Preface

This thesis is about *masqmail*, a small mail transfer agent for workstations and home networks. In October 2007 I had chosen *masqmail* for my machines because of its small size though it was a "real" mail transfer agent. *masqmail* served me well since then and I have found no reasons to change.

Unfortunately, the *masqmail* package in *Debian*, which is my preferred GNU/Linux distribution, is unmaintained since the beginning of 2008. Unmaintained packages are likely to get dropped out of a distribution if critical bugs appear in them. Although *masqmail* had no critical bugs, this was a situation I definitely wanted to prevent.

Using my diploma thesis as a "power-start" for maintaining and developing *masqmail* in the future was a great idea. As it came to my mind I knew this is the thing I *wanted* to do. — I did it! :-)

The overall goal of this document is to revive *masqmail* in usage and development. *masqmail* was not developed further in the last five years although the world of email has changed during this time. Hence quite some work needed to be done.

I decided to start at the basis and analyze the environment and *masqmail* throughout to end up in concrete plans of what should be done and how it should be done to turn *masqmail* into a modern mail transfer agent again.

The actual implementation of the proposed changes will follow-up this thesis. Here solutions are identified, described, discussed, and recommended but not implemented. I have been working in the code and have fixed bugs during the time I worked on the thesis, though.

This document is primary written with an audience of *masqmail* developers and developers of other mail transfer agents in mind. But users of *masqmail* and everyone who is interested in email systems in general may find this thesis an interesting literature, too.

However, at least basic knowledge about Unix and C programming is a prerequisite for chapters three, four, and five. Kernighan and Pike's "The UNIX Programming Environment" [**?**] is a valuable source to gain information about Unix. Programming in the C language is best learned from Kernighan and Ritchie's "The C Programming Language" [**?**].

Organization

The document consists of six chapters, each covering a delimited part of the overall topic and building upon the content and results of previous chapters. The first three chapters lead into the topic and create a solid base where the second part builds upon. The chapters four and five form the central part of the thesis as they focus on *masqmail*.

Chapter 1 **introduces** *masqmail* to the reader. It presents the properties, goals, advantages, and problems of the program. Basic concepts of the email technology are also described and later assumed to be known.

Chapter 2 **analyzes the market** of electronic communication and email. This chapter gives sound reasons for the sense of future development of *masqmail* by showing that email will remain an important technology in the future. It tries to identify future trends, too.

Chapter 3 **deals with mail transfer agents** (MTAs) which are the most important entities of the email transport structure. MTAs are defined, classified, and the most important ones are presented and compared.

Chapter 4 **focuses on *masqmail*'s present and future**. It is the core of the thesis. Requirements are identified and lead to a list of pending work tasks. Then possible strategies for future development are discussed.

Chapter 5 **describes improvement plans** which are based on decisions in chapter four, in more detail. A proposed architecture for a redesigned *masqmail* is presented, too.

Chapter 6 **summarizes** the most important results and closes the thesis.

Conventions

The following typographic conventions are used in this thesis:

1. *Italic shape* is used to emphasize text, to introduce new terms, and for names, including product, host, and user names, as well as email addresses.

2. For names of persons Small Caps are used.

3. File and path names, contents of files, and output from programs are displayed in `Typewriter font`.

References to external resources are marked using one of three styles, distinguished by the type of resource.

1. References to books, articles, and documents of similar kind, look like this: [**?**]. The letters represent the author(s) (here Kernighan and Pike), while the number represents the year of publication (here 1984).

2. Websites are different from documents as they are less some text written by some author but more a place where information is gathered. Website may also change from time to time, thus the date of access is given to indicate the version to which was referred. References to websites have such appearance: [**?**].

3. *Request for Comments* are those documents that define the Internet. They are referenced directly by their unique number. For instance: RFC 821.

The Bibliography is located at the end of the thesis. It also includes a list of the relevant RFCs and how they can be retrieved.

Acknowledgments

First, I want to thank Oliver Kurth for writing *masqmail*; I build upon his work. My second thanks goes to professor Markus Schäffter, my advisor. He was the one who made this thesis possible by putting faith in me and this topic. I very much enjoyed the time with him.

I thank Christian Langbein and professor Volkmar Kese for teaching me important lessons about structure. You are so right, it is all about: structure, structure, structure.

My Dad and my friend Julian Forster took time for me so I could explain various parts of the thesis to them; this was important, thanks. James Stenard was of great help in questions about the English language, thanks. Roger Schietzel double-checked all web addresses and ISBNs for validity, thanks for covering this bulky task.

Henry Atting, Joachim Breitner, Marc Geis, Jochen Roth, and Hans-Jörg Schaaf (in alphabetical order) had a look at my thesis and returned comments and suggestions—each one was valuable. Thank you all.

Not to forget is everyone who discussed with me on mailing lists and in private communication, and my family for backing me.

There is also an institution that needs to be praised: The *Württembergische Landesbibliothek* in Stuttgart; it was the most productive place to work and the most impressive one, too.

But the most support I did receive from Lydi. I am deeply grateful for your patience and sacrifice during the last months; for your motivation and encouragement; and for the ease I found in your arms. Thank you!

markus schnalke

Chapter 1

Introduction

This chapter introduces some basic email concepts that are essential for understanding the remainder of the thesis. Then *masqmail*—the program of interest—is presented. History, typical usage, and the function it provides are described. After an explanation of *masqmail*'s relevance, its weaknesses are pointed out. Solving these weaknesses is the topics that is covered throughout this thesis.

1.1 Email prerequisites

Electronic mail is a service on the Internet and thus, like other Internet services, defined and standardized by *Requests For Comments* (short: RFCs) under management of the *Internet Engineering Task Force* (short: IETF). RFCs are highly technical documents and it is not required that the readers of this thesis are familiar with them.

This section gives an introduction into the basic internals of the email system in a low-technical language. It is intended to make the reader familiar with the essential concepts of email as they are essential throughout the thesis.

Mail agents

This thesis will frequently use the three terms: MTA, MUA, and MDA, naming the three different kinds of nodes of the email infrastructure. Here, they are explained with references to the "snail mail" system which is known from everyday life. Figure 6.1 shows the relation between those three mail agents and the way an email message takes when passing through the system.

MTA: *Mail Transfer Agents* are the post offices for electronic mail. The basic job of an MTA is to transport mail from senders to recipients, or more pedantic: from MTA to MTA. *sendmail*, *exim*, *qmail*, *postfix*, and, of course, *masqmail* are MTAs. MTAs are explained in more detail in chapter 3.

MUA: *Mail User Agents* are the software users deal with. A user writes and reads email with it. The MUA passes outgoing mail to the nearest MTA. Also the MUA displays the contents of the user's mailbox. Well known MUAs are *Mozilla Thunderbird* and *mutt* on Unix systems, and *Microsoft Outlook* on *Windows*.

MDA: *Mail Delivery Agents* correspond to postmen in the real world. They receive mail, destined to recipients they are responsible for, from an MTA, and deliver it to the mailboxes of those recipients. Many MTAs include an own MDA, but independent ones exist: *procmail* and *maildrop* are examples.

[Figure 1 about here.]

Mail transfer with SMTP

Today most of the email is transferred using the *Simple Mail Transfer Protocol* (short: SMTP), which is defined in RFC 821 and the successors RFC 2821 and RFC 5321. A good entry point for further information is [**?**].

A selection of important concepts of SMTP is explained here.

