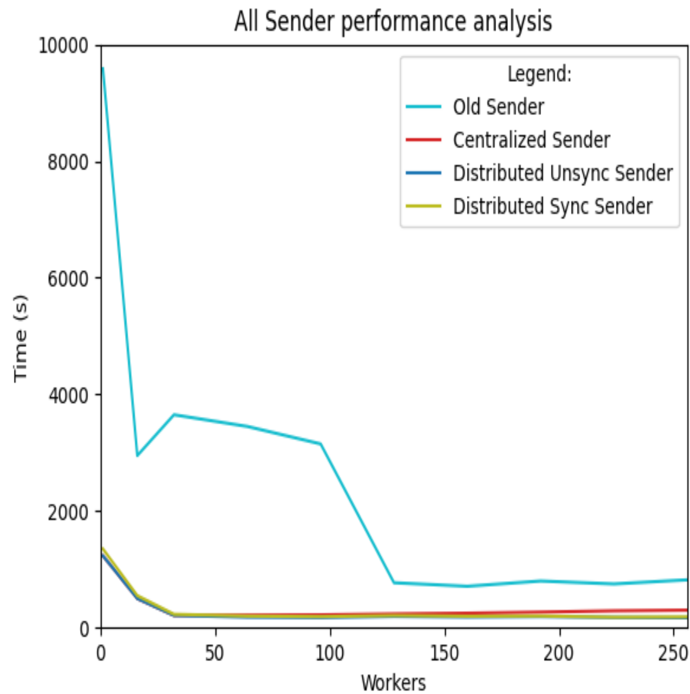


Figure 5.4 shows the effect of worker synchronization. The times did not change so rapidly, and the minimum changed for 256 workers from the value 170 to 185. It looks like for each number of workers, the sending time increased by approximately 10 - 20%.

5.5 Comparison of all variants

In this Section, there is presented a comparison of all four curves measured above, denoted as old Sender, centralized Sender, distributed unsynchronized Sender, and distributed synchronized Sender.

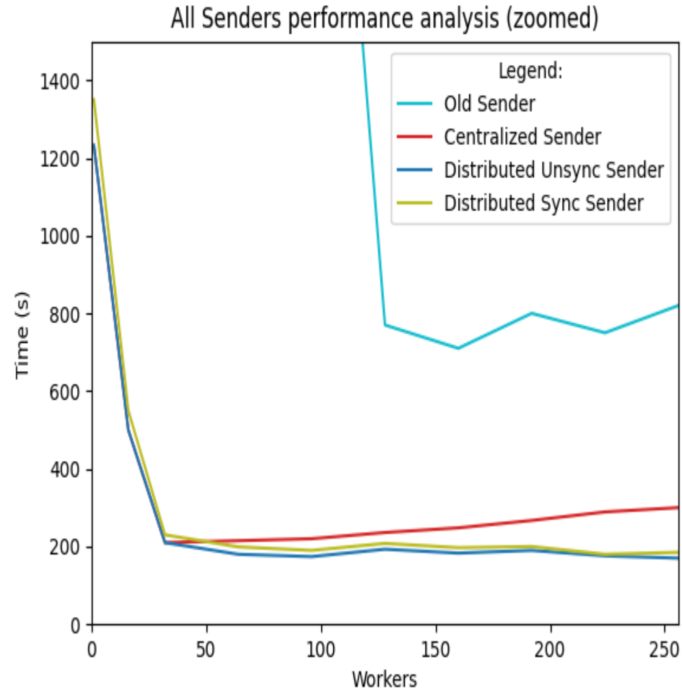
Figure 5.5: All Senders performance analysis



As depicted in Figure 5.5, the exact times of the new version curves exhibit considerable variability in comparison to the old version curve. The optimal measured times for each variant, ranked in ascending order, are as follows:

- **Distributed Unsynchronized Sender** - `SenderWorkers`: 256, the exact time: 170 (the initialization time: 110, the sending time: 60)
- **Distributed Synchronized Sender** - `SenderWorkers`: 256, the exact time: 185 (the initialization time: 110, the sending time: 75)
- **Centralized Sender** - `SenderWorkers`: 32, the exact time: 210 (the initialization time: 110, the sending time: 100)
- **Old Sender** - `SenderWorkers`: 160, the exact time: 710 (the initialization time: 50, the sending time: 660)

