Towards a Better Understanding of Internet Protocol Standardization

An Analysis of the IETF Email Archives

Cezary Radoslaw Jaskula

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2021

Towards a Better Understanding of Internet Protocol Standardization

An Analysis of the IETF Email Archives

Cezary Radoslaw Jaskula

Towards a Better Understanding of Internet Protocol Standardization

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Internet standards are vital for ensuring that the modern global network functions smoothly and efficiently. Yet, the process that leads to the creation of these standards is not frequently undergoing a systematic analysis. The high-level goal of this project is to develop methods and tools that will allow us to analyze the arguments and decision-making contained within the Internet Engineering Task Force email archives. By doing this we hope to gain a better understanding of the design process underlying both modern and old internet protocols.

In order to accomplish this, the IETF email archives were parsed from their raw state and ingested into a customized, semi-structured, full-text database building on the open-source Apache Solr framework. This made it possible to define and calculate various statistics that are showcased in the thesis. It also allowed for an attempt at conversation tracking to be made. The program used to track threads was developed from scratch, and has proved itself effective at tracking conversations based on the email headers.

While the results from this endeavor are promising, they only constitute a first step towards the automatic detection of arguments. The next step, the textual analysis of the actual email content, can now be taken using this database without having to spend time on error detection and correction, email header parsing, and so forth.

Acknowledgements

I would like to acknowledge my supervisors, Professor Michael Welzl and Stephan Oepen for their guidance, feedback, and patience. Throughout the entire process they have given good advice, constructive critique and their honest opinions about the project. This has helped greatly in shaping the final product. I want to thank them for making it an enjoyable experience through their sense of humour and genuine interest in the project.

I would like to say a special thank you to Mikael Nielsen Røykenes and Helge Gjølberg Aschem for taking the time to proofread my thesis. I also thank them, and Thomas Kraugerud, for their continued friendship and support.

Finally, I would like to express my sincere gratitude to my mother for her unconditional encouragement and support. Without her, none of this would have been possible.

Cezary Radoslaw Jaskula
Oslo, May, 2021

Contents

Chapter 1

Introduction

Internet standards are vital for ensuring that the modern global network functions smoothly and efficiently. Yet, the process that leads to the creation of these standards is not frequently undergoing a systematic analysis.

The organization responsible for the development of these standards is called the Internet Engineering Task Force, or just IETF. This organization consists of volunteers of various backgrounds working together to develop new standards, and updating outdated ones. The work done by the organization's participants is mostly done by the use of mailing lists. It is in these mailing lists that various topics are discussed, proposed and critiqued.

The correspondence sent to these mailing lists is automatically archived by the IETF. These archives are publicly available for anyone to see and study as they see fit. The IETF even has tools in place that make traversing the archives a more pleasant experience; the IETF datatracker is one of those tools.

This approach has some downsides, as the archiving of messages itself is not perfect. It is not uncommon to find messages that are malformed, or are missing parts. Spam is also an issue that makes reading the archives a challenging task, if done by hand. Another downside is the lack of any form of conversation tracking. While mailing lists are created with the intention of discussing a specific topic, there are no mechanisms in place to prevent users from discussing other things, and then there is the issue of underlying topics. Spam is also a problem, and has found its way into many, if not all, of the mailing lists in the archives. All these issues combined make it very difficult for a regular person to make out the arguments and decision making that goes into the creation of any specific standard. This means that only a handful of people who have actively participated in the creation of a specific standard have the knowledge of which arguments were used for and against the creation of those standards. This means that reasons for why internet standards are they way they are, is knowledge not easily available to the general public, leading to a situations where users and developers abide by standards, without understanding why there standards are they way they are.

The high-level goal of this project is to develop methods and tools that will allow us to analyze the arguments and decision-making contained within the Internet Engineering Task Force email archives. By doing this we hope to gain a better understanding of the design process underlying both modern and old

internet protocols.

This thesis is divided into nine chapters, the first one being the introduction. Chapter two presents more detailed background information about how the IETF works, how the email messages are archived, and the challenges associated with the archival format.

Chapter three is about transforming the raw data from the email archvies into a well defined format that can later be used to extract information, such as statistics, and conversation threads contained within the archives.

Chapter four describes the actual process of parsing the email archives into the format described in chapter three. It describes the tools and methods used, as well as some of the challenges encountered under the developments of the parsing tools.

Chapter five shows how the database program "Solr" can be used, both by code and by hand, in order to acquire new information from the transformed email archives.

Chapter six showcases some of the statistics that can be calculated with the Solr program. It also discusses the quality of the data at hand, as well as the improvements made to said data.

Chapter seven is about tracking conversation threads. It describes the steps that were taken in order to extract data needed for the tracking process, as well as the tracking process inself, and its results. An attempt at tracking conversations containing the "Last call" key phrase is made in this chapter, and statistics regarding the results returned by the tracking program are presented as well.

Chapter eight is dedicated to connecting the authors of RFCs to their emails in the email archives. This chapter contains discussion regarding the extraction of authors' names, email addresses, as well as the working groups their RFCs belong to.

This thesis sometimes refers to a GitHub repository containing the source code for programs used in various instances, this repository can be accessed at the following link:

`https://github.com/CezaryRJ/Master_code`

Chapter 2

Background: the IETF

IETF stands for Internet Engineering Task Force, and is an organization composed of network designers, vendors, scientists and operators concerned with evolving the internet architecture, as well as ensuring smooth operation of the internet. The participants are organized in "Working groups". These groups are organized based on research topics in specific areas. The work is mainly done by the use of mailing lists, as well as organized meetings three times a year. Working groups are grouped into areas, and managed by Area Directors, or "AD"-s. ADs are members of the "Internet Engineering Steering Group" (IESG).

2.1 RFCs

RFC stands for "Request for comments" and is a type of publication used by the IETF containing documents, comments and discussions concerning internet protocols, procedures, programs, concepts and more. The IETF has put in place a search engine for easy access to these publications which can be found at

`https://www.rfc-editor.org/search/rfc_search.php`

Alternatively, the entire RFC stream can be found at the following link:

`https://www.rfc-editor.org/search/rfc_search_detail.php?stream_name=IETF&page=All`

An RFC always begins its life as an "internet draft. These can be submitted by anyone and there are 2 ways of doing so:
The first one is to use the IETFs I-D Submission Tool, or IDST for short, to submit the internet draft.
The second is to simply send the Internet draft to the following email address:

`internet-drafts@ietf.org`

The IETF however recommends using the first method mentioned as manually validating an internet draft sent in by mail takes significantly more time

than processing it automatically. The drafts are however not reviewed in any way, as long as they fulfill the submission rules.

2.2 The lifecycle of an internet draft

When an internet draft is submitted, it is first checked to see if it fulfills the submission rules, and is accepted if it does. At this point the internet draft will get a title that follows the following format:

Draft-lastname-intendedworkinggroupname-nameofidea-00

The last two numbers represent the revision number, meaning that if a new internet draft is submitted, and goes as far as being revised, its new title will end in 01 and so on. All internet drafts are therefore considered to be a work in progress, and must be cited as so. Citing an internet draft is not always a good idea, as all internet drafts older than 185 days are removed. This is not guaranteed to happen, we can still find internet drafts from over 20 years ago with a simple google search, there is however no guarantee that any specific internet draft will survive.

For an internet draft to become an RFC, a working group has to decide to work on it. If this happens the internet drafts title is again changed, this time the indendedworkinggroupname is changed to the "ietf-" and the actual name of the group that will be working with this internet draft, the author's name is also replaced, meaning it then looks like the following:

draft-ietf-workinggroupname-nameofidea-00

The last two digits form a number that is incremented as described previously, but the counter is reset to zero whenever the name changes (i.e., also at working group acceptance). Once a working group has decided to focus on a specific internet draft it is highly likely that it will end up as an RFC. RFCs in contrast to internet drafts do not change nor expire, meaning that we can easily find any RFC ever published by the IETF on their website.

As previously mentioned, work in the IETF is mostly done in working groups. These groups focus is mostly concerned with the development of RFC's, and by extension, new internett standards. There are however other groups that focus more on the administrative side of the IETF. Sutch groups provide administrative support, and in general make sure that everything runs smoothly. Following is a list of some of these groups.

2.3 Groups and committees

2.3.1 Internet engineering steering group

As the name implies, this group is in charge of steering the IETF, mainly focusing on technical management and the internet standards process. The IESG consists of the Area Directors (ADs) who are selected by the Nominations Committee (NomCom) and are appointed for two years at a time.

2.3.2 Internet research task force

The Internet research task force is a separate, yet at the same time, parallel organization to the IETF that focuses on long term research topics. This organization is concerned with researching internet protocols, applications, architecture and technology. The members of this organization are divided into research groups, where they get the long term membership needed for research and collaboration. As with the IETF in general, participation in this group is done only by individuals, and not groups or organizations.

The difference between this organization and the IETF is that the IRTF is concerned with long term topics, whereas the IETF is more focused on short term problems, as well as developing standards.

2.3.3 Tools team

The Tools team consists of volunteers from the IETF that develop tools needed by the IETF. The team leader, as well as the team members are selected by the General Area Director. The team investigates open source tools that can help other members of IETF to work more efficiently. When no suitable tools are found, the team develops and maintains custom tools, standardized tool are however greatly preferred.

Some of the Tools team projects are:

Datatracker
The IETF Datatracker is the biggest custom tool developed and maintained by the Tools team. It is the most visible and heavily-used tool of the IETF. The Datatracker is used to upload Internet-Drafts, manage their review and approval, manage meeting materials, and manage working groups. Feedback on the Datatracker is actively solicited, and the Tools team is constantly working on bug fixes and enhancements.

Postconfirm
The Postconfirm system employs a variety of verification methods to discard unwanted email.

Mail archive tool
The Mail Archive tool provides advanced, easy-to-use archive searches for all IETF lists, past and present. This tool is under active development, with many new features planned.

Mailman
Mailman is a general-purpose mailing list management framework. It is used by the IETF to archive communication done by email, as well as manage the various mailing lists.

Xml2rfc
A tool widely used by the IETF community to write internet drafts.

RFCdiff
The RFCdiff tool allows easy comparison of the changes between two versions of plain text documents.

Source: https://www.ietf.org/about/groups/tools/

2.3.4 Internet Architecture Board

The Internet Architecture Board is a committee of the IESG.
Their responsibilities are:

IESG confirmation
Confirming the IETF chair directors as well as IESG Area Directors

Architectural oversight
Providing oversight and comments on aspects of the architecture of protocols and procedures used by the internet

Standards process oversight and appeal
Providing oversight over the process of creating network standards, as well as acting as an appeal board for complaints of improper execution of said standards

External Liaison
The IAB is responsible for representing the IETF's interests in liaison relationships with other organizations concerned with issues relevant to the world-wide-web.

Advice to ISOC
Acting as a source of guidance to the Board of Trustees and Officers of the Internet Society concerning procedural, technical, architectural and policy matters.

2.3.5 Directorates

Directorates are comprised of experienced members of the IETF and often serve as advisors for IETF work. A directorate is defined by the IETF as follows:

> "In many areas, the Area Directors have formed an advisory group or directorate. These comprise experienced members of the IETF and the technical community represented by the area. The specific name and the details of the role for each group differ from area to area, but the primary intent is that these groups assist the Area Director(s), e.g., with the review of specifications produced in the area."

Source: https://www.ietf.org/about/groups/directorates/

2.4 The IETF email archives

2.4.1 Mailing-lists

The IETF uses mailing lists to discuss whatever it may be that needs to be discussed, these mailing lists may be specific to a working group, topic, or something else entirely. There are 4 main categories of mailing lists.

General IETF discussion list

This list has two main purposes. The first one being furthering the development of standard through discussion, as well as discussion regarding IETF's direction, policies and procedures. This is considered the most general of IETF's mailing lists, and as such, many various topics are allowed to be discussed here. This is however only intended for the initial discussion of topics. Once the discussion falls into the area of a specific working group, the discussion should be redirected to said working groups mailing list.

Announcement Lists

These are mailing lists intended for distribution of announcements,and are not intended for discussion. The four main announcement lists are as follows:

I-D Announce
I-D Announce receives announcements about actions taken on Internet Drafts currently being considered by the IETF .

IETF Announce
IETF Announce receives announcements about IETF meetings, the activities and actions of the IESG, the RFC Editor, and the NomCom, and other announcements of interest to the IETF community.

IPR Announce
IPR Announce list receives announcements when IPR disclosures are uploaded to the IETF website.
IESG Agenda Distribution IESG Agenda Distribution list receives the "Preliminary Agenda" for each IESG biweekly teleconference.

Working Group Lists

Each working group within the IETF has its own mailing list. These lists are used to discuss documents, as well as any other topics that may be relevant to a given working group.

Non-Working Group Lists

These lists are used for topics that may be considered useful or interesting to the members of IETF, but are not directly connected to any working group.

All correspondence to and from the mailing lists described above is archived and published in IETF's own website.

2.4.2 The mbox format

The mbox format is a file format used by most, if not all modern unix distributions to store collections of messages in pure text. By definition, each message contained in an mbox file starts with "From", a space, and the sender's email address and date. This is also referred to as an emails envelope. The envelope is not a header however, each message should, in addition to the envelope, also contain headers required by the RFC standard.

Example:

- From MAILER-DAEMON Fri Jul 8 12:08:34 2011
- From: Author <author@example.com>
- To: Recipient <recipient@example.com>
- Subject: Sample message 1

The messages are stored in the order they arrived, meaning that various parts of MIME multipart messages may appear out of order.

The mbox format is not limited to just one format, rather it is used to describe a whole family of formats stemming from the original mbox format. Just some of these alternative formats are mboxo, mboxrd, mboxcl, and mboxcl2. The reason why these formats exist stems from the shortcomings of the original mbox format.

In the original mbox format, all of the messages are stored in a single file, with the delimiter/separator being a line starting with the word "From" followed by a blank space (as found in the envelope). This means that if a given message contains this pattern in any other place than the envelope, the message will be split in two, with each part being treated as an individual message. This will cause at least one of the parts to lack its headers, and most likely make no sense on its own. The alternative formats can be summed up as various ways to avoid this from happening, usually by adding a ">" in front of any other occurrences of the word "From" in the message.

The IETF mbox files are of the original mbox format. In order to solve the issue of boundaries they have developed a script called cleanarch. This script automatically adds a ">" in front of any line starting with "From ", like the formats described above. While this script was developed to be used in conjunction with the Mailman framework, it is written in Python, meaning it can be modified and adjusted to work without it.

For the purpose of this project this script has been modified to run on its own, and add a "|" sign, instead of a ">", as this sign is often used to represent quotation, and there would be no way of distinguishing a corrected line from a quoted one should this sign be used.

Chapter 3

Data and database preparations

In order to start working with the text archive, a quick way to access it is vital, as the per file overhead will quickly add up given the sheer amount of files contained within the archives. To accomplish this the wget command, found in mostl unix distributions was used. This command allows the user to recursively download the content of a given url address. For this project, this command will be used to download all of the mbox files available on IETF's website on to local storage.

The experiments, statistics calculations as well as any other case where execution of code was necessary has been run on a server provided by the institute of informatics at the University of Oslo. Said server is at the time of writing running RedHat linux version 6.10.

3.1 wget

Wget has been used to recursively download the entire archive. One thing that needs to be taken into account when recursively traversing anything, are loops. This fortunately is not a problem in this case, as there are no outgoing links in the archives. The only url's that can be found are in the mbox files themself, and the wget command does not inspect those files. Unless a different location is specified the wget command will use the current location of the terminal as the root for the downloaded structure. For more information on the wget command please refer to the Linux manual.
The final command looks as follows:

wget -r `https://ietf.org/mail-archive/text/`

With

`https://ietf.org/mail-archive/text/`

beeing the url of the IETF text archives.

As of the time of downloading (10. October 2020) this archive contains 77224 mbox files. This number is expected to increase in the future.

3.1.1 Assumptions

In order to extract the data a few assumptions have to be made.

1. All of the emails follow RFC 2822 [Res01]. This is the most recent RFC that specifies the names of email headers. This RFC has been updated by RFC 5322 [Res08],as well as RFC 6854 [Lei13]. The changes made in these updates do not concern the names of the fields.
2. There will be exceptions to assumption 1, and the parser should be able to handle such situations
3. All relevant data should be extracted.
4. If needed, adding new fields to the database should be made relatively simple, as there is no sure-fire way to predict which fields may or may not become relevant in the future.

3.1.2 Tools

For the purpose of parsing and extracting the data, the Python programming language is used due to its abundance of modules, and ease of use.

The three main modules used in this project are:

- email
- mailbox
- pysolr

The mailbox module is responsible for iterating over messages in a given file, as well as extracting the content of the headers. The email module is responsible for field specific operations, which will be described in detail later. Pysolr is responsible for interfacing with the desired Solr instance.

3.1.3 Parser outline

1. Scan a given directory for files with a given file extention, in this case, .mail or .mailcl
2. Initialize an array that will be used to store the finished product
3. Then, for each file, do the following
   (a) Load the given mbox file
   (b) Create an iterator by using the mailbox module
   (c) Then, for each message in the file, do the following

i. Add a given field to the document, if the extraction was successful, assign the parsed value to it, with respect to the rule regarding address field rule mentioned earlier. Do this for every field that is a part of the document layout.

ii. Add the processed document to storage

4. Upload to Solr

3.1.4 Designing a full text database

In order to transform pure text into something one can make sense out of and search, it is first necessary to design a system that will allow for such functionality. Under normal circumstances, this system would essentially be a search engine. Usually a search engine keeps track of tokens which are indexed in a reverse index. Results in such search engines are usually calculated based on the similarity of tokens derived from the input string. The order in which these tokens occur also usually plays a role in how the results are calculated.

A modern search engine will also classify its token by the use of machine learning techniques. For the purpose of this project, a program that offers all of the functionality of a search engine, in addition to many other features, is used. This program is called Apache Solr. It allows the user to feed it data in a user specified format, and process it according to the users wishes. It is important to note that even if Solr is given data that breaks the defined format, it will not throw errors or crash. Instead It will try to guess how to best process the new information. For the sake of consistency, readability, and accuracy, this functionality was not utilized in this project outside of the setup process.

3.2 Setting up Solr

Before Solr can be run, a few things need to be in place. Firstly, Solr is written in java, therefore, java must be present on the machine that is going to host the database. The most recent version of Solr can be acquired from the following site:

`https://lucene.apache.org/solr/downloads.html`

A modern web browser is also recommended in order to use the Solr web interface. This is not strictly necessary, but it makes using Solr a much more pleasant experience.

Once Solr is downloaded, extract the contents of the compressed file, and do the following.

1. In `\solr\solr\server\solr\`, create a new folder and name it whatever you want the name of the Solr core to be.
2. Copy the "conf" folder from `\solr\server\solr\configsets\default\` to the folder you just created
3. Start Solr, this will depend on what operating system is used

(a) for Linux, use `bin/solr` start

(b) for Windows use `bin\solr.cmd` start

4. Once Solr has started, open up a web browser, and go to

   (a) localhost:8983 to access Solr's graphical interface

   (b) Select "Core Admin" -> "Add core"

   (c) Input the name of the folder previously created, and select "Add Core"

At this point the newly created core is ready to receive data.

3.3 Designing a Solr Schema

In order to make proper use of Solr, a schema has to be provided by the user. By default, a schema is contained within the default configuration files that are needed for the Solr core to function. The schema defines how the database looks. This means what kind of fields it has, the type of a given field, and by extension, what kind of data is accepted in that field, as well as how that field can be queried.
A set of tokenizers and filters can also be attached to a field for further processing and indexing of the input data.

This means that a "layout" for a document needs to be defined first. This is not strictly necessary as Solr will automatically add new fields and adjust their rules according to what it sees fit for that field. While this is not very useful for this project, it is there, and it means that there will be no crashes if a field that is not defined in the database schema is encountered. However, for ease of use, it is possible to first, upload a document, with the desired layout, and then manually adjust the settings for each field in the schema file. The schema file can only be manually changed when Solr is not up and running.

3.4 The Document format

A document is meant to represent exactly one email, in its entirety. This does not mean all of the headers, rather, all headers that have been chosen as useful to extract data from, as well as the message body. Attachments will not be included as they take up a considerable amount of not only storage space, but visual space as well. This would make the messages difficult to read, something that goes against the goals of this project.

For further research one could attempt to restore the attachments from their ASCII representation back into their original form, then parse them. That is however outside of the scope of this project.

Then, what exact information should be extracted from the messages? As much data as possible, as long as it fulfills at least one of the following requirements.

1. The data follows the standard set by the IETF
2. The data that does not follow the standard, but can still be transformed into a useful format.
3. The data that is common in all email messages

While custom headers are allowed and the support for them exists, it is impossible to predict what they may be called, or what data they may contain, making it impossible to predict how said data should be processed. They will for that reason be excluded from this project, only headers defined in the most recent RFC describing the email format will be considered.
As not all headers are considered mandatory, there will inevitably be fields that are missing, yet are expected to be present. For such occasions a default value must be specified. Not all fields will accept the same value however, therefore a default value must be specified for each individual field. This is done by the parser, and not Solr. Since Solr simply omits fields that have no value, resulting in documents that will not be uniform. It is also easier to determine that a field is empty, if there is a default value in its stead, rather than the entire field missing. So while this is strictly speaking not necessary for the functionality of the database, as Solr is perfectly capable of searching for documents that do not contain a certain field, it will make reading the documents easier for the end user.

3.4.1 Date

This field contains the date of sending, as described in RFC5322 [Res08]. It is also a mandatory field, and is therefore included in the document template. The data extracted from this field will be formatted to comply with Solr's "datefield" type. This will allow queries based on date, such as, from and to a specific date. The default value for this field, must be at least older than the oldest email present in the database. To make sure that this is the case, the default date has been set to 1. of january 1900.

As many ways of formatting dates exist, there is no guarantee that the conversion to Solr's format will be successful, for this reason a "Date-raw" field will be added to the document template. This field will not be processed in any way by either the parser or Solr. The default value in this field will be "Null"

3.4.2 From

This is also a mandatory header and as such it is included in the document template. The existence of a default value in this field will be treated as an error. What is important to note about this header, as well as many other headers, is that it contains email addresses, and may as well contain the actual names of the senders. Another important aspect of this field, and other fields of the same nature, is that it is allowed to contain more than 1 address, which can again have names attached to them. For this reason a multivalued field has to

be used. This is a field that essentially behaves like a list. The elements in this kind of field are processed separately at index time. To make searching easier and more accurate, fields that can contain once or more email address are split into 3 separate fields in the database, those are as follows

(original name of field)
This field contains the raw unprocessed data retrieved from the mbox file

(original name of field)-name
This field contains the names extracted from the original field.

(original name of field)-address
Contains all of the addresses found in the "From" field.

To make sure that no data is lost during processing, the "From" field itself will remain untouched.