First the *store-and-forward* transfer concept. This means mail messages are sent from MTA to MTA, until the final MTA (the one which is responsible for the recipient) is reached. The message is stored for some time on each MTA, until it is forwarded to the next MTA.

This leads to the concept of *responsibility*. A mail message is always in the responsibility of one system. First it is the MUA. When it is transferred to an MTA, this MTA takes over the responsibility for the message, too. The MUA can then delete its copy of the message. This is the same for each transfer—from MTA to MTA and finally from MTA to the MDA—the message gets transferred and if the transfer was successful, the responsibility for the message is transferred as well. The responsibility chain ends at a user's mailbox where he himself has control on the message.

A third concept is about failure handling. At any step on the way an MTA may receive a message it is unable to handle. In such a case this receiving MTA will *reject* the message before it takes responsibility for it. The sending MTA still has responsibility for the message and may try other ways for sending the message. If none succeeds the MTA will send a *bounce message* back to the original sender with information on the type of failure. Bounces are only sent if the failure is expected to be permanent or if the transfer still was unsuccessful after many tries.

Mail messages

Mail messages consist of text in a specific format. This format is specified in RFC 822, and the successors RFC 2822 and RFC 5322.

A message has two parts, the *header* and the *body*. The header of an email message is similar to the header of a (formal) letter. It spans the first lines of the message up to the first empty line. The header consists of several lines, called *header lines* or simply *headers*. They specify the sender, the recipient(s), the date, and possibly further information. Their order is irrelevant. Headers are named like the colon-separated start of those lines, for example the "`Date:`" header. A user may write the header himself but normally the MUA does this job.

The body is the payload of the message. It is under full control of the user. From the view point of the SMTP protocol, it must consist of only 7-bit ASCII text. But arbitrary content can be included by encoding it to 7-bit ASCII. MIME is the common SMTP extension to handle such conversion automatically in MUAs.

Following is a sample mail message with four header lines (`From:`, `To:`, `Date:`, and `Subject:`) and three lines of message body.

```
From: markus@host01
To: alice@host02, bob@host03
Date: Sun, 11 Jan 2009 18:20:01 +0100
Subject: A sample mail message

This is the content of the message.

Further empty lines can be included.
```

Email messages are put into *envelopes* for transfer. This concept is also derived from the real world so it is easy to understand. The envelope is used to route the message from sender to recipient. It contains the sender's address and addresses of one or more recipients. Envelopes are generated by MTAs, usually from mail header data. The user has not to deal with them.

Each MTA on the way reads envelopes it receives and generates new ones. If a message has recipients on different hosts, then the message gets copied and sent within multiple envelopes, one for each host.

The sample message would lead to two envelopes, one from *markus@host01* to *alice@host02*, the other from *markus@host01* to *bob@host03*. Both envelopes would contain the same message.

1.2 The *masqmail* project

The *masqmail* project was initiated by Oliver Kurth in 1999. His aim was to create a small MTA that is especially focused on computers with dial-up Internet connections. Throughout the next four years he worked steadily on it, releasing new versions every few weeks. During the active phase of development 53 version have been released. In average, this is a new version every 20 days.

This thesis is based on the latest release of *masqmail*—version 0.2.21, dated November 2005. It was released after a 28 month gap of inactivity. The source code of 0.2.21 is the same as of 0.2.20, with only build documents modified. The homepage of *masqmail* [**?**] does not include this latest release, but it can be retrieved from the *Debian* package pool[1] [**?**].

masqmail is covered by the *General Public License* (short: GPL) version two or any later version [**?**]. This qualifies *masqmail* as Free Software [**?**].

Kurth abandoned *masqmail* after 2005 and no one adopted the project since then. Thus, the author of this thesis decided to take over responsibility for *masqmail* now. He received Kurth's permission to do so in private telephone conversation with Kurth on September 4, 2008.

The program's new homepage [**?**] includes a collection of available information about this MTA.

1.2.1 Target field

Kurth's intention when creating *masqmail* is best told in his own words:

> MasqMail is a mail server designed for hosts that do not have a permanent internet connection eg. a home network or a single host at home. It has special support for connections to different ISPs. It replaces sendmail or other MTAs such as qmail or exim. [**?**]

It is intended to cover a specific niche: non-permanent Internet connection and different *Internet Service Providers* (short: ISPs).

Although it can basically replace other MTAs it is not *generally* aimed to do so. The package description of *masqmail* within *Debian* states this more clearly by changing the last sentence to:

> In these cases, MasqMail is a slim replacement for full-blown MTAs such as sendmail, exim, qmail or postfix. [**?**]

The program is a good replacement "in these cases" but not generally, since it lacks essential features for running on openly accessible mail servers. It is primarily not secure enough for being accessible from untrusted locations.

masqmail is best used in home networks which are non-permanently connected to the Internet. It is easy configurable for situations which are rarely solvable with the common MTAs. Such include different handling of mail to local or remote destination and respecting different routes of online connection. These features are explained in more detail in section 1.2.2.

While many other MTAs are general purpose MTAs, *masqmail* aims on special situations. Nevertheless, it can be used as general purpose MTA, too. Especially this was a design goal of *masqmail*: To be a replacement for *sendmail* or similar MTAs.

masqmail is designed to run on workstations and on servers in small networks, like they are common in SOHOs (*Small Offices/Home Offices*).

[1] The URL is:
`http://ftp.de.debian.org/debian/pool/main/m/masqmail/masqmail_0.2.21.orig.tar.gz`

Typical usage scenarios

This section describes three common setups that make sensible use of *masqmail*. The first two are shown in figure 6.2.

[Figure 2 about here.]

Imagine an Internet-connected home network consisting of some workstations.

Scenario 1: If no server is present, every workstation would be equipped with *masqmail*. Mail transfer within the same machine or within the local net works straight forward using direct transfer. Outgoing mail to the Internet is sent to an *Internet Service Provider* (short: ISP) for relaying whenever the router goes online. The configuration of *masqmail* would be the same on every computer; only host names would differ. To receive mail from the Internet requires a mailbox on the ISP's mail server. Mail needs to be fetched from the ISP's server onto the workstation using the POP3 or IMAP protocol.

Scenario 2: In the same network but with a server, one could have *masqmail* running on the server and using simple forwarders (see section 3.1) on the workstations to transfer mail to the server. The server would then, dependent on the destination of the message, deliver locally or relay to an ISP's server for further relay. This setup does only support mail transfer to the server but not back to a workstation. However, this can be solved by mounting the user's mailbox from the server to the workstation or by using POP3 or IMAP. Mail transfer from the ISP to the local server needs POP3 or IMAP as well.

Scenario 3: A third scenario is unrelated as it is about notebooks. Notebooks are usually used as mobile workstations. One uses them to work at different locations. With the increasing popularity of wireless networks this becomes more and more common. Different networks demand for different setups: In one network it is best to send mail to an ISP for relay. In another network it might be preferred to use a local mail server. A third network may have no Internet access at all, hence using a local mail server is required. All these different setups can be configured once and then used by simply telling the online state to *masqmail*, even automatically within a network setup script.

In general, all kinds of usage scenarios within a trusted network are possible. Important to notice is that mail can not be sent from outside into the trusted network then. For using *masqmail* on notebooks it is suggested to only accept mail from local users because notebooks are often in untrusted environments.

Limitations

Although *masqmail* is seen as a replacement for other general purpose MTAs, it should not be used on large mail servers. The reasons are that it implements only a basic subset of features and that its performance and security is not as good as needed for such usage.

