U N I V E R S I T Y O F T A R T U
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science speciality

Peeter Jürviste

Email Information Concentrator

Bachelor Thesis (6 EAP)

Supervisor: Ulrich Norbisrath, PhD

Author: ..... "....." June 2010
Supervisor: ..... "....." June 2010

Allowed to defence
Professor: ..... "....." June 2010

TARTU 2010

Contents

Acknowledgments

I have come this far thanks to people around me who have helped me in different ways. First and foremost, I wish to thank Ulrich Norbisrath for his great advice on writing the thesis, technical guidance and encouragement.

I am also very grateful to Tõnu Tamme for providing me with the needed input to improve the Prolog Parser further and for his valuable proofreading. It has been a great pleasure working with Dmitri Danilov on integrating my Graph Parser with his 3D graph visualization application. I thank him for reading my thesis and proposing changes as well. I would also like to express gratitude to Georg Singer for sending me the resources about related projects.

Lastly, I wish to thank my parents who have always supported me in what I am doing and never lost faith in me.

Introduction

Email is one of the most successful and widely used computer applications yet devised. It has been around for decades and is used by individuals as well as organizations all over the globe. However, nowadays we are facing a growing message overload problem. This paper is a part of the corresponding research mainly located in an area called knowledge management with an aim to discover possibilities to handle this problem.

This field of research is currently very relevant because the number of emails sent is growing every year and people not only want to receive less spam, they also want to be able to find the messages they are looking for, group, categorize and sort them the way they need. On the other hand, current message tools (web-based applications and email clients) are not capable to deal with this increased amount of messages effectively. This project supports the research undertaken here by supporting the structured input of mailbox contents.

An electronic mail message consists of two components, the message header, and the message body, which is the email's content. In this work, I develop an Email Information Concentrator. The Email Information Concentrator aims to deliver emails via IMAP and parse the relevant parts of messages to other formats, such as an Email Graph or Prolog statements. The output of the Email Information Concentrator is used as an input for further research.

I decided to undertake this project since it provided me with an opportunity to combine theoretical studies with some code-writing. What is more, I do not have a background in Python programming and it is therefore very exciting to learn a new programming language. The third reason is probably being part of a bigger research project in the field of knowledge management which could revolutionize the way electronic mails are processed.

One of the most challenging tasks ahead is presumably parsing HTML tags inside message body. What is more, new difficulties might emerge when composing an Email Graph to represent a mailbox and its contents and linking it with our graph 3D visualization software.

The thesis is divided into four chapters. After this introduction, the first chapter defines the message overload problem and describes related work and projects. In addition, the structure of the email message format and the IMAP protocol are described. It is followed by a requirement analysis for the Email Information Concentrator. The design and implementation of the system are discussed in the third chapter. It also looks at future tasks: what are the main challenges ahead as we see them now? The fourth chapter contains instructions aimed at the end-user to get the best out of this library.

Chapter 1

Related work and theoretical background

This chapter defines the message overload problem as well as lists reasons why it is necessary to do research in this field. There is an overview of projects related to the Email Information Concentrator in section 1.2. In addition, I describe the structure of the email message format and the IMAP protocol.

1.1 The message overload problem

Most people active in the digital age will have discovered how much digital information is trying to capture their attention everyday. One type of this kind of information is electronic mail. Starting with online email accounts, we suddenly realize, that we actually do not have to delete any messages anymore, we can just archive them. Space, especially hard disk space, is really cheap. The result is that we store or archive most of our emails, creating a huge very coarsely sorted repository. Categorization like business and private are not anymore enough. We see a number of documents that are cutting across categories. Often we would like to categorize only certain paragraphs of a document in a specific way and others differently.[12] Creating separate mailboxes for every occasion is not an option because it would be virtually impossible to manage.

The amount of information (documents, web pages, emails, chats), we are working within our professional as well as private sphere increases daily and current information management tools are not capable to deal with this amount of information effectively.[12] Even worse, despite the fact that the amount of information available is growing on a daily basis, this does not make us more productive. It slows us down. Consequently, it gets more complicated to find what we are looking for on the one hand and valuable documents are saved in repositories without ever being touched again.

Back in 1996 Steve Whittaker and Candace Sidner looked into the email overload problem and defined it as using email for tasks it has not been designed for. Originally designed as a communications application, email is now being used for additional functions, that it was not designed for, such as task management and personal archiving.[16] They also argue that email overload creates problems for personal information management: users often have cluttered inboxes containing hundreds of messages, including outstanding tasks, partially read documents and conversational threads. Furthermore, user attempts to rationalize their inboxes by archiving are often unsuccessful, with the consequence that important messages get overlooked, or "lost" in archives.[16]

Ten years later (2006), members of Microsoft Research team published a paper on the key changes in this field of knowledge management. They say that while inboxes are roughly the same size as in 1996, people's email archives have grown tenfold.[9] Even now, it seems likely that email overload will continue to grow since email programs continue to be a catch-all storage and communication medium for many other tasks, like document transfer and staying aware of online content using RSS feeds.[9] Fisher, Brush and others emphasize that there remains a need for future innovations to help people manage growing archives of email and large inboxes, i.e. we urgently need to do extensive research in advanced information management technologies.

1.2 Related and derived projects

Quentin Jones, Gilad Ravid and Sheizaf Rafaeli have studied the impact of information overload on human behaviour. Based on the analysis of over 2.65 million postings to 600 Usenet newsgroups over a 6-month period, they have found evidence for the assertion that individual strategies for coping with information overload have an observable impact on large-scale online group discourse. Their main findings were:

- Users are more likely to respond to simpler messages in overloaded mass interaction
- Users are more likely to end active participation as the overloading of mass interaction increases
- Users are more likely to generate simpler responses as the overloading of mass interaction grows [10]

Out of all the information provided, I was particularly interested in how they identify messages as replies or parent messages. This kind of message processing was not a part of the Email Information Concentrator but it could give ideas for reconstructing discussion threads.

