Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Jan Kundrát

IMAP E-mail Client

Department of Software Engineering

Supervisor: Mgr. Vlastimil Babka
Study Program: Computer Science, Programming

2009

I'd like to thank my supervisor, Mgr. Vlastimil Babka, for his numerous advices during the writing of this thesis, Ms. Anna Adamcová for her great patience and support, and my parents for supporting my studies.

I hereby declare that I wrote this thesis myself using the referenced sources only. I also agree with lending and publishing of this thesis.

Prague, May 29, 2009 Jan Kundrát

Contents

Název práce: IMAP E-mail Client
Autor: Jan Kundrát
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Vlastimil Babka
E-mail vedoucího: Vlastimil.Babka@mff.cuni.cz

Abstrakt: Předložená práce popisuje implementaci pokročilého klienta pro práci s poštou pomocí protokolu IMAP. Hlavním zaměřením je podpora relevantních standardů s důrazem na efektivní implementaci; využívá se pokročilých vlastností IMAPu, jako například IDLE notifikací, parsování zpráv na straně serveru či lokálního ukládání e-mailů s možností práce offline. Spojení s IMAP serverem je realizováno protokolem TCP s volitelnou možností TLS šifrování či přes lokálně bežící proces. Projekt umožňuje psaní a odesílání jednoduchých mailů, podporováno je odesílání pomocí protokolu SMTP a sendmailem. Cílová platforma je framework Qt na Linuxu, avšak aplikace je portovatelná i na jiné platformy.

Klíčová slova: IMAP, e-mail, MIME, Qt, SMTP

Title: IMAP E-mail Client
Author: Jan Kundrát
Department: Department of Software Engineering
Supervisor: Mgr. Vlastimil Babka
Supervisor's e-mail address: Vlastimil.Babka@mff.cuni.cz

Abstract: This thesis describes the implementation of an advanced IMAP e-mail client, Trojita. Focusing on standards compliance, the client supports a wide range of IMAP features crucial for an efficient implementation such as IDLE notifications, server-side message parsing and caching of retrieved body data for offline operation. Connecting to the IMAP server is supported over TCP sockets, optionally secured via the TLS encryption, as well as through a local process. Basic message composition and sending via both SMTP and a local sendmail instance is supported. The target platform is the Qt framework on Linux, but the application is reasonably portable.

Keywords: IMAP, e-mail, MIME, Qt, SMTP

Chapter 1

Introduction

Although being surpassed in popularity by other technologies like the World Wide Web and peer-to-peer applications, the electronic mail is still one of the few services that an average user can name when talking about the Internet. There are many ways how to send and receive e-mail, some of them being more popular than the others. In this thesis, we try to resurrect the older approach to reading e-mail, that is, using an standalone application.

In the recent years, the improvements in the field of web browsers and related technologies allowed rapid development of new features that were not conceivable just five years ago. Using these new functions, the popularity of web-based e-mail readers grew accordingly. However, using a proper standalone application has numerous benefits from being able to work when not connected to the Internet to providing a truly instant feedback, a feature that is still somewhat missing from most of the web applications. Managing e-mail over a standardized protocol also avoids vendor lock-in, a highly dangerous phenomenon associated with using proprietary messaging solutions. To date, the *Internet Message Access Protocol* [1] (or IMAP) is the only such protocol.

1.1 Motivation

Implementing an IMAP client is far from trivial. While there are certainly numerous stable and widely-used Mail User Agents on the market, none of them fulfills all the expectations of reasonable performance and efficiency. Indeed, most of these applications started as MUAs speaking the POP3 protocol only, adding support for IMAP later on. Typically, this support was implemented long since the initial design phase of the product was completed, resulting in suboptimal design choices [2]. Therefore, we believe there is still a place on the market for a stable, highly performing IMAP e-mail application.

1.2 Structure of the thesis

In the following chapter, we provide a gentle introduction to the world of IMAP and related standards such as MIME. Chapter 3 explains some of the design choices made during the development process of Trojita, as well as elaborates on the internal application architecture and structure. An overview of other IMAP implementations on the market is provided in chapter 4, including a comparison about how well they operate when compared to Trojita. The User's Guide is available in chapter 5. Finally, the whole thesis is concluded by a wrap-up providing an overview of what we implemented and how the result is usable.

Chapter 2

IMAP and Related Technologies

Electronic mail, or a SMTP-mail, is a public service suitable for automated message exchange among connected entities. Interoperability is guaranteed by several *Internet standards*, usually codified in the form of RFC documents. Subject to these standards are various protocols specifying the rules of what entities can communicate to each other as well as definition of the format of all transmitted messages. In this section, we provide a gentle introduction to the numerous standards which deal with this highly complex topic.

2.1 Basic Concepts

An *email message*, or a RFC-822[1] message, consists of three parts: *Envelope*, *Header* and *Body*. The *envelope* is the only relevant part for mail exchange among Internet hosts; it includes basic information like addresses of the sender and recipient. A *header* provides more detailed metadata about the message, from human-readable sender addresses and message route traces to user-defined fields. Some pieces of this information are used when the original delivery fails for some reason, like the `Return-Path` header. The *header* also serves as a basis for an *IMAP envelope data* which is explained later in chapter 2.2. Finally, a *message body* is what an average user typically

[1]As defined by the RFC 822 standard and subsequently modified by RFC 2822 [3] and others

refers to as “the e-mail”. It might contain just a plain US-ASCII text, an HTML message with embedded images or it could be a recursively defined entity with rich tree structure.

An *IMAP server* is a host in the Internet which provides access to local mail store via the standard protocol, IMAP4rev1 in this case. The IMAP server might be located in an employer’s data center or on the user’s laptop, for example.

An *MTA*[2] or an *SMTP Server* is an Internet host whose purpose is delivery of the RFC 2822 mail. MTAs in the Internet communicate with each other, determining routing information from MX-records[3] in the DNS. As a common practice nowadays, these servers also accept outgoing e-mail from their own users, provided the connection is properly authenticated, either by simple fact that it originates from a trusted network or that the user has provided her credentials, typically as a user name and password combination or in the form of an X.509 certificate.

A *Mailbox* or a *Mail Folder* is a place on the IMAP server where the messages are logically stored. A mailbox might contain number of child mailboxes if the server implementation allows it. Certain IMAP servers have to be told in advance whether the mailbox just being created is supposed to contain only other mailboxes or only regular e-mail messages, while others allow free mixing of both.

2.2 IMAP-specific Attributes

A message in the IMAP sense is *immutable*, that is, its contents (e.g., header, structure of the body or the actual message parts) cannot change, ever. Each message also has mutable attributes, the most important being *flags*.

Messages in a remote IMAP mailbox are identified by two partially independent numbers, the *sequence number* and the *UID*. Sequence numbers start at one and are consecutive, that is, they dynamically change when a message in the middle of the mailbox is removed. This means that one particular message can have multiple sequence numbers assigned over time and that one sequence number can refer to totally unrelated messages, as the numbering changes.

[2]Mail Transfer Agent

[3]Mail eXchange, a special record in the Domain Name Service which specifies what servers are responsible for accepting all incoming e-mail for the particular domain

On the other hand, the *UIDs* work differently. When a message is delivered to the mailbox, it receives a UID one greater than the previous one. After the UID is assigned, it will never change for that particular message, and the mailbox remembers the biggest assigned UID so far. This means that UIDs are never recycled over the normal life span of a mailbox and they can be used as a persistent identifier to one message in a particular mailbox.

Sometimes a non-standard situation might happen, e.g., when a third-party accessed the mailbox data behind the back of the IMAP server. Perhaps the user tried a utility program that removes huge attachment from an otherwise valuable message, or a spam classifier modified message headers directly, bypassing standard mechanisms for proper flagging. In any such case, the IMAP guarantee of immutable nature of the messages does not hold, and a compliant IMAP server has to communicate this problem to all clients. The *UIDVALIDITY* mailbox attribute serves this purpose. Whenever the UIDVALIDITY changes, a compliant client is required to throw away any cached information about a mailbox because the UID assignment starts over again.

UID and UIDVALIDITY combined together form a 64bit integer which is absolutely guaranteed to be a unique identifier for any message in the given mailbox – if the IMAP client once retrieved any immutable part from mailbox A with this 64bit identifier, subsequent queries for the part identified by the same 64bit number in the same mailbox can be satisfied from the client's cache.

It should be noted that this UID/UIDNEXT numbering is strictly per-mailbox and not per-server, and that different users accessing the same server might see the same 64bit number for totally unrelated messages. While RFC 3501 [1] strongly recommends that UIDVALIDITY should be kept constant if at all possible, a compliant IMAP client has to deal with UIDVALIDITY changes nonetheless.

The *IMAP envelope* is a data structure holding interesting values determined mostly from various message headers. It is *not* the same thing as the SMTP envelope, which typically stores less information.

The *Internal date* holds a time stamp about when the message was delivered to the IMAP server. *Size* attribute is the number of octets required for storing a full copy of message in RFC 2822 format. *Body structure* represents the tree-like structure of the MIME message.

The *IMAP flags* is an attribute holding a set of zero or more named tokens associated with the message. A flag can be either *permanent* or *session-only.* Permanent flags are stored in the mailbox itself, while the session-only flags disappear on a subsequent reconnect to the mailbox. Some of the flags are defined by the RFC 3501 standard itself, others are added by various recommendations and extension standards. An IMAP server might allow IMAP clients to store their own message flags. A special means of communicating this via the protocol exist[4].

Finally, message contents itself is distributed in form of *message texts.* Clients are free to choose from retrieving MIME message parts separately or as one blob.

2.3 MIME

Multipurpose Internet Mail Extensions [4], [5], [6], or just MIME, is a set of standards for extending the old SMTP mail from 7bit ASCII texts to rich multimedia contents. Parts of the standard deal with encapsulation of non-ASCII characters in message headers, as well as providing instructions about how to embed non-textual data in e-mail messages.

2.3.1 Message as a Container

An important aspect of MIME is its introduction of an internal structure to e-mail message bodies. Previously, a message body was just a plain text, while since the codification of this standard, it became possible to exchange more structured information. Contemporary users are probably familiar with HTML e-mails containing embedded images and a PDF attachment for seamless printing. Users routinely send e-mails with attached photographs from their holidays, or an archive containing business presentation.