- Sender
- Reply-To
- To
- Cc

Those are fields that contain email address and name pairs, therefore they will be processed in the same way the From field is. These are not a mandatory fields, as such, absence of data in those fields will not be treated as an error.

3.4.3 Sender

This field contains information about the agent responsible for transmitting the message, this field may be especially useful for tracking which individuals work for the same organization, as they will, most likely be using the same agent to transmit their messages.

3.4.4 Reply-to

This field specifies which mailbox the response to a given message should be sent to. This field is especially useful for tracking conversations as it specifies directly who is replying to who. One may wonder why the mailing list address cant be used for this purpose. The reason for this is that a conversation may take place outside of the mailing list, only to be redirected to the mailing list at some later point. It may then also be continued outside of the mailing list and so on. It is also very plausible that conversations across mailing lists are taking place regularly, and using just the mailing list address as a means to find those conversations would simply not work.

3.4.5 To

This field specifies the main recipient of a given message and like the "From" field, can also contain the actual name of the recipient in addition to their email address.

3.4.6 CC and Bcc

CC stands for Carbon Copy and is used to automatically send a copy of the message to all of the specified recipients. Bcc stands for Blind carbon copy. The difference between these two fields is the visibility on the recipients side. What this means is that given the Cc field, the recipient will be able to see all of the other recipients specified in the Cc field. This is not the case in the Bcc field, the recipient will not be able to see who else also got the message. The Bcc is for this reason a convenient way of sending out general announcements without revealing any sensitive data.

3.4.7 In-Reply-To

The in-reply-to field as defined in RFC 5322 [Res08], is a field used when creating a reply to one or more messages. This field should contain the message identifiers of all messages a given reply is replying to.

3.4.8 Message-id

This is a mandatory field that uniquely identifies a message on a global scale. They are very useful for tracking down specific messages. This field will for this reason not be processed in any way aside from removing preceding and succeeding whitespace. Absence of data in this field will count as an error.

3.4.9 References

This field contains the message id of a referenced message. This is not a mandatory field, but still a field with a unique key, and should therefore not be processed in anyway aside from the one mentioned above.

3.4.10 Subject

This is what is commonly referred to as an "unstructured field", meaning there is no concrete pattern or standard as to how data in this field should look. It can be anything from one word to a whole sentence, or just some completely random characters. With that being said it is probably safe to assume that most people would use this field in a sensible way, that being to highlight the motive or goal that lead to the emails creation. This field should therefore be processed in the same way as the payload of the message,that being as pure text. This is not a mandatory field and the absence of data in this field will not be treated as an error.

3.4.11 Comments

This is also an "unstructured field", as such it should be processed in the same way as described above.

3.4.12 Payload

This, again, is an unstructured field commonly used to contain the actual text of the email. It will therefore be processed like the 2 fields described above.

3.5 Tokenizers

We have already briefly discussed tokenizers and their effect on how results are calculated. Therefore in order to ensure that a search on a given field yields the expected results, fitting tokenizers for that field must be chosen. Luckily Solr has a set of tokenizers built right in. The tokenizers are specified on a per field basis, meaning that each field in a document can be processed differently without it affecting any of the other fields.

The tokenizer, or tokenizers for each field are specified in a database schema file. Any changes made to a fielder's choice tokenizer after the initial data upload will require a re-upload of the entire database to take effect.

3.5.1 From

As previously mentioned this field, at least in this database, will contain unprocessed data, the same string extracted from the mbox file, is the same string that will occupy this field in the database. In order to make this field useful however, Solr's standard tokenizer will be used. It treats "@" as a delimiter, meaning all email addresses will be split into several tokens, this is not a problem, as the previously mentioned address field will already contain these.

Solr standard tokenizer = solr.StandardTokenizerFactory

3.5.2 From-name

As this field contains what are essentially keywords, Solr's keyword tokenizer will be used. This also means that when searching for a given name, the exact name must be input in order for the desired results to be returned.

Solr whitespace tokenizer = solr.KeywordTokenizerFactory

3.5.3 From-address

As already established, this field will exclusively contain email addresses, for this reason Solr's email tokenizer will be used. The same will be true for other fields that contain exclusively email addresses.

Solr email Tokenizer = solr.UAX29URLEmailTokenizerFactory

The following field will be processed in the same way as the From field:

- Sender
- Reply-To
- To
- Cc
- Bcc

3.5.4 Message-id

As this is a unique identifier, and should not in any way be changed, the keyword tokenizer wil be used. The keyword tokenizer is exactly what the name implies, and produces a token that is exactly the same as the field's content, in other words, it does nothing to the data given.

Solr keyword tokenizer = solr.KeywordTokenizerFactory

3.5.5 References and In-Reply-To

These two fields are similar based on the data contained within them. While they are used for different purposes, they both contain message ids, meaning they can be parsed, and represented in the same way. As with the "Message-ID" field, these will not be changed in any way. Additionally, for each of these fields, a new field has been added bearing the same name as the original field, with a "-ID" added at the end. These new fields will contain a processed version of the original field.

Solr keyword tokenizer = solr.KeywordTokenizerFactory

Solr standard tokenizer = solr.StandardTokenizerFactory

3.5.6 Subject, Comments and Payload

As these are all fields with the same characteristics, they will be tokenized by the same tokenizer. These are all unstructured fields, meaning there is no definitive answer as to how these fields should be tokenized, therefore, Solr's standard

tokenizer will be used.

Solr standard tokenizer = solr.StandardTokenizerFactory

3.5.7 The final document template

The final document remplate is as follows:

Field	*Tokenizer*
Date	No tokenizer
Date-raw	solr.KeywordTokenizerFactory
Timezone	solr.KeywordTokenizerFactory
From	solr.StandardTokenizerFactory
From-name	solr.KeywordTokenizerFactory
From-address	solr.UAX29URLEmailTokenizerFactory
Sender	solr.StandardTokenizerFactory
Sender-name	solr.KeywordTokenizerFactory
Sender-address	solr.UAX29URLEmailTokenizerFactory
Reply-to	solr.StandardTokenizerFactory
Reply-to-name	solr.KeywordTokenizerFactory
Reply-to-address	solr.UAX29URLEmailTokenizerFactory
To	solr.StandardTokenizerFactory
To-name	solr.KeywordTokenizerFactory
To-address	solr.UAX29URLEmailTokenizerFactory
Cc	solr.StandardTokenizerFactory
Cc-name	solr.KeywordTokenizerFactory
Cc-address	solr.UAX29URLEmailTokenizerFactory
In-Reply-To	solr.StandardTokenizerFactory
In-Reply-To-ID	solr.KeywordTokenizerFactory
Message-ID	solr.KeywordTokenizerFactory
References	solr.StandardTokenizerFactory
References-ID	solr.KeywordTokenizerFactory
Comments	solr.StandardTokenizerFactory
Subject	solr.StandardTokenizerFactory
Content	solr.StandardTokenizerFactory
Mailing-list	solr.KeywordTokenizerFactory
File-location	solr.StandardTokenizerFactory

Chapter 4

Parsing

Parsing is a word commonly used to describe the process of generating new data, based on other input data. The act of generating machine code from a given programming language has parsing as one of its steps. In the case of this project, parsing means converting pure text, into a form that Solr can understand. This practically means that the way the parser outputs its data, as well as what kind of data it outputs, depends on what is in the Solr schema.

4.1 Tools

As previously established, Python is used to implement the parser due to easy access to a vast selection of modules. The portability Python provides is also a welcome addition. While there are languages that generate faster programs, they do not provide the functionality that Python does. While this functionality could have been implemented, it would have taken too much time, and would be unwise as the tools needed are already available to use free of charge.

4.1.1 Python modules

As previously mentioned Python was chosen for this project due to its selection of modules. The modules used to implement the parser are as follows.

4.1.2 email

This package provides functionality for parsing the individual heads fields. It is for example used to convert the "Date" field into a format Solr can understand. This package conforms to RFC 5233 [Res08]and RFC 6532 [YSF12], as well as RFC 2045 [FB96a], RFC 2046 [FB96b], RFC 2047 [Moo96], RFC 2183 [TDM97], and RFC 2231 [FM97] regarding the MIME format.

4.1.3 mailbox

The mailbox package is responsible for iterating over mbox files. It also allows the user to extract headers based on their name. This means it is in fact possible to extract custom headers, this is however not utilized in this project. The documentation for this module does not specifically state what RFC this module conforms to. The "Message" objects this module implements, do however. The documentation specifically states:

> "If message is a string, a byte string, or a file, it should contain an RFC 2822-compliant [Res01] message, which is read and parsed."

4.1.4 pysolr

This is a package used for interfacing with Solr, in the prasers case this package is used to upload the parsed data to a given Solr core.

4.2 Can the Python mailbox module be used?

Before we deem the mailbox module appropriate for our use, we first need to make sure that it even though it follows the RFC 2822 [Res01], which has been long made obsolete by newer RFCs, it can be still used.

In short, the module can only be used if a message that is compliant with RFC 6854 [Lei13] (which is the most recent RFC regarding the email format at the time of writing) is also, at the same time, compliant with RFC 2822 [Res01].

The answer is both yes and no, due to the way the Python module works. For example, the "get()" method that is used to fetch a specific header, does not actually care what the header line contains. It simply returns the value of the field with a given name.

To give an example, let us say we have a message, where the "From" header looks like the following:

"From: test.com"

This is not RFC2822 [Res01] compliant way of creating and filling in a header field as the left hand side of the "@" sign is missing, as well as the "@" sign itself. The Python module however will still return "test.com" from the "get("From")" call, even though it is not compliant.

If that is the case, what part of the message needs to comply with the RFC 2822 [Res01]?

All kinds of separators need to comply. As we are extracting many messages from a single file, the boundaries of a given message need to be one hundred percent correct. If this is not the case, errors such as missing headers, mixed messages or other unpredictable behaviour may occur.

According to RFC 2822 [Res01], inside a message, there is only 1 boundary, that being the boundary separating the headers from the body.

From the RFC 2822 text:

> *"A new line that separates the headers from the body A message consists of header fields (collectively called "the header of the message") followed, optionally, by a body. The header is a sequence of lines of characters with special syntax as defined in this standard. The body is simply a sequence of characters that follows the header and is separated from the header by an empty line (i.e., a line with nothing preceding the CRLF)."*

Source: `https://tools.ietf.org/html/rfc2822#section-2.1`

There is no mention of any changes to this rule in later RFCs, it is therefore assumed that it is still in effect at the time of writing. All names of headers need to comply, fortunately names of headers we are concerned with have remained unchanged through the years. Some older emails may lack the newer headers, which is to be expected, and the parser will be able to handle such cases. In conclusion, the changes made in newer RFCs do not get in the way of our works. Meaning the current mailbox and email modules can be utilized for this project.

4.2.1 Older standards

There still remain older standards. The IETF is an old organization, and messages from even before 1980 can be found in their database.

The earliest RFC that specifies any form of email format that exists in the IETF's database, is RFC 561 [Bhu+73], published on September 5. 1973. It lays the fundamentals of how a modern email message should look like. Most importantly, it specifies what the headers should be called. What it does not specify, is which headers are required to be present. This means that one cannot determine, with one hundred percent certainty, that an email from this time conforms to its RFC. Which in turn means that missing headers in those emails do not violate any standards (and should technically not be considered an error). However, it is safe to assume that in order for a message to be properly transmitted, certain information still needs to be provided, such as the recipient address and the sender address.

Another important thing to note is that the RFC specifically states that the case (upper/lower) of headers does not matter. The Python mailbox module matches headers with no regards to upper or lower case by default. It also has a method that returns a list of all fields with the same name. If any email is found to have several fields with the same name, the contents of those fields will be appended, and processed as described earlier.

In short, as long as the boundaries and names of headers in a given email are correct, the mailbox module will be able to properly parse it, regardless of the RFC.

4.3 The parser code

The "mailparser.py" file contains methods used in the parsing process. This file by itself only contains code responsible for parsing exactly 1 mbox file, and returning the result. This is done to allow the main program to control the various steps needed in order to properly transform the email archives.

First, the needed modules are imported. The imported modules are not limited to the ones discussed earlier, but are still needed and utilized in the parsing process. As these modules are more commonly used, and their function is mostly self explanatory, they will not be discussed in detail, rather, the use of specific method originating from these modules will be explained as needed.

```
import os
import email
from email.utils import *
import mailbox
import string
import time
from dateutil.parser import *
```

The "parsefile" method is the "main" method, it is the method that converts an mbox file into a data structure that can be directly uploaded to Solr. It takes 2 input arguments:

`Fileinn` is the location of the mbox file
`Mailing_list` is the name of the mailing list to which the current mbox file belongs

This is extracted by the caller program from the file path of the mbox file. This is done for efficiency reasons, as the mbox files are stored in folders named after the mailing list they belong to, the extraction process only needs to happen once per folder. This also means that the "Mailing-list" field will never be empty.

The mailbox module is first used to transform the given mbox file into an iterable data structure. Next an iterator is created, and the return variable is initialized as a list.

```
def parsefile(fileinn,mailing_list) :

        #print(fileinn)

        box = mailbox.mbox(fileinn)

        iter = box.iterkeys()

        out = []
```

The program then enters a loop which executes for as long as there are items to iterate over. Each item is first extracted from the data structure created by the mailbox module, and a dictionary is initialized and assigned to the "mail" variable. This variable holds, the exact data that is uploaded to Solr. This also means that it needs to correspond exactly to what is specified in the Solr

schema. All of the field names need to be present to accomplish this, empty fields will have their previously specified default value assigned to them. All of the data must also follow the format defined by Solr.

```
msg = box.get_message(key)

mail = {}
```

First, the "Date" field is extracted. This is done by invoking the "get()" method contained in the "Message" object. A "Message" object is what the mailbox module creates and uses to store and alter messages. The "get()" method takes 1 argument, that being the name of the header it should return. It is important to note that the case of the input argument does not matter in this case as the method will match the input argument with no regards to upper or lower case.

If the method fails to locate the requested header the default return value is returned, this value is "None".

```
tmp = msg.get("Date")
if not tmp is None:

        mail["Date"] = parseDate(tmp)
        mail["Timezone"] = getTimezone(tmp)

else :#Default values
        mail["Date"] = "1900-01-01T0001:00"
        mail["Timezone"] = -9999
```

4.3.1 parseDate() and getTimezone()

These two methods are responsible for transforming the "Date" header into a format that Solr can understand and make use of. They both take exactly one argument, which should be the content of the "Date" header. The parseDate() method uses the "Time" module and its strftime() method to transform the input arguments into a given format. The "strftime()" method takes 2 arguments, the first being a string specifying the output format, and the second being a tuple representing a point in time.

The way of specifying the desired output format of "strftime()" can be found in the official Python documentation, in this case, the following formatting is used, as it is the exact format of Solr's datepointfield field type:

`%Y-%m-%dT%H:%M:%S`

- `%Y` Year with century as a decimal number.
- `%m` Month as a decimal number [01,12].
- `%d` Day of the month as a decimal number [01,31].
- `%H` Hour (24-hour clock) as a decimal number [00,23].

- `%M` Minute as a decimal number [00,59].
- `%S` Second as a decimal number [00,61].

These variables are replaced by their respective value in the second argument.

It is important to note that in the code, a "parsedate()" method is used as the second argument to "strftime()". This is not a recursive call, but a call to the method contained in the email.utils module. This method transforms the "Date" header content into a 9-tuple that can be passed directly to methods such as "mktime()" or "strftime()".

The documentation for this method specifically states that it attempts to parse the date according to the rules in RFC 2822 [Res01]. If this is not possible, it will attempt to guess the correct way to parse the given argument. This also means that information taken from this field should be taken with a grain of salt as there is no guarantee that this information is 100% correct. For this reason a field called "Date-raw" will be included in the document layout, as previously stated.

If, at any point, the parsing fails, the default value is returned.

4.3.2 addAddressField(name,mail,msg)

This method is responsible for adding and correctly formatting all the fields that can contain name and address pairs. As previously discussed it will add 3 fields per method call, those fields being the:

- "name of the field"
- "(name of the field)-name"
- "(name of the field)-address"

The method takes 3 arguments: name, mail and msg.

- "Name" being the name of the field
- "mail" being the current document
- "msg" being the current message

First, the method attempts to extract the field with a matching name. This is done to check if said field actually exists. If the check fails, the default value is applied to all 3 fields, and the method exits. If the check is successful, the extracted field is added to the document. This is the raw, unprocessed field that was previously discussed.

```
tmp = msg.get(name)
if not tmp is None:

        mail[name] = tmp
```

Next, the `get_all()` method is called. This method is contained within the "Message" object. It takes 2 input arguments, first one being the name for the field one wishes to attempt to extract, and the second one being an empty list. This method returns a list of all values for the field with the given name.

The result of `get_all()` is then passed to the `getaddresses()` method, found in the email.utils module. This method returns a list of 2 tuples, each of these tuples contains the name at index 0, and the email address at index 1. It is important to keep in mind that there is no guarantee that this will always be the case. The method will return tuples with empty indexes without warning, meaning it is necessary to check each tuple for empty values before assigning them to their respective field.

```
tmp = msg.get_all(name, [])
tmp = getaddresses(tmp)
```

The parser then continues as described, checking for empty values in the 2 tuples, and adding them to their respective temporary field. Finally, a check to see if the temporary variables are empty is performed to verify if anything at all has been extracted, if this is not the case the default value "Null" is assigned. The temporary variables are then added to the document with their proper names, and the method terminates.

```
tmp_name = []
tmp_address = []


while i < len(tmp) :

        if tmp[i][0] != '' :
                tmp_name.append(tmp[i][0])


        if tmp[i][1] != '' :
                tmp_address.append(tmp[i][1])


        i = i + 1

if not tmp_name: #if empty then add null
        tmp_name.append("Null")

if not tmp_address : #if empty then add null
        tmp_address.append("Null")


mail[name + "-name"] = tmp_name
mail[name + "-address"] = tmp_address
```

4.4 addMultiIdField()

This method is responsible for parsing the "References" and "In-Reply-To". As the name of the method suggests, it is responsible for parsing fields that may

or may not contain several message id's. This method was developed as neither the mbox module or the email utils module were successful in extracting information from these fields properly. Often returning strings that were either malformed, or returning only one value, where several are present.

To understand how and why this method is as it is, we first need to look at the definition of the syntax of a message id as defined by RFC 2822 [Res01]:

msg-id = [CFWS] "<" id-left "@" id-right ">" [CFWS]

Here, [CFWS] essentially means a set of separators previously defined in the RFC. This is not very helpful however, as it was quickly found out that in many cases, this definition simply does not hold. Aside from this, a message id is defined as 2 strings sandwiched between a "<" sign and a ">" sign, with a "@" in the middle. As this bears resemblance to an email address, the "getaddresses()" was tested as a candidate for parsing of the two fields. The results were poor however, so this option was dropped.

Instead, the previously mentioned modules are used to retrieve the fields in their raw state, to be parsed in the following way. First, the modules simply try to extract the raw data, and if the extraction fails the default value "Null" is applied instead.

```
tmp = msg.get(name)#pull data
#was anything pulled?
if not tmp is None and tmp != "":

        mail[name] = tmp

        tmp = msg.get_all(name)
```

Then, the extracted string is split according to a regular expression.

```
for value in tmp:

        ids = re.split('␣|,|\t|\n|\r',value)
```

Doing this still leaves one problem however, as one may notice the regular expression does not take into consideration situations where there are no separating characters, a situation that is quite common. In this case, the message ids need to be split based on the less and greater than signs as defined by RFC 5322 [Res08]. The "split()" method is used to separate the message id's in this case, the downside to using this method is the fact that it removes the specified delimiter from the string, essentially changing it from its original form. This is unacceptable in our case as modifying an id will make finding it impossible. For this reason, any string that has successfully been split into two or more parts, needs to go through a process of adding these characters back in. This is quite simple to do. If the resulting list is of length 2, then the first element in the list will be missing a "greater-than" sign on its right hand side, and the last element will be missing a "less-than" sign on its left hand side. If the list is of length 3

or more any elements that are not first or last in the list will be missing both of these characters.

```
for x in ids:

        a = x.split("><")

        if len(a) > 1:
                out.append(a[0] + ">")
                out.append("<" + a[-1])

                for y in a[1:][:-1]:
                        out.append("<" + y + ">")

        else:
                out.append(x)
```

The final challenge of parsing this field is to make sure that garbage data does not find its way in. In some cases for example, things like dates and another random string are thrown into these fields, and while they may have served their purpose at the time, they are not desirable in our case. In order to prevent said undesirable string from getting into the final product, a simple filter was created. This filter is based directly on the definition of a message id found in RFC 5322 [Res08], with some slack. While RFC 5322 [Res08] strictly defines a message id as being sandwiched between a "greater-than" and a "less-than" sign with a "@" symbol in the middle, there are cases where this rule is only partially followed. In order to get the most complete results, the filter will only allow strings that either start and end with the "greater", "less than" signs, or contain at least one instance of the "at" sign.

```
final = list()

for x in out:
        if "@" in x:
                final.append(x)
        elif len(x) > 1 and x[0] == "<" and x[-1] == ">":
                final.append(x)
```

Finally the method adds two new fields, one being a raw version of the extracted data, and the other being the results of the process described. The field names used are that of the original field, with the processed version having an "-ID" attached at the end.

```
if len(final) > 0:
        mail[name + "-ID"] = final
else:
        mail[name + "-ID"] = ["Null"]
```

If this process does not yield any results, the default value "Null" is assigned instead.

4.4.1 Handling multipart messages

In order to distinguish MIME multipart messages from their regular counterparts, a check is made to see if the message is a multipart message. The "is_multipart() " is used for this purpose.

If this method returns "False", the content field is extracted as one part. If the method returns "True", each part of the message is checked for its content type. All parts with the "text/plain" content type are then appended onto each other. The final message is then appended to a list, ready for upload to Solr.

It is important to note that the act of connecting various pieces of a MIME messages together is done by the mailbox module automatically.

```
if not msg.is_multipart() : #Text mail

        mail["Content"] = box.get_message(key).get_payload()

        out.append(mail)

else : #MIME mail

        mail["Content"] = ""
        for part in msg.walk():
                if part.get_content_type() == "text/plain":
                        mail["Content"] += part.get_payload()

        out.append(mail)
```

4.5 Common errors and fixes

4.5.1 Encoding errors

A common problem one will encounter while trying to parse the archives are unicode encoding errors. These are caused by unknown characters contained within the mbox files. For example:

> Error type = UnicodeEncodeError
> 'ascii' codec can't encode character `\textbackslashufffd` in position 1121: ordinal not in range(128)

One curious thing to note about these errors is that the vast majority of them are caused by the same character, namely `\textbackslashufffd`. This is a special replacement character.