Figure 5.6: All Senders performance analysis (zoomed)

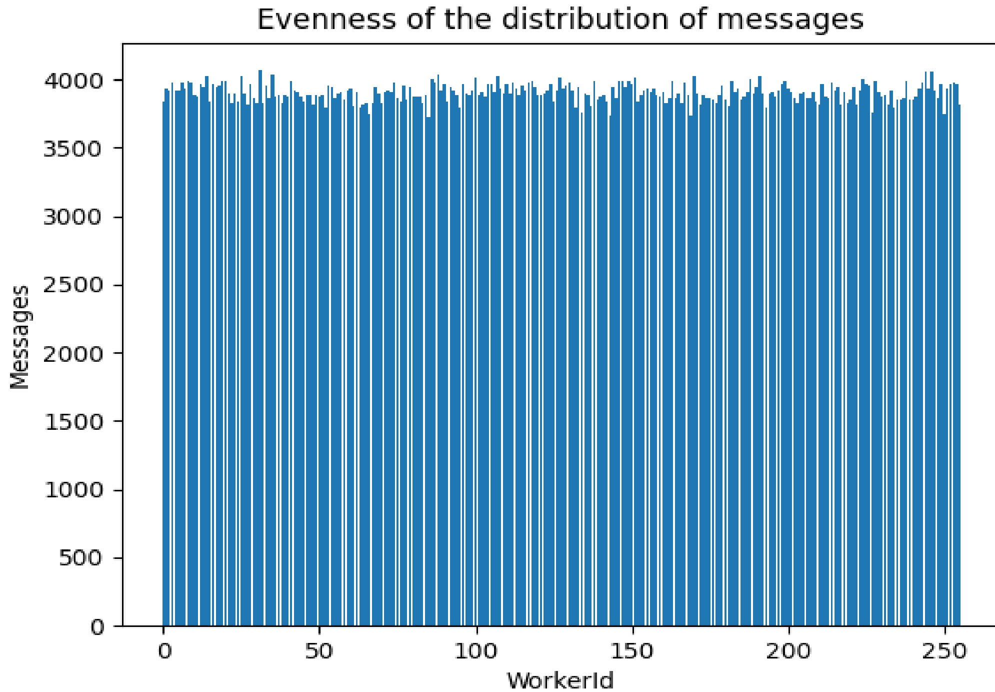


To be able to see the differences between the three curves from the new version, there is made zoomed Figure 5.6 of the previous one that depicts these differences primarily.

5.5.1 Evenness of the distribution of messages

As the last analysis in this section, it would be valuable to show measurement from the evenness of the distribution of messages among 256 `SenderWorkers`. Figure 5.7 depicts the exact number of messages assigned to each `SenderWorker`.

Figure 5.7: Evenness of the distribution of messages



The fewest messages were assigned to `SenderWorker` with Id 85 and value 3728. On the other hand, most messages were assigned to `SenderWorker` with Id 31 and value 4070. This measurement graph clearly proves that the hash algorithm SHA-512 used in the new version of the Sender component distributes messages evenly with only slight deviation.

5.5.2 HAPUBLIC server

This section will focus only on measuring request processing times for PUBLIC and HAPUBLIC servers. The PUBLIC server is used from the old version of Mailtrain, and the HAPUBLIC from the new version. Processing one request contains reading a file path from the database according to URL arguments and then sending a requested file.

It is helpful to note that although the PUBLIC server runs only on one node as a single thread process, the processing alone is parallel because all requests are processed by the web server asynchronously, and I/O operations are executed in Node.js in parallel by default. Because of that, it is not necessary to run more than one web server on one node for the HAPUBLIC server.

Since the architecture of the using cluster has a shared disk array among all nodes, measured times of processed requests for both servers are not significantly different. For that reason, there will be executed two different performance analysis. The first performance analysis is simulated where a request is not really executed, but the server will wait one second and then will send a successful and constant response of negligible size. The second performance analysis is accurate, where a request is really processed, and a requested file is sent as a response. The reason for making these two different versions is that it should depict the HAPUBLIC

server as horizontally scalable, and the measured times are not ideal only due to the cluster architecture.