MIME standard achieves this by defining a *content type* for message bodies. This content type, usually set by an RFC 2822 header of a message, determines how a compliant mail client displays the message. The old-school non-MIME mails have a default MIME type of `text/plain`, an unformatted text in 7bit US-ASCII encoding.

Another portion of the standard defines generic-purpose containers, i.e., abstract content types which contains other message parts within. Using these containers, a message can suddenly evolve from an unstructured blob of text to a deep tree of unrelated body parts.

[4]The `PERMANENTFLAGS` response code

Most of these containers are defined in RFC 2045 [4] and 2046 [5], one was added in RFC 2387 [7].

The `multipart/alternative` encoding provides a mean of transmitting a message that comes in several different formats, but each of them contains the same data. The MUA is expected to pick one of the underlying parts and display it, perhaps allowing the user to override this selection as a nice bonus.

The `multipart/mixed` is a generic catch-all MIME type for multipart messages. If a compliant MUA sees a multipart content type that it can not recognize, it should behave as if it was the mixed multipart. A typical action is showing all embedded parts next to each other.

There are several more content types defined, but describing them in detail is out of scope for this thesis.

2.3.2 International Characters in Messages

Original SMTP mail as defined by RFC 822 is suitable only for transferring 7bit data. Nowadays, 8bit transport channels are far more common, both for representation of international characters and for transfers of binary data. Therefore, the MIME family defines *transfer encodings* for conversion of generic 8bit binary data into a 7bit character stream and a portable way of expressing national characters using 7bit ASCII characters only.

For the former, two standard encodings are available, the *Quoted printable* and *Base64* (as well as a "fake" *binary* or *8bit* encoding for transport of binary data over less restrictive tunnels), the latter is supported by a similar approach, as defined in RFC 2047 [6].

2.3.3 MIME Support in IMAP

A large part of work involving the MIME support can be performed by the IMAP server. This is especially true for message structure parsing, where the IMAP specification provisions ways to explore the "tree" of a message, as well as methods for retrieving the resulting body parts separately (and even including byte ranges for more convenient download).

The IMAP protocol does not, however, handle the rest of MIME support, nor much of other header parsing. While it is common to ask the server for just a named subset of RFC 2822 mail headers, the returned data are not pre-parsed in any way. Therefore, much of the work from coalescing several header parts to decoding international character data is left as an exercise to the client.

Fortunately, various message attributes in the IMAP protocol can serve as a substitute for parsing the RFC 2822 headers. Sender and recipient data is available from the IMAP `ENVELOPE`, message size as `RFC822.SIZE`, IMAP itself provides extensions for message threading [8], etc.

2.4 IMAP Protocol Flow

Upon a successful connection to a remote IMAP server, client might choose to authenticate itself if the server has not already pre-authenticated it automatically under a particular user account. After that or in case the authentication is not required, the connection enters the *authenticated* state. In this phase, no mailbox is selected and only a subset of commands is valid. Clients can, for example, ask for listings of the mailbox tree, get quick information about number of messages in a particular mailbox or otherwise manage mailboxes as a whole. They have to, however, *select* a mailbox in order to do anything else, like retrieving messages or marking them as read.

A mailbox can be opened as read-only or for both reading and writing, provided the authenticated user has sufficient privileges. Retrieving messages can be done in both modes, but write operations (like storing a new message to the mailbox or manipulating the message flags) require read-write access.

2.4.1 Commands and responses

IMAP is a line-oriented text protocol. Anything that a server sends to its client is called a *response* while all data flowing in the opposite direction are known as *commands*. Each command generated by client begins with a unique identifier called *tag* which is followed by the actual command identifier. Most commands also accept (or require) a *parameter* which might be a string literal, a number, an atom, a parenthesized list of other types, etc. Transmission of generic string literals requires an explicit confirmation from the server unless the LITERAL+ [9] extension is active. Support for this extension is crucial for good performance, as if absent, most commands would be delayed by at least network round-trip-time and no support for pipelining of such commands would be possible. Clients are free to issue several commands without waiting for their completion, subject to certain ambiguity rules[5].

Each command usually leads to at least one server response, the *tagged reply*. This response contains the same tag as the client sent in the original command, allowing the client to find out whether command failed or succeeded. Some commands may result in sending further replies, the *untagged replies*. An important detail of the protocol is that such untagged replies are allowed to occur at almost any time during the conversation and *not* only as a result of a particular command. A compliant client must be prepared for this situation and handle it correctly, for there is no difference between those unilaterally generated and "requested" replies.

Each tagged reply indicates a completion of a particular command. If the command succeeded, an `OK` status is received, if it failed, server returns a `NO`. Should the server had troubles figuring out the client's intention, a `BAD` response is returned. More fine-grained control over results is achieved via *response codes*, textual entities with strictly defined meaning. Examples of such response codes are "mailbox opened in read-only mode", "specified character set no supported" or "please show this error message to the user".

2.4.2 Mailbox Synchronization

For efficiency reasons, each client usually keeps some data in its persistent cache. Good candidates for caching are message texts, a copy of body structure or perhaps even flags from the previous session. Because some server replies, namely the `EXPUNGE` which informs the client that a message has been removed from the mailbox, refer to messages only by their sequence numbers and clients usually identify messages by UIDs in order to allow message texts caching, a compliant client has also to keep its UID-to-sequence number mapping up-to-speed with the server. Doing so for just a visible (in the GUI) part of the mailbox might seem tempting, but the risks involved therein (like not knowing what to purge from local cache as a result of received EXPUNGE without extra `FETCH`) are high and usually outweigh possible initial savings.

An IMAP server fully conforming to RFC 3501 is required to sent a few "status" replies when a mailbox is selected. Interesting fields for the mailbox synchronization are updates on UIDVALIDITY, UIDNEXT and EXISTS. After receiving all three, client has enough information to decide

[5]See section 5.5 of RFC 3501

about what actions are necessary to bring the particular mailbox back to a fully synchronized state. Clients only have to deal with mutable message attributes, which are the sequence number, the UID number and message flags.

If the UIDVALIDITY value changed, the client is required to throw away any data it might have in its local cache that refers to this particular mailbox. Some UIDs might have changed, message parts got modified or the client just happens to talk to a buggy server implementation. This situation is then identical to syncing form an initial state where client has no knowledge about the target mailbox. In the following text, we will assume that the UIDVALIDITY value is the same as on the previous synchronization.

If the UIDNEXT value did not change either, no messages were delivered to the mailbox since the last time. If the EXISTS number is the same as well, there have been no changes at all and the only item to re-synchronize are message flags. If the EXISTS number changed, some messages were permanently removed. Clients usually re-fetch their UID-to-sequences mapping as a whole, but implementations which prefer to transfer the lowest amount of data over the number of iterations over the network could use an algorithm similar to binary search to find out which regions of the UID mapping were left intact and which require re-synchronization.

If the UIDNEXT changed[6] and the ΔUID is the same as $\Delta EXISTS$, some new messages arrived, but none were deleted and the client only has to ask for UIDs of these new messages (and FLAGS for all messages in the mailbox). If the increments of UIDNEXT and EXISTS differ, a fallback to generic synchronizing approach where client retrieves UIDs of all messages in the mailbox is necessary. It is also reasonable to transfer flags as a part of this step.

If there were no cached information or a UIDVALIDITY change occurred, clients would typically fetch a complete UID mapping and message flags.

After performing these steps and having received all relevant replies, the client is in a fully synchronized state and ready for receiving mailbox updates.

2.4.3 Changes to Mailbox

The mailbox that is selected is typically of some interest to the user. The original IMAP RFC mandated that all clients must be prepared for receiving

[6]It could only grow up, because UID numbers are strictly ascending

updates about number of messages contained therein, removal of old ones and deliver of new, but due to the lack of support by clients at that time, this feature was rarely fully utilized. Instead, the *NOOP* command was widely used as a polling check for new messages. However, this had all the disadvantages of a polling-based approach, especially unnecessary increase of network traffic, lower possibility to enter deep sleep for battery powered devices and even delays in mail delivery, for some clients were rather hesitant to poll for changes too often. A solution to this problem is presented in RFC 2177 [10] where client enters a special mode (which is syntactically similar to synchronizing literals that are used when support for LITERAL+ [9] is absent) to tell the server that it can really send real-time notifications to the client.

Changes to the number of messages in the mailbox are communicated mainly by two kinds of untagged replies, *EXISTS* and *EXPUNGE*. The EXPUNGE reply contains a sequence number of the message that has been permanently removed from the mailbox and immediately causes all messages with sequence number higher than the message being deleted to decrement their assigned sequence numbers by one. This means that clients have to work with sequence numbers, unless they are prepared to update their UID map on each expunge, which would be a rather expensive approach. The EXISTS reply is used to inform the client that new message was delivered, or it could follow an EXPUNGE to provide a redundant information to the client about number of messages in the mail store. Under no circumstances could it be used to decrease the number of messages in current mailbox, as this is already handled by EXPUNGE.

2.4.4 Fetching and Manipulating Messages

Once selected, messages in the mailbox can be queried and downloaded to the client. The *FETCH* command serves for retrieving all data from body structures and IMAP envelopes to actual message parts. Message flags are updated by the *STORE* command and its variants. Existing messages can be copied to other mailboxes with the *COPY* command, new messages can be stored into a mail folder by the *APPEND* command.

2.4.5 Queries Against Other Mailboxes

Because most graphical MUAs want to show the user more than one mailbox at a time, at least in a collapsed view with counts of new or unread messages, and frequent switching among several mailboxes is not a cheap operation, IMAP also provides a command for determining state of other than currently selected mailbox, the *STATUS* command. Using this command for querying state of a currently selected mailbox is explicitly forbidden by the RFC, for a conforming client already has all required information to compute all possible message counts. Another reason for disallowing STATUS on current mailbox is that in some server implementations, this operation might cause a process to open the mailbox once more which might in turn provide inconsistent results and confuse the client even more.

2.4.6 Searching, Sorting and Threading

Even the base specification of the IMAP protocol provides facilities for searching the messages stored in current mailbox. This was extended over time by ways to request specific message ordering or threading [8]. Most e-mail clients therefore do not need to have all message bodies and headers fetched and available in a local cache in order to allow their users to search through messages. However, this searching and sorting is not available in offline clients which have to implement their own code for such purposes, or disallow these operations when network access is disabled.