As for identifying replies, they constructed an algorithm utilizing the degree to which reply indicators in the subject line, the message body, and the message headers each correctly identified a message as a reply.[10] The algorithm was computed as follows: if the subject line contained "re:" or "reply:", it was considered a reply. If a reply string (a line with the format similar to "Name <e-mail address of earlier poster> wrote:") was not found in the subject, then messages were required to have a score of 0.8 or higher to be considered as a reply. More than two lines with indenting result in a weighting of 0.6. "In reply to:" was given a weight of 0.5. Having message content that indicated that it was forwarded was given a weight of 0.4. Finally, a message reference was given a weight of 0.3. The exact weights were chosen heuristically.[10]

Quentin Jones, Gilad Ravid and Sheizaf Rafaeli also claim that 16% of messages coded as replies did not have references, which means that threading could not be reconstructed by solely relying on the references contained in the message headers.[10] To identify parent messages they proceeded as follows: find existing parent-child message relationships by matching messages containing a "reference:" header field with messages having the same "message-ID". If this was not successful, a similar step was taken for the "in reply to:" field. Third, various searches were made to match the subject line

after reply indicators such as "re:" had been removed. Finally, the Oracle SQL extension "`connect by`" command was used to construct the discussion threads for all the newsgroups from the message-reply pairs.[10]

Lewis and Knowles from AT&T Labs pointed out even earlier (1997) that message header referencing is not a reliable means of identifying and reconstructing discussion threads.[11] They claim that inconsistencies between email clients, loose standards, creative user behaviour, and the subjective nature of conversation make threading systems based on structural information only partially successful. Lewis and Knowles propose that this situation is unlikely to change, and that threading of electronic messages be treated as a language processing task.[11]

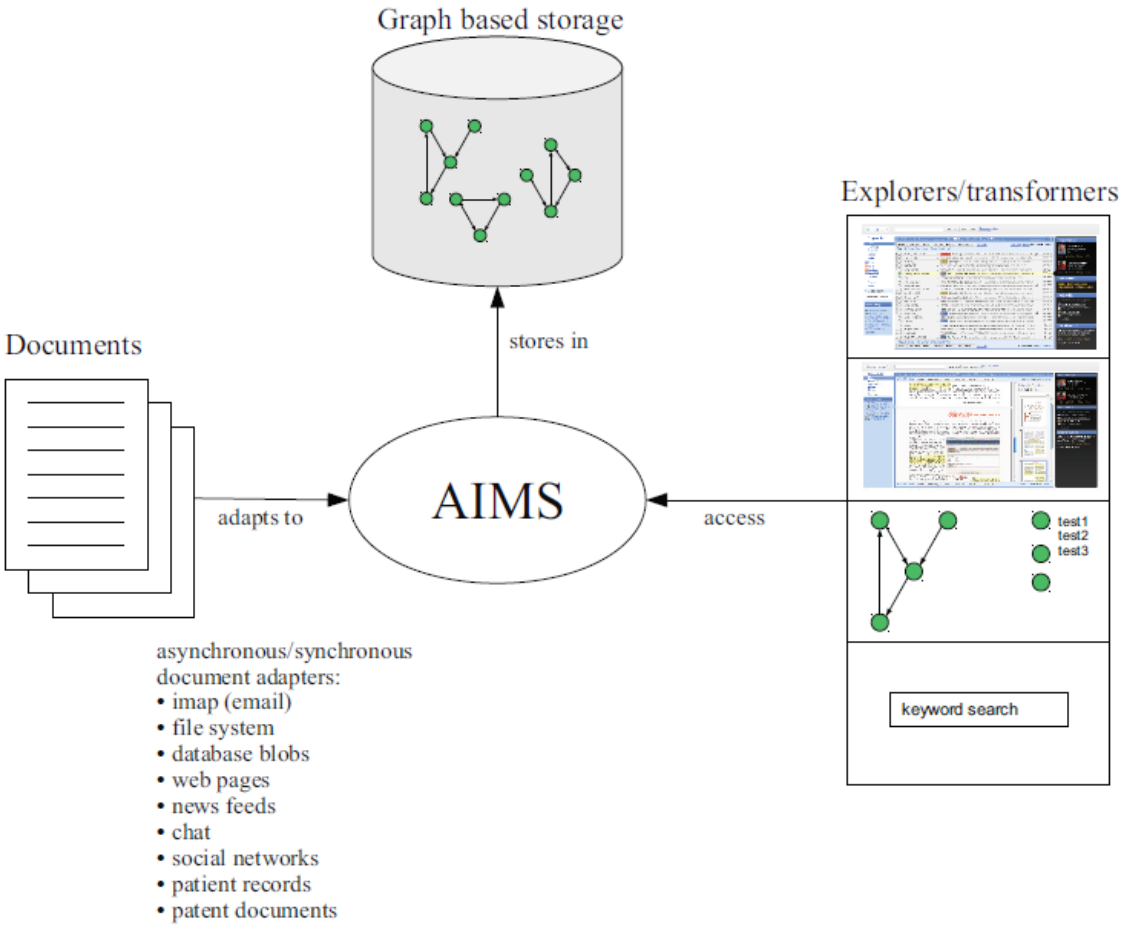

Figure 1.1: Coarse AIMS project architecture (taken from [12])

The Email Information Concentrator is a part of the Advanced Information Management System (AIMS) research project. The AIMS project addresses the growing message overload caused by digital documents, web pages, blogs, emails, chat, Facebook, and Twitter that we are experiencing. It will combine and research appropriate technologies from text mining, graph based storage and exploration and advanced user interfaces.[12] In Figure 1.1 is a coarse overview of the architecture of this project. It puts the employed technologies into context: documents are delivered by the means of asynchronous or synchronous document adapters, adapted to the AIMS and stored in graph databases. Explorers and transformers can access this data through the AIMS.

Tõnu Tamme is doing knowledge extraction with Prolog out of large e-mail repositories. His main goal is to provide better tools for information and knowledge extraction from large e-mail repositories. His application uses human language technology in order to categorize and classify content of email messages. Simple frequency heuristic on the paragraphs and semantic linguistic tools like WordNet allow to build topic-based relations between paragraphs of messages. Using semantic relations between topics complex search queries can be composed to find relevant information from emails.[15] The Email Information Concentrator provides input for Tõnu's application as Prolog statements.