The author, Kurth, warns on the old project's website about using *masqmail* to accept connections from the Internet because of the risk of being an open relay:

> MasqMail is not designed to run on a host with a permanent internet connection. It does not have the ability to check for spam mail and it will relay everything from everywhere to everywhere. Use another mail server such as exim for permanent connections. **[?]**

The actual problem is not the permanent Internet connection but listening for incoming mail on it. If a firewall is closed for incoming mail, then the permanent Internet connection is no problem. To use *masqmail* for permanent Internet connections it needs to be secured with care.

The Internet is the common example for an untrusted network but other networks may be untrusted, too.

1.2.2 Features

This thesis regards version 0.2.21 of *masqmail*. This is the last version released by Oliver Kurth.

The source code

masqmail is written in the C programming language. The program, as of version 0.2.21, consists of 34 source code and eight header files which contain about 9 000 lines of code[2]. Additionally, it includes a *base64* implementation (about 300 lines) and *md5* code (about 150 lines). For systems that do not provide *libident*, this library is distributed as well (circa 600 lines); an available shared library has higher precedence in linking, though.

The only mandatory dependency is *glib*—a cross-platform software utility library, originated in the GTK+ project. It provides safe replacements for many standard library functions, especially for the string functions. It also offers handy data containers, easy-to-use implementations of data structures, and much more.

Some parts of *masqmail*'s functionality can be included or excluded at compile time by defining symbols. To enable maildir support for example, one has to add `--enable-maildir` to the configure call. Otherwise the concerning code gets removed during preprocessing.

[2]Measured with *sloccount* by David A. Wheeler [**?**].

With *masqmail* comes the small tool `mservdetect`; it helps setting up a configuration that uses the *mserver* system for online state detection. Two other binaries get compiled for testing purposes: `readtest` and `smtpsend`. These three additional programs use parts of *masqmail*'s source code; they only add a file with a `main()` function each.

Features

masqmail supports two channels for incoming mail:

1. Standard input which is used when `masqmail` (or the `sendmail` link) is executed on the command line
2. A TCP socket which is used by local or remote clients that talk SMTP

The outgoing channels for mail are:

1. Direct delivery to local mailboxes (in *mbox* or *maildir* format)
2. Local pipes to pass mail to a program (e.g. to MDAs or to gateways to FAX or UUCP)
3. TCP sockets to transfer mail to other MTAs using the SMTP protocol

Figure 6.3 shows this as a picture. (The "online state" input is explained a bit later.)

[Figure 3 about here.]

Outgoing SMTP connections feature SMTP-AUTH and SMTP-after-POP authentication but incoming connections do not. Using wrappers for outgoing connections is supported. This allows encrypted communication through a gateway application like *openssl*.

Mail queuing is essential for *masqmail* and thus supported of course, alias expansion is also supported.

The *masqmail* executable can be called by various names for sendmail-compatibility reasons. As many programs expect the MTA to be located at `/usr/lib/sendmail` or `/usr/sbin/sendmail`, symbolic links are pointing from there to the *masqmail* executable. Furthermore does *sendmail* support calling it with a different name instead of supplying command line arguments. The best known of these shortcuts is `mailq` which is equivalent to calling it with the argument `-bq`. *masqmail* recognizes the shortcuts `mailq`, `smtpd`, `mailrm`, `runq`, `rmail`, and `in.smtpd`. The first two are inspired by *sendmail*. Not implemented yet is the shortcut `newaliases` because *masqmail* does not generate binary representations of the alias file.[3] `hoststat` and `purgestat` are missing for complete sendmail-compatibility.

[3]A shell script named `newaliases` that invokes `masqmail -bi` can provide the command to satisfy strict requirements.

Additional to the MTA job, *masqmail* also offers mail retrieval services by acting as a POP3 client. It can fetch mail from different remote locations, also dependent on the active online connection. Such functionality is especially useful in a setup like *Scenario 2* on page 5.

Online detection and online routes

masqmail focuses on handling different non-permanent online connections, thus a concept of online routes is used. One may configure any number of routes to send mail. Each route can have criteria to determine if some message is allowed to be sent over it. Mail to destinations outside the local network gets queued until a suitable online connections is available.

The idea behind this concept is sending mail to the Internet through the mail server of the same ISP over which one had dialed in. It was quite common that ISPs accepted mail for relay only if it came from a online connection they managed. This means, it was not possible to relay mail through the mail server of one ISP while being online through the connection of another ISP. *masqmail* is a solution to the wish of switching the relaying mail server easily.

Related is *masqmail*'s ability to rewrite the sender's email address dependent on which ISP is used. This prevents mail from being likely classified as spam.

To react on the different situations, *masqmail* needs to query the current online state. Is an online connection available? And if it is: Which one? Three methods are implemented:

1. Reading from a file
2. Reading the output of a command
3. Querying an *mserver* system

Each method may return a string naming the route that is online or returning nothing to indicate offline state.

Mail for hosts inside the local network or for users on the local machine is not touched by this concept; such mail is always sent immediately.

1.3 Why *masqmail* is worth it

First of all, *masqmail* is better suited for its target field of operation (multiple non-permanent online connections) than any other MTA. Especially is such usage easy to set up because *masqmail* was designed for it. Many alternative MTAs were not designed for those scenarios as the following two example show: "Exim is designed for use on a network where most messages can be delivered at the first attempt." [**?**, page 30]. And: "qmail was designed for well-connected hosts: those with high-speed, always-on network connectivity." [**?**, page9].

masqmail make it easy to run an MTA on workstations or notebooks. There is no need to do complex configuration or to be a mail server expert. Only a handful of options need to be set; the host name, the local networks, and one route for relaying are sufficient in most times.

Probably users say it best; in this case Derek Broughton:

> No kidding. The whole point is that you _have_ to have an MTA and you don't want to configure Postfix/Exim/Sendmail/Qmail (almost all of which I've actually done).
>
> I now use masqmail – it's really simple, my configuration is all in debconf, it's supported by whereami, and it's really simple :-)
>
> I'm sure you can make any MTA behave nicely when offline, but it was a chore with all of them. [**?**, post #8]

Not to forget *masqmail*'s size. *masqmail* is much smaller than full-blown MTAs like *sendmail*, *postfix*, or *exim*, and still smaller than *qmail*. (See section 3.3 for details.) This makes *masqmail* a good choice for workstations or even embedded computers.

Again words of a user who chose *masqmail* as MTA on his old laptop with a 75 megahertz processor and eight megabytes of RAM:

> Masqmail appears to be a great sendmail replacement in this case. It's small and is built to support sending mail "off-line", and to connecting to the SMTP servers of several ISPs. [**?**]

masqmail is also used in a scientific project: Wolfgang Leister chose *masqmail* for the prototype implementation of the *HikerNet* [**?**]. The *HikerNet* is an adhoc network for peer-to-peer communication in otherwise network-less areas. Unfortunately, the usage of *masqmail* for the prototype is not documented. The author of this thesis received the information in private email communication with Leister in October 2008. Leister stated, he chose *masqmail* as email-to-hikernet gateway because it was well suited and easy to set up for this particular usage. Other MTAs would have been possible choices, but it was easier with *masqmail*.

Although the development of *masqmail* has been stopped in 2003, *masqmail* still has its users. Having users is already reason enough for further development and maintenance. This applies especially when the software covers a niche and when requirements for such software in general changed. Both is the case for *masqmail*.