The following revisions also specified certain extensions related to the internationalization of e-mail traffic. For example, searching for non-ASCII characters in messages would be very hard using the bare-bone SEARCH command only, as each message in mailbox can be stored in one of the myriad encodings available in the MIME standard. Subsequent revisions to IMAP therefore defined support for more intelligent searches which support unified searching through messages that arrived in different character sets.

2.4.7 Manipulating Mailboxes

Due to some historic reasons (especially the old versions of SMTP protocol being 7bit), IMAP does not use any standard encoding for mailbox names. Instead, a modified version of UTF-7 called simply *modified UTF-7* is mandated. The standard does not specify what to do if the other side uses

8bit data in mailbox name, but most implementations nowadays fall back to UTF-8, which is arguably a good thing to do.

Mailboxes can be created, deleted and renamed by IMAP clients. If the server supports it, they form a tree hierarchy. Some servers distinguish between mailboxes for storing messages and those that contain only other mailboxes, while other implementations allow free mixing of both. Unfortunately there is no reliable way to tell to which class the target server belongs, but provisions are given about how to create a folder for message and how for other mailboxes. If mailboxes form a tree, the server implementation specifies which character is used as a hierarchy delimiter. Most common choices in today's mainstream IMAP servers are dot (`.`) which is common among servers exporting Maildir folders and slash (`/`) as used by servers exporting traditional mbox UNIX mail store. This inconsistence might lead to user confusion, but cannot be avoided without breaking backward compatibility.

2.4.8 Session termination

IMAP session ends when the server sends the client an untagged `BYE` response. This might be caused either by client's request via the `LOGOUT` command or by an auto-logout timer on the server side. Server implementations are required to allow at least 30 minutes of inactivity before sending this reply[7].

2.4.9 IMAP Extensions

The IMAP protocol also has support for extending its functionality. Extensions can add new commands, replies and reply codes and even change basic aspects of the protocol, as demonstrated by the introduction of non-synchronizing literals in RFC 2088 [9]. Support for these extensions is strictly voluntary, though, and each compliant IMAP protocol entity has to function properly even when talking to an implementation supporting bare IMAP4rev1 as specified by RFC 3051.

[7]Many real networks, however, enforce much stricter policies with shorter timers. Real world client implementations therefore send keep-alive messages once in a few minutes to avoid connection termination.

2.5 Other Methods of Mail Store Access

IMAP is not the only protocol for accessing mails stored on a remote server. In this section, we will introduce some of the alternatives and discuss their advantages and disadvantages when compared to IMAP.

2.5.1 POP3

The main difference between IMAP and POP3[8] protocol is their philosophy — IMAP was designed for scenarios where user leaves all her e-mail at a central place which has authoritative answer about what messages are supposed to "be there" while POP3 expects their users to download all e-mail to a computer where the POP3 client runs and then *permanently remove the original messages from the server.* Upon each reconnect, the POP3 client downloads and deletes all new messages again. This fundamental difference leads to troubles when the user wants to use POP3 in situations it was not designed for – a classic example is accessing one mailbox from two places over time, like a work computer and a laptop at home. While there are certain ways to work around these POP3 design limitations which allow leaving a copy of each message on the POP3 server, they all remain just hot fixes instead of providing a proper and reliable solution. An excellent comparison of these two paradigms is available from Grey's paper [11].

2.5.2 MAPI

A proprietary protocol from Microsoft whose documentation got recently opened for public access after a pressure from both government an non-government organisations. It has some nice features comparable to IMAP, but offers no real advantage besides tight integration with other Exchange services (which might or actually might not be considered an advantage). Implementations from other parties than the original vendor are scarce.

2.5.3 Webmail

Several years ago, a boom of online services from well-known companies like Google or regional Internet portals grew in popularity. Despite inherent

[8]Post Office Protocol version 3

limitations of web-based user interfaces, most "casual" users of e-mail, especially the second "home mailboxes" of regular employees, are today hosted and accessed via a webmail interface. Although hardly any of these services offers any guarantee about availability or performing backups, regular users typically are not concerned about these issues. Reliability of these services is, on the other hand, typically greater than that of a typical home PC with aging hardware, and friendliness of the UI improved with advent of recent web technologies.

2.5.4 IMAP Criticism

During more than 20 years of the IMAP history, the standard has certainly accumulated some aging dust. A typical example are the synchronizing literals, originally the only way to send more complex data to the server, which require explicit server's acknowledgment before continuation. Given a typical GPRS connection which is still common in many countries, a latency of 500 ms can delay all interaction with the IMAP server considerably. Many similar aspects were fixed by subsequent modifications to the standard, others are rather unfixable design limitations.

A fine example of such limitation, at least when perceived by the contemporary optics, is the concept of mailboxes. Online messaging services like GMail introduced a new way of thinking about messages – previously a message would belong to exactly one mailbox and if the user wanted to have it available from two places (perhaps in "work related" and "mailing list" folders), she would have to store two copies. IMAP has no concept of a "link" between two messages. When one of them is flagged, the other copy is left intact. Another prehistoric feature of IMAP is its modified UTF-7 encoding which is not used anywhere else besides the IMAP protocol. Another point – IMAP usually does a good job in shielding the client implementation from the need to parse all MIME data, but certain pieces remain, like the RFC 2822 headers with international characters.

Some authors also criticize the protocol on basis of its complexity and being loosely-defined in certain aspects. An example is the IDLE [10] extension, which adds an explicit mode indicating to the server that the client is really ready to process any updates to mailbox state, while the original RFC mandated the client to be in such condition all the time. Most common MUAs also have troubles with getting all side-corners of IMAP right.

That being said, the IMAP is the only widely implemented standard for accessing an online or disconnected mail store. It has certain deficiencies, but all in all, it is a perfectly usable protocol.

Chapter 3

Trojita Design

As evidenced by numerous available MUAs on the market, there are many possible ways on how to implement a usable e-mail client. In this chapter, we explain some design choices behind Trojita.

3.1 Overview

Trojita is composed of several logically decoupled layers of independent components, see figure 3.1 on page 24. Communication between them adheres to a strictly defined interface.

On the top of the hierarchy lies the *GUI*. This part is what the user interacts with; it provides views showing all mailboxes on a server, a list of messages in the currently selected mailbox and the contents of the e-mail messages themselves. The GUI is based on the Qt framework, making use of various standard components. A substantial part is based on the Interview framework [12] and the WebKit [13] as well as certain custom widgets, especially the Message View (section 3.7).

The GUI is fed with data by the *Model*, a model-view implementation created with the Qt item views. This part works in close collaboration with the IMAP server. It is responsible for processing server replies, building a representation of data stored on the remote host and sending commands to the server in response to user actions. Most of the IMAP protocol logic is concentrated on this layer.

The Model is assisted by a *Cache*. The cache provides persistent storage for certain data which are convenient to be present on next invocation of the application. Examples of such data include message parts or mailbox state

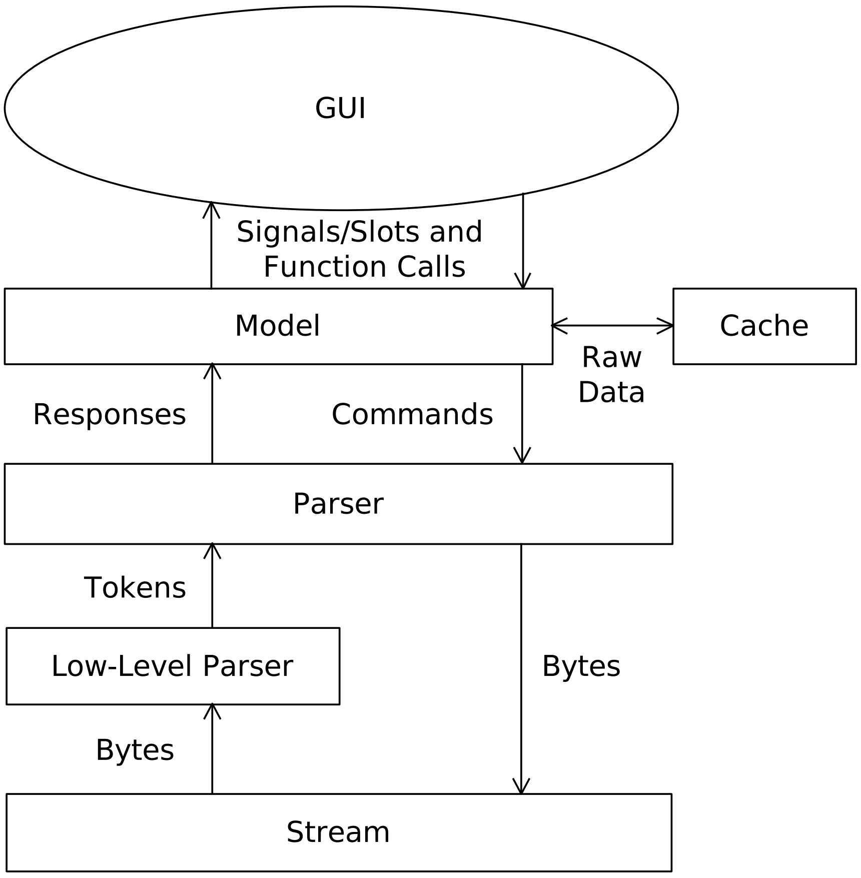

Figure 3.1: A diagram showing the Trojita architecture

information which allows a quick resynchronization after a reconnect. While the role of the cache and the data stored therein is not critical for Trojita's operation, having a local cache can considerably speed up the runtime. A layer serving a similar purpose is common among other e-mail clients as well.

Beneath the Model is the IMAP *Parser*, a component converting the stream of data flowing from the server into structured (and as the name suggests, parsed) data. The parser is assisted by a *Low-level parser*, an entity responsible for analysing the byte stream into tokens of data such as strings or numbers.

Finally, all network I/O is managed by a *Stream* and associated classes which provide a consistent interface for managing connections to the remote server.