5.5.2.1 Simulated version

The first version of measuring the performance of the PUBLIC and HAPUBLIC servers is simulated. There are sent 1000 parallel requests to the PUBLIC and HAPUBLIC servers. The x-axis indicates the number of workers or nodes (since one worker runs on each node) used for the HAPUBLIC server, and the y-axis indicates the total amount of time measured in seconds needed for processing all requests.

Figure 5.8: (HA)PUBLIC servers simulated performance analysis

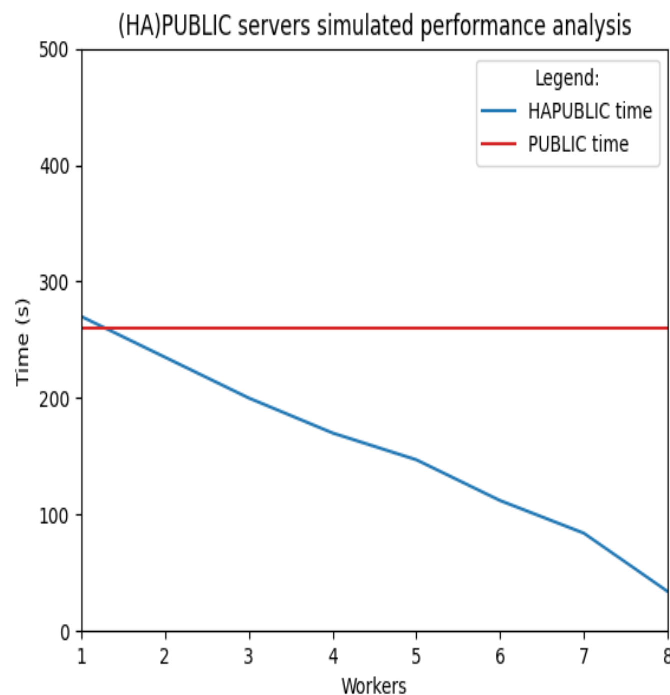


Figure 5.8 depicts that increasing the number of workers for the HAPUBLIC server decreases the processing time. The processing time remains the same for the PUBLIC server since it runs only on one node with one worker. So this measurement clearly proves that the HAPUBLIC server can scale horizontally compared with the PUBLIC server used in the old version of Mailtrain.

5.5.2.2 Real version

The second version of measuring the performance of the PUBLIC and HAPUBLIC servers is real. The testing parameters remain the same. There are sent 1000 parallel requests to the PUBLIC and HAPUBLIC servers. The x-axis indicates the number of workers or nodes (since one worker runs on each node)) used for the HAPUBLIC server, and the y-axis indicates the total amount of time measured in seconds needed for processing all requests.

Figure 5.9: (HA)PUBLIC servers real performance analysis

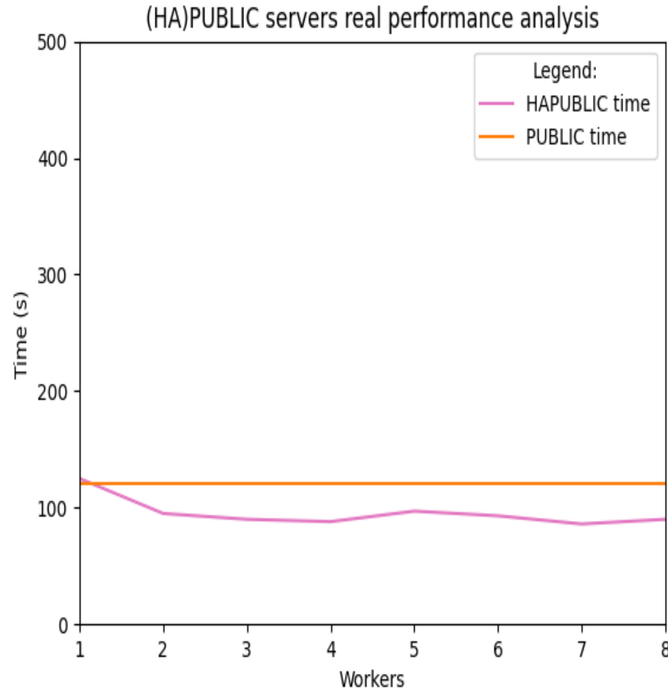


Figure 5.9 depicts that increasing the number of workers for the HAPUBLIC server does not change the processing time since the reading from the disk array is a significant bottleneck.