It is difficult to get numbers about users of Free Software because no one needs to tell anyone when he uses some software. *Debian*'s *popcon* statistics [**?**] are a try to provided numbers. For January 2009, the statistics report 60 *masqmail* installations of which 49 are in active use. If it is assumed that one third of all *Debian* users report their installed software[4], there would be in

[4]One third is a high guess as it means there would be only about 230 thousand *Debian* installations in total. But according to the *Linux Counter* [**?**] between 490 thousand and 12 million *Debian* users can be estimated.

total around 150 active *masqmail* installations in *Debian*. *Ubuntu* which also does *popcon* statistics [**?**], counts 82 installations with 13 active ones. If here also one third of all systems submit their data, 40 active installations can be added. Including a guessed amount of additional 30 installations on other Unix operating systems makes about 220 *masqmail* installations in total. Of course one person may have *masqmail* installed on more than one computer, but a total of 150 different users seems to be realistic.

One thing became clear now: *masqmail* has users. And software that is used should be developed and maintained.

1.4 Problems to solve

A program that is neglected for more than five years in a field of operation that changed during this time surely needs improvement. Security and spam have highly increased in importance since 2003. Dial-up connections became rare, instead broadband flat rates are common now. Other MTAs evolved in respect to theses changes—*masqmail* did not.

The current market situation and trends for the future need to be identified. Looks at other MTAs need to be taken. Required work on *masqmail* needs to be defined in combination with the evaluation of strategies to do this work. And a plan for further development should be created.

1.5 Delimitation

This thesis is neither a installation guide for *masqmail* nor a detailed explanation of *masqmail*'s source code. Installation and setup guides can be found on *masqmail*'s homepage [**?**].

The POP3 functionality of *masqmail* receives few regard in this document because it is not directly related to the core of *masqmail* which is being an MTA.

The *mserver* system to query the online state is also only mentioned but not regarded further. It seems best to move this functionality into a separate program which is run through the shell command interface, anyway.

Chapter 2

Market analysis

This chapter analyzes the current situation and future trends for electronic communication in general and for email in particular. First email's position within other electronic communication technologies is located. Then trends for the whole field of electronic communication are shown. Afterwards opportunities and threats in the market and trends for email are identified. The insights of these analysis result in a summary of things that are important for developing future-proof email software.

2.1 Electronic communication technologies

Electronic communication is “communication by computer”, according to the *WordNet* database of the *Princeton University* [**?**]. Mobile phones and fax machines should be seen as computers here, too. The *Science Glossary* of the *Pennsylvania Department of Education* [**?**] describes electronic communication as “System for the transmission of information using electronic technology (e.g., digital cameras, cellular telephones, Internet, television, fiber optics).”

Electronic communication needs no transport of tangible things, only electrons, photons, or radio waves need to be transmitted. Thus electronic communication is fast in general. With costs mainly for infrastructure and very low costs for data transmission is electronic communication also cheap communication. Primary the Internet is used as underlying transport infrastructure. Thus electronic communication is available nearly everywhere around the world. These properties—fast, cheap, available—make electronic communication well suited for long distance communication.

As globalization proceeds and long distance communication becomes more and more important, the future for electronic communication is bright.

Electronic communication includes the following technologies: electronic mail (email), Instant Messaging (IM), chats (e.g. IRC), short message service (SMS), multimedia message service (MMS), voice mail, video messages, and Voice over IP (VoIP).

2.1.1 Classification

Electronic communication technologies can be divided in synchronous and asynchronous communication. Synchronous communication is direct dialog with little delay. Telephone conversation is an example. Asynchronous communication consists of independent messages. Dialogs are possible as well, but not in the same direct fashion. These two groups can also be split by the time which is needed for data delivery. Synchronous communication requires nearly real-time delivery, whereas for asynchronous communication message delivery times of several seconds or minutes are sufficient.

Another possible separation is to distinguish recorded and written information. Recorded information, like audio or video data, is accessible only in a linear way by spooling and replay. Written information, on the other hand, can be accessed in arbitrary sequence, detail, and speed.

Lenke and Schmitz use the same criteria to classify *new media* [**?**]. They additionally divide into local and remote communication—the latter is presumed here—and by the number of communication participants. A classification by participant structure is omitted here, as communication technologies for many-to-many communication (like chat rooms) are usable for one-to-one (private chat) too, and ones for one-to-one (email) are usable for many-to-many (mailing lists).

Figure 6.4 shows a classification of communication technologies by the properties synchronous/asynchronous and written/recorded. Email and SMS are examples for written and asynchronous communication; IM and chats are ones for written but synchronous communication. Voice mail and video messages stand as examples for recorded asynchronous communication. VoIP represents recorded synchronous communication.

[Figure 4 about here.]

One might be surprised to find Instant *Messaging* not in the group of *message* communication. Instant Messaging could be put in both groups because it allows asynchronous communication additional to being a chat system. The reasons why it is classified as dialog communication are its primary use for dialog communication and the very fast—instant—delivery time.

Email is not limited to written information, at least not anymore since the advent of MIME, which allows to include multimedia content in textual email messages. Thus recorded information can be sent as sub parts of emails. The same applies to Instant Messaging too, where file transfer is an additional sub service offered by most systems. In general recorded information can be transmitted in an encoded textual form.

2.1.2 Life cycle analysis

Life cycle analysis are common for products but also for technologies. This one here is for electronic communication technologies. The first dimension regarded is the life time of the subject. It is segmented into the introduction, growth, mature, saturation, and decline phases. The second dimension can display market share, importance, or similar values. The graph has always an S-line shape, with a slow start, a rapidly increasing first half, the highest level in the fourth fifths, and a slowly declining end. Reaching the end of the life cycle means that the subject gets superseded by successors or the market situation changed thus it is old-fashioned.

The current position on the life cycle of some selected communication technologies is shown in figure 6.5. It is important to notice that the time dimension can be different for each technology—some life cycles are shorter than others—the shape of the graph, however, is the same.

[Figure 5 about here.]

Video messages and voice mail are technologies in the introduction phase. Voice over IP is heavily growing these days. Instant Messaging has reached maturation and is still growing. Email is an example for a technology in the saturation phase. Fax, for instance, is a declining technology.

Email ranges in the saturation phase which is defined by a saturated market. No more products are needed: there is no more growth. This means, email is a technology which is used by everyone who want to use it. It is a standard technology. The current form of email in the current market is on the top of its life cycle. The future is decline, sooner or later.

But life cycles positions change as the subject or the market changes. An examples is the *Flash* animation software [**?**]. The product's change from a drawing and animation system to a technology for website creation, advertising, and movie distribution, and the thus changing target market, made it slip back on the life cycle. If the email system would evolve to become the basis for Unified Messaging (see section 2.1.3), a similar slip back would be the consequence.

The DVD standards DVD+ and DVD− are an example for a changing market. With the upcoming next generation formats *Blu-ray Disc* [**?**] and HD-DVD [**?**], a much sooner decline of DVD+ and DVD− started, even before they reached their last improvement steps in storage size. Such can happen to email too, if Unified Messaging is a revolution to the email system instead of an evolution.

2.1.3 Trends

Following are the trends for electronic communication. They are shown from the view point of MTAs. Nevertheless are these trends common for all of the communication technology.