3.2 Model-View Architecture

Trojita's heart, the Model, implements an interface of the QAbstractItem-Model, a variant of the well-known Model-View-Controller design pattern. This architecture provides natural separation between how data are stored and how they are displayed. In case of Trojita, we went a bit ahead compared to the common approach and made essentially everything just a node, a part in one big tree which exports all data available on the server. This tree then contains all mailboxes, all messages stored in them and even the message structure and actual message body parts as its nodes. Several *filtering models* provide more user-oriented view, like "all mailboxes" or "messages in a mailbox". An alternative to this one-big-tree approach were separate models for list of mailboxes and for each opened mailbox. While the latter might make more sense at a first glance and from the perspective of a reader who just went through the IMAP protocol standard, subsequent analysis suggests that one big model actually fits the situation better, unless we wanted to deal with nasty issues that would immediately pop up as a result of sharing one Parser instance among several Models facing several mailboxes. The upper layers are not affected by this decision, as they use a strictly defined model-view interface for actual data retrieval.

The model-view architecture subsequently affects how Trojita and her Model operate. Upon successful startup, for example, Trojita will not execute many IMAP commands besides making sure that a connection to the remote IMAP server enters the authenticated state. This might involve issuing a `STARTTLS` command for establishing a secure channel, optional user authentication, etc., but none of tasks like mailbox listing or even opening some of them. This listing is only fetched as a result of a call from the GUI which asks the model what should be shown in the view representing a list of mailboxes.

After the list of mailboxes is received and processed, the model tells all the attached views that its layout has changed and that the views should update what they show to the user. The views react by requesting more information from the Model, e.g., a number of unread messages or a total message count for each visible mailbox.

When a user clicks on a particular mailbox, another view is pointed to a subset of the big model, now containing a list of messages in the mailbox. Again, the action of clicking to the mailbox does not involve any direct action in the model itself; the mailbox synchronization is initiated only after the new view asks the model about how many messages to show, and then for more data concerning those messages that are visible in the current viewport.

One important feature implemented in Trojita which is absent from most e-mail clients (especially those whose data model was designed with only POP3 in mind) is delayed on-demand fetching with an intelligent preloading. Suppose a user has a big mailbox with more than 30,000 e-mail messages. A naive client would open a connection to the server, select the mailbox and then fetch envelopes for all messages in the mailbox. Most of these messages would never be shown to the user, though, as thirty thousands messages is too much to be displayed on one screen. What Trojita does is, however, more clever – it creates thirty thousand of *fake* entries in the Model, each displaying a *loading...* sign, and ask for the real data only for those that are actually visible (with some read-ahead and read-behind for increased efficiency). This effectively minimizes the amount of network traffic as well as the local cache size and the load on the remote IMAP server, while the user barely notices any slowdown.

Being the central part of Trojita, the Model and related components is described in section 3.6 at the end of this chapter.

3.3 Parser

The *Parser* is responsible for converting raw network replies into data suitable for processing by the Model.

3.3.1 Low-level Parser

Certain low-level tasks like extraction of a string literal from the input stream are offloaded to a lower-level class, the LowLevelParser. Reading raw, unstructured byte stream as the input, its job is to extract a certain field and return it in the specified form. The resulting type of this tokenizing is not, however, determined by the LowLevelParser itself, but by the upper layers. LowLevelParser thus works as a dumb context-free extractor rather than a full parser. Should an error occur (for example when the higher-level code requests an integer, but the data at the current offset are textual), an exception is thrown. Examples of functions implemented in the LowLevelParser are `getAtom()`[1], `parseList()`[2] or `getNString()`[3]. Details about their implementation are available in the Doxygen documentation.

3.3.2 Parser

The Parser listens to events from the underlying Socket, sending queued commands to the server and processing incoming data as they arrive. During the development of Trojita (and thanks to being the first component to be actually implemented), design of the Parser changed several times. Originally it started as a threaded implementation due to certain concerns about interactivity and network I/O, but after encountering some issues with Qt's handling of asynchronous event signalling and a recommendation from one of the core Qt developers, it was refactored into a single-threaded state machine with a more conservative handling of the network I/O, which closely mimics common Qt idioms.

When a reply from the server arrives, Parser instructs the auxiliary LowLevelParser instance to read tokens from the byte stream. These tokens are then used to build a complex `AbstractResponse` instance on-the-fly. Due to the limitations of polymorphism in C++ and concerns about object lifetime and ownership, the Parser makes heavy use of C++ smart pointers, as introduced in the TR1 [14]. An alternative would be to use smart pointer implementation from the Qt framework, the QSharedPointer. It provides a similar interface, so the possibility of eliminating a dependency on TR1 implementation is open for further consideration.

This part of Trojita has an extensive unit test coverage based both on real-world traffic and artificial corner cases. Error handling is implemented via exceptions. Being a class derived from QObject, the Parser uses signals and slots for communication with the rest of the application – incoming data are read and parsed in code connected to the `readyRead()` signal from the underlying socket and availability of processed server replies is announced by emission of `responseReceived()` signal. The responses themselves are

[1]For reading an *atom*, which is usually used for transmission of integer values or ASCII words not containing "complicated" characters like spaces

[2]For retrieving a parenthesized list of other items. The other items might be atoms, string literals or even nested lists.

[3]`getNString()` returns string as parsed by `getAString()` and adds special handling for `NIL` items.

not serialized to the Qt's event delivery system, though, but remain stored in an internal queue in the Parser which provides queue-like semantics for accessing them.

3.4 Cache

A main purpose of the Cache is providing a persistent copy of certain useful data for speeding up mailbox re-synchronization upon a subsequent select as well as storing already requested message structure and message body parts.

Trojita is ready for multiple cache implementations. The implementation and interface are properly separated – an abstract class[4] specifies the only required methods of interaction between the Model and the Cache.

A simple cache implementation storing all data in operational memory with an optional possibility of persistent data store on the disk is provided. As no object life management or partial updates of the on-disk representation are implemented in this simple cache, it is not recommended for production use. Advanced caching implementations are expected to be developed after the thesis is finished. Candidates include using an sqlite database or an Qt caching object from WebKit, the `QNetworkDiskCache`. One reason for developing a simple, in-house caching solution is the fact that the Qt caching framework was introduced only in Qt 4.5 while Trojita still supports building against the 4.4 release, albeit with reduced feature set. Therefore, it was deemed that a simple custom caching framework is worth the extra work.

3.5 Streams

The Qt framework already provides an implementation of an entity with support for sequential reading and writing as well as informing the rest of the application about the availability of new data. However, no unified interface for reporting errors is provided. Therefore, most of the classes in the Stream sub-library are just thin wrappers around various `QIODevice` subclasses, adding unified signals and functions for error handling.

Trojita includes support for communication over unencrypted TCP sock-

[4]`Imap::Mailbox::AbstractCache` as defined in `Imap/Model/Cache.h`

ets, over a connection secured by SSL[5]/TLS[6] with the possibility to defer the SSL/TLS negotiation to a later point during the IMAP conversation[7], and finally over a UNIX pipe to a local process. For the latter, two implementations are actually included, first one being a wrapper around the QProcess from Qt and the other a standalone class making direct use of POSIX syscalls. The standalone implementation is no longer necessary, but was used in an earlier implementation of the Parser which made heavy use of nested calls to QIODevice's `waitForReadyRead()` method. This usage led to hard-to-debug failures in the networking code when using other QIODevices than QTcpSocket. After a communication with Qt Software support personnel [15], it was suggested to abandon these nested waitForReadyRead() calls and follow an approach common to Qt applications which simply ignore the availability of new data until there is enough bytes to process a whole response at once. In addition to more obvious and cleaner code, this eliminated the need for blocking I/O and subsequently any advantages of a threaded design of the Parser. Therefore, multithreading was abandoned and a new single-threaded network handler was implemented.

3.6 Model in Detail

The *Model* is the most complex part of Trojita. It is responsible for correctly figuring out the meaning of various IMAP server replies, requesting further data as they are to be shown to the user, managing a pool of connections to opened mailboxes as well as handling of certain non-standard situations.

3.6.1 The Tree

Being an implementation of QAbstractItemModel, the Model has to provide an interface through which the attached views and proxy models[8] can operate.

As most of other complex implementations of the QAbstractItemModel which provide a tree view with deep hierarchy, the Model maintains an

[5]Secure Socket Layers

[6]Transport Layer Security

[7]The STARTTLS command, cf. section 6.2.1 of RFC 3501 [1]

[8]Proxy models are auxiliary models which do not supply their own data, but instead provide a filtered and modified view of data published by the parent model. More on this topic is explained in section 3.6.2.

internal tree of classes representing different entities in the big tree. These entities are:

TreeItem An abstract parent class providing a common interface for all specialized implementations

TreeItemMailbox A class representing a remote mailbox on the IMAP server. Children of the mailbox are other mailboxes and a *list of messages.*

TreeItemMsgList A container holding all messages in the mailbox

TreeItemMessage One message in a particular mailbox

TreeItemPart One MIME body part of a message

Each of the mentioned classes contains enough information to be able to handle all delegated operations related to the Interview framework. For example, when the Model is asked for a displayable textual data for a particular index, it delegates the query to the corresponding TreeItem instance using virtual methods. The target instance then decides about *what is actually requested* and *how to retrieve that data.* Certain requests might be satisfied from the data already stored in the TreeItem instance itself while other require a query against the IMAP server. If the latter happens, the TreeItem asks the Model to submit a request to the IMAP server and returns some *temporary data* in response to the original request which most likely originated from the GUI. A typical temporary representation is a translated form of *"loading..."* or a question mark. As soon as the real data arrive from the server, the view is asked to update the display with new value.

To illustrate this rather complex theory on a concrete example, suppose a view just scrolled to a new message in the mailbox. The view obtained a *model index*[9] from the Model and called the Model's `data()` function to find out what should be displayed on screen. This function call was immediately propagated to the newly created TreeItemMessage instance. As the message just arrived to the mailbox, the class didn't have time to request more data about it yet. The TreeItemMessage instance is therefore in an *initial state* indicating that no attempt to load the data from the

[9]A temporary index used for referring to certain item in the Model. The model index is a standard term from the Qt Interview framework, and as such is not described in detail in this thesis. Readers are referred to excellent Qt online documentation for details.

server has been performed, and thus the `data()` implementation does not have any real answer to the original query. Therefore, it submits a request to the Model asking it to fetch the IMAP envelope of the particular message. However, the reply to this message is not guaranteed to arrive in a timely manner and the TreeItemMessage implementation cannot afford to block the GUI while it waits for the corresponding reply. Therefore, it returns an empty placeholder (or some intuitive string like the *loading...* text already mentioned) for the time being. This text is then displayed in the GUI, for the Model does not have anything better to show to the user yet. In addition to that, the TreeItemMessage changes its state to *loading* which indicates that the remote server was already asked for the data and there is no point in requesting it over and over again[10].