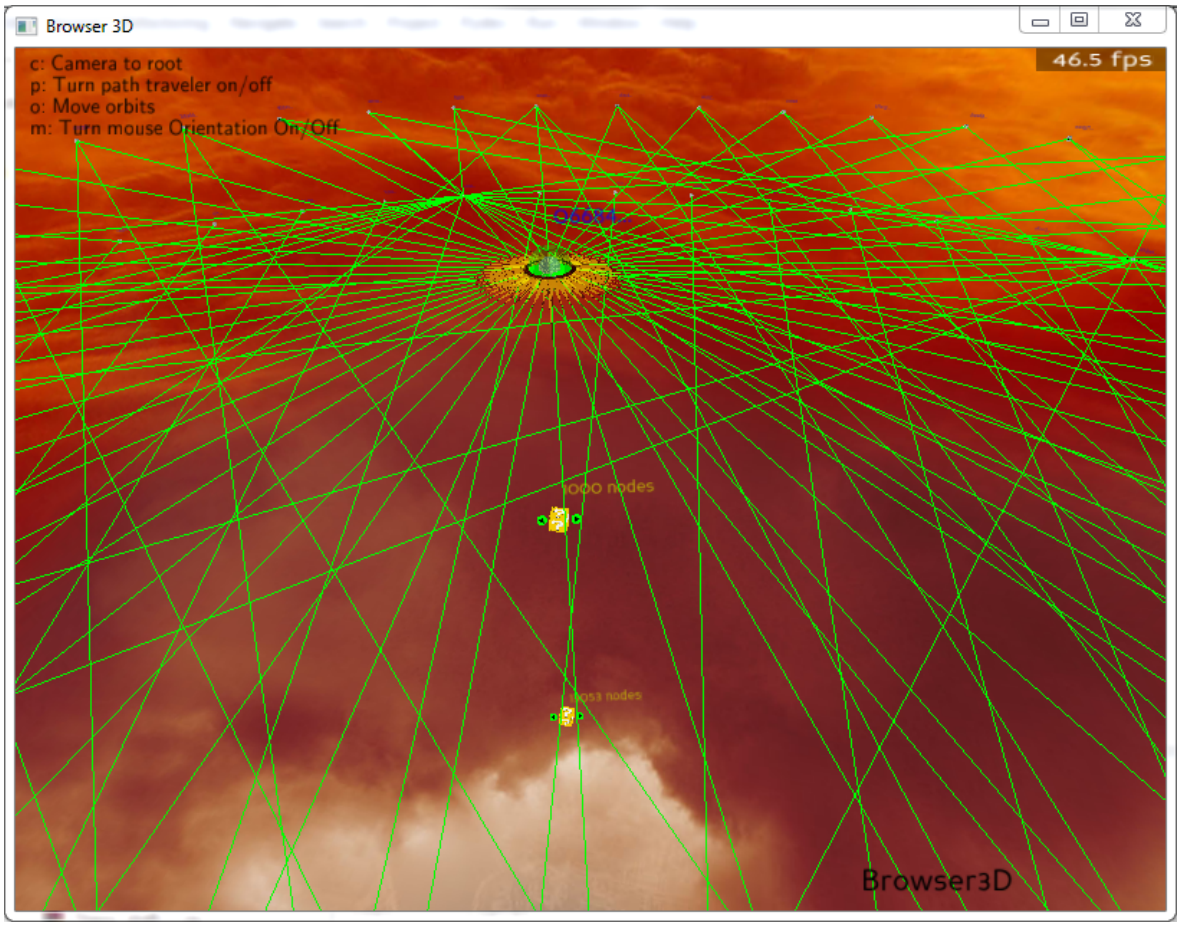

Figure 1.2: Dmitri Danilov's graph visualization

Dmitri Danilov provides several solutions for 3D graph visualization problems in context of exploratory search in his Master thesis. His implementation represents a 3D application that visualizes the directed graph with a 2.5D layout and provides the possibility to explore the graph data.[8] The base of the application is a 3D game engine Panda3D. The visualization software is compatible with my Email Information Concentrator, thus making it a perfect tool for visualizing an Email Graph in 3D space. A screenshot of the 3D Graph Exploration application is presented in Figure 1.2.

1.3 Email message format

The Internet e-mail message format is defined in RFC 5322 and a series of RFCs, RFC 2045 through RFC 2049, collectively called, Multipurpose Internet Mail Extensions, or

MIME.[4]

Internet email messages consist of two major sections which are separated from one another by a blank line:

Header — contains fields such as summary, sender, receiver, and other information about the email.

Body — the message itself as unstructured text; sometimes containing a signature block at the end. This is exactly the same as the body of a regular letter.[4]

According to RFC 5322, the only required header fields are the origination date field and the originator address field(s). All other header fields are syntactically optional. All header fields have the same general syntactic structure: a field name, followed by a colon, followed by the field body.[13] The most common header fields are the following:

Date: — specifies the date and time at which the creator of the message indicated that the message was complete and ready to enter the mail delivery system[13]

From: — specifies the author(s) of the message, that is, the mailbox(es) of the person(s) or system(s) responsible for the writing of the message[13]

Sender: — specifies the mailbox of the agent responsible for the actual transmission of the message. For example, if a secretary were to send a message for another person, the mailbox of the secretary would appear in the "Sender:" field and the mailbox of the actual author would appear in the "From:" field.[13]

Reply-To: — indicates the address(es) to which the author of the message suggests that replies be sent[13]

To: — contains the address(es) of the primary recipient(s) of the message[13]

Cc: — contains the addresses of others who are to receive the message, though the content of the message may not be directed at them[13]

Bcc: — contains addresses of recipients of the message whose addresses are not to be revealed to other recipients of the message[13]

Message-ID: — a unique message identifier that refers to a particular version of a particular message. The uniqueness of the message identifier is guaranteed by the host that generates it.[13]

In-Reply-To: — contains the contents of the "Message-ID:" field of the message to which this one is a reply (the "parent message")[13]

References: — contains the contents of the parent's "References:" field followed by the contents of the parent's "Message-ID:" field[13]

Subject: — contains a short string identifying the topic of the message[13]

Comments: — contains any additional comments on the text of the body of the message[13]

Keywords: — contains a comma-separated list of important words and phrases that might be useful for the recipient[13]

Most modern graphic email clients allow the use of either plain text (*text/plain*) or HTML (*text/html*) for the message body at the option of the user. HTML email messages often include an automatically generated plain text copy as well, for compatibility reasons.[4]

1.4 Internet Message Access Protocol

The Internet Message Access Protocol (IMAP) is one of the two most prevalent Internet standard protocols for e-mail retrieval, the other being the Post Office Protocol (POP).[6] Version 4rev1 (IMAP4rev1) allows a client to access and manipulate electronic mail messages on a server. IMAP4rev1 permits manipulation of mailboxes (remote message folders) in a way that is functionally equivalent to local folders. IMAP4rev1 also provides the capability for an offline client to resynchronize with the server.[7]

IMAP4rev1 includes operations for creating, deleting, and renaming mailboxes, checking for new messages, permanently removing messages, setting and clearing flags, Internet email message parsing, searching, and selective fetching of message attributes, texts, and portions thereof.[7] Messages in IMAP4rev1 are accessed by the use of numbers. These numbers are either message sequence numbers or unique identifiers.[7]

In the process of making the Asynchronous Delivery Agent (see Subsection 3.2.1), I had to study various client commands of IMAP defined by RFC 3501. Here is a list of some of the more common client commands:

LOGIN arguments: login, password; identifies the client to the server and carries the plaintext password authenticating this user[7]

SELECT arguments: mailbox name; selects a mailbox so that messages in the mailbox can be accessed[7]

CREATE arguments: mailbox name; creates a mailbox with the given name[7]

DELETE arguments: mailbox name; permanently removes the mailbox with the given name[7]

CLOSE permanently removes all messages that have the \Deleted flag set from the currently selected mailbox, and returns to the authenticated state from the selected state[7]

SEARCH arguments: searching criteria; command searches the mailbox for messages that match the given searching criteria. Searching criteria consist of one or more search keys, for example NEW, SUBJECT <string> , LARGER <n> , etc.[7]

FETCH retrieves data associated with a message in the mailbox[7]

IMAP4rev1 also specifies server responses for the above-mentioned (and non-mentioned) client commands.