Consolidation

There is a consolidation of communication technologies with similar transport characteristics going on, nowadays. Email is the most flexible kind of asynchronous communication technology in major use. Hence email is the best choice for transferring messages of any kind today. But in future it probably will be *Unified Messaging*, which tries to group all types of asynchronous messaging into one communication system. It aims to provide transparent transport for all kinds of content and flexible access interfaces for all kinds of clients. Unified Messaging seems to have the potential to be the successor of all asynchronous communication technologies, including email.

Today email still is the major asynchronous communication technology and it probably will be it for the next years. Unified Messaging needs similar transfer facilities as email, thus it seems to be rather an evolution to the current technology than a revolution. Hence MTAs will still be of importance in future, though maybe in a modified form.

Integration

Integration of communication technologies becomes popular. This goes beyond consolidation, because communication technologies of different kinds are bundled together to make communication more convenient for human beings. User interfaces tend to go the same direction. The underlying technologies are going to get grouped. But it seems as if synchronous and asynchronous communication can not be joined together in a sane way, thus they will probably only merge at the surface.

Communication hardware

Communication hardware comes from two different roots: On one side, the telephone, now available as mobile phones. This group centers around recorded data and dialog but messages are also supported by the answering machine and SMS. On the other side, mail and its relatives like email, which use computers as main hardware. This part centers around document messages but also supports dialog communication in Instant Messaging and Voice over IP.

The last years finally brought the two groups together, with *smart phones* being the merging hardware element. Smart phones are computers in the size of mobile phones or mobile phones with the capabilities of computers, however one likes to see it. They provide both functions by being telephones *and* computers.

Smart phones match well the requirements of recorded data for which they were designed. Text is difficult to write with their minimal keyboards, but speech to text converters may provide help in future. This leads to a need for ordinary computers for the field of exchanging text documents and as better input hardware for all written information.

It seems as if a combination of desktop computers and smart phones will be the hardware used for communication in future. Both specialized to the best matching communication technologies but with support for the others, too. Hence facilities for transferring information off and onto the devices will be needed.

Unified Communication

Unified Communication is the technology that aims to consolidate and integrate all electronic communication and to provide access for all kinds of hardware clients. Unified Communication tries to bring the three trends here mentioned together. The PC *Magazine* has the following definition in its Encyclopedia: "[Unified Communications is t]he real-time redirection of a voice, text or e-mail message to the device closest to the intended recipient at any given time." [**?**]. The main goal is to integrate all kinds of communication (synchronous and asynchronous) into one system, hence this requires real-time delivery of data.

According to Michael Osterman, Unified Communications is already possible as far as various incoming sources are routed to one storage where messages can be accessed by different clients [**?**]. But a system with an "intelligent parser of a single data stream into separate streams that are designed to meet the real-time needs of the user" is a goal for the future, he says.

The question is whether the integration of synchronous and asynchronous communication does make sense. A communication between one person talking on the phone and the other replying with an instant messenger, certainly does (assumed the text-to-speech and speech-to-text converting is fast and the quality good enough). But transferring large video messages with the same technology as real-time communication data, possibly does not.

Unified Messaging

Unified Messaging, although often used exchangeable with Unified Communications, is only a subset of it. It does not require real-time data transmission and is therefore only usable for asynchronous communication [**?**]. Unified Messaging's basic function is: Receiving incoming messages from various channels, converting them into a common format, and storing them into a single memory. The stored messages can then be accessed from different devices [**?**].

The easiest way to implement Unified Messaging is to base it on either email and convert all input sources to email messages (as attachments for instance) and store them in the user's mailbox, or use the telephone system as basis and convert text messages to speech. Both is technically possible for asynchronous communication.

Finally, a critical voice from Jesse Freund, who voted Unified Messaging on top of a hype list, published by *Wired.com* ten years ago. His description of the technology ended with the humorous sentences:

> Unified messaging is a nice idea, but a tough sell: The reason you bought a cell phone, a pager, and a fax/modem is because each does its job well. No one wants to download voice mail as a series of RealAudio messages or sit through a voice mail bot spelling out email, complete with 'semicolon dash end-parenthesis' for ;-). [?]

2.2 Electronic mail

After viewing the whole market of electronic communication, a zoom into the market of electronic mail follows. Email is an asynchronous communication technology that focuses on the transport of textual information. The market situation for email is important, because this thesis is about an MTA. Interesting questions are: Is email future-safe? How will electronic mail change? Will it change at all? Which are the critical parts? These questions matter when deciding the directions for further development of an MTA. They are discussed in this section.

2.2.1 SWOT analysis

A SWOT analysis regards the strengths and weaknesses of a subject against the opportunities and threats of its market. The slightly altered form called *Dialectical* SWOT *analysis*, which is used here, is described in [?]. SWOT analysis should always focus on a specific goal which is to reach. In this case, the main goal is to make email future-safe.

The two dimension—the subject and the market—are regarded in relation to each other by the analysis. Here the analysis shall be driven by the market's dimension. Thus first threats of the market are identified and split into being strengths or weaknesses of email. Then the same is done for opportunities of the market.

Threats

The market's main threat is *spam*, also named *junk mail* or *unsolicited commercial email* (UCE). David A. Wheeler is clear about it:

> Since *receivers* pay the bulk of the costs for spam (including most obviously their time to delete all that incoming spam), spam use will continue to rise until effective technical and legal countermeasures are deployed, *or* until people can no longer use email. [?]

The amount of spam is huge. Panda Security and Commtouch write in their *Email Threats Trend Report* for the second Quarter of 2008: "Spam levels throughout the second quarter averaged 77 %, ranging from a low of 64 % to a peak of 94 % of all email [...]" [?, page 4]. The report sees the main source of spam in bot nets consisting of zombie computers: "Spam and malware levels remain high for yet another quarter, powered by the brawny yet agile networks of zombie IPs." [?, page 1]. This is supported by IronPort Systems: "More than 80 percent of spam now comes from a 'zombie'—an infected PC, typically in a consumer broadband network, that has been hijacked by spammers." [?]. Positive for MTAs is that they are not the main source for spam, but it is only a small delight. Spam is a general weakness of the email system because it is not stoppable.

Opportunities

Opportunities of the market are large data transfers, originating in multimedia content, which becomes popular. If email is used as basis for Unified Messaging, lots of voice and video mail will be transferred. Email is weak related to this kind of data: The data needs to be encoded to ASCII which stresses mail servers a lot. Additionally a lot of traffic is generated by the *store-and-forward* transfer, which SMTP uses.

The use of different hardware to access mail is another opportunity of the market. But as more hardware gets involved, the networks become more complex. Thus the need for more software and infrastructure to transfer mail within the growing network might be a weakness of the email system.

An opportunity of the market and at the same time a strength of electronic mail is its standardization. Few other communication technologies are standardized, and thus freely available, in a similar way. Another opportunity and strength is the modular and extensible structure of electronic mail; it can easily evolve to new requirements.

The increasing integration of communication channels is an opportunity for the market. But deciding whether it is a weakness or strength of email is difficult. Due to the impossibility to integrate synchronous stream data and large binary data, it is a weakness. But it is also a strength, because arbitrary asynchronous communication data already can be integrated. On the other hand, the integration might be a threat too, because integration often leads to complexity of software. Complex software is more error prone and thus less reliable. This, however, could again be a strength of electronic mail because its modular design decreases complexity.

Figure 6.6 displays the SWOT analysis in a handy overview. It is obvious to see, that the opportunities outweigh. This is an indicator for a still increasing market.

[Figure 6 about here.]