In the meanwhile, the Model receives TreeItemMessage's request for fetching an envelope. It decides that the requested data belong to, e.g., message No. 666 in the mailbox. The total message count is 670 and no messages between 660 and 670 were fetched yet. To enhance the user experience and minimize network round-trips, the Model submits a batched request to fetch message envelopes of all these messages at once, as chances are high that most of them will be queried for data soon anyway. As a part of this request, they are all flagged as *loading* in order to prevent duplicate fetching requests.

After a while, the response from the server arrives. As usual, the Model delegates the actual processing of the reply to the TreeItemMailbox or TreeItemMessage instance. The class notices that enough data for updating its state from *loading* to *fetched* has arrived, so it marks itself as *fetched* and tells the Model to broadcast this change to the attached consumers (views). Each view, upon receiving a `dataChanged()` signal from the Model, asks for the displayable data again. Model passes the `data()` function call to the TreeItemMessage instance which discovers that it has the data already available in memory and returns the real text field (like a message Subject) this time.

The ultimate goal of this lazy population is reducing network traffic and server load, as well as the client's memory usage.

The delayed loading is employed in all places in the Model where possible. For the efficiency reasons, though, certain data items which are known to be needed shortly after each other are requested in one block. Examples of

[10]If the `data()` function concerning the same index is called again, the same placeholder data is returned, but no further requests to the IMAP server are initiated.

such data are groups of the message envelope, the IMAP internal date, the message size, the UID, the body structure and the message flags concerning a particular message. As suggested earlier, these fields are also preloaded for previous and following messages in a sequence.

One notable exception to this approach is building a list of mailboxes. Due to the way the `LIST` response is defined[11], no reliable way of requesting a partial listing of the mailbox tree is provided besides indicating that the result should include mailboxes from the specified hierarchy level only. Trojita therefore has to fetch all the names of top-level mailboxes at once. Requesting a listing of their children is then delayed until the user clicks the "expand" widget next to the mailbox name. Also the numbers indicating a total message count or the number of new messages in a mailbox are not requested for mailboxes that were not shown yet.

The tree nodes do not make use of smart pointers, instead, the parent node is responsible for properly freeing all resources associated with it, including destroying the children.

Trojita already tries to minimize the memory footprint, not fetching data until needed, yet an empty TreeItemMessage instance is still created for each message in the opened mailbox. The memory consumption associated with these operations is difficult to measure[12]. Experiments show that opening a mailbox with more than 43,000 messages increases the overall memory usage by about 6.5 MB on an x86 machine. Slightly less than one third of this memory is consumed by the QTreeView itself.

3.6.2 Proxy Models

While browsing the huge mailbox tree itself is certainly an educating experience for users wishing to become familiar with e-mail processing and the IMAP standard, it is far from being user-friendly. Therefore, Trojita provides *proxy models* for presenting a more conservative view to the remote server's resources.

A proxy model works in close collaboration with the parent model, providing a modified and filtered view of the parent's data to the attached views. Trojita uses two proxy models, one for providing a list of mailboxes and the other for listing messages in the opened mailbox. Neither of these

[11]Consult the RFC 3501 [1], section 7.2.2

[12]The increase in memory usage includes internal QTreeView data structures which could not be eliminated even if Trojita did not create any new objects for its own purposes

models contain significant intelligence, as most of the hard work is already done by the original parent Model.

3.6.3 Offline and Expensive Network Mode

In addition to the mode of operation described above, the Model can be instructed to change its behavior in order to either reduce the network traffic even further or to avoid any communication with the remote IMAP server altogether. For the former, and *expensive network* mode is provided, the latter is handled by an *offline mode*.

When offline, essentially any network traffic[13] is blocked. All requests which cannot be satisfied from the cache will fail, and the corresponding TreeItem's state will be changed to *offline*. This particular status is reflected in the GUI in an intuitive way, mostly by changing the unknown message subjects to *offline*, etc.

The expensive mode mostly eliminates various optimistic preloading, as well as changes the operation of the message view to not show all body parts automatically.

3.7 Message View

Rendering an e-mail message is not as simple as it seems. First problem that the implementation has to face is the delayed on-demand loading of the message parts. Due to the rich tree structure of MIME messages, Trojita does not know which parts are required to be shown before an attempt to render the message is made, and loading all of them in advance would be not only suboptimal in terms of network utilization, but could cause an inconvenient delay when displaying huge messages as well. Similarly, blocking for a full delivery of all required message parts when the rendering starts is not an option, as this process should not block the GUI.

The second problem originates in the complex nested structure of MIME messages. There is no limit for the maximal allowed depth of the body part nesting, so a message could have an attachment which in turn is another message containing further complex structure, easily stretching for many levels.

[13]Besides shutting down the already established connection

A first version implemented in Trojita dealt with the first problem, while leaving the second one unsolved. Since that, the message view widget was rewritten and the version presented in this thesis supports unlimited nesting of messages.

The core of both message rendering widgets is based on the WebKit [13], a standards compliant HTML rendering engine.

3.7.1 Network Manager

The QWebView class is a component providing the HTML rendering engine as well as certain auxiliary infrastructure such as a client implementation of the HTTP stack. These parts combined together contain everything required for building an application which displays HTML pages retrieved from the network via a standard protocol like HTTP. Messages stored on a remote IMAP server are not, however, usually available over HTTP. Therefore, the QWebView had to be extended in order to be usable in Trojita.

Luckily the QWebView provides a straightforward way to delegate the network processing to a custom component via the QNetworkAccessManager interface. That custom component is then responsible for handling all network requests that originate as part of the processing of the HTML page.

In Trojita, we created a special network manager[14] whose job is to provide an access to the message bodies stored on the remote IMAP server as well as filter out any access to the “regular Internet”.

The reason for the former is security and spam filtering. For example, when an HTML message contains a reference to a remote image, a naive e-mail client which allows the retrieval of external entities via HTTP would provide an accurate indication about when the message was read and even confirm the very existence of the addressee’s e-mail address to the attacker.

The most interesting part of the network manager is, however, providing an access to the body parts. When the URL to be retrieved specifies the `trojita-imap` scheme and `message` as the “host name”, the *path* of the URL is directly used as a logical address of the requested body part. If such a part exists, an auxiliary helper structure, a subclass of QNetworkReply, is returned which immediately requests the retrieval of the corresponding part from the IMAP server (or the local cache, if available therein).

In an old implementation, the QNetworkReply itself was responsible for properly handling the MIME content-type of the body part. For example,

[14]The `MsgPartNetworkAccessmanager`

it contained a code for selecting the preferred part among several available in the multipart/alternative, or including multiple commands in case of multipart/mixed. This approach, however, did not scale. Therefore, the new implementation moves this logic to the upper layer where it arguably belongs.

The QNetworkReply subclasses in the new implementation do not handle any formatting logic. They are, as their name indicates, just thin wrappers dealing with raw data I/O from the TreeItemPart instance. For handling the formatting, a new class, the `PartWidgetfactory`, factory was created.

3.7.2 Widget Factory

The `PartWidgetfactory` is a classic example of the well-known *Factory* design pattern [16]. Its only public method, `create()`, is responsible for returning a QWidget which is supposed to display a representation of the actual body part.

For some "trivial"[15] content-types, the factory returns an instance of a slightly modified[16] QWebView.

If the type is complex, a wrapping widget is created. A classic example of the wrapper is the handler of `multipart/alternative` MIME content-type. In this case, a QTabWidget whose tabs contain the actual nested parts is returned. Widgets for the nested parts are obtained by a recursive call to the `create()` function, solving the problem of the nested body parts in an elegant way.

RFC 2183 [17] introduced a method for specifying whether a body part is to be shown *inline*, that is, as any other regular part, or as an *attachment* through the Content-Disposition header. If the part is an attachment, most MUAs would not render its contents, but instead show a control for downloading the body part to the disk. Trojita works in a similar way. Unfortunately, certain e-mail clients don't distinguish among an attachment and an inline part, labeling data such as attached ZIP files as *inline*. Therefore, Trojita process the Content-Disposition header only as a suggestion if it asks for an inline display.

[15] *Trivial* in this context refers to a message part which could not be decomposed to more elementary parts. Examples of *trivial* content-types are `text/plain` or `image/jpeg` while examples of *non-trivial* ones are `multipart/mixed` or `message/rfc822`.

[16] The modifications include filtering of the mouse wheel events, because the widget is supposed to be as large as required and leave implementing a scrolling view to itself to an upper layer.

This part of Trojita is ready for further extensions. One particular extension which would be extremely useful to be implemented is support for signed and encrypted messages. Signatures are currently only recognized and indicated in the GUI, but not verified at all.

3.7.3 HTML Mail

A common format for today's commercial e-mail messages is the *HTML mail.* Building on top of the RFC 2387 [7] and RFC 2111 [18], the HTML e-mail allows grouping HTML text and related data such as images into one MIME message part.

Conforming to the above mentioned standards, Trojita supports the `cid` URL scheme. The network access manager (see section 3.7.1) is extended with support for those URLs – if a message part with the corresponding Content-ID header is found among the current message bodies, it is returned in response to a `cid:` URL.

An interesting fact about Trojita is that her support for the `cid:` URLs is more feature-complete than that of Thunderbird/2.0.0.14 which has troubles identifying certain body parts in some HTML messages[17].

3.8 GUI

No matter how advanced the underlying engine is, if the GUI does not keep up with the features offered or is not intuitive enough, users will switch to a competing product.

Trojita makes use of standard widgets from the Qt library whenever possible. Common and well-known components such as tree views builds the core of the GUI, showing list of messages and list of mailboxes. When the standard widgets did not exist or were not usable, new were implemented, re-using the existing ones as much as possible.

An example of a well-done re-use is the Message View. Building on already complex parts such as the WebKit and the associated helper classes, it combines the widgets with whom the user is probably already familiar into a big functional part. Without an unneeded user confusion, it manages to render the MIME structure of the received message faithfully.