5.6 Evaluation summary

After executing all performance tests and measuring times, it is partially shown that the new version of Mailtrain can scale horizontally for sending extensive campaigns and processing file requests in the HAPUBLIC server. Although measured times are not ideal, the new version brought rapid acceleration. It is essential to consider in these types of systems that for achieving ideal times, it would be necessary having all system components that represent bottlenecks with the same and high performance.

In this thesis, the main system components that cause bottlenecks are the SMTP server, the disk array where the MongoDB database is stored, and the cluster where `SenderWorkers` and HAPUBLIC server workers run as was said in Chapter 3. Practically it is almost impossible to meet these requirements, especially having all these components with the same performance, but the ideal cluster architecture where Mailtrain should achieve the best performance is Big Data architecture [11].

It means meeting all these requirements:

- Each node must have its own non-shared disk or own disk array with the best possible bandwidth.
- Each node must have an appropriate number of CPUs and the operation memory to be able to run as many `SenderWorkers` as possible.

- Each node in the system necessitates an independent MongoDB server that is configured for replication and sharding based on the e-mail hash. This implies that `SenderWorkers` executing on a given node solely send e-mails that are stored on that particular node.
- Having the SMTP server with the same performance as the rest of the system.

6. Conclusion

The main goals of the thesis were to implement a new architecture for Mailtrain that can scale horizontally on large data, supports high availability of all critical services, contains safe shutdown of the system, and has more readable and upgradeable source code of the Sender component. All of these goals have been accomplished and proved in the Evaluation chapter.

One of the biggest successes of this thesis is that the current version solves problems beyond the original scope. The original scope was only to make Mailtrain horizontally scalable for sending campaigns that are the most used in production (regular and RSS) and ensure high availability for critical services. The current version supports horizontal scalability not only for each type of campaign but also for each type of queued message and all critical services. The same applies to high availability. This solution made the Sender source code more readable and upgradeable since the new Sender supports all types of messages, and therefore the old Sender could be totally removed.

Further, the current version supports two modes (centralized and distributed) and thus is accessible to customers with less financial budget and more minor data. It is also satisfying to demanding customers that send many campaigns simultaneously and need to run Mailtrain in some high-performance distributed system.

6.1 Future work

The current version could be improved and extended in many ways. After analysis of the Sender component, it is evident that some are not optimal operations. This thesis mainly focuses on campaigns that are most frequently used in production (regular and RSS), as mentioned above. The consequence is that sending queued messages is not implemented with a big emphasis on optimality. For triggered campaigns, there are duplicated data for each message in MongoDB `queued` collection. Also, handling and rescheduling unsuccessfully sent queued messages are done more complicatedly. The best solution is to entirely change sending triggered campaigns as campaign messages, not as queued ones, and optimize rescheduling other types of queued messages. It would also be helpful to add another status to a campaign that would distinguish between states when a campaign is only sent and when all campaign messages are synchronized with the MariaDB database.

The next thing the current version should improve is removing the PUBLIC server and moving all requests to the new HAPUBLIC server. Since there is no need to have two different types of servers with requests used for the same purpose, the source code will be more understandable and extensible.

If a customer wants to deploy Mailtrain in the distributed mode, it can be run only under the Slurm platform. There are much more popular platforms for running distributed systems, such as Amazon EC2 Container Service, Google Kubernetes Engine, Azure Kubernetes Service, etc. It would be helpful to make Mailtrain support also these popular platforms. Support for these platforms includes creating deployment scripts and testing actual deployments or some little

change in `PlatformSolver` if needed.