In order to fix this issue, the "codecs" module is used as it allows us to specify what action should be taken when the said error is raised.

```
import codecs
codecs.register_error("strict", codecs.replace_errors)
```

What is weird about this is that " codecs.replace_errors" specifies that any characters that cause errors should be replaced with a special replacement character, which is the same character that causes the error in the first place. Nevertheless adding the line above completely solved the issue of unicode encoding errors.

4.5.2 Cleanarch

As previously discussed, the mbox format has some inherent problems with the way it stores messages. In order to solve this problem the developers of the Mailman framework have created the cleanarch script. This script is written in Python, meaning it is possible to modify it to run outside of Mailman, and this is exactly what has been done.

The script consists of 2 methods, `escape_line` and, `clean`.

4.5.3 The insertion symbol

The `escape_line()` method is responsible for writing the new symbol at the start of a line starting with a "From", if such a line is found outside on the messages envelope. The sign the IETF uses for this is ">". While this will solve our boundary problem, it introduces a new one, as ">" is often used to quote previous messages in the case of a reply.

For example:

> *Tom-PT Taylor wrote:*
> *>*
> *> I am trying to decide whether I want to be in Washington for the SIP interim*
> *> session on Nov. 7. Will anything be presented or discussed which wasn't*
> *> covered at the MMUSIC session in Oslo?*
>
> *It will be a much more in depth presentation than in Oslo. We now have several hours just for tutorials and explanations, in order to best understand the motivations behind the design decisions made by DCS. There will also be time for discussion afterwards.*
> *-Jonathan R.*

Taken from `sip\1999-10.mail`

To avoid this issue the modified cleanarch script instead adds a "|" sign at the start of the line.

4.5.4 The code

The `escape_line()` method takes 4 input arguments, 2 of which are left over and unused in the modified version. Those being, `lineno()` and, `quiet()`, in the original script these were used to specify the line number of the line being replaced, and suppress output. As this functionality is not needed for this project,

it has been removed. The remaining input arguments, `line()` and `output()` are the line itself and a boolean value specifying if output should be generated respectively.

The, `clean()` method takes two input arguments, "Inn" being the directory of the mbox files one wants to clean and "out" being the name of the output file.

The goal while modifying the program was to change as little as possible, this however means that there are some variables that don't actually have any impact on the execution of the program, but are required as input arguments. Variables that had absolutely no use have been removed.

One thing to note about this program is that it outputs to sys.stdout, not a file, by default. To avoid replacing every line, sys.stdout is simply redirected to a file.

```
sys.stdout = open(out, 'wb') #redirect stdout to file
```

This needs to be undone if sys.stdout is to write to the console again during the program's execution.

The program begins by defining 2 regular expressions for use when matching lines in a given file.

```
cre = re.compile(mailbox.UnixMailbox._fromlinepattern)

fre = re.compile(r'[\041-\071\073-\176]+')
```

When the clean() method is called, its various variables are first defined, some of which have no function outside of serving as input variables to other methods. The stdout is also redirected into the output file.

```
quiet = False
output = True
status = -1


file = open(inn,"rb")

lineno = 0
statuscnt = 0
messages = 0
prevline = None

sys.stdout = open(out, 'wb') #redirect stdout to file
```

The input file is then read line by line, and each line is checked for the existence of "From".

```
while True:

    lineno += 1
    line = file.readline()
```

```
if not line:
    break
if line.startswith('From␣'):
```

If this is true, the next line is checked to see if it is a RFC 2822 [Res01] compliant header. If this not the case, the `break_line()` method is call in order to insert the "|" character at the start of the current line, as this cannot be a valid envelope. Otherwise, the program continues without any changes to the input file.

```
fieldname = nextline.split(':', 1)
if len(fieldname) < 2 or not fre.match(nextline):

        escape_line(line, lineno, quiet, output)
```

If all checks have successfully passed the program additionally checks if the preceding line to a valid message envelope is an empty line. If this is not the case, an empty line is inserted.

```
if prevline is not None and prevline != '\n':
        sys.stdout.write('\n')
sys.stdout.write(line)
sys.stdout.write(nextline)
```

As the code inside the loop itself has not been modified in any way, a more detailed description written by the original authors, can be found within the source code.

It is important to note that this program will run only on Python 2 due to changes made to the mailbox module in Python 3. To run a Python program with Python 2 use the following command:

py -2 `<name_of_progra_file>`

Chapter 5

Using the Database

There are many ways of interacting with Solr. The first one is the graphical user interface that was previously mentioned, the second option is using the API in a language of choice. There exist Solr API's in a variety of languages. The most popular choice is most likely Java. As Solr itself is written in Java it seems natural that the same language would be the first choice for interacting with it. The Solr Java API is called Solrj,and like Solr, made by Apache. This API is however not used for this project. The reason for this being the ease of use, and availability of necessary modules in Python.
In order to communicate with Solr in Python, a module called "pysolr" is needed. This module can easily be installed by using the "pip install" command like any other Python module.

5.1 Connecting to Solr

By default, Solr binds to localhost at port 8983. In order to send and receive information to and from Solr, a "Solr" Python object must be created. This object is defined in the pysolr module and takes one input argument; the address at which the Solr instance can be found.
Example:

con = pysolr.Solr(localhost:8983)

One can then reference this object to interact with the selected Solr instance.

Each file is parsed, then returned as a list of dictionaries, representing each parsed message. This was done as it is a simple way to represent the messages in python, in addition to correctly formatting them for their upload to Solr. Each key in the dictionary represents a field name, and has the fields value assigned to it.

The ".add()" method within the "Solr" object created earlier takes a list as an input argument, and uploads contents of said list to Solr. This is also part of the reason why everything is organized as one large list by the parser.

5.2 Querying Solr

Before attempting to query, it is first important to understand how this works in Solr, as it is not as simple as using many other popular internet search engines, like "Google" or "Yahoo". As previously established, Solr holds a collection of documents. These documents have fields, in some predefined format for the data contained within them.

Solr queries are at their core, url's. Meaning that each query can be executed by accessing said url, be it from a web browser, or by using other means to open a connection. This url contains all of the parameters used to calculate the query result, those being the field (or fields) to be queried, the query term, as well as any other extra options like faceting, or sorting.

5.2.1 The Query Format

Since the queries are url's, it is absolutely possible to write these by hand. For example:

```
http://localhost:8983/solr/clean/select?q=From%3AAlan
```

This is a query on the "From" field, looking for the word "Alan". This is not recommended as there are far better methods available.

How to write queries in the graphical user interface

In order to query a Solr core from the GUI:

1. first open the interface
2. select the desired core
3. click on the query option below the core name

Then, in the upper lower part of the screen, in the text box named "q", the query can be written. The query syntax of Solr is very simple, specify the field you want to query, and add a ":" sign, followed by the query term.

A query equivalent to the one above, written in proper Solr query syntax will look like the following:

From:Alan

This syntax is the same for queries written in pysolr. Finally, clicking the "Execute query" button on the bottom left will make the response appear on the right side of the screen.

More advanced options can also be used directly from the GUI, such as faceted search.

Solr also supports some logical operators in its queries, these are

- AND
  Requires operators on each side to be present to match the query.

- OR
  Requires one, or both of the operators on each side present to match the query.
- NOT
  Requires the following term to not be present
- +
  Requires a term to be present
- -
  Prohibits a term

This very useful when executing very specific queries,. Let us say for example, we want all messages sent within a year to a specific mailing list, the query would look like this

Mailing-list:ipv6
AND
Date:[2019-01-01T00:00:00Z TO 2019-01-01T00:00:00Z+1YEAR]

This query will fetch all messages from the "ipv6" mailing list that were sent during 2019.

5.2.2 How to write queries in PySolr

Writing queries in PySolr is quite similar to the way one writes queries in the GUI, in fact, the query field itself,works exactly the same. In order to query Solr the "Solr" object from the pysolr module must be initialized, as it contains the "search" method. This method takes two input arguments, the first being a string. This is the string one would input into the GUI query field. For example, the following query, done in the GUI:

is equivalent to the following query done in pysolr:

result = solr.search("*:*",param)**

The second argument is a dictionary of parameters to apply to the query. These work in the same way they would in the GUI, except they now have to be set manually. This method gives the user access to far more parameters than the GUI does. For example, the GUI does not allow its user to specify the limit of results returned from a facet search. The code does not only that, but it also allows to set the upper and lower limit of occurrences in the results, as well as how the returned results should be sorted. The parameters also define what kind of functions one wants to utilize for the search. To activate a certain function its parameters need to be set to "on". In order to avoid having to type out the entire parameter list every time, it is possible to set a given option to

"off" to disable it. It may be a good idea to first create a "default" parameter dictionary, then copy and paste it, and adjust it as needed.

Example:

```
param = {
"debugQuery":"off",
"rows" : 10,
"facet":"on", #if this is set to "off"
"facet.field":"Date", #then no facet will be returned
"facet.limit":-1, #no facet limit
"facet.sort":"count", #sort by count
"facet.mincount":0
}
```

5.3 Solr response format

As previously mentioned Solr responds to queries in Python with a dictionary. When not using pysolr, one should specify the format the response should take. This does not refer to how the response looks, rather, what programming language will be able to read the response. It is still possible to change this, should it be needed at any point. It is also possible to change this format in the GUI, and this may be a good idea as certain formats may make the response easier to read. The variable responsible for controlling the response format is called "wt", and appears as a drop-down menu in the GUI.

5.3.1 Response Dictionary

The response received from Solr will from this point on only be discussed in the context of the Python response format.

The dictionary received from Solr has a mostly set format. As this is a dictionary, not a list, the order of the elements is irrelevant, yet it is always the same. First, the "responseHeader" key is found This key contains information about the executed query. The datastructure assigned to this key is another dictionary, and contains the following keys:

- status

- qtime
- params

"status" corresponds is HTTP code returned from the request.
"qtime" is the amount of time elapsed during the execution of the query, measured in milisecond.
"params" is the exact same dictionary that was used as the second input argument to the "search()" method of Pysolr. This can be useful for debugging.

Next is the "response" key, this is also a dictionary with a set format:

- numFound
- start
- numFoundExact
- docs

"numFound" is the exact number of documents found.
"start" is the offset in the results.The default value of this variable is 0, meaning no results are omitted.
"numFoundExact" is a boolean value, and specifies if "numFound" is an exact value or an approximate value. By default this value is set to "true". The purpose of this value is to increase performance, as Solr's documentation states:

> "Solr can skip over documents that don't have a score high enough to enter in the top N. This can greatly improve performance of search queries."

Source:
`https://lucene.apache.org/solr/guide/8_6/common-query-parameters.html`

The final key is "docs", and like the name suggests, the retrieved documents are contained within it. The data structure assigned to this key is a list, unlike the others. The documents themselves are dictionaries however.

This is the layout of the default query, without any of the augmenting options enabled. Enabling the options does not change this layout however, it only expands the returned data structure with more keys. For example, if the "facet" parameter is set to "on", the "facet_counts" key will be added to the dictionary, and the value assigned to this key is again a dictionary that will have its own values assigned to it.

This trend applies to most of the augmentation options, but as not all of them are utilized, they will not be further discussed in this document.

```
"responseHeader":{
  "status":0,
  "QTime":0,
  "params":{
    "q":"*:*",
    "_":"1603373344709"}},
"response":{"numFound":73728,"start":0,"numFoundExact":true,"docs":[
    {
      "Date":["1999-09-24T17:18:40Z"],
      "Timezone":"(EDT)",
      "From":["Majordomo@lists.research.bell-labs.com"],
      "From-name":["Null"],
      "From-address":[" Majordomo@lists.research.bell-labs.com"],
      "Sender":["Majordomo-Owner@lists.research.bell-labs.com"],
```


5.4 Faceting

One augmentation option that is heavily utilized in this project is "faceting". It is extensively used to calculate statistics, as it can be used to count the number of occurrences of tokens in a given field. Enabling the faceting option adds the "facet_counts" key to the response dictionary. Under this key yet another dictionary can be found, this time containing the following keys:

- facet_queries
- facet_fields
- facet_ranges
- facet_intervals
- facet_heatmaps

What is assigned to these keys will depend on the executed query, and for this reason, only options that were used or considered useful will be discussed.

The use of the faceting option is best explained with an example. Let's say we want to know how many messages there are in each mailing list. To get this information, one would have to count unique occurences of each token in the "Mailing-list" field.

The parameter dictionary would look like this:

```
param = {
"rows" : 0,
"facet":"on",
"facet.field":"Mailing-list",
"facet.limit":-1,
```

```
"facet.sort":"count",
"facet.mincount":0
}
```

The "rows" parameter does not have to be set to 0 in this case, but the response dictionary will be smaller this way, meaning less data has to be transferred over the network,which results in faster execution time. The "wt" parameter is also missing from the parameter dictionary. It so as pysolr handles the response formatting automatically.

Solr's response will look like the following:


```
"responseHeader":{
  "status":0,
  "QTime":2157,
  "params":{
    "q":"*:*",
    "facet.field":"Mailing-list",
    "start":"0",
    "rows":"0",
    "facet":"on",
    "_":"1603377745865"}},
"response":{"numFound":2746145,"start":0,"docs":[]
},
"facet_counts":{
  "facet_queries":{},
  "facet_fields":{
    "Mailing-list":[
      "ietf",127303,
      "i-d-announce",94429,
      "v6ops",42504,
      "httpbisa",37011,
      "sip",36625,
      "mobileip",36311,
```


5.5 Date queries

As previously mentioned, the "Date" field in the documents uses a special Solr field, with extra functionality regarding dates, and how they can be queried. It was also mentioned that this field follows a very strict syntactic format. The same holds for queries regarding this field. Recall also the difference between "Datefield" and "DateRangeField" as the difference between these two comes into play at this point.
There are 2 main ways of interacting with the "Date" field, as it is implemented in this project. The first one is querying a specific date. This is unfortunately where things get a little bit impractical. In order to query a specific date the

entire string representing a specific point in time, all the way down to seconds, has to be specified.

For example:

```
1999-09-24T17:18:40Z
```

will return all documents with this exact date. When writing queries for one specific date, the date itself needs to be "quoted". By this I mean it needs to have the " sign both at the start and end of the date.
Example:

```
"Date":["1999-09-24T17:18:40Z"]
```

The "Z" at the end must also be present. This character represents the time-zone, with "Z" specifically referring to the UTC timezone.

This is not very useful however, as one would only in very specific cases seek out messages with a date that specific. In order to write more general queries, for example, queries that look for messages sent on a specific date the "date range" portion can be utilized.
When writing date range queries, the entire date query must be contained within the "[" and "]" brackets.

The general syntax is a follows:

Date:[date1 TO date2]

Example:

Date:[1999-09-24T17:18:40Z TO NOW]

This query will return all messages sent between the specified date, and the time of querying. Another useful feature if this field is date math and keywords. Date math allows for easy calculations without doing the actual calculations ourselves. In order to simplify these calculations, it is possible to use keywords, just of these keywords are:

- NOW
- DAYS
- MONTHS
- YEARS

One can use these to easily calculate date ranges, for example:

Date:[1999-09-24T17:18:40Z TO 1999-09-24T17:18:40Z+3DAYS]

This query specifies a range of 3 day, from the given starting point. This can also be done in reverse.

Date:[1999-09-24T17:18:40Z TO 1999-09-24T17:18:40Z-3DAYS]

This query again specifies a date range of 3 days, however, now with the left hand side date being the end of the range.

Chapter 6

Calculating statistics

Now that a way to query Solr for information has been established, the actual extraction of useful data can commence. While Solr is a very powerful and efficient full text database, it lacks a lot of the more advanced features of, for example, SQL. It simply lacks commands like “max()”, "count()", “unique()” and so on. While this is not a huge problem, the job these commands would have done for us in SQL, will have to be done by hand in Python.
Those familiar with Solr may say that Solr does have a “stats” component, and they are absolutely right. This component is only available for a few select field types however. In our case, only the “Date” field has a “stats” component. So while Solr does have some support for statistics, it’s not very useful for this project.

6.1 Error rate

As previously mentioned, the parsing process is not perfect, and neither are the archives themself. The database will contain errors, be it errors caused by the parser, errors made by IETF’s archiver program, or messages that contained errors from the very beginning. In order to provide accurate statistics about the data, it is necessary to first find out how many messages contain errors in a given field. Then, what is considered an error? For this project, an error will be defined as the absence of data or existence of a default value in fields that according to RFC 5322 [Res08], are not allowed to be empty. This will essentially give us an idea of how well the data has been parsed, and the quality of the data provided by the IETF. Errors like syntax not being in accordance with the specification, and other deviations from RFC 5322 [Res08] will not be taken into account in this project, as the goal is not to correct the email archives.

Not all fields can be used to calculate error rate in this way however, like mentioned in chapter 3, not all fields are mandatory. The mandatory fields are as follows:

- Date
- From

These are the two primary fields that will be used to calculate an error rate.

The destination fields are somewhat different in this case. While none of them are explicitly mandatory, a message cannot be sent without a destination. For this reason, any message that holds a default value in all 3 of its destination fields, will be considered an error. The 3 destination fields as described in RFC 5322 are:

- To
- Cc
- Bcc

6.1.1 File statistics

All of the statistics and experiments from this point onward were done based on the IETF archies downloaded on 23. october 2020. The downloaded directory contains 80519 files, 66015 of which are .mail files, it is these files that are parsed by the parser, and which the statistics and experiments are based upon.

In this project, there are only 3 categories of files that are of any use.

1. Files that are definitely mbox files. The files with the ".mail" extension fall into this category
2. Files that do not meet the requirements of the first category, but could still potentially be mbox files. These are defined as files that do not have a file extension, or have one that does not match the first category, and at the same time start with the word "From". One thing to note about this category is that it will never contain empty files, as empty files cannot contain the word "From".
3. Files that do not have the ".mail" file extension, and do not start with the word "From".

Files in category 3 will be ignored in this project, and while it is possible that some of these are malformed mbox files, a manual check of these files shows that the vast majority of these files are index files.

The downloaded directory has a size of 30150405836 bytes (30.15GB), and these are distributed across the 3 categories as follows.

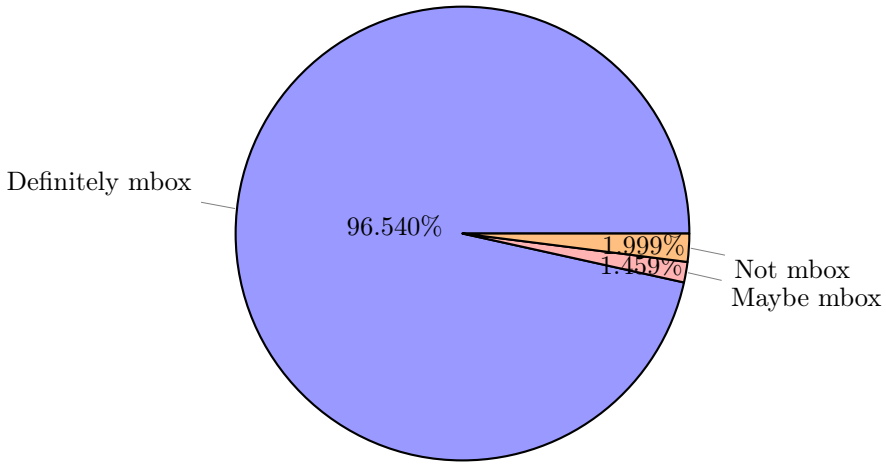

Figure 6.1: Categorization of downloaded mbox data.

96.54% of the data are categorized as 'definitely mbox' (approximately 2.9 GB), and 1.46% and 1.99% are in the 'maybe mbox' and 'not mbox' categories, respectively. A break-down of these numbers is shown in Figure 6.1

In order to determine the quality of files in category 2, two fresh Solr cores were created. Both of these cores use the exact same schema as the cores that hold files from category 1. The raw unprocessed files were parsed and uploaded to test core 1, while the files processed by the cleanarch script were parsed and uploaded to test core 2.

6.1.2 Choosing a starting point

Before statistics based on time can be calculated, a concrete point in time has to be chosen as the beginning. This is no easy task, as there are many messages with fake or incorrect dates. The earliest date cannot be set to any earlier than the default value chosen for the date field, which is first of January 1900.

One way of solving this problem is to simply set the starting point to the date on which the IETF was founded, that date being 16. January 1982. It is safe to assume that since the organization did not exist, there was no one to keep track of the messages and archive them. It is however still possible that there was some communication regarding the organization prior to this date, for this reason, the default date for all statistics has been set to first of January 1980.

In fact, by using that date as a starting point, a grand total of 13348 (0.00461% of the database) messages are lost, most of which seem to be either broken, incorrectly formatted, or simply spam.

These messages can be found by executing the following query in Solr:

Date:[0000-01-01T00:00:00Z TO 0000-01-01T00:00:00Z+1980YEAR]

These messages are still included in the statistics, they are however bundled in a "pre 1980" category, and will not be calculated on a year by year basis.

6.2 Error report

With the error definitions in place, these are the results:

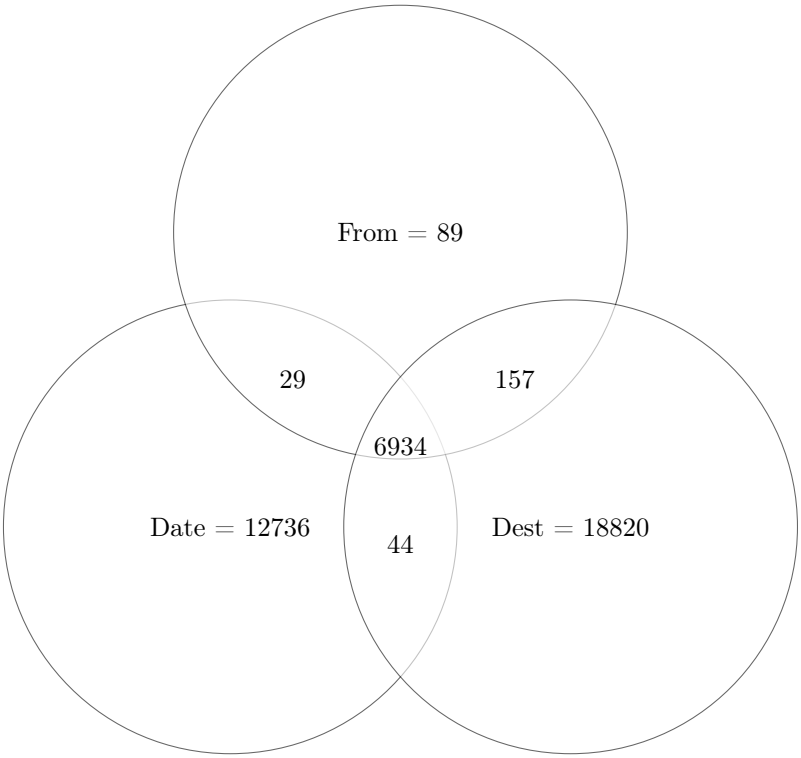

Figure 6.2: Distribution of errors across messages obtained from the raw files found in the IETF email archives. Each type of error is represented by a circle, the overlapping areas represent messages with the respective combination of errors.