[17]For example those from the `http://mon.itor.us/` monitoring service.

The mail composer widget closely mimics the *Write New Mail* window from Mozilla Thunderbird. The recipients matrix was implemented in a similar way, providing users with a familiar experience and reducing the need of getting accustomed to yet another software.

Similarly, the icons indicating message state (such as being flagged for deletion, already replied to, forwarded to another recipient, etc.) are indicated by standard icons. Trojita will, thanks to the QtIconLoader [19], try to load standard icons as per the relevant FreeDesktop.org standard [20], and for those that are absent from the system, a built-in icon from Thunderbird is used.

Drag and drop is used for copying and moving messages between mailboxes. No support for drag and drop outside of the Trojita is available yet, mainly due to a lack of widely accepted standard MIME type for doing so.

Chapter 4

Other IMAP Implementations

Albeit the guarantee of interoperability is arguably the most important aspect of a standardization process, certain e-mail clients perform better than the others. In this section, we try to provide a basic overview of selected e-mail implementations.

This overview is by no means complete, for there are many more e-mail client implementations than pages in this thesis. We therefore focus on highlighting issues observed during the course of interoperability testing and on a comparison with Trojita.

Each client was tested against an instance of the Dovecot IMAP server [21]. INBOX contained roughly five thousands messages, most of them were trivial text/plain e-mails with just one line of the message body. Other mail folders were however filed with certain testing messages representing a typical e-mail usage of the author of this thesis. Samples include GnuPG signed mail (the multipart/signed MIME type) and a special "MIME Torture Test" [22] message whose purpose is to test deep MIME part nesting as well some common decoder errors.

4.1 Alpine 2.0

Alpine [23], a successor of `pine`, is a console e-mail client by Mark Crispin, the principal author of the IMAP protocol. As such, its qualities are widely acknowledged. When coupled with the UW-imapd, an IMAP server implementation from the University of Washington from the same author, they offer some non-standard features such as multi-mailbox search. The IMAP part of the code base is available as the `c-client`, an implementation of the

IMAP protocol in the C programming language.

The testing process showed that Alpine performs as an *online client*, that is, a client with no local persistent cache for retrieved message parts. These parts are cached for the session life time, but any downloaded data will be fetched again on a subsequent Alpine invocation. A nice feature of Alpine is, on the other hand, that it does not fetch any data about messages not currently visible on screen at all, achieving the lowest amount of initial network traffic among all tested IMAP implementations.

As a surprise comes the marketed lack of support for the `IDLE` IMAP command. Crispin argues [24] that in today's world of NAT[1], a real client will have its connection to the server sewered by a network device much sooner than before the 30 minutes timeout period guaranteed by the RFC, and suggests to use `NOOP`-based polling instead. This opinion is arguable, and most e-mail clients' authors do not seem to share it.

4.2 Mutt 1.5.19

The second console e-mail client, the Mutt [25], is a popular text-based application. Controlling its user interface is slightly less intuitive than that of Alpine, but the advanced users praise its speed and the fact that the GUI does not block their operations.

Similar to Alpine, the Mutt is an *online client* without any persistent IMAP cache for downloaded message parts. In addition, Mutt will not issue the `IDLE` IMAP command unless specifically told so in the configuration file. As a reason for this are stated issues with certain server implementations[2]. Even when IDLE is enabled, Mutt only enters this mode when showing a list of messages in the mailbox and not when viewing one message.

4.3 KMail 3.5

Being the default e-mail client in KDE 3.5, the KMail [26] has a broad user community. However, its IMAP support might sometimes seem a bit rusty –

[1]Network Address Translation

[2]Interestingly enough, Dovecot is blamed, while new version of Dovecot contain a workaround claimed to "fix Mutt issues", so figuring out which party to blame is left as an exercise to the reader.

there is, for one, no support for IDLE[3], and quite a number of bugs against the IMAP implementation is reported at its Bugzilla.

In fact, the KMail contains two distinct implementations of IMAP, the first one being an *online mode* and the second one a *disconnected mode*. In the disconnected mode, all message bodies selected for being available when offline are transfered and all MIME parsing is deferred to the client, no matter what structure the e-mail message has.

In the online mode, a first surprise is KMail's handling of the IMAP message flags. In spite of using the proper command forms for *updating* certain individual flags instead of blindly specifying the resulting state of *all* flags for a particular message, the operations are apparently aimed too wide, targeting most of the messages in the mailbox even when some of them were not shown to the user yet. In addition, KMail needlessly introduces its own flag names even for those flags whose names are either standardized or their use widely supported. An example is setting both `KMAILFORWARDED` and `$FORWARDED` flags instead of just the former, semi-standardized [27] one.

The body parts of the complex MIME message from the MIME Torture Test are retrieved separately and the rendered representation of the message corresponds to the original body structure, which is an impressive result that no other e-mail client in this test (besides Trojita) achieved. The `text/plain` messages are, however, fetched as one compact entity along with their headers.

Common to both implementations, the mailbox hierarchy discovery is performed as recommended by the IMAP best-practices [28], i.e., using the `''%''` wild character instead of `''*''`.

KMail 1.11.1 with KDE 4.2.1 performed in a similar way to the already discussed version.

4.4 Mozilla Thunderbird 2.0

Mozilla Thunderbird [29] has been the e-mail client of choice for the author of this thesis. The GUI is excellent, not only well-looking, but also extremely convenient to use. However, its support for detecting changes to the mailbox performed by another session leaves much to be desired – sometimes it is

[3]The authors state that this limitation is present due to the architecture of the KDE's KIO framework which does not really allow similar modes of operation but is designed for client-initiated actions only.

prone to not updating the displayed state based on what is received from the server.

Thunderbird by default shows only *subscribed mailboxes*, i.e., those that were flagged by the user as interesting in some way. Their discovery is performed using the `''*''` wild mask, which might potentially result in a large list to fetch. On the other hand, assuming that the user did not mark too many mailboxes is probably acceptable. The MIME parsing is deferred to the e-mail client, as a common choice among the graphical Mail User Agents. We assume that this is a direct result of support for other protocols than IMAP which do not provide any form of server-side MIME support, and the client therefore has no reason not to decompose the message itself after receiving the full body in one part.

Thunderbird by default opens several connections to the IMAP server and uses IDLE for asynchronous notifications about new messages in each of them. The limit on number of concurrent connections is configurable [30], which is probably the only reasonable approach to figuring out how many of them will not yet be considered an abuse by the server operators[4].

4.5 Mailody

Mailody [31] originally started as a standalone IMAP e-mail client, an alternative to KMail. During the development of KDE4, its code base became the foundation of the IMAP support in Akonady, a framework for PIM[5] information exchange in the then-upcoming KDE4 desktop.

We were unable to test the version identified as "1.5.0-beta1" on KDE 4.2.1 as it crashed with a segmentation fault each time we tried to make it talk to our testing Dovecot instance. A similar result was observed when trying the older 0.5.0 version on KDE 3.5.10.

From the IMAP traffic dumps obtained before the application crashed, it is evident that while the authors of Mailody certainly thought about how to implement IMAP properly (as documented, e.g., by an attempt to group

[4]Client developers might argue that leaving tens of concurrent connections to the server open at the same time, each of them in the IDLE mode in a particular mailbox, is actually *better* for the server than multiplexing the access using just one or a few of IMAP threads. The client developers are of course right, but the server administrators usually cannot be expected to have such a deep understanding of the IMAP protocol. Indeed, most of them would prefer to ban such a user.

[5]Personal Information Management

a large message set to smaller parts), it is not a finished product yet. As an example, IMAP allows packing of a contiguous sequence into an interval specified by both boundaries. Mailody fails to use this feature, requesting the range by enumerating all values contained therein. It also does not seem to notice the IMAP envelope at all, fetching the corresponding headers directly. While the latter it not an error, strictly speaking, it at least makes the reader of the source code wonder what led to that particular choice.

Earlier versions of Mailody had troubles even with mailbox names being passed as literals instead of quoted strings. This might be caused by the fact that the source code used rigid regular expressions for parsing of server replies which did not account for each possible response variation. For example, such a naive parser would have troubles recognizing a reply containing a response code[6], which is free to be present in most situations.

4.6 Outlook Express

The Outlook Express is the built-in e-mail client in various versions of Microsoft Windows. The version we tested was shipped with Windows XP, and is the latest release of the Outlook Express ever[7].

The IMAP support in the Outlook Express is suboptimal, to say the least. An example of the "efficiency" of the protocol implementation is how the Outlook Express behaves when the user has selected several folders to be subscribed. At first, the application opens multiple connections to the server. Each of these connections then changes the subscription status of *one* mailbox, enters IDLE for a short time and finally terminates.

When the application opened a mailbox with roughly five thousand test messages, it began an immediate full download of all of them. Each messages was fetched by a dedicated command, adding one useless parameter to the list of items to fetch, and the IDLE mode was entered and immediately left between each two commands. That accounts for circa five thousand attempts to enter the IDLE mode, each of them lasting several milliseconds.

The GUI part of the application leaves much to be desired, too. A notable example is its handling of `multipart/signed` message parts – if the signature is made according to MIME/OpenPGP [32], Outlook Express

[6]Consult Section 2.4.1 for more on the concept of response codes

[7]The product got re-branded to *Windows Mail* and subsequently to *Windows Live Mail*

displays an empty message with two attachments, the signed body and the signature itself. This effectively prevents most users from actually reading the PGP-signed e-mails, as they do not expect the "real message body" to be available as an attachment only. Another problem exists with the display of the X-509 signed e-mails which contain a valid signature that cannot be verified. If such a signature is found, the Outlook Express displays a misleading warning about a possible security threat to the *local computer* instead of calmly informing the user about the fact that the certificate of the CA[8] is not found and thus the signature could not be verified. An extra click or two is needed to display the actual message body. This again presents a big obstacle to the actual usage of the signed e-mail on the Internet, for the Outlook Express still has a substantial market share.

4.7 Windows Live Mail

After the deprecation of the Outlook Express during the development phase of Windows Vista, a new e-mail application called *Windows Mail* has been introduced. However, it was not being shipped for a long time, as yet another successor, the *Windows Live Mail*, got introduced. Windows Live Mail is expected to be the e-mail application distributed with the upcoming Windows Seven operating system.