Another service that could be expanded with other options is an SMTP server. There are only three types of SMTP servers that the current version supports. It is a centralized and serial SMTP server, Zone-MTA, and AWS SES. There are some other popular services for sending e-mails, such as Elastic Email, Mailjet, etc. Starting to support them would also improve the popularity of Mailtrain.

Bibliography

[1] *AWS SES Cloud Email Service*. URL: https://aws.amazon.com/ses/.

[2] Eve; Banks Alex; Porcello. *Learning React*. O'Reilly Media, 2017. ISBN: 978-1491954621.

[3] *Comparing The Differences - MongoDB Vs MariaDB | MongoDB*. URL: https://www.mongodb.com/compare/mariadb-vs-mongodb.

[4] *Comparing The Differences - MongoDB Vs MySQL | MongoDB*. URL: https://www.mongodb.com/compare/mongodb-mysql?fbclid=IwAR2h0H2H6B0Vc4izYK52TDjb0j_BdpqBpfs-qtWfPmMKdisiG9i4spr7Rew.

[5] Mustafa Elgili. "Load Balancing Algorithms Round-Robin (RR), Least-Connection and Least Loaded Efficiency". In: (Oct. 2017).

[6] *Express - Node.js web application framework*. URL: https://expressjs.com/.

[7] *HAProxy technologies*. URL: https://www.haproxy.com/.

[8] Guy Harrison and Michael Harrison. *MongoDB performance tuning: Optimizing MongoDB databases and their applications*. 1st ed. Berlin, Germany: APress, 2021. ISBN: 978-1484268780.

[9] Yousef Al-houmaily and George Samaras. "Two-Phase Commit". In: (Jan. 2009), pp. 3204–3209. DOI: 10.1007/978-1-4899-7993-3_713-2.

[10] Thomas Hunter and Bryan English. *Multithreaded Javascript: Concurrency beyond the Event Loop*. O'Reilly Media, Inc, USA, 2021. ISBN: 978-1098104436.

[11] W.H. Inmon, Daniel Linstedt, and Mary Levins. "Chapter 4.2 - What Is Big Data?" In: *Data Architecture (Second Edition)*. Ed. by W.H. Inmon, Daniel Linstedt, and Mary Levins. Second Edition. Academic Press, 2019, pp. 73–80. ISBN: 978-0-12-816916-2. DOI: https://doi.org/10.1016/B978-0-12-816916-2.00011-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128169162000115.

[12] Hui Lei. "TCC: State of the Transactions". In: *IEEE Transactions on Cloud Computing* 7 (Jan. 2019), pp. 1–4. DOI: 10.1109/TCC.2019.2892015.

[13] Greg; Lim. *Beginning node.Js, express MongoDB development*. Independently Published, 2019. ISBN: 978-1078379557.

[14] Shaimaa Khalifa Mahmoud et al. "A comparative analysis of Cross Site Scripting (XSS) detecting and defensive techniques". In: *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*. 2017, pp. 36–42. DOI: 10.1109/INTELCIS.2017.8260024.

[15] *MariaDB MaxScale*. URL: https://mariadb.com/kb/en/maxscale/.

[16] John O'Gorman. *The Linux process manager - the internals of scheduling, interrupts and signals: The internals of scheduling, interrupts and signals*. en. Chichester, England: John Wiley & Sons, 2003. ISBN: 978-0470847718.

[17] John; Rhoton. *Programmer's Guide to Internet Mail: SMTP, POP, IMAP, and LDAP*. Digital Press, 1999. ISBN: 978-1555582128.

[18] *Slurm Workload Manager*. URL: https://slurm.schedmd.com/.

[19] Joseph Soon et al. "Implementing of Virtual Router Redundancy Protocol in a Private University". In: *Journal of Industrial and Intelligent Information* 1 (Jan. 2013), pp. 255–259. DOI: 10.12720/jiii.1.4.255-259.

[20] *Spider Storage Engine Overview*. URL: https://mariadb.com/kb/en/spider-storage-engine-overview/.

[21] Meiliana Sumagita and Imam Riadi. "Analysis of Secure Hash Algorithm (SHA) 512 for Encryption Process on Web Based Application". In: 7 (Sept. 2018), pp. 373–381.