There are 38 809 erroneous messages in total, and they make up 1.3322% of the entire database.

Year	From error	Destination error	Both	Total messages per year	Percent per year
1980	0	0	0	43	0.000000%
1981	0	0	0	1	0.000000%
1982	0	0	0	0	0.000000%
1983	0	0	0	0	0.000000%
1984	0	0	0	1	0.000000%
1985	0	0	0	1	0.000000%
1986	0	0	0	0	0.000000%
1987	0	0	0	16	0.000000%
1988	0	0	0	25	0.000000%
1989	0	0	0	1	0.000000%
1990	0	0	0	14	0.000000%
1991	0	0	0	2	0.000000%
1992	0	0	0	7	0.000000%
1993	0	0	0	76	0.000000%
1994	0	0	0	46	0.000000%
1995	0	3	0	125	2.400000%
1996	2	3	0	390	1.282051%
1997	0	8	0	6794	0.117751%
1998	0	63	0	32945	0.191228%
1999	0	217	4	47930	0.461089%
2000	1	360	3	61694	0.590009%
2001	3	276	0	103630	0.269227%
2002	37	1175	11	124445	0.982763%
2003	11	2319	6	139301	1.676944%
2004	9	2863	75	120907	2.437411%
2005	9	3774	1	138451	2.733097%
2006	1	4450	16	138231	3.231547%
2007	3	2728	15	211617	1.297627%
2008	5	47	1	146306	0.036225%
2009	0	22	0	170228	0.012924%
2010	0	41	0	163604	0.025061%
2011	0	230	18	129324	0.191766%
2012	0	76	0	122797	0.061891%
2013	0	17	0	132088	0.012870%
2014	6	30	2	146231	0.025986%
2015	2	16	0	133848	0.013448%
2016	0	28	3	118598	0.026139%
2017	0	43	0	130735	0.032891%
2018	0	11	2	136456	0.009527%
2019	0	5	0	126353	0.003957%
2020	0	10	0	109885	0.009100%

Table 6.1: Error rate over time.

6.2.1 The effects of Cleanarch

As previously described, cleanarch is a script developed by the IETF to correct the shortcoming of the mbox format. By simply using two Solr cores, one containing the parse of the clean data, and the other containing the raw data the effects of cleanarch can be clearly observed. The modified cleanarch script also outputs a new clean file for every .mail file in any subdirectory, meaning it is also possible to observe the effects Cleanarch has on the file size, even if this difference is not expected to be major.

Like expected the difference is negligible:

Category	Count
mail bytes	29107339665
mailcl bytes	29107362679

The difference is 0,023 megabytes.

The post parse difference however, is greater.

Type of files	Solr document count
Clean	2893656
Default	2913059

The clean archives produce 2893656 Solr documents
The default archives produce 2913059 Solr documents.
The difference is 19403 documents.

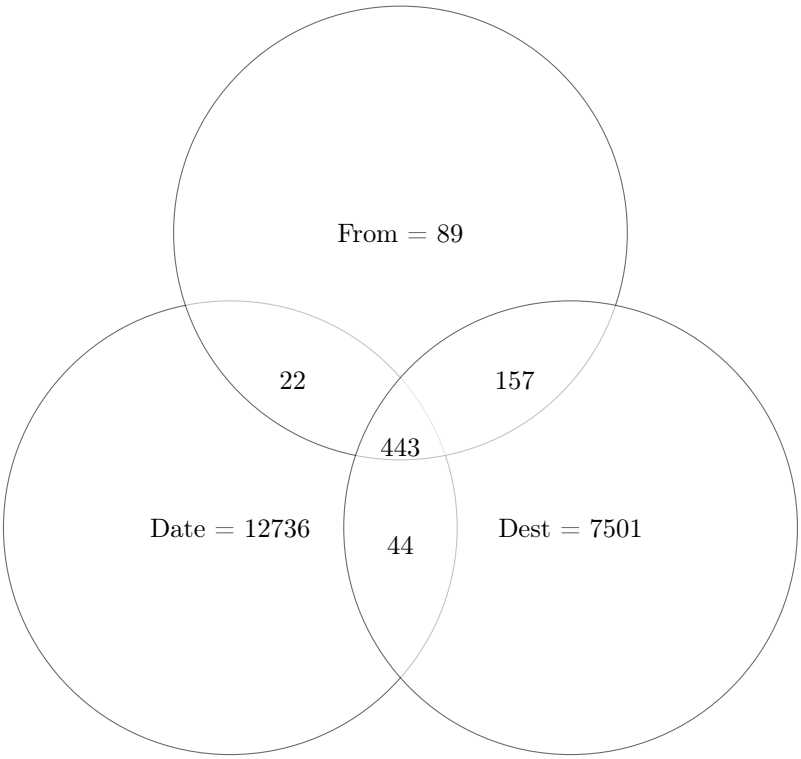

Figure 6.3: Distribution of errors across messages obtained from the clean files produced by the cleanarch program. Each type of error is represented by a circle, the overlapping areas represent messages with the respective combination of errors.

There are 20 992 erroneous messages in total, and they make up 0.725% of the entire database.

We can see that CleanArch successfully eliminates 54% of all errors, mostly in messages that have all 3 types of errors simultaneously.

6.2.2 A closer look at category 2

As previously mentioned not all files in the IETF email archive carry the ".mail" extension. This does not mean that unmarked mbox files do not exist, as we saw previously. When parsed, these unmarked files produce a total of 107136 Solr documents.

The biggest difference between category 1 and 2 is the error rate. When looking at filenames of category 2 it is quite common to see the ".broken" file extension, implying these files were excluded from the rest for a reason.

This does not mean that these files are useless however, because as long as there is some context to a message, it still holds value. Recall chapter 5, in the parsing process, some of the information is extracted from the filename,thus there will always be some fields with information. This also means that as

long as the directory structure of the archive is consistent, at the very least the mailing list of a message will always be known. However, the directory structure is not always consistent. For all category 1 files, the leaf node folder has the name of the mailing list. This means that it is safe to simply split the directory string and use the last item in the resulting list as the mailing list name.

The unmarked files do not always follow this structure, and in some cases they are contained in yet another subfolder. For this reason a small change to the parser had to be done; it now uses the n-th position in the split directory string as the mailing list, instead of using the last.

The error analysis of category 2 files returns the following results:

Error combination	Value
ALL	11219
From + Date + Dest	7297
From + Date	13
From + Dest	0
Date + Dest	2
From	12
Date	93
Dest	3802

Table 6.2: Distribution of errors in files from category 2.

The 5 largest sources of errors in category 2 are the following mailing lists:

Error combination	Value
All	7150
From + Date + Dest	7065
From + Date	1
From + Dest	0
Date + Dest	1
From	2
Date	10
Dest	71

Table 6.3: Stats for mailing list avt.

Error combination	Value
All	2529
From + Date + Dest	0
From + Date	0
From + Dest	0
Date + Dest	0
From	0
Date	0
Dest	2529

Table 6.4: Stats for mailing list isis-wg

Error combination	Value
All	756
From + Date + Dest	9
From + Date	0
From + Dest	0
Date + Dest	0
From	0
Date	0
Dest	747

Table 6.5: Stats for mailing list capwap.

Error combination	Value
All	293
From + Date + Dest	0
From + Date	0
From + Dest	0
Date + Dest	0
From	0
Date	0
Dest	293

Table 6.6: Stats for mailing list tap.

Error combination	Value
All	165
From + Date + Dest	130
From + Date	0
From + Dest	0
Date + Dest	0
From	9
Date	26
Dest	0

Table 6.7: Stats for mailing list i-d-announce.

These 5 mailing lists are responsible for 97% of all errors in category 2 files. The remaining mailing lists contain between 0 to 3 errors each, for the full results please see the github repository.

6.2.3 Cleaning category 2

As with category 1 the cleanarch script was used to try and fix errors in the mbox files. The results were quite different from category 1. The most significant difference is the decrease in documents count. As the purpose of cleanarch is to fix incorrectly separated messages in a mbox file, the expectation is that the amount of documents should increase, not decrease.

The raw files generate 107136 files after parsing. The clean files generate 106030 files, which is 1106 fewer. The error analysis tells a similar story:

Category	Value
From + Date + Dest	7193
From + Date	13
From + Dest	0
Date + Dest	2
From	11
Date	93
Dest	2811
ALL	10123

Table 6.8: Distribution of errors in clean files from category 2.

As we can see the error count has dropped by 1106, which is very close to the total decrease in document count, which is 1096. This could mean that instead of fixing boundaries between messages, it actually does the opposite.

The average length of the “Content” field in the raw files is 2479.721. The same average for the clean files is 2483.450 Which confirms that cleanarch does indeed make messages longer in this case.

6.2.4 Error rate over time

The error rate over time in category 2 raw files is as follows:
(Only years with errors are shown)

Year	Error count
1993	1
1994	1
1995	1
2006	9

Table 6.9: Errors in the "From" field found in the raw category 2 files.

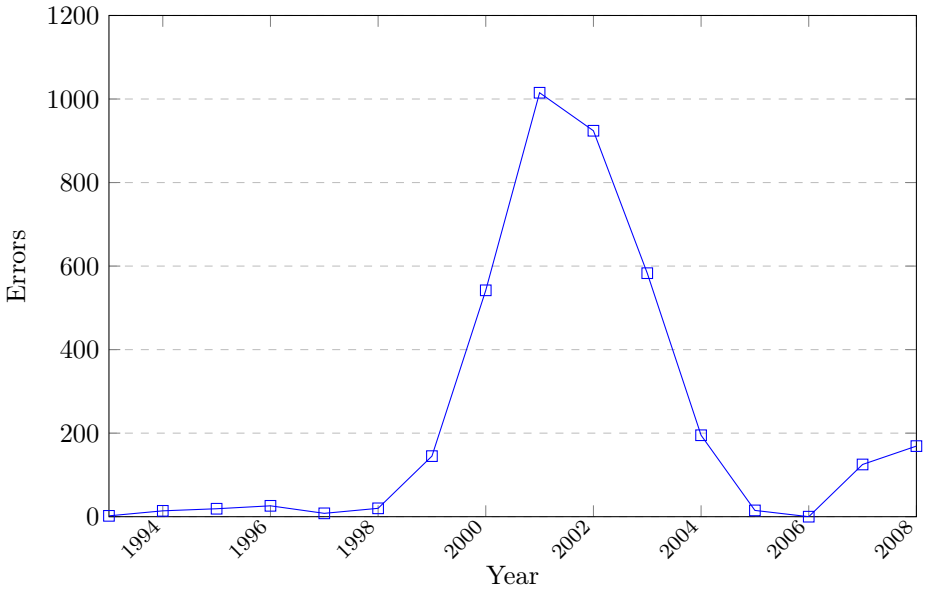

Figure 6.4: Destination errors per year

The results show errors from the years they first appeared to the last years they appeared. It is also important to remember that these can only show errors in messages that do not also have an error in their date field. As just the “avt” mailing list has 7076 messages with date errors, the amount of messages depicted in these tables is significantly smaller than the total amount of messages with errors.

In conclusion, category 2 files have a much larger frequency of errors than their category 1 counterparts. Those errors however are concentrated in just 5 out of 118 mailing lists found in these files. For this reason it was decided that adding these files to the main Solr core would still be beneficial to this project. All statistics from this point on will be calculated using a core containing clean files from both category 1 and 2.

6.3 Final error rate

The final error rate in the core used for calculating statistics from this point in is depicted in the figure 6.5.

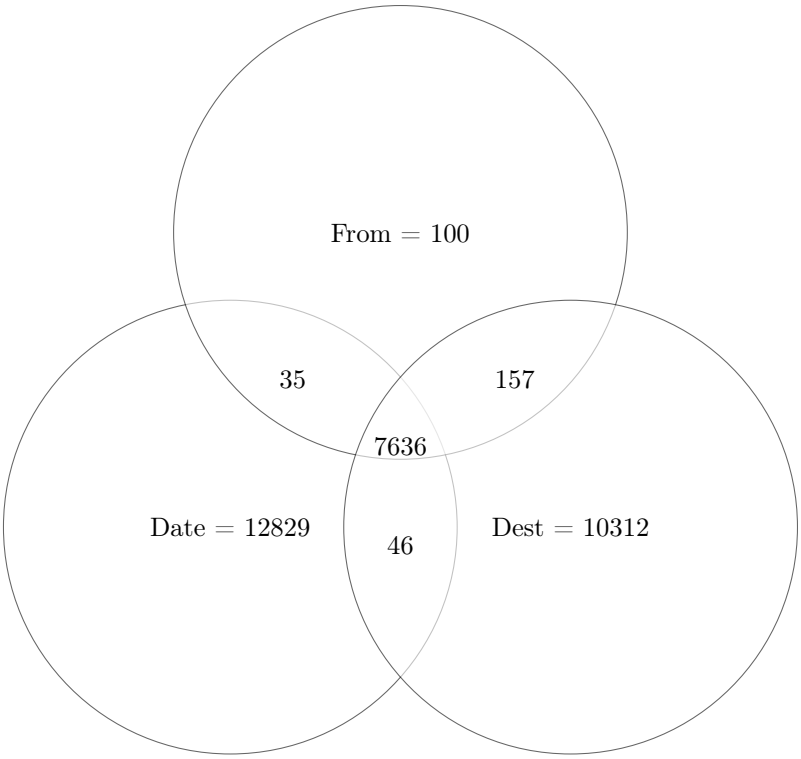

Figure 6.5: Error rate in the main and final Solr core.

6.3.1 Implementing some aggregation functions

Like mentioned Solr does not support sql functions like min(), max(). This is not entirely correct however, Solr has support for these functions, they can be used on the various fields in the documents. For example

max(field1,field2)

However, they cannot be used on the results returned by a facet field query. For this reason more general versions of these functions have been implemented to streamline the calculation of statistics. All of the functions below take a Solr object as the first arguments, so a connection has to be established before these functions can be used. The following methods have been implemented:

max(solr,field)

min(solr,field)

avg(solr,field)

unique(solr,field)

6.4 General statistics

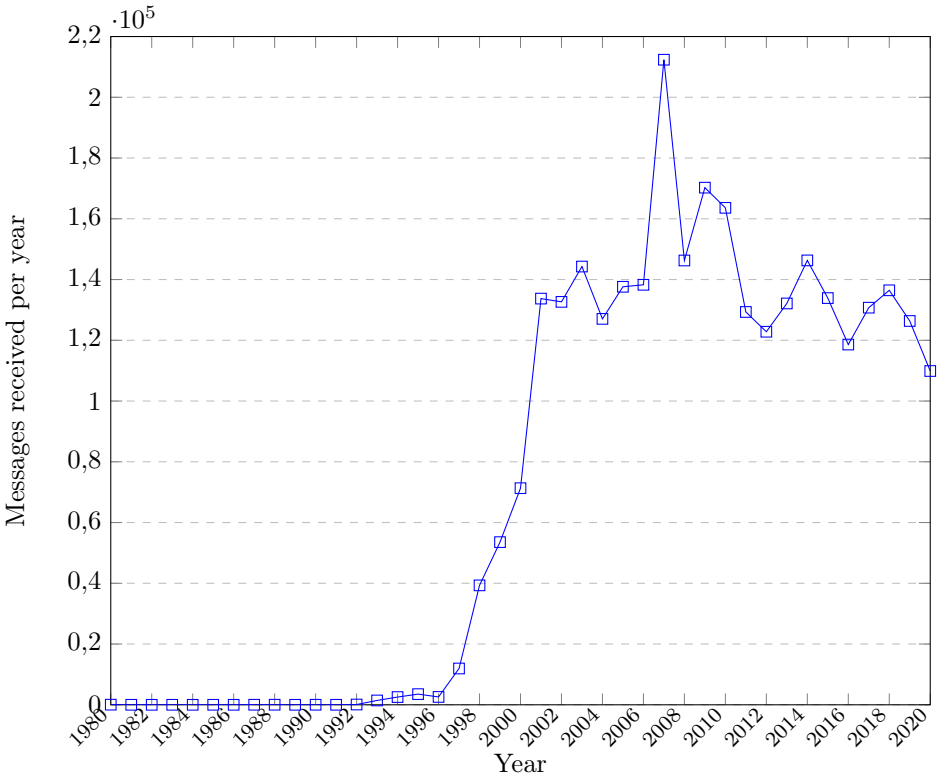

Figure 6.6: Messages were sent per year.

As we can see in figure 6.6, there are close to no messages sent between the years 1980 and 1990. This may be due to the archival system not being as well developed as it is today, resulting in a major loss of data. Another possibility is that email was not as widely used as it is today. During this period communication media like fax, phone calls, and paper mail were more commonly used. Since this decade is essentially devoid of data, the earliest year included from this point on is 1990.

6.4.1 Top 20 contributors

Ranking	address	Value
1	internet-drafts@ietf.org	190901
2	notifications@github.com	41934
3	iesg-secretary@ietf.org	27170
4	black_david@emc.com	14365
5	rfc-editor@rfc-editor.org	14198
6	brian.e.carpenter@gmail.com	13491
7	marcelrf@bellsouth.net	11485
8	julian.reschke@gmx.de	10500
9	christer.holmberg@ericsson.com	10230
10	noreply@github.com	8953
11	stephen.farrell@cs.tcd.ie	8567
12	martin.thomson@gmail.com	8522
13	cabo@tzi.org	7915
14	dromasca@avaya.com	7606
15	jari.arkko@piuha.net	7482
16	j.schoenwaelder@jacobs-university.de	7306
17	feedback@oneoffice.jp	7194
18	paul.hoffman@vpnc.org	7108
19	mcr+ietf@sandelman.ca	7101
20	mnot@mnot.net	7043

Table 6.10: Top 20 posters.

As we can see in the graph and table above, there are some email addresses that do not seem to belong to any one individual, instead their names imply that they are automatically generated messages. There is no surefire way to know which messages exactly come from some form of automatic messaging system, it is however possible to make an educated guess. For this reason, a blacklist of addresses that are believed to not belong to individuals and found in the top fifty most active addresses, has been created in order to improve the quality of the data displayed. This blacklist was manually created by the author with help from Michael Welzl..

The blacklist contains the following addresses:

- internet-drafts@ietf.org
- notifications@github.com
- noreply@github.com
- iesg-secretary@ietf.org
- ietf-secretariat-reply@ietf.org
- rfc-editor@rfc-editor.org
- noreply@ietf.org

- trac@tools.ietf.org
- feedback@oneoffice.jp

With these addresses excluded, the top 20 table looks quite different:

Ranking	address	Value
1	black_david@emc.com	14365
2	brian.e.carpenter@gmail.com	13491
3	marcelrf@bellsouth.net	11485
4	julian.reschke@gmx.de	10500
5	christer.holmberg@ericsson.com	10230
6	stephen.farrell@cs.tcd.ie	8567
7	martin.thomson@gmail.com	8522
8	cabo@tzi.org	7915
9	dromasca@avaya.com	7606
10	jari.arkko@piuha.net	7482
11	j.schoenwaelder@jacobs-university.de	7306
12	paul.hoffman@vpnc.org	7108
13	mcr+ietf@sandelman.ca	7101
14	mnot@mnot.net	7043
15	adrian@olddog.co.uk	6811
16	randy@psg.com	6316
17	ekr@rtfm.com	6135
18	john-ietf@jck.com	5979
19	housley@vigilsec.com	5910
20	fred@cisco.com	5857

Table 6.11: Top 20 posters, after filtering out the blacklisted addresses.

From this point on, the graphs and tables in this thesis will have these addresses excluded from them, unless explicitly stated otherwise.

6.4.2 Top 10 mailing lists

Ranking	address	Value
1	ietf	134157
2	i-d-announce	103544
3	quic-issues	54104
4	v6ops	44450
5	ips	40603
6	avt	40454
7	dmarc-report	40007
8	ipv6	38591
9	httpbisa	38133
10	mobileip	37475

Table 6.12: Most active mailing lists, measured by the total amount of messages sent.

6.4.3 Top mailing list per year

Year	Mailing-list	Count
1990	webdav	3
1991	idn	1
1992	ipsec	78
1993	avt	1067
1994	avt	1650
1995	ipsec	1638
1996	avt	1540
1997	ietf	2574
1998	ietf	13383
1999	ietf-announce-old	7177
2000	sip	5739
2001	ips	25894
2002	mobileip	7591
2003	mobileip	15731
2004	marid	8742
2005	ietf	6271
2006	i-d-announce	7151
2007	ietf	6313
2008	i-d-announce	5072
2009	i-d-announce	5428
2010	i-d-announce	6342
2011	i-d-announce	6477
2012	i-d-announce	6985
2013	ietf	8614
2014	i-d-announce	7148
2015	i-d-announce	7044
2016	i-d-announce	6696
2017	quic-issues	9143
2018	dmarc-report	16635
2019	quic-issues	15433
2020	quic-issues	14328

Table 6.13: Most active mailing list per year, measured by the total amount of messages sent per year.

6.5 Crosstalk

In order to gain a better understanding of how working in the IETF is done, it is important to understand how working groups and people interact with each other. There are many ways of defining what an interaction, in the context of this project, is. For this reason several options have been defined and explored.

Definition 1

An interaction is defined as the act of sending a message to a mailing list,

performed by any entity.

This definition essentially says that if an address appears in the "From" field, it is considered an interaction from an entity identified by the address found in this field.

The simplest way of checking how many mailing lists an entity has interacted with is checking how many different mailing lists an email address appears in.

While this method does not give any specifics regarding groups or people, it sheds some light on the average crossinteraction per email address, whomever that address may belong to. The database also makes it possible to see how this has developed over time.

6.5.1 The results

Based on all unique values found in the "From-address" field, each value appears on average in 1.52 mailing lists. The following graph shows how this developed over time.