Resulting strategies

The result of a SWOT analysis is a set of strategies that advice how to best react on the identified opportunities and threats, dependent on whether they are strengths or weaknesses of the subject. These strategies are what should be done to achieve the overall goal—here making email future-safe.

Threats of the market that are weaknesses of the subject should be avoided if possible, or one should prepare against them if they are impossible to avoid. As spam is unavoidable, email must prepare against it. The goal is to reduce spam to a bearable level. Spam fighting, with currently used protocols, is a war where the good guys must lose. Investing high effort will result in few gain. Hence enough spam protection should be provided, but not more. New concepts and protocols will change this fight; they must be in use before email has become unusable.

Threats that are strengths of the subject should be confronted. Here none were identified.

For opportunities of the market that are weaknesses of the subject, solutions should be searched. Large data transfers and infrastructures with nodes that move within the network, are of such kind. As a lot of potential is available, it should be used to develop solutions, to remove the weaknesses.

Finally, opportunities that are strengths of the subject. These are standardization, modularity, and extendability. They should be exploited to go even further, these are the key advantages of email.

2.2.2 Trends for electronic mail

Nothing remains the same, neither does the email technology. Emailing in future will probably differ from emailing today. This section tries to identify possible trends that affect the future of electronic mail.

Provider independence

Today's email structure is heavily dependent on email providers. This means, most people have email addresses from some provider. These can be providers that offer email accounts in addition to their regular services, for example online connections. AOL and *T-Online* for instance do so. Or specialized email providers that commonly offer free mail as well as enhanced mail services for which one has to pay. Examples for specialized email providers are GMX and *Yahoo*.

Outgoing mail is send either with the web mail client of the provider or by using an MUA which sends it to the provider for relay. Incoming mail is read with the web mail client or retrieved from the provider via POP3 or IMAP to the local computer to be read using the MUA. This means all mail sending and receiving work is done by the provider.

The reason therefore is originated in the time when people used dial-up connections to the Internet. A mail server needs to be online to receive email. Sending mail is no problem, but receiving it is hardly possible with an MTA which is few time online. Internet service providers had servers that were all day long connected to the Internet. So they offered email service, and they still do.

Nowadays, dial-up Internet access became rare; the majority of the users has broadband Internet access. As a flat rate is payed for it, the time being online does not affect costs anymore, even traffic is unlimited. Today it is possible to have an own mail server running at home. The remaining technical problem is the changing IP addresses one gets assigned every 24 hours[1]. But this is solvable with one of the dynamic DNS services; they provide the mapping of a fixed domain name to the changing IP addresses.

Home servers become popular for central data storage and multimedia services, these days. Being assembled of energy efficient hardware, power consumption is no big problem anymore. These home servers will replace video recorders and CD music collections in the near future. It is also realistic that they will manage heating systems and intercoms, too. Given the future leads to this direction, it will be a logical step to have email and other communication provided by the own home server as well.

After years in which MTAs have not been popular for users, the next years might bring the MTAs back to the users. Maybe in a few years nearly everyone will have one, or many, running at home.

Pushing versus polling

The retrieval of email is a field that is also about to change these days. The old way is to fetch email by polling the server that holds the personal mailbox. This polling is normally done in regular intervals, often once every five to thirty minutes. The mail transfer from the mailbox to the MUA is initiated from the user side. The disadvantage herewith is the delay between the arrival of mail on the server and the time when the user finally has the message on his screen.

To remove this disadvantage, *push email* [**?**] was invented. Here the server is not polled every few minutes about new mail, but the server pushes new mail directly to the client on arrival. The transfer is initiated by the server. This concept became popular with smart phones; they were able to do emailing but the traffic caused by polling the server was expensive.

The concept works well with mobile phones where the provider knows about the client, but it does not seem to be a choice for computers, since the provider needs to have some kind of login to push data to the user's computer. Push email, however, could swap over to computers when using a home server and no external provider. A possible scenario is a home server which receives mail from the Internet and pushing it to own workstations and smart phones. The configuration could be done by the user by using some simple interface, like one configures his telephone system to have different telephone numbers ringing on specified phones.

[1] At least this is the situation in Germany.

Another problem is when multiple clients share one mailbox. This is only solvable by working directly in the server's mailbox, which causes lots of traffic, or by storing at least information about read messages and the like there.

New email concepts

Changing requirements for email communication lead to the need for new concepts and new protocols that cover these requirements. One of these concepts to redesign the email system is named *Internet Mail 2000* [?]. It was proposed by Daniel J. Bernstein, the creator of *qmail*. Similar approaches were independently introduced by others, too.

As main change, the sender has the responsibility for mail storage; only a notification about a mail message gets sent to the recipient. The recipient can then fetch the message then from the sender's server. This is in contrast to the SMTP mail architecture where mail and the responsibility for it is transferred from the sender to the receiver. (See page 2 for the *store-and-forward* principle.)

MTAs are still important in this new email architecture, but in a slightly different way. They do not transfer mail itself anymore, but they transport the notifications about new mail to the destinations. This is a quite similar job as in the SMTP model. The real transfer of the mail, however, can be done in an arbitrary way, for example via FTP or SCP.

A second concept, this one primary to arm against spam, is David A. Wheeler's *Guarded Email* [?]. It requires messages to be recognized as Ham (non-spam) to be accepted, otherwise a challenge-response authentication will be initiated.

Hashcash by Adam Back—a third concept—tries to limit spam and denial of service attacks [?]. It requests payment for email. The costs are computing time for the generation of hash values. Thus sending spam becomes expensive. Further information about *Hashcash* can be found on [?].

New concepts, like the ones presented here, are invented to remove problems of the email technology. *Internet Mail 2000*, for instance, removes the spam problem and the problem of large message transfers.

2.2.3 Important properties in future

Easy configuration Provider independence through running an own mail server at home asks for easy configuration of the MTA. Providers have specialists to configure the systems, but ordinary people do not. Solutions are either having some home service system for computer configuration established with specialists coming to ones home to set up the systems; like it is already common for problems with the power and water supply systems. Or configuration needs to be easy and fool-proof, so it can be done by the owner himself. The latter solution depends on standardized parts that fit together seamlessly. The technology must not be a problem itself. Only settings that are custom to the users environment should be left open for him to set. This of course needs to be doable using a simple configuration interface like a web interface. Non-technical educated users should be able to configure the system.

Complex configuration itself is not a problem if simplification wrappers provide an easy interface. The approach of wrappers to make it look easier to the outside is a good concept in general. It still lets the specialist do complex and detailed configuration while also a simple configuration interface to novices is offered. *sendmail* took this approach with the *m4* macros [**?**]. Further more is this approach well suited to provide various wrappers with different user interfaces (e.g. graphical programs, websites, command line programs; all of them either in a questionnaire style or interactive).

Performance When MTAs become popular on home servers and maybe even on workstations and smart phones, then performance will be less important. Providers need MTAs that process large amounts of mail in short time. There is no need for home servers and workstations to handle that much mail; they need to process far less email messages per time unit. Thus performance will probably not be a main requirement for an MTA in future, given they mainly run on private machines.

Flexibility New mailing concepts and architectures like push email or *Internet Mail 2000* will, if they succeed, require MTAs to adopt the new technology. MTAs that are not able to change are going to be sorted out by evolution. Thus it is important *not* to focus too much on one use case, but to stay flexible. Allman saw the flexibility of *sendmail* one reason for its huge success (see section 3.2.2).