[22] Muhammad Syafrudin et al. "Chapter 5 - An Open Source-Based Real-Time Data Processing Architecture Framework for Manufacturing Sustainability". In: *Sustainability* 9.11 (2017). ISSN: 2071-1050. DOI: 10.3390/su9112139. URL: https://www.mdpi.com/2071-1050/9/11/2139.

[23] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007. ISBN: 978-0-13-239227-3.

[24] *What are ACID Properties in Database Management Systems?* URL: https://www.mongodb.com/basics/acid-transactions.

[25] *ZoneMTA (internal code name X-699)*. URL: https://github.com/zone-eu/zone-mta.

List of Figures

2.1 Database schema 12
2.2 Mailtrain architecture 15
2.3 Sender architecture 19
2.4 SenderMaster activity diagram 20
2.5 E-mail sending procedure 21
2.6 PUBLIC server 23
2.7 Sending e-mails use case diagram 24

3.1 Sender performance analysis 29

4.1 Distributed Mailtrain design 34
4.2 Distributed Sender design 40
4.4 Synchronizer activity diagram 41
4.5 Campaign statuses 42
4.6 Scheduler activity diagram 43
4.7 DataCollector activity diagram 44
4.8 SenderWorker activity diagram 45
4.10 SenderWorker state diagram 47
4.11 HAPUBLIC server design 51

5.1 Old Sender performance analysis 59
5.2 Centralized Sender performance analysis 60
5.3 Distributed unsynchronized Sender performance analysis 61
5.4 Distributed synchronized Sender performance analysis 62
5.5 All Senders performance analysis 63
5.6 All Senders performance analysis (zoomed) 64
5.7 Evenness of the distribution of messages 65
5.8 (HA)PUBLIC servers simulated performance analysis 66
5.9 (HA)PUBLIC servers real performance analysis 67

List of Abbreviations

ACID - Atomicity, Consistency, Isolation, and Durability
API - Application Programming Interface
AWS - Amazon Web Services
AWS SES - Amazon Web Services Simple Email Service
CPU - Central Processing Unit
CSV - Comma Separated Values
CTA - Call To Action
DKIM - DomainKeys Identified Mail
GDPR - General Data Protection Regulation
GUI - Graphical User Interface
HA - Highly Available
HTML - Hypertext Markup Language
HTTP - Hypertext Transfer Protocol
HTTPS - Hypertext Transfer Protocol Secure
IP - Internet Protocol
JSON - JavaScript Object Notation
MJML - Mailjet Markup Language
MTA - Message Transfer Agent
RAM - Random Access Memory
RSS - Really Simple Syndication
SMTP - Simple Mail Transfer Protocol
SQL - Structured Query Language
TCC - Try, Confirm, Cancel
TZ - Time Zone
URL - Uniform Resource Locator
UTC - Coordinated Universal Time
VRRP - Virtual Router Redundancy Protocol
WYSIWYG - What You See Is What You Get
XSS - Cross Site Scripting
YAML - Yet Another Markup Language

A. Attachment

Enclosed herewith is a USB flash drive that contains the necessary components for deploying the new version of Mailtrain on a virtual machine operating on Ubuntu 18.04. The deployment will be facilitated by Oracle VM VirtualBox, a publicly accessible virtualization tool offered at no cost. The USB drive includes all needed components, such as the virtual machine, source code, and instructional video outlining all steps involved in the deployment procedure, including setting up the virtual machine, running Mailtrain, and generating randomized data.

The centralized mode is used to build Mailtrain, and the generated data is intended for testing purposes.

All file descriptions:

1. **guidance.mp4** - the whole guidance step-by-step in MP4 video format for setting up the virtual machine, running Mailtrain, and generating random data.
2. **mailtrain/** - Git repository of Mailtrain source code, including all versions. The old version of Mailtrain is stored in branch `v2`, and the new version of Mailtrain is stored in branch `v3`.
3. **ubuntu-18.04.6-desktop-amd64.iso** - ISO image of Ubuntu 18.04 virtual machine.