1.5 Roundup

In this chapter I defined the message overload problem and studied related and derived projects. The studied resources confirm that the message overload problem is the subject of a number of research projects. I also investigated the email message format and the IMAP protocol.

Chapter 2

Requirement analysis

Before the implementation phase, requirements were collected and assessed. The following chapter both defines what the Email Information Concentrator is supposed to do as well as outlines the qualities it will possess. Main scenarios are also listed.

2.1 Functional requirements

The Email Information Concentrator is required to fulfill the following functional requirements:

- Importing electronic mail messages using secure IMAP

  This means that the user can download the contents of electronic mail messages from an IMAP mail server over an SSL encrypted socket.

- Parsing message header and body fields

  Extract all relevant header fields, filenames of attachments from the message header. In addition, the mailbox parser must output the contents of the message body.

- Constructing email graph with NetworkX

  Find a way to generate an Email Graph out of the mailbox and its messages. An external package NetworkX can be used for adding nodes and edges to the graph.

- Producing Prolog statements out of mailbox contents

  A user can generate Prolog statements that represent the content of the mailbox. This output includes header, attachment and body information.

- Providing simple heuristics for paragraph information extraction

  Devise a heuristic for detecting where one paragraph ends and another starts in *text/html* as well as *text/plain* messages.

- Extracting relevant word, line and paragraph information from message body

  Parsers output message body in a structured way: paragraphs, lines and words.

2.2 Non-functional requirements

Usability:

- Email delivery agents must display the progress of importing messages to the end-user

  For example, if a user imports her mailbox, she can see how many messages have been downloaded as well as the number of total messages in the mailbox.

- Mailbox parsers must notify the user when they need additional user input or have finished execution

  A user is notified like: "Parsing complete! Press Enter to terminate." Also, the system asks for credentials if needed.

- It must be usable for computer science professionals with previous Python experience after reading the manual and scenarios

  If somebody has a degree in computer science and has programmed in Python before, the provided manual along with scenarios ought to enough to use the application.

Reliability:

- The application may not contain any known critical functional errors on delivery

  All known critical functional errors must be fixed before delivery. This does not apply to non-functional nor unknown functional errors.

- The Email Information Concentrator must not crash if set up and used correctly

  The Email Information Concentrator may not crash if the user follows every instruction in the manual and in scenarios.

Performance:

- All implemented parsers must be able to parse 2500 messages in less than 10 seconds on a modern computer (dual-core CPU 2GHz or more, 2 GB RAM or more)

  If you have a modern computer that passes the given hardware requirements, parsing 2500 messages must take less than 10 seconds in most cases (51% or more)

Support:

- End-users will not be personally assisted

  There will be no helpline, wiki, blog or any other resource in addition to the manual and scenarios.

2.3 Scenarios

In this section I cover the most frequent scenarios in detail. These include importing a mailbox, using the Prolog Parser and the Graph Parser.

2.3.1 Importing a mailbox

As a user, I can import my mailbox via secure IMAP and have it saved on my hard drive. I open the `EmailConcentrator.py` file and comment in the following lines:

```
async = AsynchronousDeliveryAgent.AsynchronousDeliveryAgent
('servername', portnumber, login, password, 'outputfilename')
async.fetchMail()
```

I make sure that the server name, port number and output file name are correct for me. For example:

```
async = AsynchronousDeliveryAgent.AsynchronousDeliveryAgent
('imap.gmail.com', 993, login, password, 'test.mbox')
async.fetchMail()
```

Then I save the file, run it, type my user name and password when prompted and my mailbox gets transferred into my computer.

2.3.2 Generating Prolog statements

As a user, I can generate Prolog statements from my mailbox which I have imported earlier. To do that, I simply open the `EmailConcentrator.py` file and comment in the following lines:

```
prolog = PrologParser.PrologParser
('mailboxname', 'outputfilename')
prolog.displayMail()
```

I verify that my mailbox is on the project root folder and specify the correct settings like that:

```
prolog = PrologParser.PrologParser
('myMailbox.mbox', 'myOutput.pl')
prolog.displayMail()
```

After that I save the file, run it and the system will produce Prolog statements out of my mailbox and save them where I wanted.

2.3.3 Generating an Email Graph

As a user, I can generate an Email Graph from my mailbox which I have imported earlier. To do that, I simply open the `EmailConcentrator.py` file and comment in the following lines:

```
egraph = GraphParser.GraphParser
('mailboxname', 'outputfilename')
egraph.pickleGraph()
```

I change the mailbox name and output file name if necessary and make sure that I have the mailbox file on my project root folder along with the `EmailConcentrator.py` file and other files.

```
egraph = GraphParser.GraphParser
('myMailbox.mbox', 'myGraph.txt')
egraph.pickleGraph()
```

Upon saving and running the file, this pickles an Email Graph object which I can later read in my programs like:

```
import pickle
import networkx
graph = pickle.load(open('myGraph.txt', 'rb'))
```

This operation also depends on the presence of `BodyParser.py`, `EmailGraph.py` and Beautiful Soup which can be downloaded from `http://www.crummy.com/software/BeautifulSoup/`. I have successfully loaded an Email Graph.

2.4 Roundup

In this chapter I reviewed the functional and non-functional requirements for the Email Information Concentrator. A selection of common scenarios were also listed as a reference.

Chapter 3

The Email Information Concentrator

In this chapter I describe the design and implementation of the library. Many code samples as well as a class diagram are presented to give an overview of what has been done in this project.

3.1 Design

This application consists of two abstract modules that operate separately: a mail delivery agent and a mailbox parser. Each abstract module has different implementations (see Figure 3.1). For instance, two different parsers are implemented, one yielding a graph representation of the mailbox and the other corresponding Prolog statements. All current parsers use the Body Parser to parse email body. It yields a message split into lists of paragraphs, lines and words.