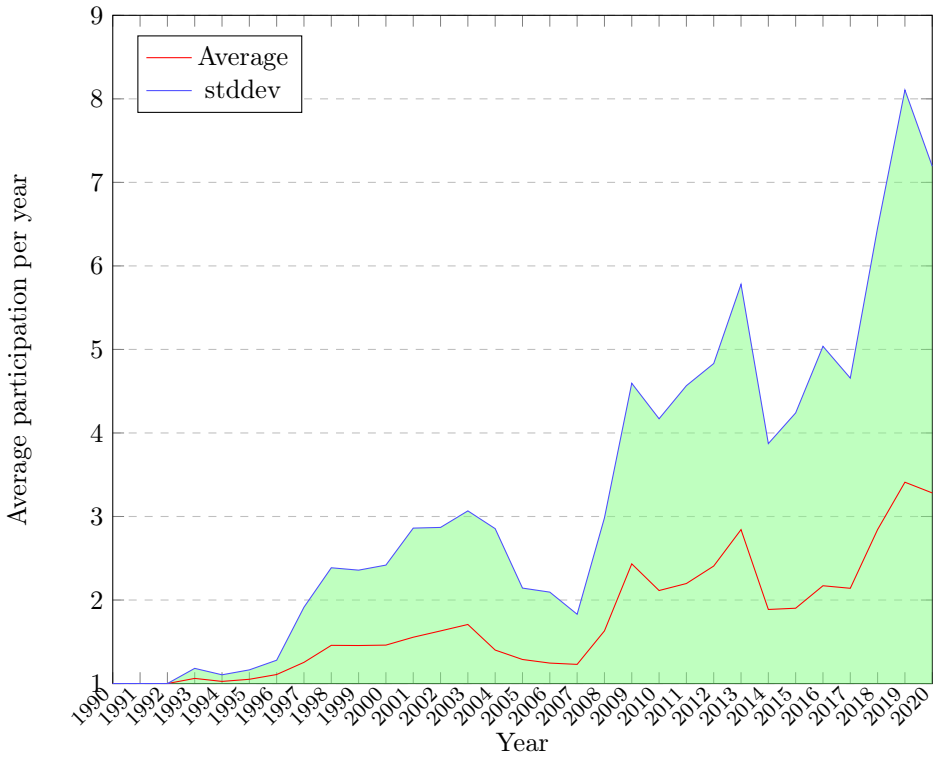

Figure 6.7: The number of mailing lists an email address appears in, per year.

How is this distributed? Figure 6.8 shows how many entities are participating in how many working groups, and figure 6.9 plots the same data in the form of a Cumulative Distribution Function (CDF).

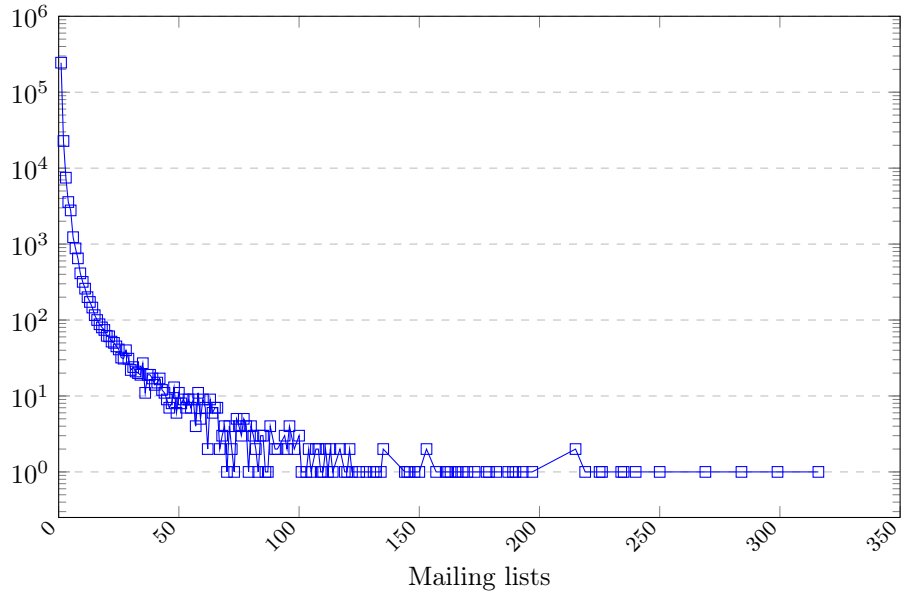

Figure 6.8: The number of users sending messages to one or more multiple mailing lists.

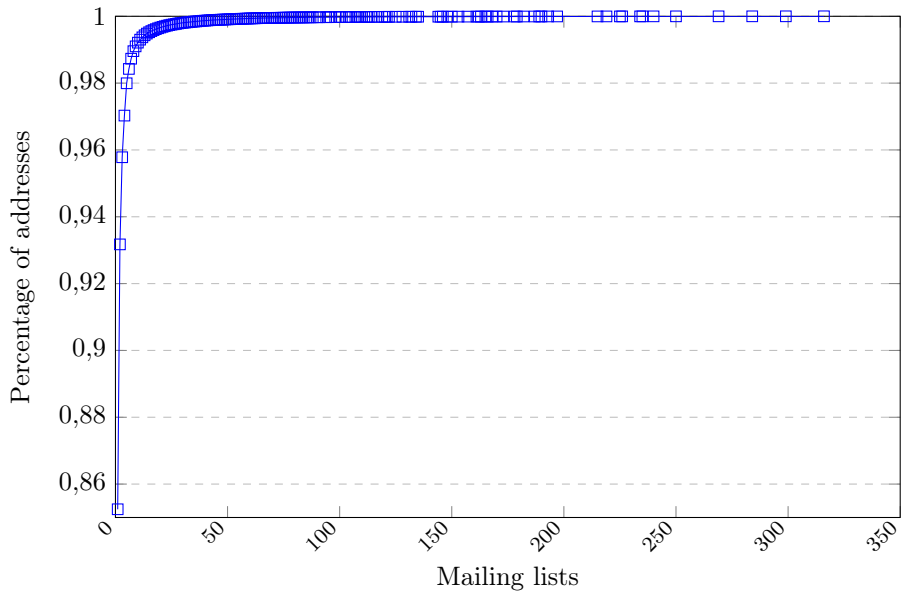

Figure 6.9: Cumulative Distribution Function (CDF) of mailing lists an email address apears in.

As we can see the the vast majority of entities only ever contact a few mailing lists.

Considering the 0 to 50 range in the figure 6.8 and 6.9, how many messages on average does an entity that is contacting x different mailing lists send?

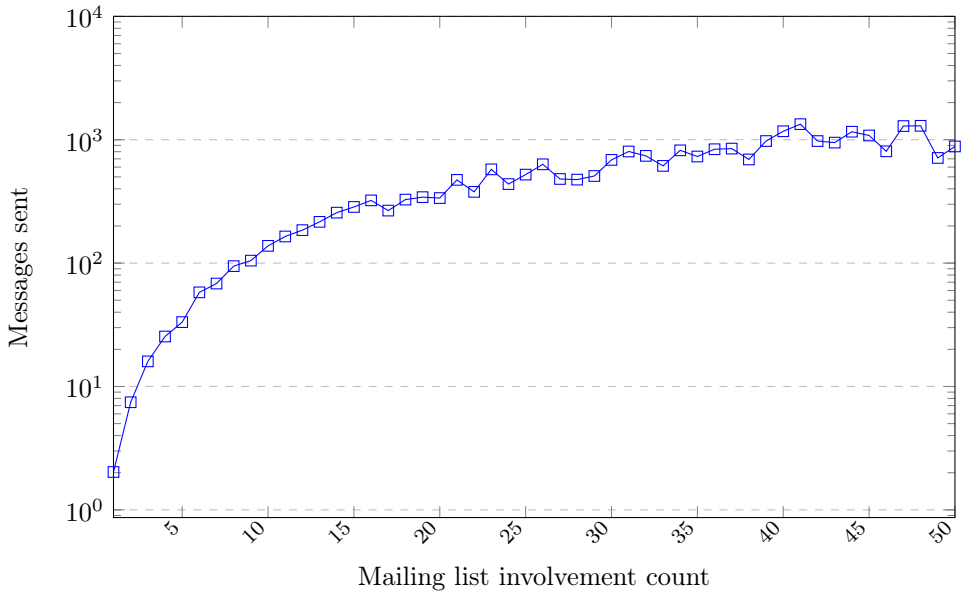

Figure 6.10: Messages sent by email addresses that appear in 1-50 mailing lists.

The graph flats out at around 1000, indicating that even if someone interacts with a wide variety of mailing lists, the number of messages sent, stays around the same number. This may be due to the limited amount of time any given person can spend sending emails.

6.5.2 Calculating cross talk

From the figure 6.10 we can see that on average an entity interacts with more than 1 mailing list, and just calculating the average in this case does not tell the entire story. In order to get a better look at cross-mailing-list interactions, some adjustment had to be made to both the code, and the definitions:

Definition 1

Cross talk in the context of this project is defined as "an interaction by an entity with a mailing list that is not its home".

Definition 2

An entity's home mailing list is the mailing lists in which its identifier (in this case, the email address) appears the most.

With the definitions in place, calculating crosstalk is very similar to calculating the average. The only difference between the 2 is the fact that the most

frequent component is ignored in one of the plots.

Finally, on average how many messages are sent to mailing lists that are not considered the home of an entity, and how has this developed over time?

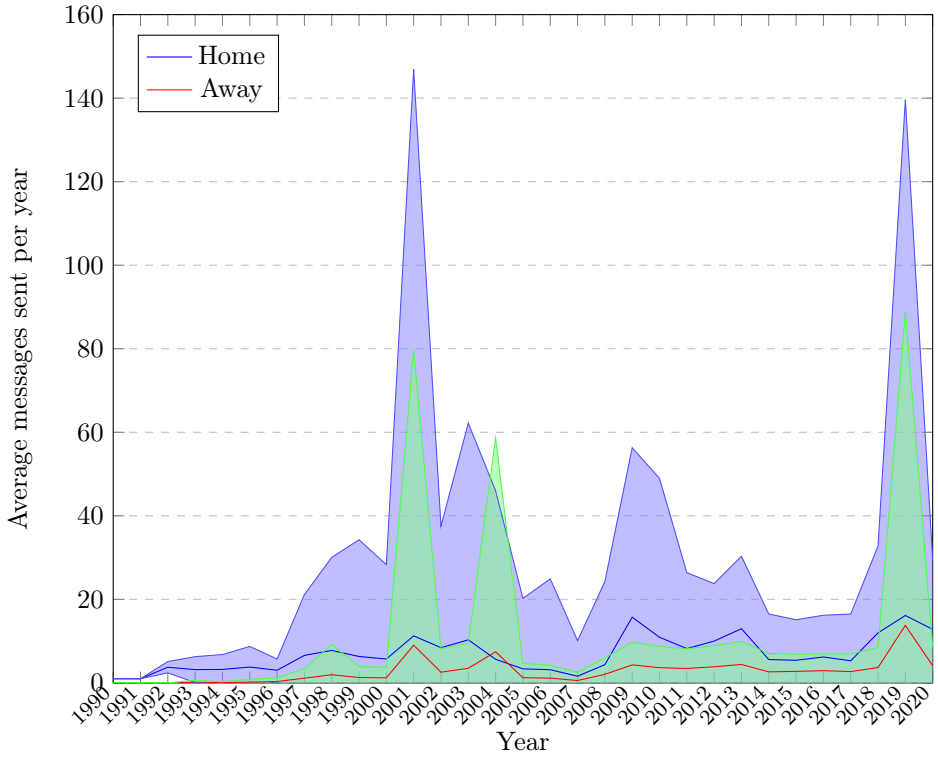

Figure 6.11: Messages sent to an address's home mailing list, as well as any other mailing lists that are not their home. Blue and green shaded areas show the standard deviation for the "Home" and "Away" cases, respectively.

Chapter 7

Tracking threads

7.1 Motivation

In order to gain better understanding of how RFCs come to be, it is vital to know what kind of discussion usually ends with the publication of a new RFC. What kind of arguments were used? What kind of response did said arguments evoke? Were they positive, negative, or somewhere in between?

Before these questions can be answered a definition of a conversation thread needs to be specified. The simplest way of defining a conversation thread is by using the "reply-to" and "in-reply-to" fields normally found in email messages. Like discussed in a previous chapter, these fields directly refer to another message that was sent at an earlier point in time.

The most recent RFC defining these fields and their inner workings is RFC 5322 [Res08], which describes these fields as follows

> *The "In-Reply-To:" and "References:" fields are used when creating a reply to a message. They hold the message identifier of the original message and the message identifiers of other messages (for example, in the case of a reply to a message that was itself a reply). The "In-Reply-To:" field may be used to identify the message (or messages) to which the new message is a reply, while the "References:" field may be used to identify a "thread" of conversation.*
>
> *When creating a reply to a message, the "In-Reply-To:" and "References:" fields of the resultant message are constructed as follows:*
>
> *The "In-Reply-To:" field will contain the contents of the "Message-ID:" field of the message to which this one is a reply (the "parent message"). If there is more than one parent message, then the "In-Reply-To:" field will contain the contents of all of the parents' "Message-ID:" fields. If there is no "Message-ID:" field in any of*

the parent messages, then the new message will have no "In- Reply-To:" field.

The "References:" field will contain the contents of the parent's "References:" field (if any) followed by the contents of the parent's "Message-ID:" field (if any). If the parent message does not contain a "References:" field but does have an "In-Reply-To:" field containing a single message identifier, then the "References:" field will contain the contents of the parent's "In-Reply-To:" field followed by the contents of the parent's "Message-ID:" field (if any). If the parent has none of the "References:", "In-Reply-To:", or "Message-ID:" fields, then the new message will have no "References:" field.

Note that in the definition of the "references" field, it is stated that it can be used to track a "thread" of conversation, exactly what we are trying to do here. However, considering the note following the definition:

Note
Some implementations parse the "References:" field to display the "thread of the discussion". These implementations assume that each new message is a reply to a single parent and hence that they can walk backwards through the "References:" field to find the parent of each message listed there. Therefore, trying to form a "References:" field for a reply that has multiple parents is discouraged; how to do so is not defined in this document.

Specifically, we cannot assume that a reply only has a single parent. Like mentioned, a thread may split at any moment for any reason, it is for this reason safer to use the "In-Reply-to" field to track threads.

7.2 Definitions

Before further discussing the challenges that arise when trying to track conversation threads, some definitions have to be put in place. These definition are as follows:

7.2.1 Node

A node is defined as a unique message id.

7.2.2 Parent

A parent is defined as the set of message ids contained within any given messages "In-Reply-To" field. No further restrictions are placed upon this definition,

meaning that for example, a node is perfectly capable of being its own parent.

7.2.3 Child

A child is defined as a set of message ids belonging to messages that have a message id in their "In-Reply-To" field. Like with the definition of a "Parent", a node can be its own child.

7.2.4 Stub

A stub is a node with no connecting edges.

These definitions are based on the intuitive understanding of how a conversation works, and are also based on the underlying assumption that a reply to a message cannot take place before the original message is created.

With all these definitions in place, the final product will be a one or several bidirectional graphs. From here graph traversal algorithms like Djikstras can be utilized to collect information about conversations.

7.3 Extrapolating edges

In a perfect world there would always be a bidirectional edge from one node to another, or no edge at all. This is however not the case. This can be due to data errors, corruption, a parsing error, or the data simply missing from the very beginning. It is safe to assume that if any given node A is child of node B, then node B is also parent of node A. The child of a node will always be younger than its parent and vice versa. For this reason, a bidirectional edge will be added between A and B as long as the criterion for at least one direction is present.

7.4 Implementing the thread tracker

Implementing the thread tracker is best done by utilizing Solr as much as possible. The first challenge one will encounter is the fact that we do not know at what point in a thread a given message is without further investigation. This means that the tread tracker needs the ability to "grow" the thread both forwards and backwards. This is already made possible by the definitions above.

The direction in which the thread tracked program decides to "grow" first is irrelevant, as the process does not create new nodes; rather, it links them. The process does not alter the existing data in any way, it simply creates new

information based on what it already knows.

As previously mentioned, there is no real way of knowing when a "thread", in terms of conversation related to a specific topic ends. It is absolutely possible that a thread, as defined in this project, continues for a very long time, and discusses many different topics. One could say that the change of topic happens when the "Subject" field changes, and while this is not an unreasonable suggestion, there is no way to know for sure. There is also the problem of replies adding a "Re:" and other string to a "Subject" field without any pre-defined format. For this reason the "Subject field will not be taken into account for now.

It is also important to note that the result of this endeavour heavily relies on the "In-reply-to" field actually having data in it, currently the value "null" is present in 1368596 out of 2893656 documents, that being roughly 47% of all messages. This does not necessarily mean that the data is bad, the "in-reply-to" is not a mandatory field, so the absence of data cannot be considered an error, it is perfectly possible that a lot of messages are simply never replied to.

7.4.1 Representing the results

Since for the time being, only the "In-reply-to" field will be used to track down threads, a timeline representation makes some sense, as it logically speaking makes sense that a reply to a message must have been sent at a later point in time than the message being replied to.

This fact can potentially be used to extrapolate dates of messages that are missing them. For example, if message A was sent on the first of november 2020, and message B, which is a reply to message A, does not have a proper date, we at least know that it is older than message A. If in addition, there is a message C, that is a reply to message B, we get a specific period of time where message B must have been sent. This of course will not yield a perfectly accurate time, however, being able to tell the year and month when a message was sent is still a huge improvement over not being able to tell anything.

The problem with a timeline is the fact that a message can have several replies that may have been sent at different points in time. This fact makes a tree representation more suitable. This representation has however another weakness: for a tree to meet the requirement of being a tree, the nodes on a given level cannot be connected to each other. This is not a property that can be guaranteed in this case due to the uncertain nature of the data at hand. With that in mind, a more accurate representation would be a graph as a graph does not have to maintain the property mentioned previously. In a graph, one is free to connect nodes in any way one pleases. For all these reasons the graph representation has been chosen.

7.4.2 The thread tracking program

The following is a high level description of how the thread tracking program works.

- First, using Solr, obtain a list of all message ids in the database. This of course excludes all messages that do not meet this requirement, as mentioned previously.

- Then for each message id, create a node object that will represent the message with a specific id. Then, for each node object, query Solr to obtain message ids for the nodes parents and children. Finally print all to a file and upload the results to a Solr core.

7.4.3 The code, phase 1

The first step in creating a thread tracking program is building the conversation graph. The graph building program calculates both parents, and children of all nodes. This is all finally saved to a structured file ready to be read and processed further by another program.

One thing to note here is that this step is strictly speaking, not necessary as all needed information regarding a node's parent and children is already stored within Solr. The process of extracting and parsing this information from Solr is a very computationally heavy task that takes several hours to complete. Essentially, what the following code is doing, can in a way be seen as caching, in order to speed up future endeavours.

Each node is stored as the following object.

```
class node: #the node class
    def __init__(self, id,parents,children,seen):
        self.id = id
        self.parents = parents
        self.children = children
        self.seen = seen
```

id being the message id, ***parent*** being message id's of the parent messages, ***children*** being a list of message id's of the children, and ***seen*** being simply a way to mark already seen nodes when traversing the graph.

7.4.4 get_parent()

This is the first of 2 main methods. As the name suggest, it is this method that is responsible for calculating a given node's parents.

The query parameters are as follows:

```
def get_parent(node_inn,solr):
```

It takes two input arguments, the current node to be used as the base of the calculations, and an already set up Solr object that will be used to interface with a Solr instance of choice.

```
param = {"debugQuery":"off",
"rows" :1,
"fl":"In-Reply-To-address"
}
```

Notice that this query does not make use of the facet function, and is limited to returning only 1 document at most. The underlying assumption here is that even if there are several copies of the same message in the database, they should only differ in the content of their "Mailing list" field, as all other fields are pulled directly from the mbox file.

The "fl" parameter is also set to the "In-Reply-To-address" field in order to limit the amount of data that needs to be sent by Solr, and hence quicken execution significantly.

Next is a series of attempts at obtaining the best results from the database. A Solr query, in this case, can fail in two different ways. The first is a syntax error, these are usually caused by illegal characters, and cause an exception to occur. The other one, is the results set being empty. This is considered an error in this case, as all queries executed by this program are based on the message ids it has obtained at the start of execution from the same database, meaning that every message id should at the very least have one message associated with it. Both of these errors, however, have the same solution. The solution is the message id with quotation marks, as this makes Solr parse the query string as one "unit". This is still not enough, as some message id-s contain a colon. This is a reserved character that cannot occur in queries, unless escaped with a backslash. Any backslashes that are in the original message id also need to be escaped with the same backslash.

If any errors are encountered on the first try, the message-id is slightly modified as described above, and a new attempt is made. If this attempt also fails, the message id is logged as an error, printed to the terminal, and printed to a log file at the end of execution for further investigation.

```
res = solr.search("Message-ID:" + "\"" + node_inn.id + "\"",**
    ↪ param)

if res.raw_response["response"]["numFound"] == 0:

    try:
        res = solr.search("Message-ID:" + "\"" + node_inn.id
            .replace(":","\:").replace("\"","\\\"") + "\"",**param
                ↪ )
```

Recall that the fl parameter refers to the "In-Reply-To-address" field in the database, now recall from chapter 5, that said field is a multivalued field, which Solr returns as a Python list. If a satisfactory response is obtained from Solr, this list is iterated over to check if an element in the list is a message id that matched one of the messages obtained at the start of execution. This may sometimes fail as message ids are usually surrounded by a less-than and greater-than sign. This is in some cases removed during parsing. In order to ensure the best

possible result, a second attempt at matching the element is made; this time, with the surrounding characters added back in. If the second attempt fails, the element is ignored.

```
for x in res.raw_response["response"]["docs"][0]["In-Reply-To-
    ↪ address"]:

    if x in nodes:

        node_inn.parents.add(x)
        nodes[x].children.add(node_inn.id)


    else:

        if ("<" + x + ">") in nodes:

            node_inn.parents.add("<" + x + ">")
            nodes["<" + x + ">"].children.add(node_inn.id)
```

7.4.5 get_child()

This is the second method, and as the name suggests its purpose is to get the children of a given node. Its input arguments are the exact same as with the previous method, the node whose children we want to calculate, and a Solr object that will be used to communicate with Solr. In contrast to the get_parent() method, this one utilizes Solr's facet function. The following is the parameter list used for queries in this method

```
param = {"debugQuery":"off",
"rows" :0,
"facet":"on",
"facet.field":"Message-ID",
"facet.limit":-1,
"facet.sort":"count",
"facet.mincount":1
}
```

The get_child() method follows the same philosophy in terms of obtaining the best possible result from the database, meaning it makes several attempts if the previous one was not satisfactory.

The first attempt is made with the unmodified message id found in the node object passed to the function. If this query returns 0 results, one of 2 actions is taken based on the format of the id string. Like mentioned in the previous method, message ids are usually surrounded by a greater-than and less-than sign like so "<messageid>". These 2 surrounding characters are in some cases removed during the parsing process. This method queries the "In-Reply-To-address" field, meaning that using values from the "Message-ID" field directly has a chance of not actually matching anything. If this happens and the message

id has the previously mentioned format, a new attempt is made with the first and last character of the message id removed, as well as the illegal characters.

```
if res.raw_response["response"]["numFound"] == 0:

    if node_inn.id[0] == "<" and node_inn.id[-1] == ">":

        try:
            res = solr.search("In-Reply-To-address:" + "\"" +
                ↪ node_inn.id[1:][:-1].replace(":","\:") + "\"",**
                ↪ param)
```

If more than 2 attempts fail, the message is printed to a file as well as the terminal for further investigation at a later point.