Unfortunately, most of the issues mentioned in the section about Outlook Express still stand. Both applications apparently use not only the same (or remarkably similar) IMAP code, but also a not-so-different GUI layer – while the overall appearance has certainly changed since Outlook Express, the issues with presentation of, e.g., the PGP/MIME signed e-mail messages persist. We will therefore end this section by an observation that the IMAP experience is almost the same as with the Outlook Express.

4.8 Trojita

Trojita is an IMAP client with support for various mail access modes. It can be told not to request data which were not loaded from the network yet, supports an efficient re-synchronization mode which tries to minimize the network traffic and tries to deal with even the most advanced MIME messages reasonably. The IMAP conversation could still be improved, though

[8]Certificate Authority, an entity which issues certificates to other entities

– for example, an initial LIST is issued even before the user is properly authenticated.

In general, we believe that we have implemented the IMAP protocol completely and properly. We tried to follow the IMAP RFC to the letter, but were sometimes forced to relax our requirements a bit, for example when talking to the Cyrus IMAP server which regularly sends replies which do not contain a textual part at all, in spite of the protocol grammar mandating its presence. We try to handle most situations gracefully, e.g., by resorting to a full mailbox re-synchronization when Trojita detects that the mailbox state changed in other ways than the server already acknowledged.

When compared to Alpine and Mutt, Trojita supports an offline message cache which allows its operation even when the connection to the remote IMAP server cannot be obtained. Its support for IDLE is also arguably better than that of Mutt, entering the IDLE more often.

Comparing with the messaging platform from Microsoft would not be fair. It is commonly accepted that the Outlook Express managed to implement IMAP in such a broken way that any further comment would be redundant.

The KMail offers a nice GUI and integration with the rest of the KDE desktop. Its IMAP support is, however, not that polished up. If Trojita had the GUI and related features of KMail, it would be hard to beat in terms of usability.

Finally, Mozilla Thunderbird is quite an impressive application. Even though we might have certain reservations to the overall Mozilla architecture, the GUI is user friendly and clean, and the application just works most of the time. In certain corner-cases, such as when working over a low-bandwidth network connection, Trojita has potential for outperforming the Thunderbird thanks to the employment of server-side MIME parsing. In fact, Trojita has already bitten Thunderbird when dealing with one particular HTML-formatted message which Mozilla fails to render properly, while Trojita handles it without any problem.

A conclusion of Trojita's overall usability is provided in the end of the thesis.

Chapter 5

User's Guide

5.1 About Trojita

Trojita is a standards-compliant IMAP Mail User Agent. Designed with portability and efficiency in mind, it allows the user to operate over congested network lines with high latency, while still providing sufficient comfort.

5.2 Installation

5.2.1 Prerequisites

In order to build Trojita from source, several libraries and helper programs are required to be present on the system.

1. CMake 2.6 or later
2. Qt 4.4 or later
3. Working implementation of the TR1 C++ standard such as GCC 4.3 or newer or the Boost libraries

A recommended version of the Qt framework [33] is at least 4.5. Trojita, however, includes compatibility code for older versions of Qt. The minimal supported version is Qt/4.4. Any older release cannot be supported, as the WebKit HTML engine which we use for e-mail rendering was added during the 4.4 development. If users choose to build against 4.4, some Trojita features might be missing.

Unlike most Qt projects, Trojita uses CMake [34] instead of `qmake`, especially due to author's familiarity with the former system.

Trojita makes use of some standard C++ features that were adding during the TR1 [14] standardisation, namely the `std::tr1::shared_ptr` smart pointer. A working implementation of said class is required. Trojita build system will make use of compiler's built-in TR1 support, if available, and resort to using Boost if the compiler does not support this extension.

Some third-party code is also bundled in the source code distribution. A list of such products including the reason for not just using an external library follows:

1. QwwSmtpClient [35] is a Qt SMTP client library written by Witold Wysota. Released to public only in mid May 2009, obtaining a package of it might prove challenging. The copy in Trojita's codebase also contains some fixes which did not made it to the publicly released version yet.

2. ModelTest [36], a class for verifying certain assumptions of the Interview Model-View framework of Qt. Qt Software (former Trolltech) does not ship it with Qt sources, so Trojita includes a copy for developers' convenience. This feature is not turned on by default in release builds.

3. QtIconLoader [19], another helper library from Qt Software. Its purpose is to provide a unified icon set, as per relevant XDG standards.

4. KCodecs [37], an implementation of UTF-7, Base64, Quoted-Printable and related RFC 2047 standards. These functions are reasonably self-contained, so they are bundled with Trojita instead of depending on quite heavyweight `kdelibs`.

5.2.2 Obtaining source code

Source code is available from the CD (see A) attached to this thesis or online at `http://trojita.flaska.net/`.

5.2.3 Installing

Unpack the source tarball, prepare the build environment and launch the compile process by entering these commands:

1. `tar -xf trojita-1.0.tar.bz2`
2. `cd trojita-1.0`
3. `mkdir _build`
4. `cd _build`
5. `cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ..` (the `..` are not a typo)
6. `make`

After a while, a new binary, `trojita`, will appear in the build directory. Launch it to start Trojita.

5.3 Using Trojita

5.3.1 Configuration

All configuration is, thanks to the Qt's QSettings, stored in target platform's common storage location. The only supported mean of configuring Trojita is through its built-in configuration dialog that will automatically appear on the first run. It can also be launched any time later from the application menu.

On the first page, user is asked to provide identity data. These are used for composing e-mail messages.

Second page is the most important one in configuration of Trojita. It is the place where the user tells the application which IMAP server is it supposed to talk to, which credentials should be used for authentication, etc.

One unusual thing which is not common among other MUAs is the choice of how to connect to the IMAP server. Most users are probably familiar with connections over TCP, optionally secured by SSL/TLS, but the last option, the *Local Process*, is unique to Trojita. If selected, it replaces the usual TCP connection to the IMAP server with launching a local process, which is in turn used instead of the remote IMAP server. The most interesting use case is to set up SSH connection to the remote server (along with proper SSH keys) and use that as a secure, reliable and convenient way for accessing e-mail, without having to enter passwords all the time. For example, to launch an instance of Dovecot IMAP server on host `mail.example.org`, user should enter `ssh mail.example.org dovecot --exec-mail imap`.

The last page serves for configuring the message composition and sending. Trojita supports mail submission via SMTP[1] and local `sendmail`-compatible MSA[2] application.

5.3.2 Basic usage

After the configuration is complete, the user is presented with a familiar user interface. The left pane presents a view of remote mailboxes, or folders, on the IMAP server. The top widget contains a list of messages in the currently opened folder, and finally the bottom area is reserved for displaying the actual e-mail messages.

Depending on the network parameters and server load, it might take a while before the list of mailboxes appears. The delay is only significant when launching Trojita for the first time, for the retrieved list of mailboxes is cached on a persistent disk cache for later use.

After the list of mailboxes is loaded, user can click on them to open them in the *Message List* widget on the upper right corner of the application window. After having selected a mailbox, the list of messages appears shortly. Again, the loading is faster next time Trojita is launched.

Depending on user settings, Trojita typically will not transfer all message metadata upon opening a mailbox. This feature is implemented to conserve both server's and client's resources and to be nice to the network traffic. In a typical mode, that is, when not being told to be extra-mean with resources, Trojita fetches only metadata of those messages that are displayed on the visible screen region, and some more to make scrolling smoother. More on this topic is said in section 3.6 of this thesis.

Message contents is transfered on-demand only, that is, when user selects a message for viewing. As before, all persistent contents once downloaded from the server is cached in a persistent on-disk cache for future use. This applies to both message metadata and the real contents of e-mails.

[1]Simple Mail Transfer Protocol

[2]Mail Submission Agent[38], a local gateway whose purpose is forwarding e-mail to the network of MTAs

5.3.3 Managing Messages and Mailboxes

Messages displayed to the user are automatically marked as read after some delay. The idea behind this feature is to allow users to peek on a particular message, shortly judge its contents, classify it as a "process later" and then move on to another one. This is a common approach shared with other MUAs as well.

Manipulating with *message flags* (see chapter 2.2) is possible via the context menu (appearing on the message list) as well by standard keyboard shortcuts. The `M` key toggles the *Read* status and `Delete` key schedules messages for removal.

In consistency with the usual IMAP approach (and unlike the common modus operandi on mainstream desktop MUAs), Trojita has no concept of *Trash* [3] for messages. Instead, each message can be marked as deleted. If flagged as such, Trojita displays a graphical indicator next to the message subject. A deleted message will not be physically removed from the mailbox until the user explicitly requests such an action. This operation, called *expunge* in the IMAP dictionary, is available from the application menu. Expunging happens on one mailbox at a time, there is no way to tell the IMAP server to permanently remove all deleted messages from all mailboxes.

To create new mailbox, user is expected to use the context menu on the left *mailbox view* pane. Due to technical limitations, some IMAP servers distinguish among folders that can contain other mailboxes and folders for message store only. Additionally, there is no reliable way for the IMAP client to tell whether the remote server is affected by this limitation or not. Therefore, Trojita uses the same approach as other common MUAs and ask user what kind of mailbox she wants to create.

Mailbox removal is triggered from the context menu as well. Due to the possibly serious consequences of this action (there is no way to undelete a removed mailbox), user is presented with a confirmation dialogue asking her whether she is serious in her intent to remove the mailbox.

Messages can be copied or moved between mailboxes by simple drag-and-drop operations. To place a copy of one or more messages into another mailbox, simply select the messages to copy in the opened mailbox and then drag them with mouse to the target mailbox. If user wants to move them

[3] A *Trash* is a mailbox that presents a view of all deleted messages. It might or might not be a real mailbox on the server, or just a virtual folder implemented purely in the e-mail client.

instead of performing a copy, she is supposed to hold the `Shift` key. This behavior is determined by the Qt framework, so it might vary among the supported platforms.

5.3.4 Message Composition

Message Composition widget is triggered by the corresponding item in the menus. The displayed dialog should be familiar to most users.

5.3.5 Checking Mail

Trojita automatically checks the current folder for new messages. If the remote server supports the IDLE extension [10], the method used is extremely friendly to both network and server resources. In cases the server does not support this feature, Trojita resorts to polling with all it disadvantages.