Security Another important requirement for all kinds of software is security. There is a constant trend coming from completely non-secured software, in the 70s and 80s, over growing security awareness, in the 90s, to security being a primary goal, now. This leads to the conclusion that software security will be even more important within the next years. As more clients get connected to the Internet and especially more computers are listening for incoming connections (like an MTA in a home server), there are more possibilities to break into systems. Securing of software systems will require increasing effort in future.

Out-of-the-box usage *Plug-and-play*-able hardware with preconfigured software can be expected to become popular. Like someone buys a set-top box to watch Pay-TV today, he might be buying a mail server box in a few years. He plugs the power cable in, inserts his email address in a web interface, and selects the clients (computers or smart phones) to which mail should be send and from which mail is accepted for relay. That’s all. It would just work then, like everyone expects it from a set-top box today. Secure and robust software is a precondition for such boxes to make this vision possible.

In summary: Easy configuration, as well as the somehow opposed flexibility, will be important for future MTAs. Also will it be security, but not performance. MTAs might become more commodity software, like web servers already are today, with the purpose to include it in many systems and the need of minimal configuration.

2.3 Summary

It seems as if electronic mail or a similar technology has good chances to survive the next decades.

Asynchronous communication It is assumed that it always will be important to send information messages. Those can be notes from people or notifications from systems. No other current available communication technology is as suitable for this kind of information transfer, as email, SMS, voice mail, or any other asynchronous communication technology. Synchronous communication, in contrast, is focused on dialog and typically interrupts people. The here needed kind of messages should not interrupt people, unless urgent, and they do not need two-way information exchange. Although synchronous communication could be used for transferring messages, it is not the best choice. The best choice is an asynchronous technology. Thus at least one asynchronous communication technology is likely to survive.

Email and Unified Messaging Whether email will be the surviving one, is not possible to know by now. It currently seems likely that Unified Messaging will be the future for asynchronous communication. But Unified Messaging is more a concept than a technology itself. This concept will base upon one or many underlying transport technologies, like email, SMS, and the like. Its goal is to integrate the transport technologies in order to hide them from the user's view. Currently, email is the most used asynchronous electronic communication technology. It is matured, flexible, and extendable, as well as standardized. These advantages make email a good base transport technology for Unified Messaging. Anyhow, whether email will be the basis for Unified Messaging or not, MTAs are a software which is needed for all asynchronous communication methods: programs that transfer messages from senders to recipients.

Unified Communication Unified Communication, as next step after Unified Messaging, is about the integration of synchronous and asynchronous communication channels. It seems *im*possible to merge the two worlds on basis of email in an evolutionary way. As only a revolutionary change of the whole email concept would make that merge possible, it is best to ignore it. New designed technologies are usually superior to heavily patched and bent old technologies, anyway. A general merge of synchronous and asynchronous communication has good chances to be fatal for email.

Until Unified Communication will become reality—if ever—electronic mail has a good position, also as basis for Unified Messaging.

SWOT analysis Not only the market influences email's future safety, but also must the email technology itself evolve to satisfy upcoming needs. Actions to take were discovered by using the SWOT analysis. These are: Prepare against spam. Search solutions for large data transfers and increasing growth and ramification of networks. Exploit standardization, modularity, and extendability.

Trends Also needed is awareness for new trends like: Provider independence, new delivery concepts, and completely new emailing concepts that introduce new protocols. Easy configuration, as well as the somehow opposed flexibility, will be important, but not performance. Security will be essential.

What kinds of MTAs will be needed in future? Probably ones running on home servers and workstations. This is what *masqmail* was designed for. The dial-up Internet connections, which are central to *masqmail*'s design, become rare, but mobile clients that move between different networks do need similar concepts, too. This makes *masqmail* still be a good MTA for such usage. Additionally, *masqmail* is small and it is much easier to configure for setups that are common to workstations and home servers, than other MTAs.

MTAs might become more commodity software, like web servers already are today, with the purpose to be included in many systems with only minimal configuration.

masqmail is a valuable program for various situations. Some setups became rare, but others are expected to become popular in the next years. It is expected that *masqmail*'s niche will rather grow than shrink.

Chapter 3

Mail transfer agents

After having analyzed the market for email and having identified upcoming trends, in the last chapter; this chapter takes a look at MTAs—the intelligent nodes and thus the most important parts of the email infrastructure. The MTAs will be grouped by similarities first. Then the four most popular Free Software MTAs will be presented to the reader in a short overview and with the most important facts. The end of this chapter is a short comparison of these programs.

3.1 Types of MTAs

"Mail transfer agent" is a term that covers a variety of programs. One thing is common to them: They transfer email from a sender to one or many recipients.

This is how Bryan Costales defines an MTA:

> A mail transfer agent (MTA) is a highly specialized program that delivers mail and transports it between machines, like the post office. **[?]**

The Free Dictionary is a bit more concrete on the term:

> Message Transfer Agent - (MTA, Mail Transfer Agent): Any program responsible for delivering e-mail messages. Upon receiving a message from a Mail User Agent or another MTA, [...] it [...] delivers it to any local addressees and/or forwards it to other remote MTAs (routing) for delivery to remote recipients. [?]

Dent and Hafiz agree [**?**, page 19] [**?**, pages 3-5].

Common to all MTAs is the transport of mail; this is the actual job. Besides this similarity, MTAs can be very different. Some of them have POP3 and/or IMAP servers included. Some can fetch mails through these protocols. Others have all features one can think of. And maybe there are some that do nothing else but transporting email.

Following is a classification of MTAs into groups of similar programs, regarding what is viewable from the outside.

Relay-only MTAs

Also called *forwarders*. This is the most simple kind of an MTA. It transfers mail only to defined *smart hosts*[1]. Relay-only MTAs do not receive mail from outside the system and they do not deliver locally. All they do is transfer mail to a specified smart host for further relay.

Most MTAs can be configured to act as such a *forwarder*. But this is usually an additional functionality.

One uses this kind of MTA to give a system the possibility to send mail without the need to do a lot of configuration. In a local network, usually the clients are set up with relay-only MTAs, while there is one mail server that acts as a smart host. The "dumb" clients send mail to this mail server which does all further work.

Example programs in that group are: *nullmailer*, *ssmtp*, and *esmtp*.

Groupware

Normally the term "groupware" does not mean one single program, but a suite of programs. They build a framework which is then populated with various modules that provide the actual functionality. Modules for mail transfer, file storage, calendars, resource management, Instant Messaging, and more, are commonly available.

These program suites are used if the main work to do is providing integrated communication facilities and team working support for a group of people. Mail transfer is only one part of the problem to solve. The most common scenario are companies. They use *groupware* to provide adequate services for their teams to work efficiently. But one may use *groupware* on the home server for the family members, too.

Examples for groupware are: *Lotus Notes*, *Microsoft Exchange*, and *OpenGroupware.org*.

"Real" MTAs

There is a third type of MTAs in between the minimalistic *relay-only* MTAs and the feature loaded *groupware*. Those programs may be named "real MTAs", or "proper MTAs", though there is no common name. They are what is meant with the term "mail transfer agent"—programs that transfer mail between hosts.

[1] *smart host*s are mail servers that receive email and route it to the actual destination.

Common to them is their focus on the email transfer, while they are able to act as smart hosts. Their variety ranges from ones mostly restricted to mail transfer (e.g. *qmail*) to others having interfaces for adding further mail processing modules (e.g. *postfix*). This group covers everything in between the other two groups.