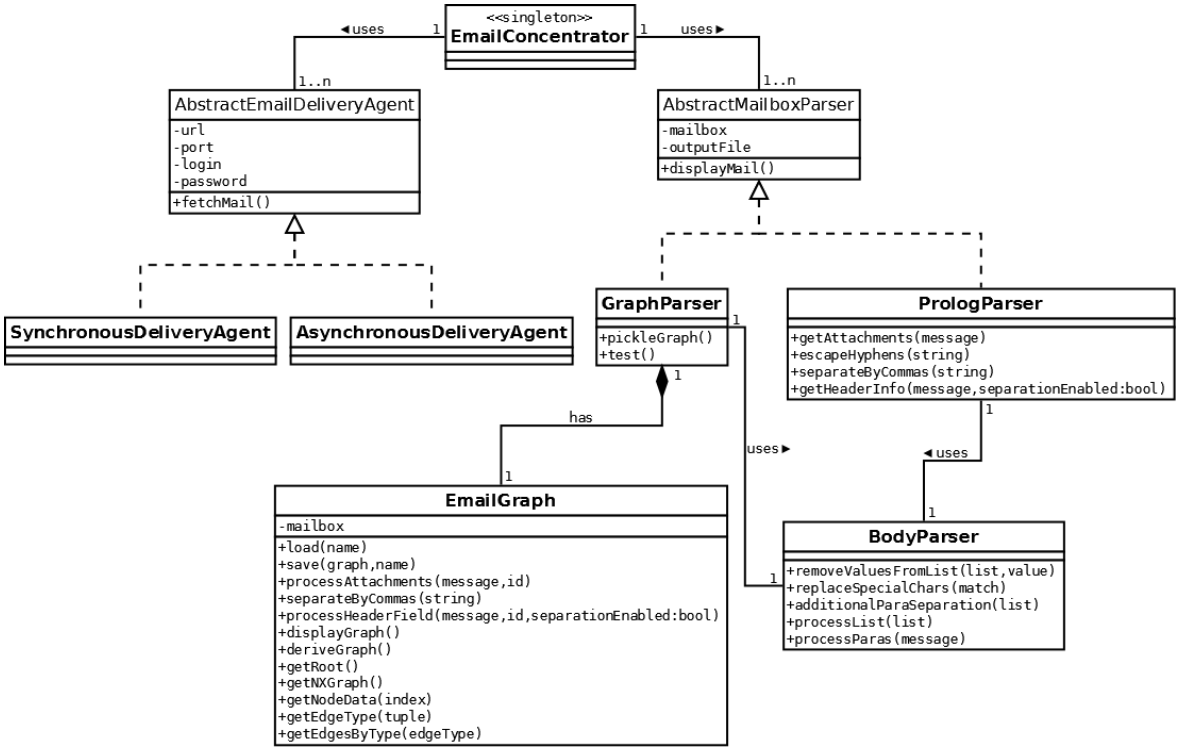

Figure 3.1: Class diagram

The program is designed to support new implementations for these modules if the need arises. The Email Information Concentrator is coded in Python 2.6.5. Two external packages are also used, namely Beautiful Soup and NetworkX, for parsing HTML contents in messages and for the creation of email graphs correspondingly.

3.2 Implementation

3.2.1 Email delivery

I have a class `AbstractEmailDeliveryAgent` that contains an abstract function `fetchMail()`. All delivery agents are supposed to extend that class and implement its abstract functions. At the moment, there is just one such implementation named `AsynchronousDeliveryAgent`. Its sole purpose is to fetch email messages using the IMAP protocol over an SSL encrypted socket. I used a class called `IMAP4_SSL` of the `imaplib` module from the Python Standard Library. This class encapsulates a connection to an IMAP4 server and implements a large subset of the IMAP4rev1 client protocol (see Section 1.4).[2] `SynchronousDeliveryAgent` was also planned in the design of the Email Information Concentrator but was later postponed due to lack of time. Here, the word "asynchronous" means getting all messages at once and then transforming them. "Synchronous" can be seen as getting messages which were changed since the last time.

```
def fetchMail(self):
        box = mailbox.mbox(self.destination,
        factory=None, create=True)
        parser = Parser()
        server = imaplib.IMAP4_SSL(self.url, self.port)
        server.login(self.login, self.password)
        messages = server.select()[1][0]
        print 'Importing all', messages, 'messages.'
        resp, data = server.search(None, 'ALL')
        for num in data[0].split():
                resp, data = server.fetch(num, '(RFC822)')
                label = num + "/" + messages + ": " + resp
                print label
                emailMessage = parser.parsestr(data[0][1])
                box.add(emailMessage)
        server.close()
        server.logout()
        print 'Fetching complete.'
```

The code above uses a Parser object. This object takes either a string representation of a message (for instance, a string downloaded from an email server in my case) or a file containing flat text of an email and turns it into the appropriate Python "Message" object representing the email.[14] I also define a secure IMAP connection, open a new session and fetch all messages from the "Inbox" folder (`server.select()[1][0]` defaults to that). A user can see the progress of the operation: how many messages in total, how many of them already transferred.

3.2.2 Prolog parser

One of the implemented parsers is the Prolog Parser. All parsers extend a class `AbstractMailboxParser` and implement its function `displayMail()`. The overall structure of its output data is:

```
mbox([
        message([
                {body}
                {header}
                {attachments}
        ]),
        {other messages}
]).
```

Here is a simplified (some header fields omitted) real-life example of this output:

```
mbox([message([[[['first', 'of', 'all', 'you', 'are', 'more', '
   liberal', 'than', 'Farrakhan', 'and', 'secondly', 'he'], ['
   correct', 'the', 'tea', 'baggers', 'and', 'radical', 'white',
   'republicans', 'would', 'love'], ['have', 'obama', 'killed']],
    [['On', 'Mar', '2, ', '8:27', 'pm, ', 'Dave', '<K19j...@yahoo
   .com>'], ['...Speaking', 'to', 'an', 'estimated', '20, 000', '
   followers', 'of', 'the', 'black'], ['movement', 'at', 'the', '
   United', 'Center', 'on', 'Sunday, ', 'the', '76-year-old'], ['
   said, ', '"The', 'white', 'right', 'is', 'trying', 'to', 'set'
   , 'Barack', 'up', 'to']], [['You', 'received', 'this', '
   message', 'because', 'you', 'are', 'subscribed', 'to', 'the',
   'Google', 'Groups'], ['Political', 'Forum"']], [['To', 'post',
    'to', 'this', 'group, ', 'send', 'email', 'to']], [['To', '
   unsubscribe', 'from', 'this', 'group, ', 'send', 'email', 'to'
   ]], [['For', 'more', 'options, ', 'visit', 'this', 'group', '
   at']]], ['Message-ID:', '<02aa8aa5-d826-4098-91b9-2
   fe24d742377@g7g2000yqe.googlegroups.com>'], ['In-Reply-To:', '
   <4b6cafb8-b95c-473f-aa33-a10b2a67810e@t41g2000yqt.googlegroups
   .com>'], ['References:', '<4b6cafb8-b95c-473f-aa33-
   a10b2a67810e@t41g2000yqt.googlegroups.com>'], ['Subject:', 'Re
   : Liberals like Farrakhan need racism to keep the money coming
    in'], ['From:', '"mike [ the proud liberal / socialist ]
   532!"<littlemike532@gmail.com>'], ['Sender:', '
   abc_politics_forum@googlegroups.com'], ['Reply-To:', '
   abc_politics_forum@googlegroups.com'], ['To:', 'Political
   Forum<abc_politics_forum@googlegroups.com>'], ['Date:', 'Wed,
   3 Mar 2010 01:39:53 -0800 (PST)']])]).
```