Finally, since the facet function is used, the list returned by Solr needs to have the facet counts removed. While this is happening, the program, like in the "get_parent()" method, matches the message ids against the very first list of message ids obtained at the start. If a match is found, the message id is added to the parent list of the input node.

```
while i < len(res.raw_response["facet_counts"]
    ["facet_fields"]["Message-ID"]):

    if res.raw_response["facet_counts"]
        ["facet_fields"]["Message-ID"][i] in nodes:

        node_inn.children.add(res.raw_response["facet_counts"]
            ["facet_fields"]["Message-ID"][i])

        nodes[res.raw_response["facet_counts"]
            ["facet_fields"]["Message-ID"][i]].parents.add(
                ↪ node_inn.id)


    i = i + 2
```

With these two methods in place, it is now possible to build a graph of conversations held across the IETF.

7.4.6 Removing stubs

There is room to improve this approach. First of all, since the goal is to track a conversation, nodes that have no incoming or outgoing edges can be omitted, as they can only be accessed from themselves, and are not part of any other tree/graph. Doing this will decrease the amount of data that needs to be stored.

This is done by the thread tracking program right after phase 1, when writing the results to a file.

7.4.7 The save format

Due to the uncertain nature of data in this project, a well structured way of saving the graphs and nodes to a file had to be created. This format ensures

that all nodes and graphs are read and written correctly. The format in both nodes and graphs is based on delimiters that were chosen by trial and error, and do not appear in the "Message-ID" field at any point. If these characters are to appear in any future iterations of the database, the save format will have to be adjusted to account for this.

The delimiter characters, and what they mean is as follows.

Nodes

Δ = start of a node, this character is always followed by the message-id of said node.

λ = marks the end of a nodes message-id.

Γ p = marks the start of a node's parents, all following nodes are to be assigned accordingly

Γ c = marks the start of a node's children, all following nodes are to be assigned accordingly

α = marks the start of a message id that is either a child or a parent of the current node

Ω = marks the end of a message id that is either a child or a parent of the current node

§ = marks the end of a node. When this character is encountered, the node should have been read in its entirety. This character is always followed by two empty lines.

Example:

Δ
<200207032102.g63L2B203351@astro.cs.utk.edu>
λ
Γp
α
<5.1.0.14.2.20020703172727.03e210d0@joy.songbird.com>
Ω
Γc
α
<A8SJGB6GEA6YSYUNJ1ZLFDA1U3YTRNL.3d25477c@www>
Ω

α
<20020708100834.A13615@bailey.dscga.com>
Ω
α
<5.1.0.14.2.20020704000946.039f2780@joy.songbird.com>
Ω
§

Graphs:

As previously mentioned, trees are ascribed their own unique id, this id is always the first thing that is read, and also function as a delimiter.

—**(int)** = The start of a tree, the "(int)" can be replaced with any integer, this line is always followed by an empty line, then another integer signifying the current level.

α = Same as nodes

Ω = Same as nodes

Δ = Signifies the end of a level

§ = Signifies the end of a tree, this character is always followed by exactly 4 empty lines

7.4.8 Solr cores

Two new Solr cores were made as well in order to accommodate the results of this calculation. The cores are aptly named "graphs" and "nodes". Documents contained within the "graphs" core only have four fields, two of which are automatically added by Solr, and bear no difference on the other fields:

Graphs:

- *graph-id*
  This field contains the id number of a given graph
- *nodes*
  This is a multivalued field, it contains message id's of all nodes that belong to a graph.

The two fields added by Solr are "id" and "version".

The "nodes" core contains documents that have 5 or 6 fields. Since Solr is perfectly capable of searching for documents that are missing a specific field, it

was chosen to omit default values for fields in this core. For example, if a node does not have any parent, instead of its "parent" field containing a default value, which was up to now "Null", the field will simply be missing. In order to query Solr for documents that do not have a certain field, a "-" sign must be placed before the name of the field when writing the query. For example, the following query will return all documents without the "parent" field.

-parent:*

With that cleared up, the documents in the "nodes" core have the following fields.

- node-id
  This is the id that uniquely identifies a message. For each unique value in this field, there is at least one message in the main database core. The same message can appear in several places. Announcement messages for example, sent to several mailing lists, will have several "copies" in the database, with the difference being the contents of their "Mailing-lists" content. There could be other differences as well since the messages were parsed from different files, the parsing process itself may have had varying degrees of success in extracting the data. This is just one of many instances where the database could be improved upon. Tracking down copies of the same messages in different places, then using the extracted data to form one complete document. The "Mailing-list" field could for example be changed to a multivalued field that would contain names of all mailing lists said message had been found in. Doing this may also improve the results of tracking threads.

- graph-id
  This field simply indicates which graph a node belongs to. For each node, there is exactly one graph it belongs to.

- Parent and child
  These two fields are both functionally equal. They are both multi valued fields in the database, and contain message ids of other nodes. Notice the difference between a node-id and a message-id. A message-id, as found in the main database core, is a value directly extracted by the parser, belonging to a message. A node id is a message-id that has at least one connection/edge to another node. For each node-id, there is always a corresponding document in the main core, this is however not the case the other way around.

The two automatically added fields are present here as well.

Finally, the two cores have the following tokenizers for their non-standards fields:

Graphs:

Field	*Tokenizer*
graph-id	No tokenizer
nodes	solr.KeywordTokenizerFactory

Nodes:

Field	*Tokenizer*
node-id	solr.KeywordTokenizerFactory
graph-id	No tokenizer
parent	solr.KeywordTokenizerFactory
child	solr.KeywordTokenizerFactory

With the layout of the nodes in place, here are some statistics regarding the data contained within.

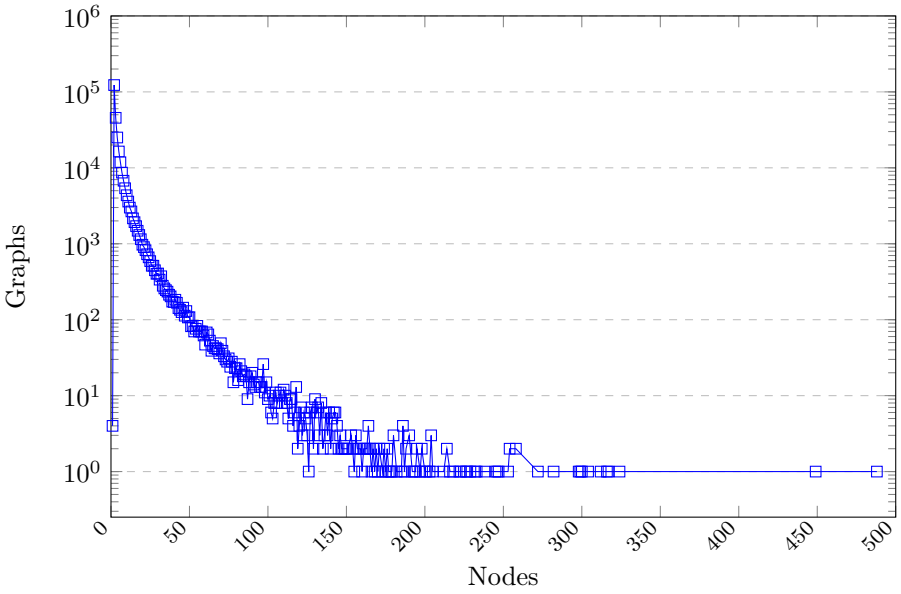

Figure 7.1: The number of graphs of a certain length contained in the database.

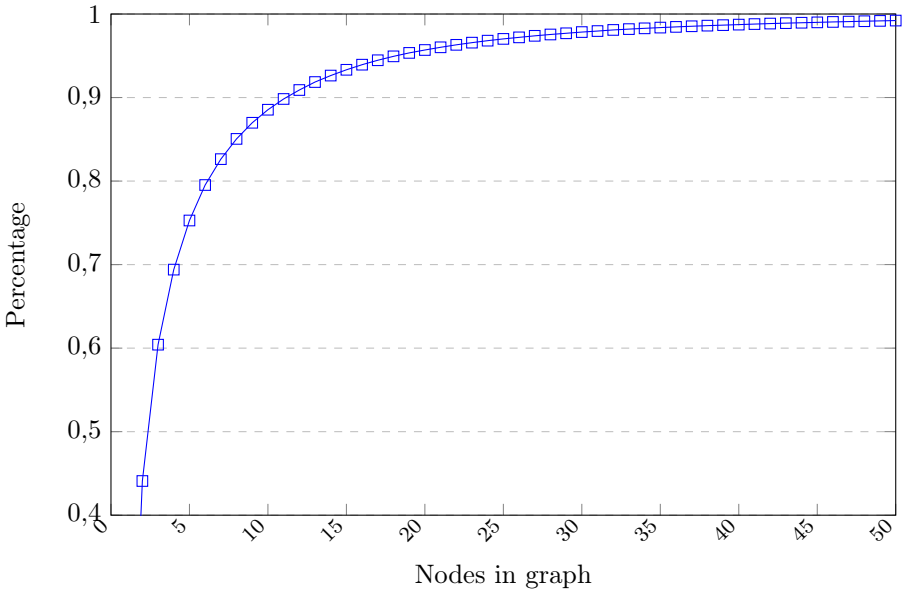

Figure 7.2: Cumulative Distribution Function of graphs contained in the database, for graphs containing up to 50 nodes.

As we can see from the figures above, the vast majority of graphs in the database are quite small, usually being only 2 to 4 nodes large. This means that extracting conversations from these graphs is going to be limited, and it should be expected that the conversation tracking process will reflect this fact.

7.5 Returning a subgraph

Before a conversation thread can be calculated from the newly obtained data, a better idea of exactly how the tracking process will function is needed. The assumption here is that a user who is interested in a particular topic, would want a conversation on said topic. This by itself is not trivial, as the current implementation of the database only returns documents that represent a single message. Before we are able to return a conversation thread, a definition of a start and end of a thread needs to be defined. This definition will depend on what exactly one is looking for, and should anyone wish to continue this work, alternative definitions of these will be a good place to start experimenting. For now, the definition will be based on their intuitive understanding, while keeping in mind the currently available data as well as the database implementation.

A natural way to search for a message related to a specific topic would probably be to use the "Subject" field. Its use is defined in RFC 822 [Cro82]
`https://tools.ietf.org/html/rfc822#section-4.7.1`

> *4.7.1. SUBJECT*
>
> *This is intended to provide a summary, or indicate the nature, of the message.*

So, given the assumption that most people use this field in the way it's meant to be used, it should be a good indicator of when the topic of a conversation changes.

Another field that is most likely used to discuss a certain topic, is the "Content" field. This is this field that contains the actual message the email is trying to convey. In the database, it is this field that holds the message body. The body of a message, does not have a strict definition, and is generally defined as "unstructured text".

Another potential source of information on a topic relevant to a message are the attachments of a given message. Like discussed in chapter 5, these are not included in the current implementation of the database. Should anyone wish to continue this project however, the attachments, their restoration, and analysis is certainly one area to potentially research further.

This finally means that there are two fields in the database that can be used to, to some degree, obtain information regarding the nature of the message. The goal of this project is to obtain better understanding of how internet standards come to be. A potential user will likely search for a specific topic. But how does one link a search, with the conversation graphs? This is certainly up for debate, but here is how it was done for this project.

7.5.1 Linking queries to conversation graphs

In order to link a regular Solr query to the conversation graph, a query is needed. For the sake of example, the "last call" will be used as the query term, as these imply urgency, possibly referring to the publication or submission of an RFC.

First, a facet query on the field "Message-ID" with the term "last call" is made, on either the "Subject" field, the "Content" field or both, is executed. As Solr uses cosine similarity to calculate results, it may be beneficial to require the query term to be present in both fields, as it may increase accuracy. This query will then return a list of message ids relevant to the query term.

This list will then be matched with the conversation graphs, where a matching program will attempt to fetch all conversation sequences relevant to the search results.

7.5.2 Feature requirements

In order for this to be done quickly and efficiently, there are certain requirements that need to be satisfied.

- a quick way to look up which graph any given node belongs to.
- a quick way to look up nodes in any given graph.

The first requirement is best solved by, yet again, using Solr. A new core has been created for this purpose. This, in contrast to many other already established cores, is a very simple one. The schema consists of only three fields: the tree id, a multivalued field containing ids of all nodes belonging to a tree, and the default Solr documents id field. This makes it possible to query the multivalued field for ids obtained from the first query, and quickly determine which graphs are relevant to said query.

The second requirement can be reasonably solved in two different ways. Recall that the conversation graph program prints its results to a structured file, said file can be read and searched as needed. The other alternative is to yet again create a new Solr core. This core would need to have a schema that can replicate the node format previously defined in the conversation graph program. This is a simple task, as the nodes object variables are easily replicable by multivalued fields in Solr. Since Solr has been a mainstay in this project, the second option was chosen.

7.5.3 The conversation tracking program

The following is a high level description of how the program in charge of calculating the results will function.

1. Execute a facet query on the "Message-ID" field, with the desired query term on the "Subject" and/or "Content" fields as previously discussed.
2. Convert the results of this query into a set of message ids.
3. Match the contents of said set with the query holding the conversation graphs, and determine which messages belong to which graph.
4. For each returned graph, try to find a path through all of the messages belonging to said graph, with a predetermined look ahead of at least 1.
5. Return all paths of length of more than 1.

The results returned by this program can finally be used to attempt to match their nodes with the publications of new RFC.

7.5.4 Ordering a graph

With the general idea of how the process of calculating a thread will work, it is now possible to properly define the start and end of a thread. There is, however, not one definition that is perfect. A natural way of picking a starting point for a conversation could simply be the oldest message in said conversation, this is not always correct however, as there are many messages in the database that either do not have a date, or have a date that is clearly wrong, be that messages from the future, or messages from a time long the invention of.

Another definition could be the same as that of a root, as previously defined. This has another problem however. That definition yet again hinges on the assumption of sequential flow of time. The assumption made when making said definition a node's child is always younger than its parent simply does not hold if we go by dates found in the database.

Another definition of the start of a thread, that does not depend on date, could instead depend on the parent/child relation. This is still not a perfect way of defining a start, as it is possible to send a reply to a message before the message being replied to is sent, if said messages message-id is known beforehand. This is simply something that will have to be kept in mind. This is not something that happens by accident, and it is highly unlikely that a legitimate participant of the IETF would orchestrate such a situation.

With all those things in mind, two options were considered as to how to sort the subgraphs returned by the thread tracing program.

The first option is simply using the dates attached to each node. Given a subgraph, the nodes will be sorted from oldest to youngest and displayed in this order. The second option is an enhanced version of option 1. As not all nodes have legitimate dates, if such a node is found, the program will attempt to retrieve a better date, or, at the very least, an interval of time for said node. This new period of time will be calculated based on its immediate neighbours, that being its parents and children. The program will attempt to find the smallest interval possible that does not violate the rules of sequential flow of time. This means that, children of a node that are older than parent nodes will be ignored, and vice versa.

7.5.5 The results

The following graphs depict some statistic regarding the resulting conversation threads.

Amount of conversations found = 23254
Average length = 4.567945299733379
Standard deviation = 6.931045907316653

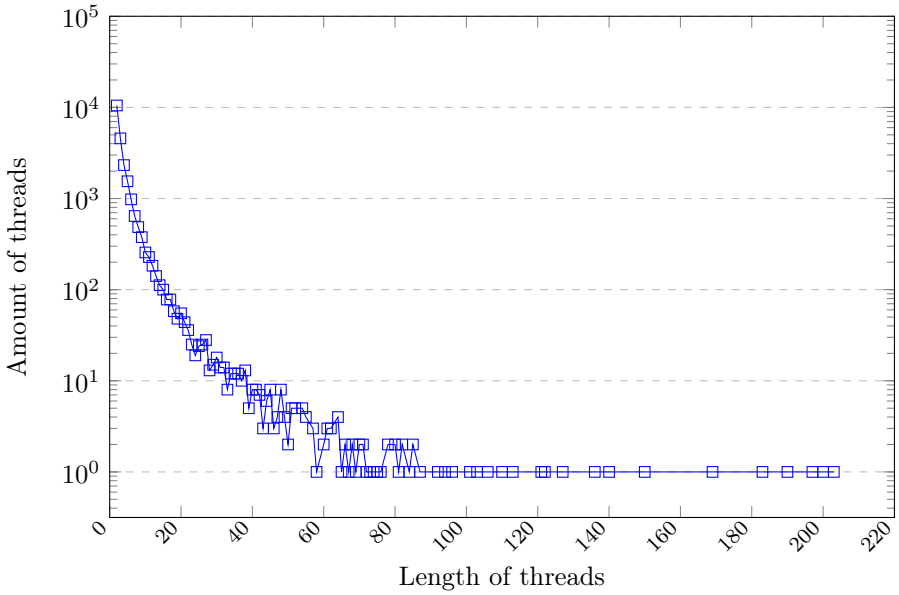

Figure 7.3: The length distribution of found conversation threads

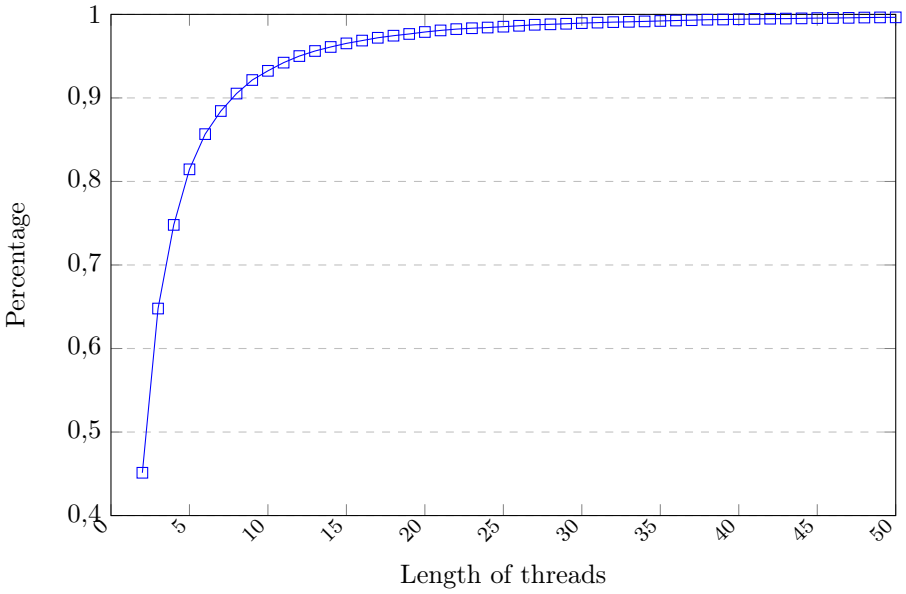

Figure 7.4: Cumulative distribution function of the length of conversation threads, for threads containing up to 50 nodes.

As an example, let us take a look at a short 2 node thread from the "last call" result set. The current implementation of the thread tracker only outputs the message ids and dates, meaning it is impossible to read the thread directly from the terminal. For this reason the content field, along with a few other

select fields have been chosen to be displayed below.

2012-12-13 19:45:31

From-name =>['Ted Lemon']
From-address =>['Ted.Lemon@nominum.com']

> Personally, I prefer minimizing the new message headers, and
> using options where possible.=20

It seems to me that we have three people saying theyd́ like to use an exist= ing header format, one saying we should define a new format, and one, Med, = probably not caring either way.
That being the case, Med, can you spin the draft one more time?

I think you can go back to the text you had, and just add the requirement f= or the link-address option. You'll also have to define the link-address o= ption and explain how it should be used. Maybe you can talk Bernie into p= roviding some text... :)

Once you've done that, let's re-spin the draft and do another last call, an= d see if we get any more responses. I'm hoping that Kim, Bernie and Tomek= will weigh in on the next last call, since they've helped to figure out ho= w to proceed with the draft.

The reply to this message is as follows.

2012-12-17 08:26:33

From-name=>['Null']
From-addres'=>['mohamed.boucadair@orange.com']

Hi Ted, all,

I updated the draft with the suggested changes. Please check the diff:=20

http://www.ietf.org/rfcdiff?url2=3Ddraft-ietf-dhc-triggered-reconfigure-02

Cheers,
Med=20

While this thread in itself may not bring any large revelation to the table, it does showcase an interaction taking place, and the thread tracking program

working as intended.

Coming back to the "last call" example, the following graph shows the average duration of a conversation thread, from the first message to the last in days.

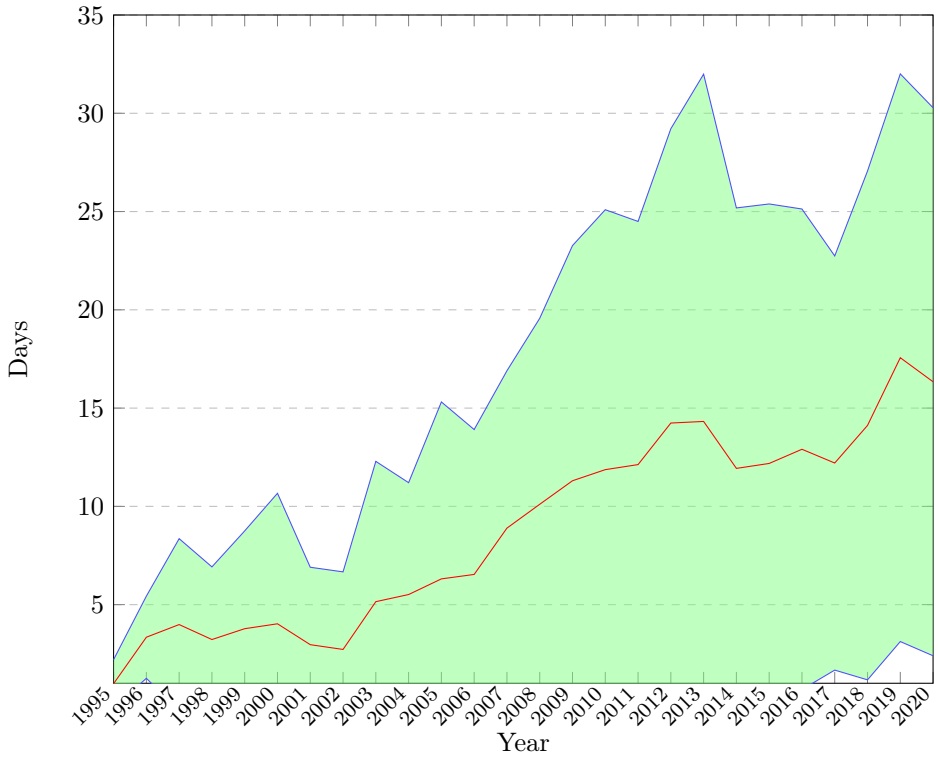

Figure 7.5: Average duration of a conversation in days

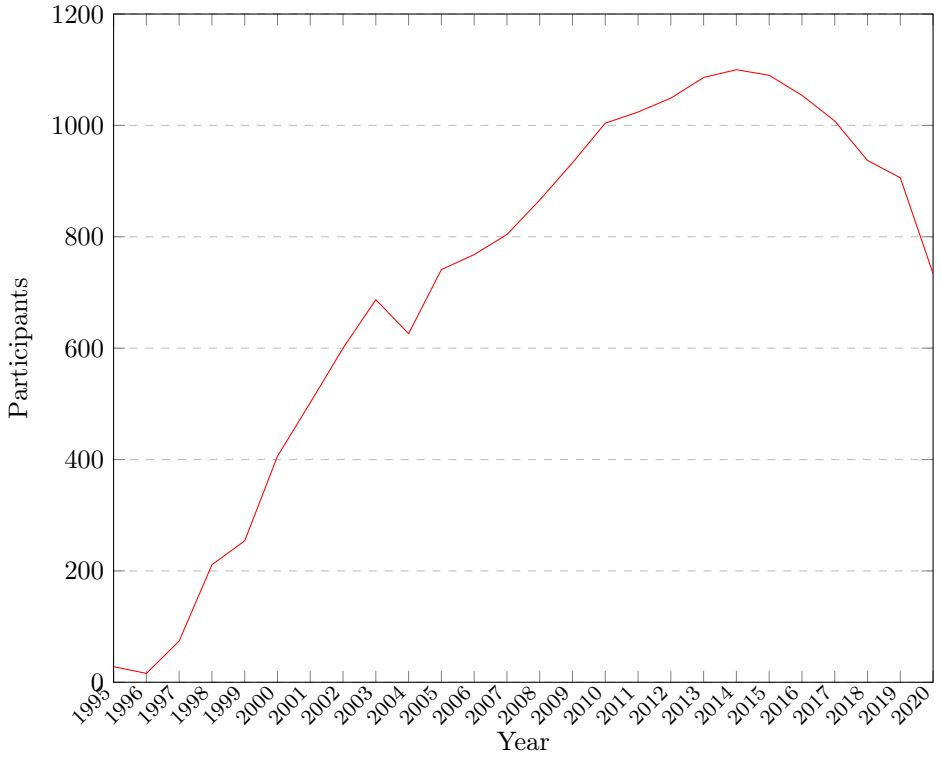

Figure 7.6: Amount of participants in conversations containing the "last call" phrase.