5.4 Advanced usage

5.4.1 Partial Message Fetching

Thanks to unique features of the IMAP protocol, it is not necessary to fetch the whole message in order to display just a fragment of it. Trojita handles this automatically, fetching message parts on demand. It also will not transfer message metadata (like its RFC 2822 envelope) unless really needed.

5.4.2 Offline Mode

When put into *offline mode*, Trojita will prevent itself from accessing the network at all. Users will be still able to view already transfered data, but no checks for new messages will be performed and those that are not stored in local offline cache will not have actual message bodies displayed until the user selects either *online* or *expensive network* mode from the menu.

5.4.3 Expensive Network

Some users connect to their IMAP servers over a slow or otherwise unsuitable network. Examples include dial-up connections or various mobile solutions like GPRS[4] where the overall convenience of having all messages immediately available is usually held back by the need to conserve network resources. In such environments, it might be desirable to fetch only the absolute minimum data required to provide properly synchronized view of the remote mailbox.

In this situation, the user is expected to choose the *Expensive Network* mode from the application menu. Trojita will not preload data to improve interactivity, nor will it show all message parts by default. Upon viewing a message, only parts smaller than a certain size are transfered automatically, others will be loaded after clicking a button. This on-demand loading is implemented separately for each part, allowing great flexibility for the user to choose what is interesting and what can be ignored.

5.4.4 Online Mode

Online mode, the common mode of operation, is what users are familiar from other IMAP clients. Trojita employs intelligent preloading of interesting contents to minimize waiting times, yet does not blindly download everything, including parts that will not be ever shown.

5.4.5 Local Cache

All data once transfered form the network[5] are kept in a persistent offline on-disk cache. If a message data is transfered twice, it means that at least one of the following conditions was met:

1. The server could not guarantee the immutability of messages. This might happen when some process touches on-disk data internal to the IMAP server in a non-standard way.

2. User has changed her settings. Whenever any data used to access the remote server changes, the only safe thing to do is to throw away anything in the local cache and start over again.

[4]General packet radio service (GPRS) is a mobile data service typically offering low bandwidth, high latency Internet connection with billing based on the amount of transfered data

[5]Subject to some restrictions like maximal message part size and cache parameters

5.4.6 Full Tree View

As an optional part of the GUI, the *full tree view*, activated by the corresponding item in the menu, provides a full view of the server contents, including mailbox hierarchy, messages in mailboxes and mailbox parts. This widget is a nice demonstration of a complex nature of MIME messages – while most users are probably familiar with the concept of attachments, the truth is that MIME defines a fully recursive structure that one message can consist of. For example, it is perfectly possible to have a message composed of six parts, each of them being a forwarded message, and each of these forwarded messages being rich-structured as well. Trojita does its best at formatting the message in a sane and familiar way, but unfortunately some senders do not really follow the same philosophy and produce messages whose structure leaves much to be desired.

More on this matter can be found in section 3.2.

Chapter 6

Conclusion

The goal of this thesis was to demonstrate that an IMAP e-mail client could be usable, yet implement the relevant RFCs properly and behave in an efficient way.

After an introduction to the world of IMAP and other Internet messaging standards in the second chapter of the thesis, we described the design of Trojita, the application implemented in this thesis, as well as provided a comparison with other commonly used IMAP e-mail clients.

While it is evident that the GUI of Trojita is not as mature as the interface of other, more established Mail User Agents such as Mozilla Thunderbird or the KMail, the core IMAP engine design is sound. Already offering complete support for the IMAP4rev1 protocol as defined by RFC 3501 [1], Trojita implements most important IMAP extensions such as IDLE [10] and conforms to various IMAP client implementation recommendations which accumulated over more than twenty years of the IMAP history. Special emphasis is placed at transferring only the required data, minimizing network transfers as well as memory consumption. As a result of these optimizations, Trojita operates swiftly even when faced with huge mailboxes containing tens of thousands of e-mail messages.

Mailbox re-synchronization uses as much data from the previously recorded mailbox state as permitted by the standards while the persistent offline cache with a pluggable architecture allows storing of already downloaded message parts among sessions and enables offline browsing of the remote server's contents when no network connection is available.

The rendering of the MIME messages provides a faithful reflection of the MIME structure, yet sustains an acceptable level of user-friendliness.

A feature unique to Trojita is its support of operation over a UNIX pipe to the local process instead of through a TCP socket. When combined with SSH and the ssh-agent, it allows a convenient password-less operation without sacrificing security.

Finally, Trojita, as presented in this thesis, is a result of several years of the author's involvement in the area of Internet messaging. Trojita, being written in a modular way, provides a mature design platform, enabling the convenient and rapid integration of new features. It is the ultimate goal of the author to deliver a fully usable IMAP client aimed at advanced users who need to work with lots of e-mail in an effective way. Therefore, he plans to continue the maintenance of the project and develop new features in the years to come. After the GUI is matured, whole range of possibilities for new functions is open, from message encryption to supporting even experimental IMAP extensions.

Thanks to its unique design which employs both well-known software design patterns as well as new features offered by the modern Qt toolkit, Trojita certainly has the potential of becoming a competitive IMAP client on the open source desktop scene.

Bibliography

[1] Crispin, M.: Internet Message Access Protocol - Version 4rev1, *RFC 3501*, March 2003,
`http://tools.ietf.org/html/rfc3501`

[2] Crispin, M.: Ten Commandments of How to Write an IMAP client, September 2006, Washington,
`http://www.washington.edu/imap/documentation/commndmt.txt.html`

[3] Resnick, P. (editor) et al.: Internet Message Format, *RFC 2822*, April 2001,
`http://tools.ietf.org/html/rfc2822`

[4] Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, *RFC 2045*, November 1996,
`http://tools.ietf.org/html/rfc2045`

[5] Freed, N., Borenstein, N..: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, *RFC 2046*, November 1996,
`http://tools.ietf.org/html/rfc2046`

[6] Moore, K.: MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text, *RFC 2047*, November 1996,
`http://tools.ietf.org/html/rfc2047`

[7] Levinson, E.: The MIME Multipart/Related Content-type, *RFC 2387*, August 1998,
`http://tools.ietf.org/html/rfc2387`

[8] Crispin, M., Murchison, K.: Internet Message Access Protocol - SORT and THREAD Extensions, *RFC 5256*, June 2008, `http://tools.ietf.org/html/rfc5256`

[9] Myers, J.: IMAP4 non-synchronizing literals, *RFC 2088*, January 1997, `http://tools.ietf.org/html/rfc2088`

[10] Leiba, B.: IMAP4 IDLE command, *RFC 2177*, June 1997

[11] Gray, T.: Message Access Paradigms and Protocols, September 1995, online at `ftp://ftp.cac.washington.edu/mail/imap.vs.pop`

[12] Qt Software: The Interview Framework, `http://doc.trolltech.com/4.5/qt4-interview.html`

[13] Qt Software: QtWebKit Module, `http://doc.trolltech.com/4.5/qtwebkit.html`

[14] ISO/IEC TR 19768, C++ Library Extensions

[15] QProcess could close the reading pipe, thus killing the child process with SIGPIPE, *a thread on the qt-interest mailing list*, `http://lists.trolltech.com/pipermail/qt-interest/2009-May/006341.html`

[16] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1995

[17] Troost, R., Dorner, S., Moore, K.: Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field, *RFC 2183*, August 1997, `http://tools.ietf.org/html/rfc2183`

[18] Levinson, E.: Content-ID and Message-ID Uniform Resource Locators, *RFC 2111*, March 1997, `http://tools.ietf.org/html/rfc2111`

[19] Qt Software, February 2009, `http://labs.trolltech.com/blogs/2009/02/13/freedesktop-icons-in-qt/`

[20] Dawes, R., The FreeDesktop Project: Icon Naming Specification v. 0.8.90,
`http://standards.freedesktop.org/icon-naming-spec/icon-naming-spec-0.8.90.html`

[21] Dovecot, Secure IMAP server,
`http://www.dovecot.org/`

[22] Crispin, M.: The MIME Torture Test Mailbox,
`ftp://ftp.cac.washington.edu/imap/mime-examples/torture-test.mbox`

[23] Alpine Messaging System,
`http://www.washington.edu/alpine/`

[24] Does pine make use of the IMAP IDLE command?, *discussion board*,
`http://objectmix.com/pine/206216-does-pine-make-use-imap-idle-command.html`

[25] The Mutt E-mail Client,
`http://www.mutt.org/`

[26] KMail: Kontact Mail,
`http://kontact.kde.org/kmail/`

[27] Melnikov, A.: Common IMAP keywords, *Expired Internet Draft*,
`http://tools.ietf.org/html/draft-melnikov-imap-keywords-03`

[28] Sirainen, T., Cridland, D. et al.: Best Practices for Implementing an IMAP Client, `http://www.imapwiki.org/ClientImplementation`

[29] Mozilla Thunderbird,
`http://www.mozillamessaging.com/en-US/thunderbird/`

[30] Mozilla Developer Center: IMAP,
`https://developer.mozilla.org/en/IMAP`

[31] Mailody,
`http://www.mailody.net/`

[32] Elkins, M. et al.: MIME Security with OpenPGP, *RFC 3156*, August 2001
`http://tools.ietf.org/html/rfc3156`

[33] Qt Reference Documentation,
`http://doc.trolltech.com/4.5/`

[34] CMake Documentation,
`http://www.cmake.org/cmake/help/documentation.html`

[35] Wysota, W.: QwwSmtpClient released, May 2009,
`http://blog.wysota.eu.org/index.php/2009/05/13/qwwsmtpclient-released/`

[36] Qt Software, ModelTest, February 2008,
`http://labs.trolltech.com/page/Projects/Itemview/Modeltest`

[37] The KDE Project: kdelibs,
`http://api.kde.org/4.x-api/kdelibs-apidocs/`

[38] Gellens, R., Klensin, J.: Message Submission, *RFC 2476*, December 1998,
`http://tools.ietf.org/html/rfc2476`

Appendix A

Contents of the CD

The attached CD-ROM is an integral part of this thesis. The contents of the top-level `trojita` directory is the following:

bin Compiled binaries of Trojita suitable for running on several Linux flavours

doc A copy of each publicly available document referenced in the bibliography

doxygen An automatically generated API documentation of Trojita

src The source code of Trojita

thesis A printable PDF file with the text of this thesis