The Prolog Parser uses Body Parser to get process email body which is discussed in subsection 3.2.4 in more detail. It uses a function `getHeaderInfo()` to extract header fields such as "Subject", "Message-ID", "From", etc. These are written into the given output file according to the required Prolog statements structure. The following code gets a message as an input parameter and prints out all attachment names.

```
def getAttachments(message, fileOut):
        for part in message.walk():
                filename = part.get_filename()
```

```
if not filename:
        filename = part.get_param
        ('name', None, 'content-type')
if filename != None:
        fileOut.write("\t\t['Attachment:', "
        + repr(escapeHyphens(filename)) + "], \n")
```

3.2.3 Graph parser

The Graph Parser is an email parser with the aim of generating a NetworkX graph. This parser was needed by our research group as an input for graph 3D visualization. By definition, a graph is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). I use graphs to make a complex system from the data parts of the email messages. The idea is to keep only unique mail message data parts and create relations for the duplicates. This technique allows to reduce the data duplication.

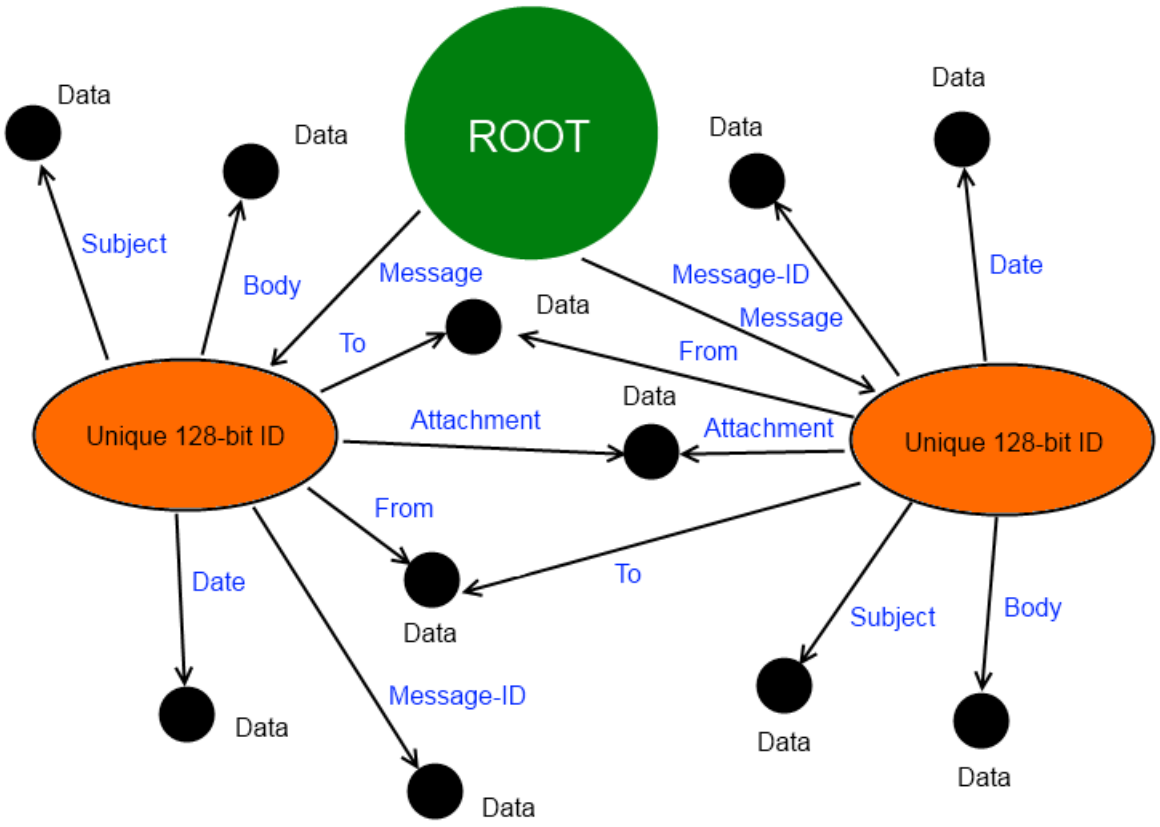

Figure 3.2: Email Graph model

Every mailbox has a root node, message nodes and data nodes. A unique id was generated for every message in mailbox and the root node as well. That way we can link multiple mailboxes in the future should we want to analyze their contents relative to each other. Edges were added between the mailbox root node and message nodes (represents a message) in that mailbox with type "Message". In order to link messages to their data information (header fields, attachment names, structured body information), data nodes were created to store that information. Edges were added

between these message nodes and their data attributes. Every edge has a descriptive label attached to it, such as "Attachment", "To", "Received", etc. This model is shown in Figure 3.2 and was implemented in `EmailGraph.py` file.

I used the Python package NetworkX (`http://networkx.lanl.gov/`) to represent a graph. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It supports both undirected and directed graphs and has the following classes: `Graph` for undirected simple graphs, `DiGraph` for directed simple graphs, `MultiGraph` and `MultiDiGraph` for graphs with parallel edges. [5] I decided to use `MultiDiGraph` because email messages contain several header fields that can have identical information, for example "Sender" and "Reply-To". However, there can be no self-loops in our Email Graphs.

Also, I added functions for basic lookup operations: `getRoot()` returns mailbox root node's index, `getNXGraph()` returns the NXGraph (type `MultiDiGraph`), `getEdgeType(tuple)` returns the type(s) of a given edge tuple and `getEdgesByType(type)` returns all edges of a given type.