Something to note about this graph is how the length of the "last call" thread has been steadily increasing from the very beginning. This could be for several reasons, in the early days, as previously noted, not everything was archived, meaning the threads, if any, would naturally be shorter. In fact, no threads before the year 1995 were even found. As the archiving developed, more and more messages were properly archived, and the length naturally increased. This however only applies to the very early messages, and does not in itself explain why the length kept increasing.

The reason could be that as time goes by, and RFCs are made to either update old RFCs, or to define entirely new standards, their complexity increases. This would in turn increase the amount of work and time that has to be put into creating them, resulting in longer threads.

The steady increase in participants also likely contributed to the increased length of threads. In fact, in figure 7.5 we can see that the amount of participants started to decline in 2014, figure 7.6 shows a decrease in thread duration at that time as well. The thread duration did recover around 2017 however, while the amount of participants continued to decline. This means that the remaining people are doing more work now than ever (possibly due to Covid-19), or that there is not as much work to be done as there was in the previous years.

Chapter 8

Connecting RFC authors to the email archives

In order to gain a better understanding of how exactly a person contributes to the IETF, both by authoring RFCs, and partaking in discussion, we first need get a better grasp on what exactly a person is, in the context of this project. Recall the documents files from chapter 5. Several of them were specifically designed to contain the names that were found with their corresponding email address. And while these are now readily available and ready to be used, they are not necessarily a good representation of a person.

First of all, a way to determine a person exactly, or at the very least with a high probability, is needed. One set of names that are one hundred percent guaranteed to not only be actual people, but also active members of the IETF that have already contributed to the organization, are authors of RFCs.

A .bib file for citing RFCs has been created by Dr.-Ing. Roland Bless. This file contains not only the names of authors, but the date of publication as well, which can be used to bind RFC publications closed to conversation threads. The .bib file can be found at the following address.

`https://tm.uka.de/~bless/bibrfcindex.html`

This file not only makes it possible to cite RFCs in LaTeX, it is also a very well structured file that can be parsed into Solr for easy access. Doing this makes it possible to search for RFCs in Solr in the same way as the email archives. It makes it possible, for example, to compare email activity in the archives to the amount of RFCs published in the same timeframe. This again will make it possible to check if a conversation thread ends in an RFC publication.

The names of authors in RFCs are also very well structured, unlike what we find in the email archives. In the RFC documents, the name of the author, or authors, is always abbreviated, with the exception of their last name.

This however, is a problem, as a person's last names and initials are simply not good enough to identify them, at least not on the global scale this project is working on. Common names like "Kowalski", "Ødegård" and similar are very likely to appear far more frequently than others, and authors with these last names would most likely be swallowed up in a sea of others with the same name.

In fact, only about 55% of all authors can be correctly identified by using this method. And last but not least, the .bib file does not contain any information regarding the email addresses of authors.

In conclusion, the data found in the .bib file is not sufficient for establishing a connection between the RFC authors and their messages in the email archives.

8.1 IETF Datatracker

In contrast to the .bib files, the RFC documents found in the IETF datatracker contain not only full names of the authors, but sometimes their email addresses as well. However, yet again, in contrast to the .bib file and the email archives, there is not an easy way to extract this information.

Two different approaches to obtaining this data were considered for this project:

Approach 1:
The first approach is essentially the same way that was used to download the email archives. Make a program that would automatically download all of the RFC pages, then parse them in order to obtain the desired data.

Approach 2:
Scrape data directly from the page by using a Python module with such functionality.

For this project, the second approach was used as it ensures that the extracted data is always up to date, as well as saving on storage space and file seek overhead.

8.1.1 The web scraper

The scraper was developed with the help of the "BeautifulSoup" Python module. The high level description of this program is as follows:

1. Connect to the specified url.
   This is simple to do as the urls follows a very simple format like so:
   `https://datatracker.ietf.org/doc/rfc0000/`

   One can simply start at 0 and iterate over all addresses by increasing the RFC id counter by 1 each time. If the RFC with a given id does not exist, the connection attempt will fail, and the exception has to caught.

2. By using the previously mentioned module, parse the page, and look for a "href" attribute that starts with "/person/". This attribute specifically holds the email address of a person. In some cases, the email address is simply not there. In those cases, the name of the author is contained within this attribute instead. These cases are easy to distinguish, as a person's name always consists of at least two strings separated by a space.

The way html encodes spaces is by replacing them with a “%20”, this means that if a string obtained from the “href” attribute contains said substring, it is not an email address, and can be ignored.

3. The previously mentioned href attribute is always followed by a “span class” attribute, with no name, again with the help of the beautifulsoup module, this attribute can be retrieved.

8.1.2 Further data extraction

Now that there exists a concrete set of name and address pairs that are guaranteed to belong to actual individuals, it is possible to use the email archives to extract even more data about them. Firstly, a single email address per name may not tell the entire story, people tend to change their email address for a wide variety of reasons, like changing workplace or their position at the company. This means that in order to get all messages sent by a specific person, all of those email addresses are needed. The same goes for names, it is not impossible for a person to change their name. It is not unusual to change your last name when getting married for example.

This overall means that the definition of a person needs to be adjusted to account for these facts. Up until now the unspoken assumption about “people” was that they are uniquely identified by their name and email address pair. This no longer holds if more addresses or name changes are introduced into the mix.
For this reason, a person was assigned an id that uniquely identifies them.

But how does one go about finding the previously mentioned email addresses? The answer to this question depends on what is chosen to identify a person in the email database, versus the people obtained by scraping the IETF datatracker.

Essentially, if one assumes that the names found in the datatracker are unique identifiers, then the email archives can be used to obtain other email addresses a person has used, given they signed their emails with their actual name. Similarly, if we assume that an email address uniquely identifies a person, any other names a person may have used, can be found.
Both of these approaches involve some risks. For example, given the second assumption, if a name is ever used in conjunction with a “shared” email address, the names of all other people to have ever used said address would all be regarded as one. One of the previously blacklisted email addresses, for example "notifications@github.com" is such an address.
One way of avoiding this, aside from manually blacklisting said address, could be to arbitrarily limit the amount of names that can be associated with an address. While this is strictly speaking not a very scientific solution, it should at the very least remove the most contested email addresses.
For this reason, a list of the top 40 most contested email addresses was created and curated by hand to serve as a blacklist. The entire list can be found below.

Address	*Amount of names*
notifications@github.com	536
noreply@ietf.org	540
atompub-archive@megatron.ietf.org	452
aaa-archive@lists.ietf.org	320
eap-archive@megatron.ietf.org	500
calsch-archive@ietf.org	560
kink-archive@lists.ietf.org	432
ecm-archive@ietf.org	296
calsch-archive@megatron.ietf.org	208
cdi-archive@ietf.org	448
drums-archive@ietf.org	398
eap-archive@ietf.org	336
ldapbis-archive@megatron.ietf.org	578
dnsext-archive@ietf.org	324
capwap-archive@ietf.org	504
aaa-archive@ietf.org	348
ediint-archive@ietf.org	302
ldapbis-archive@lists.ietf.org	738
frnetmib-archive@ietf.org	270
conneg-archive@ietf.org	390
aft-archive@ietf.org	276
ipdvb-archive@megatron.ietf.org	206
rescap-archive@ietf.org	278
kink-archive@ietf.org	250
calsch-archive@lists.ietf.org	268
rsvp-archive@lists.ietf.org	380
v6ops-archive@ietf.org	216
ipdvb-archive@ietf.org	294
web-archive@ietf.org	224
webdav-archive@ietf.org	218
dnsind-archive@ietf.org	276
eos-archive@ietf.org	364
ipvbi-archive@ietf.org	208
weird-archive@ietf.org	236
issll-archive@lists.ietf.org	352
idmr-archive@ietf.org	216
ipp-archive@megatron.ietf.org	204
provreg-archive@ietf.org	218
tcpsat-archive@ietf.org	232
16ng@ietf.org	212

A similar situation takes place with the first assumption. Simply put, if the assumption does not hold, and there is a name overlap between several authors in the IETF datatracker, a lot of unrelated email addresses will end up being assigned to one name.

An arbitrary limit on how many different email addresses a name may belong to may be set. In addition, a limit on edit distance from any of the already

known email addresses may be set in order to improve results.

Despite this, the assumption of names of RFC authors being unique identifiers was made. This was done since the probability of two people having the exact string as their name is very low given the number of RFC authors. This allowed to retrieve a more complete set of addresses.

Connecting threads to RFCs can be done in several ways. In order to be able to link RFC to threads, one needs to know when any given RFC was published. One way of obtaining this information could be, scraping this information from the IETF datatracker. This is not possible however, as the date present in the datatracker, is the "last updated" date, not the date of publication. In RFC 822 [Cro82] for example, this date is set to 9. of July 2020, which is clearly not the publication date. There is another way, the previously mentioned .bib file, does in fact contain the proper publication dates. The only downside to using this .bib file is the fact that the dates contained within do not specify the day of publication, only the year and month. This should still be sufficient in tracing down which RFCs were published around the time a conversation thread took place. In order to make this data easily accessible a new Solr core was created, and a simple parser for the .bib file was developed. This core contains only very simple documents with only five fields each, the date of publication, the RFC identifier, the title, the working group name, and the name of the author.

In order to get the best possible results, both the .bib file and the scraper used previously were combined and used to create the documents. The .bib file was used as a way to keep track of RFC identifiers, as well as their publication date, and the datatracker was used to retrieve names and email addresses, like it did previously.

The same program is also responsible for extracting a more complete set of addresses. This is done due to the amount of time it takes to scrape all the required information from the web, and as the full names of authors are needed in both cases, it was seen as a good idea to kill two birds with one stone.

The outline of the program is as follows.

- Parse the .bib file, obtain a list of RFC that can later be used to create the appropriate url for the web scraper. Extract the date.
- For each RFC in the .bib file, use the scraper to obtain the name of the author. At this point, the documents are ready to be uploaded to Solr.
- For each name extracted from the IETF datatracker, look up email addresses associated with that name. Depending on what option was chosen, either filter the results using the previously presented black list, or see how many names are associated with any address in the results, and remove those that do not meet the requirements.

The results of this program were yet again uploaded to Solr, to two new cores respectively named "authors" and "rfc". With these two in place it became possible to link messages sent by any given RFC author, to messages they have sent around at that time.

The schemas for the non-standard fields in these cores can be found below.

The "author" core schema:

Field name	*Tokenizer*
name	solr.KeywordTokenizerFactory
address	solr.KeywordTokenizerFactory
wg	solr.KeywordTokenizerFactory

The "rfc" core schema:

Field name	*Tokenizer*
date	No tokenizer
number	No tokenizer
author	solr.KeywordTokenizerFactory
title	solr.StandardTokenizerFactory
wg	solr.KeywordTokenizerFactory

8.1.3 The code

As described earlier, the first step of the program is to read the .bib file. This file contains a given amount of "structures" representing information regarding an RFC. This format can be seen below:

```
@misc{rfc1,
  author="S. Crocker",
  title="{Host Software}",
  howpublished="RFC 1",
  series="Internet Request for Comments",
  type="RFC",
  number="1",
  year=1969,
  month=apr,
  issn="2070-1721",
  publisher="RFC Editor",
  institution="RFC Editor",
  organization="RFC Editor",
  address="Fremont, CA, USA",
  url="https://www.rfc-editor.org/rfc/rfc1.txt",
  key="RFC 1",
  doi="10.17487/RFC0001",
  }
```

This is easily done as it is a very well structured file. It has a downside however: many of the "fields" in said file all contain the same information, and not all fields are interesting either. The only fields that are relevant to this project are:

- Author
- Title
- Year

Strictly speaking, the title of the RFC is not necessary, but has been added as it may prove useful in the future.

Part 1, parsing the .bib file

An attempt at splitting each line of the .bib file using a "=" as the delimiter is made in order to split the line into two components, its field name, and the data contained within. The first element of this list is always assumed to be the fields name.

The first element still need its preceding whitespace removed before it can be properly matched. The removal of this whitespace is done by the "strip()" method.

The first field the program looks for is the "author" field, as this is the field that always precedes any other fields in the .bib file. This field always contains one or more names, which are, as previously described, just the author's initials followed by their last name. If there is more than one name in this field, they will always be separated by an "and", this means that it is easy to parse this field into a list, by using the "split()" method and using " and " as the delimiter.

```
while i < len(a): #parse the bib

    doc = dict()

    tmp = a[i].split("=")

    while tmp[0].strip() != "author":

        i = i + 1

        tmp = a[i].split("=")

    tmp_author = tmp[1].split(" and ")
    authors_out = list()
```

The next step depends on whether the split actually split the string to several parts. If it did not, the first character of the name string will have a quotation sign at the start, as well as a newline, a comma, and another quotation mark at the end. These are easily removed by slicing the string. This is a bit more complicated if there are 2 or more names. The first one will always begin with a quotation mark, and the last one with a newline, a comma and a quotation mark, which are again removed by slicing the respective strings. Any string located between the start and end of the list do not need any further processing:

```
    if len(tmp_author) == 1:
        authors_out.append(tmp_author[0][1:][:-3])
    else:

        authors_out.append(tmp_author[0][1:])

        for x in tmp_author[1:][:-1]:
            authors_out.append(x)
```

```
        authors_out.append(tmp_author[-1][:-3])

    doc["author"] = authors_out
```

The next two fields,"number" and "title", do not need much processing aside from slicing away the character at the beginning and at the end:

```
    while tmp[0].strip() != "title":

        i = i + 1

        tmp = a[i].split("=")
    doc["title"] = tmp[1][2:][:-4]


    while tmp[0].strip() != "number":

        i = i + 1
        tmp = a[i].split("=")
    doc["number"] = tmp[1][1:][:-3]
```

The last remaining field, namely "date", needs to be converted into the Solr format, the fact that these dates only describe the year and month of publication is taken care of during the conversion. The program slices up the string into its two components, the year and the month. Since the month field in the .bib file does not contain the numeric representation, but a text based one, it needs to be converted into a numeric one in order to meet the Solr format. This is done by the "getMonth()" method, that is essentially just a series of "if" checks. Since there is no day, hours, or minutes in the .bib file, the day has been set to the first one, and the time has been set to 0 hours, 0 minutes, and 0 seconds. This needs to be taken into consideration when linking RFC publications to dates found in the main database core, as the dates in the documents produced by this parser are better described as "month long periods of time" rather than exact dates:

```
    year = tmp[1][:-2]

    i = i + 1
    tmp = a[i].split("=")
    month = get_month(tmp[1][:-2])

    doc["date"] = year + "-" + month + "-01T00:00:00Z"
    rfc_list[doc["number"]] = doc
```

Part 2, scraping the IETF datatracker

Now that the .bib file is fully parsed and loaded, the next step is to scrape data from IETF datatracker by using the previously mentioned Beautifulsoup module. This part of the program makes use of the parsed .bib file to keep track of existing RFC numbers. As this part of the program is connected to a remote address, there is always a chance that said attempt may fail. In this case, the extraction fails. This was not an option in this case, only a perfect

result would be satisfactory. For this reason, the program is made to retry as many times as needed. This is dangerous to do however, as it is absolutely possible that the attempt never actually succeeds, resulting in an endless loop. This could be easily prevented by setting a certain number of times the program is allowed to retry it. Despite this, said solution was not used, and the program did terminate, and successfully scraped all requested data.

```
for x in tqdm(rfc_list):
    #print(x)
    retry = True
    while retry:
        try:
```

First, the program opens a connection to the specified url, and the contents are parsed by the Beautifulsoup module:

```
vgm_url = 'https://datatracker.ietf.org/doc/rfc' + str(x) + '/'
html_text = requests.get(vgm_url,timeout=5).text
soup = BeautifulSoup(html_text, 'html.parser')
```

Next a list of all href attributes is made. In this list, the elements that start with "/person/" is the one containing the email addresses of the authors. These are added to a list.

```
for a in soup.find_all('a', href=True):
    if a.get('href')[:8] == "/person/":
        adr.append(a.get('href')[8:])
```

In order to retrieve the names, a list of all classes is retrieved. Curiously, the class with no name is the one containing the names of the authors. One thing to note here, is that the names and addresses always come in the same order. The first address found, will always belong to the first name found.

```
for a in soup.find_all('span', {'class' : ''}):
    names.append(a.text)
    a.get(a.text)
```

The name retrieval can "fail" in a sense: simply, if there is no name to be found, the string "Unknown" will be in its place. If this is not the case, the names retrieved from the datatracker are used to replace the names of authors retrieved from the .bib file in their respective RFC:

```
            if names[0] != "Unknown":
                rfc_list[x]["author"] = names[:-1]
```

Finally, if any addresses were retrieved, the name and address are stored for later use. This also means that authors who did not have any address associated with them, will be absent from the author database. This mainly applies to the RFCs

from one to one thousand. For example, the author of RFC 1 [Cro69], according to the .bib file is "S. Crocker", as there is no address to this name in the IETF datatracker, said author is absent from the author database.

```
while i < len(adr):

    try:
        ppl[names[i]].add(adr[i])
    except KeyError:
        ppl[names[i]] = set()
        ppl[names[i]].add(adr[i])
```

The working group of an RFC is extracted in a similar manner.

```
for a in soup.find_all('a'):
    tmp = str(a).split("/")
    if tmp[1] == "wg" and tmp[-3] == "about":

        rfc_list[x]["wg"] = (tmp[2])
```

At the very end, the "retry" variable is set to "False" in order to break the loop.

Part 3, further data extraction

The last part of the program is responsible for extracting a more complete set of email addresses for the authors. As such, the program begins by establishing a connection to the mail database core, where this information resides. Next the complete blacklist is loaded and the extraction process begins. The extraction process is simple yet has some unfortunate quirks. In order to retrieve more email adresses one must first search for all documents with the author's name in the "From-name" field. As one may recall from chapter 4, this is a multivalued field. This means that the faced function cannot be used, as documents with more than one value in this field would return all of their value as the result. This would mean addresses that do not belong to the authors, would be assigned to them anyway. One way of avoiding this is only taking messages with one singular value in this field into consideration. Unfortunately, the version of Solr used for this project does not support queries based on the amount of values in the multivalued field. This leaves only one option: making the program check the length by itself. This however means that the entire results set needs to be checked. This is significantly slower than using the facet function, but is the only way to get the desired results. The "fl" parameter has been set to the "From-address" field in order to speed up the process.

```
param = {"debugQuery":"off",
"rows" : 2147483647,
"facet":"off", #"off" means: no facet will be returned
"fl":"From-address",
}

solr = connect_to_solr(sys.argv[1])#connect to solr
```

```
blacklist = open("blacklist.txt","r").readlines()


print("Data extraction")
for x in tqdm(ppl):

    res = solr.search("From-name:" + "\"" + x + "\"", **param)


    for y in res.raw_response["response"]["docs"]:

        if len(y["From-address"]) == 1:
            ppl[x].add(y["From-address"][0])


    for y in blacklist:

        try:
            ppl[x].remove(y[:-1])
        except:
```

The results this program produces are finally uploaded to their respective Solr cores, those being named “authors” and “rfc”.

8.1.4 Statistics

With these two cores in place it is now possible not only to link authors to their sent messages, but also to extract statistics regarding the authors. Here is just some of the information that can be extracted from these two cores.

The following table shows the 10 most active authors, that is, the authors that have their names listed in the largest number of RFCs.

Ranking	*Name*	*RFC count*
1	Russ Housley	96
2	Donald Eastlake	95
3	Keith McCloghrie	92
4	Henning Schulzrinne	90
5	Hannes Tschofenig	85
6	Yakov Rekhter	78
7	Jonathan Rosenberg	72
8	Adrian Farrel	71
9	Paul Hoffman	70
10	Gonzalo Camarillo	70
11	Marshall Rose	65
12	Fred Baker	65
13	Alexey Melnikov	60
14	John Klensin	59
15	Mohamed Boucadair	54
16	Eric Rosen	54
17	Dave Thaler	54
18	Brian Carpenter	54
19	Bernard Aboba	54
20	Carlos Pignataro	52

Table 8.1: Top 20 authors of RFCs.