Finally, it is obvious that the results of the Graph Parser need to be saved somehow. Here I use Python object serialization module `pickle`. According to Python 2.6.5 documentation, the pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy.[1] Notice that one must open a file for pickling or unpickling an Email Graph in binary mode:

```
f=open(name, "wb")
```

or

```
f=open(name, "rb")
```

3.2.4 Line and paragraph information extraction

Line and paragraph information extraction can be considered a fundamental part of the Email Information Concentrator. It is implemented in `BodyParser.py` and is currently used both in Prolog and Graph Parser. As shown below, the Body Parser produces a double-list. The outer list is a paragraph. It can have one or more lines. Lines are in essence lists of words.

```
[['Afghanistan', 'on', 'Monday', 'announced', 'a', 'ban', 'on',
   'news', 'coverage', 'showing', 'Taliban'], ['attacks, ', '
   saying', 'such', 'images', 'embolden', 'the', 'Islamist', '
   militants, ', 'who', 'have'], ['launched', 'strikes', 'around'
   , 'the', 'country', 'as', 'NATO', 'forces', 'seize', 'their'],
    ['southern', 'strongholds.']], [['The', 'announcement', 'came
   ', 'on', 'a', 'day', 'when', 'the', 'NATO-led', 'International
   '], ['Security', 'Assistance', 'Force', '(ISAF)', 'fighting',
   'the', 'Taliban', 'reported', 'six', 'of'], ['its', 'service',
    'members', 'had', 'been', 'killed', 'in', 'various', 'attacks
   .']], [['Journalists', 'will', 'be', 'allowed', 'to', 'film',
   'only', 'the', 'aftermath', 'of', 'attacks, '], ['when', '
   given', 'permission', 'by', 'the', 'National', 'Directorate',
```

```
'of', 'Security', '(NDS)'], ['spy', 'agency, ', 'the', 'agency
', 'said.', 'Journalists', 'who', 'film', 'while', 'attacks',
'are'], ['under', 'way', 'will', 'be', 'held', 'and', 'their',
 'gear', 'seized.']]
```

The overall idea is to extract paragraphs, lines and words from a message. With that in mind, I had to determine where one paragraph or line ends and another starts. *text/plain* and *text/html* messages had to be treated differently. In the first place HTML messages contained a lot of unwanted meta-information for formatting which was often invalid HTML. Plain text was cleaner in that sense but it could still have elements of HTML as character entities, for instance " " or "Õ". In addition, I decided to remove greater than (">") characters from line beginnings to focus on human-generated content. However, this can only be considered a temporary solution until we need that meta-information. Therefore, the absence of these characters will be a serious problem later and this feature might have to be removed (see Section 3.3). Another fundamental difference between these two text types is the fact that there is no such thing as a line in *text/html* messages. Therefore, plain text messages have another layer of analysis: lines.

A simple heuristics was developed to find where paragraphs start and end in *text/plain* messages. Like any heuristic, these experience-based techniques that help in problem solving do not guarantee 100% success. However, in most cases they are applicable. The algorithm itself is following:

```
1) Introduce paragraph boundaries on empty lines.
2) If there were three or less
empty lines in a message, do the following:
        3)Iterate over all existing paragraphs and lines.
                4)If the last word in a line ends
                with full stop, comma, exclamation mark or quotes,
                mark this as the new paragraph ending.
```

An external module called Beautiful Soup (`http://www.crummy.com/software/BeautifulSoup/`) was used for messages with content type *text/html*. Beautiful Soup is a Python HTML/XML parser designed for quick turnaround projects. The way I used it was first getting all textual elements (no HTML tags nor attributes) by:

```
soup = BeautifulSoup.BeautifulSoup(''.join(part.get_payload()))
initialList = eval(str(soup.findAll(text=True)))
```

Here "part" denotes a part of a multipart message. According to Beautiful Soup documentation, you can pass in the special value True, which matches every tag with a name: that is, it matches every tag.[3] This code fragment produces the initial list which was processed further by me. I replaced HTML character entities with human-readable characters, removed some of the more common errors introduced by Beautiful Soup and split the list on new line characters into sublists (paragraphs). Some of that work was done by Beautiful Soup, my task was to find other meaningful paragraph breakpoints. Finally, I had to make sure that both parsing methods (*text/plain* and *text/html*) return output with the same structural patterns (words inside lines, lines make up paragraphs).

3.3 Future work

Email Information Concentrator will be the base for further works in the information management for emails. Especially the Synchronous Delivery Agent will be one of the next steps. It could make the overall user experience much better and reduce the network load as it will download messages which were changed since the last time as opposed to fetching the whole mailbox. The file `SynchronousDeliveryAgent.py` already exists in the project root folder but is not implemented yet.

In addition, reconstructing discussion threads remains a challenging but required task. This will also mean reintroducing greater than (“>”) characters to the Body Parser output that were removed for clarity reasons. Because it is proven in existing studies that identifying and reconstructing discussion threads cannot be done reliably by solely applying a header analysis, it is necessary to choose something more complex for the task.[11] The starting point could be an algorithm which searches for reply indicators in the subject line, the message body, and the message headers each in order to correctly identify a message as a reply or a parent.

Finally, many optimizations are still pending to speed up the handling of large-scale mailbox traffic. One of the very first steps here is a thorough performance testing to spot possible bottlenecks in the implementation of the Email Information Concentrator. Preparations for this have been done as well: I was subscribed to six active mailing lists and collected nearly 50 000 messages over the period of two months. The collected data can be a valuable input for testing the whole application’s performance under heavy load as well as measuring the detection accuracy of conversation threads.

3.4 Roundup

In this chapter I described the design and implementation of the Email Information Concentrator. The application is currently being used by other researchers for importing and parsing mailboxes. A list of tasks for the future was also given.

Chapter 4

User manual

This chapter explains how to set up and use the Email Information Concentrator.

In order to start using the Email Information Concentrator, you have to abide by the following rules:

1. Download and install Python 2.6.5 (`http://www.python.org/download/`). It might work on older versions of Python but there is no guarantee. This application is not officially compatible with Python 3.x versions.

2. Download and install NetworkX (`http://networkx.lanl.gov/download.html`).

3. Download the source code from the project website (see Appendix A). Create a folder for the Email Information Concentrator and copy all the files there.

4. Download Beautiful Soup (`http://www.crummy.com/software/BeautifulSoup/`) and extract it where the Email Information Concentrator is (only the `BeautifulSoup.py` file needs to be accessible).

5. Configure the `EmailConcentrator.py` file to your needs as shown in scenarios (section 2.3).

6. Run the `EmailConcentrator.py` file by typing "`python EmailConcentrator.py`" on the command line or just double-click the file (works on Windows).

7. When it has finished its work, you will be prompted to press Enter.

8. If you want to analyze something else, start over!

Conclusion

As the main outcome of my Bachelor thesis, an application called Email Information Concentrator and a corresponding library were designed and developed. It consists of two abstract modules that operate separately: a mail delivery agent and a mailbox parser. Each abstract module has different implementations. For instance, two different parsers were implemented, one yielding a graph representation of the mailbox and the other corresponding Prolog statements. The program is designed to support new implementations for these modules if the need arises. Two external packages are also used, namely Beautiful Soup and NetworkX, for parsing HTML contents in messages and for the creation of email graphs correspondingly.

My library includes tools that can extract message header fields defined by RFC 5322. In addition, it can also extract relevant word, line and paragraph information from the message body. A simple heuristic was developed for the extraction of paragraphs from *text/plain* and *text/html* messages. Also, I developed a model for representing an Email Graph. My idea is to keep only unique mail message data parts and create relations for the duplicates. This technique allows to reduce the data duplication. The output of the Email Information Concentrator can be used as an input for further research in related research projects.

The Email Information Concentrator will be the base for further works in the information management for emails. Especially the Synchronous Delivery Agent will be one of the next steps. It could make the overall user experience much better and reduce the network load as it will download messages which were changed to since the last time as opposed to fetching the whole mailbox. In addition, reconstructing discussion threads remains a challenging but required task. Finally, many optimizations are still pending to speed up the handling of large-scale mailbox traffic.

The Email Information Concentrator has already successfully established itself as an integral part of the knowledge managemant research. Other researchers are using it in their branches for importing and parsing mailboxes.

Elektronposti informatsiooni koondaja

Peeter Jürviste

Bakalaureusetöö (6 EAP)

Sisukokkuvõte

Elektronpost on üks edukamaid ja enamlevinud rakendusi arvutile, mis kunagi leiutatud. Tänapäeval seisame aga silmitsi üha suureneva info ülekoormuse probleemiga. Käesolev töö on osa teadmushalduse vallas tehtavast uurimistööst eesmärgiga info ülekoormuse probleemiga toime tulla. Töö tulemusena valmiski elektronposti informatsiooni koondaja. Viimase eesmärgiks on elektroonilise kirjakasti sisu allalaadimine ja kindlal viisil parsimine, luues näiteks vastava sisuga graafi või Prologi sisendlaused.

Rakendus koosneb kahest osast: meilide kättetoimetamise agent ning kirjakasti parser. Mõlema abstraktse mooduli tarvis on loodud erinevaid teostusi. Olulisemad neist on asünkroonne meilide kättetoimetamise agent, parser kirjakasti sisust kindlaksmääratud omadustega graafi genereerimiseks ning parser elektroonilise postkasti sõnumite konverteerimiseks Prologi lauseteks. Programm on disainitud pidades silmas seda, et uusi realisatsioone oleks lihtne lisada ja liidestada olemasoleva rakendusega, kui vastav vajadus peaks kunagi tekkima. Elektronposti informatsiooni koondaja on kirjutatud programmeerimiskeeles Python. Lisaks kasutati ka järgmisi väliseid teeke: 1) Beautiful Soup - HTML sisu parsimiseks sõnumite sees; 2) NetworkX - emaili graafi loomiseks.

Valminud arvutiprogramm suudab täita järgmisi ülesandeid: laadida alla elektrooniline kirjakast meiliserverist, kasutades turvalist IMAP protokolli (*IMAP over SSL*); leida sõnumite seest emaili päise elemendid ja manuste nimed; eraldada sõnumi sisust seal leiduvad üksikud sõnad, read ja tekstilõigud; genereerida postkasti sisu pealt vastav graaf. Lõigupiiride kindlakstegemiseks töötati välja spetsiaalselt selleks otstarbeks mõeldud lihtne heuristika. Mõeldi välja mudel, kuidas emaili graafina kujutada, püüdes vähendada seejuures andmeliiasust sõnumi osade graafis hoidmisel.

Hetkel on rakenduse peamine kasutusala teadmushalduse alastes uurimisprojektides, kus selle abil saab pakkuda teistele tööriistadele sisendinfot. Ehkki valmis reaalselt töötav programm, saab seda oluliselt edasi arendada nii jõudluse kui ka lisavõimaluste poolelt. Prioriteetsemad tööd tuleviku jaoks on näiteks sünkroonse meilide kättetoimetamise agendi loomine osana elektronposti informatsiooni koondajast ning meilide ahelate tuvastamine ja kujutamine graafis. Need ülesanded jäävad tulevikku.

Kokkuvõttes võib tööd pidada kordaläinuks, sest saadi valmis töötav rakendus, mida kasutatakse juba ka reaalselt projekti teistes harudes. Näen elektronposti informatsiooni koondajas suurt perspektiivi tuleviku jaoks, kui seda sihipäraselt edasi arendada.

Bibliography

[1] 11.1. pickle — python object serialization — python v2.6.5 documentation. Available from: `http://docs.python.org/library/pickle.html` [Last visited May 26, 2010].

[2] 20.10. imaplib — IMAP4 protocol client — python v2.6.5 documentation. Available from: `http://docs.python.org/library/imaplib.html` [Last visited June 2, 2010].

[3] Beautiful soup documentation. Available from: `http://www.crummy.com/software/BeautifulSoup/documentation.html` [Last visited May 26, 2010].

[4] e-mail - wikipedia, the free encyclopedia. Available from: `http://en.wikipedia.org/wiki/E-mail` [Last visited May 30, 2010].

[5] Graph types — NetworkX v1.1 documentation. Available from: `http://networkx.lanl.gov/reference/classes.html` [Last visited May 26, 2010].

[6] Internet message access protocol - wikipedia, the free encyclopedia. Available from: `http://en.wikipedia.org/wiki/Internet_Message_Access_Protocol` [Last visited May 30, 2010].

[7] M. Crispin. RFC 3501 - internet message access protocol - version 4rev1, 2003. Available from: `http://tools.ietf.org/html/rfc3501` [Last visited May 30, 2010].

[8] Dmitri Danilov. *3D Graph Exploration*. Master thesis, Tartu, 2010.

[9] D. Fisher, A. J. Brush, E. Gleave, and M. A Smith. Revisiting whittaker & sidner's email overload ten years later. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, page 312, 2006.

[10] Q. Jones, G. Ravid, and S. Rafaeli. Information overload and the message dynamics of online interaction spaces: A theoretical model and empirical exploration. *Information Systems Research*, 15(2):194–210, 2004.

[11] D. D Lewis and K. A Knowles. Threading electronic mail: A preliminary study. *Information Processing & Management*, 33(2):209–217, 1997.

[12] Quretec Ltd. Advanced information management system (AIMS), 2009.

[13] Paul Resnick. RFC 5322 - internet message format, 2008. Available from: `http://tools.ietf.org/html/rfc5322` [Last visited May 30, 2010].

[14] Michael Swanson. Page 2 - python email libraries: SMTP and email parsing. Available from: http://www.devshed.com/c/a/Python/Python-Email-Libraries-SMTP-and-Email-Parsing/1/ [Last visited May 29, 2010].

[15] Tõnu Tamme, Ulrich Norbisrath, and Georg Singer. Improvement of email handling through human language technology. Riga, Latvia, October 2010.

[16] S. Whittaker and C. Sidner. Email overload: exploring personal information management of email. In *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground*, page 276–283, 1996.

Appendix A

Resources

The **PDF version** of my Bachelor thesis along with the **source code** and other resources are available at the project's web site:
`http://ulno.net/projects/emailconcentrator`.