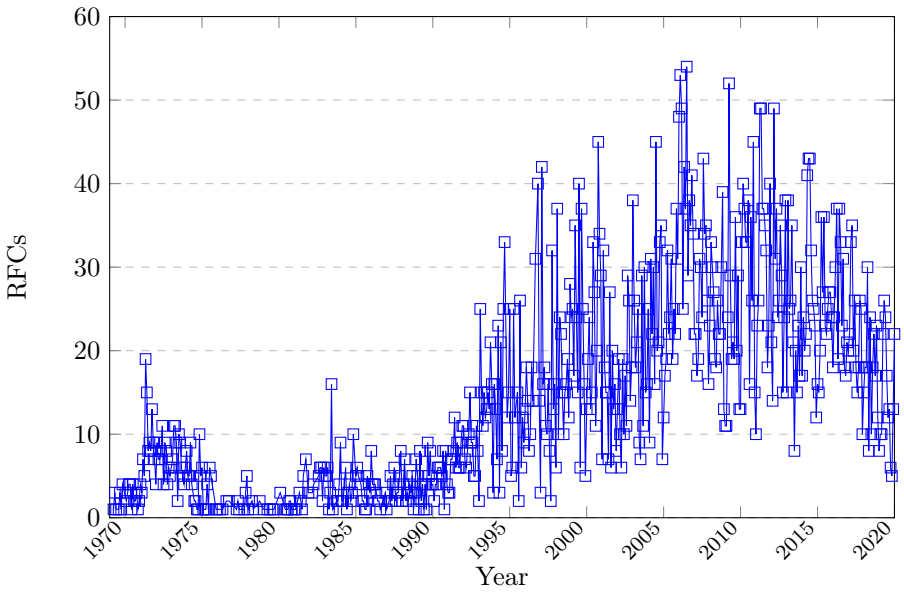

Figure 8.1: RFCs published.

What about email activity? It would seem natural that more messages are sent by the authors leading up the publication of a new RFC. As previously

discussed, RFC publications can only be tracked on a monthly basis. This is not a problem for Solr however, as it is easily able to search for documents contained within a period of time. Since RFCs can only be tracked in this way, the amount of emails sent by authors will also be tracked on a monthly basis.

What about author involvement? Are authors the main participants in discussions or do they just create the RFC documents, without participating much in the discussions themselves ? To answer this question several cases were investigated. The graph below shows how many of the top one hundred posters each year were authors. The percentage was calculated on a per-address basis. This means that if an author, as defined in the "author" core, used several of his email addresses in a given year, they would be counted several times.

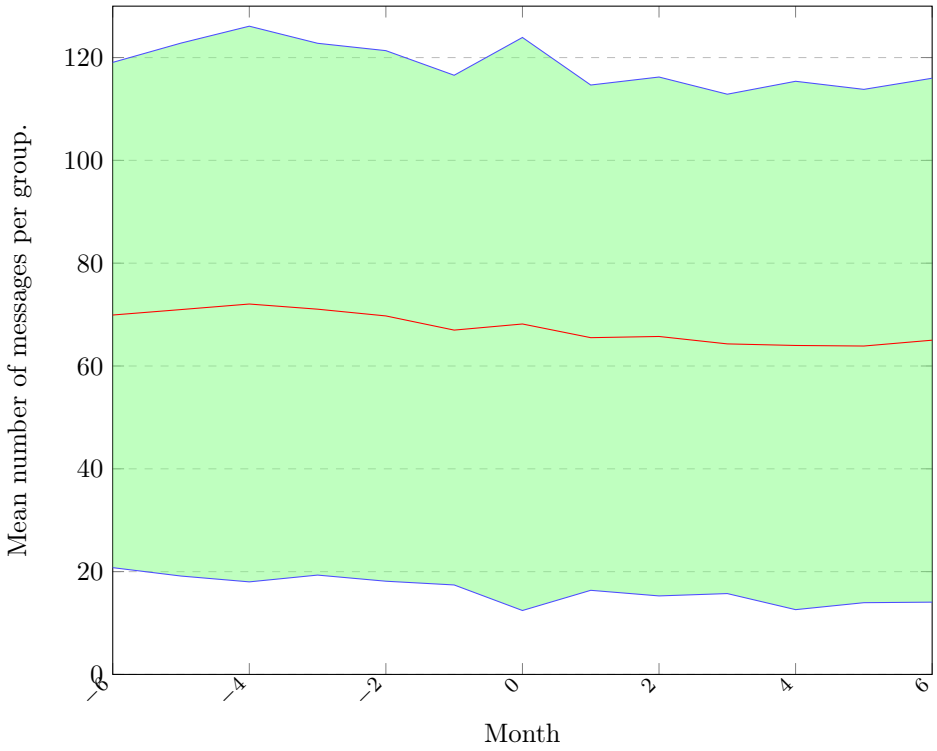

Figure 8.2: Amount of messages sent in a working group, around the time of publication of an RFC from that group. The shaded area indicates the standard deviation.

One thing to keep in mind about this graph, as well as all that follows, is that not all working group names from the RFC match up with names of mailing lists in the email archives, and for this reason cannot be linked. This means that roughly one third of all mailing lists found in RFCs are not represented in the graphs. For a full list of said working groups, see the github repository.

The graph above depicts the average email activity in a working group around the time of a RFC publication from that group. The data show clearly

that there is essentially no change in email activity, around a publication. A possible reason for this is the fact that RFCs need to be approved by the RFC editor before they can be published. This is usually a quite lengthy process, meaning the working group is usually done with their discussion by the time the RFC is actually published.

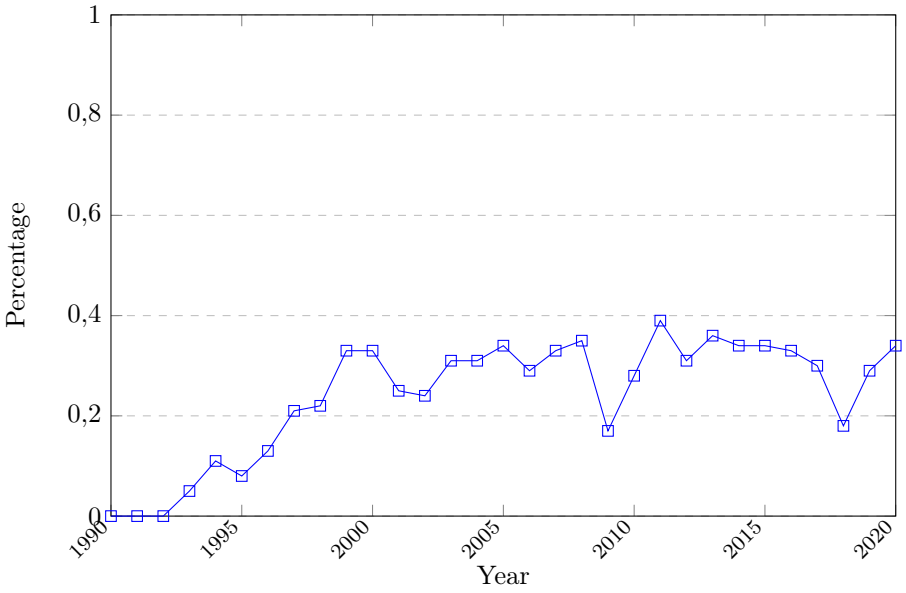

Figure 8.3: Top 100 email addresses belonging to authors of RFCs

All graphs regarding linking email adresses of authors to messages in the email archives, start at the year 1992. This is caused by the lack of data. RFCs in the datatracker do not contain the addresses of authors prior to said year, making it impossible to link authors of said RFCs to their messages. Those messages may very well be in the archives, there is simply no way of knowing given the current implementation.
Data in the following years is also sparse, which is likely the cause of the large variation seen in those years.

The data shows a quite promising trend, given the relatively small amount of authors, as compared to the rest of all posters, they still take up around 25 to 40 positions in the top 100 posters. This means that authors are indeed very active participants that are responsible for a significant amount of messages sent each year. It also means that the IETF is more than just a forum that authors use to discuss among themselves. They definitely are some of the most active users, but definitely not the only ones.

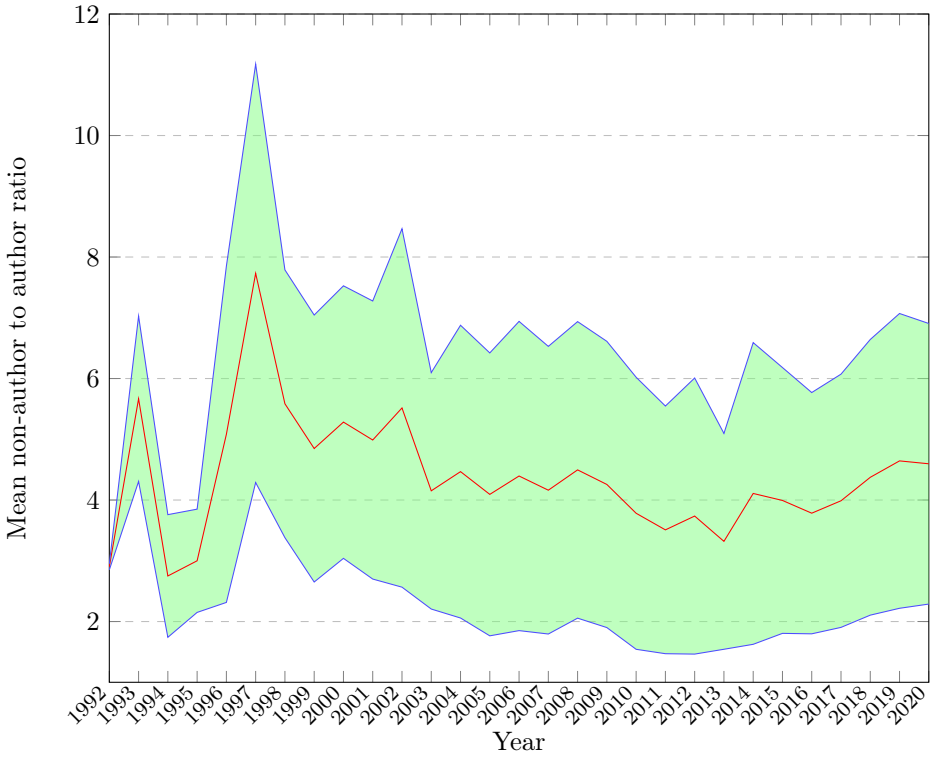

Figure 8.4: The mean of the ratio of non-authors to authors in the top 20 posters in a working group. The shaded area indicates the standard deviation.

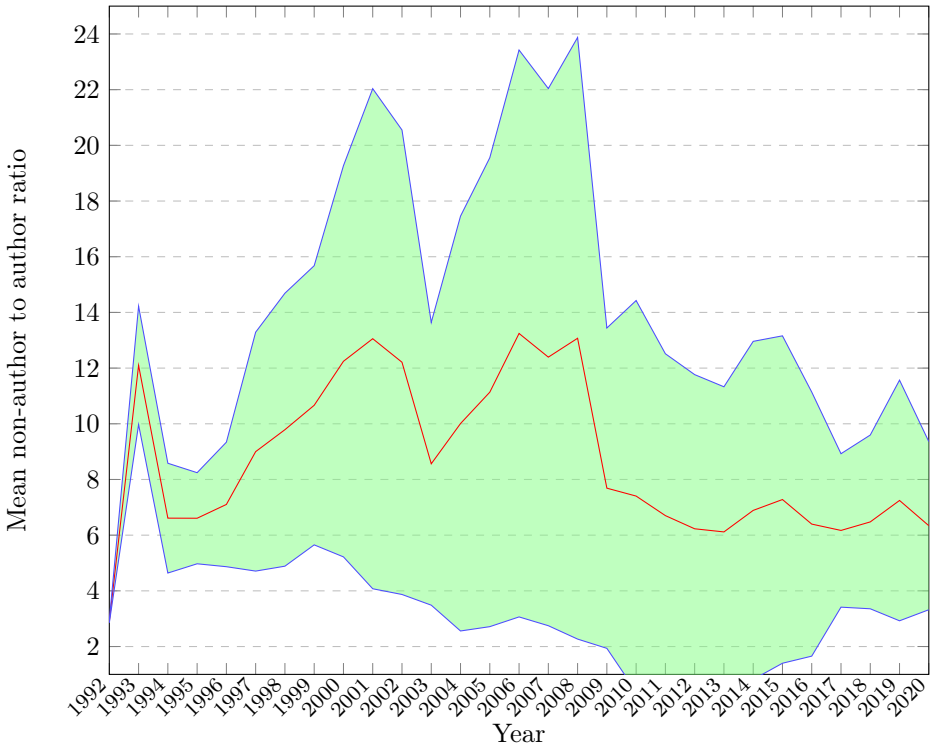

Figure 8.5: The mean of the ratio of non-authors to authors in the top 100 posters in a working group. The shaded area indicates the standard deviation.

Overall figures 8.4 and 8.5 tell the same story. Authors are active participants in a larger discussion, and while the exact amount of authors has varied somewhat through the years, they have always been some of the most active users. Reaching its peak around 2012-2013, and reaching the lowest point at around 2018, the amount of authors has been sharply increasing since.

8.1.5 Search strategy

Knowing this opens up for new possibilities when it comes to conversation tracking. One major "flaw" of the method previously described in this chapter is its reliance on keywords. An RFC does not get its number until it is published. This means that using for example "RFC 822" as a query term, will return conversations mentioning said RFC, which again means that they must have occurred after its publication. While this in itself can be useful, it does not provide anything useful in regards to how said RFC came to be.

With the now added option of investigating specific authors one can look for the conversation an author of any given RFC had before the RFC was published. But how does one do this? The current implementation, as said queries the "Subject" and "Content" fields, then uses the results returned by Solr to calculate conversation threads. This however, can be quite easily changed; as the current implementation uses a set of nodes to calculate threads, simply providing a more sutiable set of nodes will result in better results. But in that

case, what kind of queries are likely to give the best results? A good starting point would be to query a specific time frame before an RFC was published. The exact size of the timeframe, will of course vary from RFC to RFC, but 1 year is expected to at the very least be a good starting point. The next improvement would be to focus on the authors of the RFC one is trying to investigate. The underlying assumption here is that the author will generally be the "driving force" behind a conversation, and if a suitable value for the thread tracking program's lookahead is chosen, the responses and replies from other participants are likely to be included in the thread. Finally, the two fields that were originally used are still relevant, as they tell something about the nature of a conversation. What to query for in these fields is up for debate, generally speaking, querying for the number of an RFC will not yield good results. Keywords relating to the desired topic are expected to be much more effective. For example,if we wish to look for threads that may have led to the creation of RFC 2822 [Res01], querying RFC 2822 [Res01], will not work. However, querying for a keyword such as "email standard" or "message format" is expected to result in much more relevant threads when combined with the two previously discussed query options.

Chapter 9

Conclusion

To sum up the entire project, the mail archives have been parsed and transformed into a searchable state in an advanced text search database. The parser used for this purpose is not limited to parsing the IETF email archives; any archives in the mbox format can be parsed using this software. The messages in the email archives have been linked together using the available data to graphs representing a relation between messages. Said graphs have been used in an attempt to track down desired conversation threads that may contain information regarding the creation of specific RFCs. While an answer has not yet been found and there is still much work to do, the groundwork has been laid for further research, should anyone wish to continue this work. Here are some pointers as to how to improve the project in various ways.

Spam detection

The email archives are unfortunately full of messages that do not bear any meaningful or useful information in regard to the goals of this project. Many of these are automated messages trying to sell various things such as medication, or cheap cloned products. Removing these from the database would greatly improve the quality of the data within. As we saw earlier, looking for highly contested email addresses is a good way to start.

Machine learning

As the current thread tracker relies on the results passed on by Solr, figuring out a good search strategy, and using the results returned by using said strategy, may be a good way to start implementing machine learning in an attempt to retrieve better results. The results passed from asol are after all calculated with cosine similarity (as depending on the configuration of a given field), which is often used as a baseline that a given machine learning algorithm needs to beat.

Improving the parsing process

The parsing process will always be a highly important stepping stone for this project. As we saw in chapter 7, a lot of messages have flaws, and are missing vital data in their fields. Remedying this problem is surely to improve the result and conversation that can be retrieved from the database.

Restoring and parsing the attachments

Attachments carried by messages were left out of this project, mainly due to time constraints. They are a whole new source of untapped information that is ready to be restored into its original form, and parsed.

A better idea of a person

Being able to ascribe the messages sent by several email addresses to a single person can be very valuable. We already saw a fragment of it when working on tracking the work of authors specifically. This is however not a trivial task, and probably goes hand in hand with spam detection, as removing the work of automated processes, such as notification messages, or simply spam. Trying to sell some unspecified product at allegedly very attractive prices will increase the probability that a given message was in fact sent by an actual human being. How exactly this is done is not yet clear, however, organizing all email addresses in the database into categories could be a first step. For example, looking for patterns in email addresses of automated messages. This is also an instance where machine learning could prove to be very useful, as it has already been, and is being used to identify individuals matched on many aspects of their behaviour.

Establishing linear time

A major issue with both the graphs, nodes, and conversation tracking is the fragile concept of time, at least as it seems to be in the current database. A way to properly order the emails sequentially, and giving proper, or at least better estimation of when they were actually sent, would likely help to improve the results. Establishing, enforcing, and resolving family tree rule violations is also expected to be highly beneficial, as once we know that the child of a parent is guaranteed to be younger, they can be ordered appropriately.

These are just some of the suggestions from the author of this thesis. There probably are many other things that could be changed and altered in order to improve the overall quality of the project.

Chapter 10

Appendix

10.1 Known issues and fixes

Issue 1:

Some of the strings returned by Solr from the address field are not immediately searchable due to how Solr manages special characters and spaces. This is a problem as it is these strings or "tokens" that are returned by the facet function, and are often used to calculate statistics for this project.

Example:
"Dose <zeroconf-archive"@lists.ietf.org
"Dose <kink-archive"@megatron.ietf.org
"Dose <dnsext-archive"@ietf.org
"greetingll-yours.net"@129-79-115-30.dhcp-bl.indiana.edu
"greetingll-yours.net"@mail.difusoradigital.com.ar
".com"@truemail.co.th
".org"@afa.org.ar
"Prof. Jesus Dolphin"@core3.amsl.com

To remedy this a method was created to restore these strings to a form accepted by Solr. This method is called "fix_adr(inn)", takes a string as input, and outputs a more Solr friendly version. This method should only be used if a query with a non processed string fails.

Issue 2:

Similar to issue 1 there are some characters that are not accepted by Solr. This also depends on where in the string these characters are found. Characters that cannot be at the beginning of a string are as follows:

+,-, ,!

Characters that are not allowed in any position in the string are as follows:

",/,[,],{,},;:

The method mentioned in issue 1 has been expanded to also handle these characters, and should be used in the same circumstances as issue 1.

10.2 List of Solr cores

author

This "author" core contains a set of authors, with their names, and email addresses.

Field name	*Tokenizer*
name	solr.KeywordTokenizerFactory
address	solr.KeywordTokenizerFactory

rfc

This “rfc” core contains information about RFCs, including their authors, time of publication, id and title.

Field name	*Tokenizer*
date	No tokenizer
number	No tokenizer
author	solr.KeywordTokenizerFactory
title	solr.StandardTokenizerFactory

graphs

This core is used to store the conversation graphs.

Field	*Tokenizer*
graph-id	No tokenizer
nodes	solr.KeywordTokenizerFactory

nodes

This core contains the nodes of the conversation graphs. Each node has information regarding iself, as well as its immediate neighbours, that being its parents and children.

Field	*Tokenizer*
node-id	solr.KeywordTokenizerFactory
graph-id	No tokenizer
parent	solr.KeywordTokenizerFactory
child	solr.KeywordTokenizerFactory

ietf-archive-final-v2

This is the main core, it is this core that contains the parsed email archives, and it is this core that was used for calculating statistics, tracking conversations and many other things.

Field	*Tokenizer*
Date	No tokenizer
Date-raw	solr.KeywordTokenizerFactory
Timezone	solr.KeywordTokenizerFactory
From	solr.StandardTokenizerFactory
From-name	solr.KeywordTokenizerFactory
From-address	solr.UAX29URLEmailTokenizerFactory
Sender	solr.StandardTokenizerFactory
Sender-name	solr.KeywordTokenizerFactory
Sender-address	solr.UAX29URLEmailTokenizerFactory
Reply-to	solr.StandardTokenizerFactory
Reply-to-name	solr.KeywordTokenizerFactory
Reply-to-address	solr.UAX29URLEmailTokenizerFactory
To	solr.StandardTokenizerFactory
To-name	solr.KeywordTokenizerFactory
To-address	solr.UAX29URLEmailTokenizerFactory
Cc	solr.StandardTokenizerFactory
Cc-name	solr.KeywordTokenizerFactory
Cc-address	solr.UAX29URLEmailTokenizerFactory
In-Reply-To	solr.StandardTokenizerFactory
In-Reply-To-name	solr.KeywordTokenizerFactory
In-Reply-To-address	solr.UAX29URLEmailTokenizerFactory
Message-ID	solr.KeywordTokenizerFactory
References	solr.KeywordTokenizerFactory
Comments	solr.StandardTokenizerFactory
Subject	solr.StandardTokenizerFactory
Content	solr.StandardTokenizerFactory
Mailing-list	solr.KeywordTokenizerFactory
File-location	solr.StandardTokenizerFactory

Bibliography

[Cro69] S. Crocker. *Host Software*. RFC 1. IETF, Apr. 1969. URL: http://tools.ietf.org/rfc/rfc0001.txt.

[Bhu+73] A.K. Bhushan et al. *Standardizing Network Mail Headers*. RFC 561. IETF, Sept. 1973. URL: http://tools.ietf.org/rfc/rfc0561.txt.

[Cro82] D. Crocker. *STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES*. RFC 822. IETF, Aug. 1982. URL: http://tools.ietf.org/rfc/rfc0822.txt.

[FB96a] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC 2045. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2045.txt.

[FB96b] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2046.txt.

[Moo96] K. Moore. *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*. RFC 2047. IETF, Nov. 1996. URL: http://tools.ietf.org/rfc/rfc2047.txt.

[FM97] N. Freed and K. Moore. *MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations*. RFC 2231. IETF, Nov. 1997. URL: http://tools.ietf.org/rfc/rfc2231.txt.

[TDM97] R. Troost, S. Dorner, and K. Moore. *Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field*. RFC 2183. IETF, Aug. 1997. URL: http://tools.ietf.org/rfc/rfc2183.txt.

[Res01] P. Resnick. *Internet Message Format*. RFC 2822. IETF, Apr. 2001. URL: http://tools.ietf.org/rfc/rfc2822.txt.

[Res08] P. Resnick. *Internet Message Format*. RFC 5322. IETF, Oct. 2008. URL: http://tools.ietf.org/rfc/rfc5322.txt.

[YSF12] A. Yang, S. Steele, and N. Freed. *Internationalized Email Headers*. RFC 6532. IETF, Feb. 2012. URL: http://tools.ietf.org/rfc/rfc6532.txt.

[Lei13] B. Leiba. *Update to Internet Message Format to Allow Group Syntax in the "From:" and "Sender:" Header Fields*. RFC 6854. IETF, Mar. 2013. URL: `http://tools.ietf.org/rfc/rfc6854.txt`.