Real MTAs include *sendmail*, *exim*, *qmail*, and *postfix*.

Other segmenting

MTAs can also be split in other ways.

Due to *sendmail*'s significance in the early times of email, compatibility interfaces to *sendmail* are important for Unix MTAs. The reason is that many mail applications simply assume the *sendmail* MTA to be installed on the system. Being not *sendmail-compatible* may not matter for some fields of action, but makes the program ineligible for serving as a general purpose MTA on Unix systems. Hence being sendmail-compatible is a major property of an MTA. MTAs without *sendmail-compatible* interfaces, or at least compatibility add-ons, will not be covered here. One example for such a program is *Apache James*.

Another separation can be done between Free Software MTAs and proprietary ones. Many of the MTAs for Unix systems are Free Software. Only these are regarded throughout this thesis, because comparing Free Software with proprietary or commercial software is not what typical users of programs like *masqmail* do. Comparison with non-free programs may be a point for large Free Software projects that try to step into the business world. Small projects, mostly used by individuals at home, need to be compared against other projects of similar shape. The document is seen from *masqmail*'s point of view—an MTA for Unix systems on home servers and workstations—so non-free software is out of the way.

masqmail's position

Now, where does *masqmail* fit in? It is not groupware nor a simple forwarder, thus it belongs to the "real MTAs". Additionally, it is Free Software and is sendmail-compatible to a large degree. This makes it similar to *sendmail*, *exim*, *qmail*, and *postfix*. *masqmail* is intended to be a replacement for those MTAs.

But: It was not designed to be used as a general replacement for them. (See: section 1.2.1) In fact, *masqmail* is only a replacement *in some situations*. This primary excludes working in an untrusted environment.

3.2 Popular MTAs

This section introduces a selection of popular MTAs; they are the most likely substitutes for *masqmail*. All are sendmail-compatible "smart" Free Software MTAs that focus on mail transfer, as is *masqmail*.

The programs chosen to be compared are: *sendmail*, *exim*, *qmail*, and *postfix*. They are the most important representatives of the regarded group.

3.2.1 Market share analysis

MTA statistics are rare, differ, and good data is hard to collect. These points are bad if good statistics are wanted. Thus it is obvious there are only few available.

Table 6.1 shows the most used MTAs determined by three different statistics. The first was done by Daniel J. Bernstein (the author of *qmail*) in 2001 [**?**]. The second is by Simpson and Bekman in 2007 and was published on *O'ReillyNet* [**?**]. And the third is from *MailRadar.com* with unknown date[2] [**?**].

[Table 1 about here.]

All surveys show high market shares for the four MTAs: *sendmail*, *exim*, *qmail*, and *postfix*. Only the *Microsoft* mail server software and *IMail* have comparable large shares. Other Free Software MTAs (*smail*, *zmailer*, MMDF, *courier-mta*) are less important and seldom used.

The three surveys base on different data. Bernstein took 1 000 000 randomly chosen IP addresses, containing 39 206 valid hosts; 958 of them accepted SMTP connections. The *O'ReillyNet* survey used only domains owned by companies; in total 400 000 hosts. *MailRadar* scanned 2 818 895 servers, leading to 59 209 accepted connections.

All surveys show *sendmail* to be the most popular MTA. *postfix*, *qmail*, and *exim* are among the top six in each. *exim* has slightly smaller shares than the other two. The four programs together share more than half of the market according to Bernstein and the *MailRadar* statistics. *O'ReillyNet* has their share to be somewhere between a third and the half. This uncertainty comes from the large amount of unidentifiable MTAs.

The 22 percent of *mail security layers* in the *O'ReillyNet* survey is remarkable. Mail security layers are software guards between the network and the MTA that filter unwanted mail before it reaches the MTA. This increases security by filtering malicious content and by blocking attacks against the MTA. The large share here may be a result of only regarding business mail servers. The problem concerning the survey is the disguise of the MTAs that run behind the security layer. It seems wrong to assume equal shares for the MTAs behind the guards as for the unguarded MTAs, because mail security layers will be more often used to guard weak MTAs, as strong ones do not need them so much. This needs to be kept in mind when looking at the *O'ReillyNet* survey.

The date of the *Mailradar* statistics is not known; a mail to *Mailradar* with a request for information has not been replied, unfortunately. However, it seems quite sure that the statistics were published after 2001, caused by the

[2]The footer of the website shows "Copyright 2007" but more likely does this refer to the whole website.

sendmail and *postfix* shares. But to decide whether before or after the one from *O'ReillyNet* would be just guessing. Possibly it receives constant input and thus displays a current state.

3.2.2 The four major Free Software MTAs

Now follows a small introduction to the four programs chosen for comparison. *masqmail* is not presented here as it was already introduced in chapter 1. Longer introductions, including analysis and comparison, were written by Jonathan de Boyne Pollard [?].

sendmail

sendmail is the best known MTA, since it was one of the first and surely the one that made MTAs popular. It also was shipped as default MTAs by many Unix system vendors [?].

The program was written by Eric Allman as the successor of his program *delivermail*. Allman was not the only one who was working on the program. Other people developed own versions of it and a variety of flavors came up, especially in the late eighties when Allman was inactive [?, page 5].

sendmail is designed to transfer mails between different protocols and networks, this lead to a very flexible, though complex, configuration.

The program was first released with BSD 4.1c in 1983. The latest version is 8.14.3 from May 2008. The program is distributed under the *Sendmail License* as both, free and proprietary software.

Further development will go into the project *MeTA1* which succeeds *sendmail*. The former name of this new project was *sendmail X*.

More information can be found on the *sendmail* homepage [?] and in the, so called, *Bat Book* [?].

exim

exim was started in 1995 by Philip Hazel at the *University of Cambridge*. It is a fork of *smail-3*, and inherited the monolithic architecture which is similar to *sendmail*'s. But having no architecture-given separation of the individual components of the system did not hurt. Its security is quite good [?].

exim is highly configurable, especially in the field of mail policies. This makes it easy to specify how mail is routed through the system and who is allowed to send email to whom. Interfaces to integrate spam and malware checkers are provided by design, too.

The program is Free Software, released under the GPL. The latest stable version is 4.69 from December 2007.

One finds *exim* on its homepage [?]. The standard literature is Hazel's *exim* book [?].

qmail

qmail is seen by its community as "a modern SMTP server which makes sendmail obsolete" [?]. It was written by DANIEL J. BERNSTEIN, starting in 1995. His primary goal was to create a secure MTA to replace the popular, but vulnerable, *sendmail*. His own words are: "This is why I started writing qmail: I was sick of the security holes in sendmail and other MTAs." [?].

qmail first introduced many innovative concepts in MTA design. The most obvious contrast to *sendmail* and *exim* is its modular design. But *qmail* was not the first modular MTA. MMDF, which predates even *sendmail*, was modular, too. Regardless of MMDF's modular architecture, *qmail* is generally seen as the first security-aware MTA [?].

The latest release of *qmail* is version 1.03 from July 1998. Afterwards, in November 2007, *qmail*'s source was put into the *public domain*. This made it Free Software.

Because of BERNSTEIN's inactivity, though the requirements changed since 1998, "[a] motley krewe of qmail contributors (see the README) has put together a netqmail-1.06 distribution of qmail. It is derived from Daniel Bernstein's qmail-1.03 plus bug fixes, a few feature enhancements, and some documentation." [?].

qmail's homepages are [?] and [?]. The best book about *qmail*, from BERNSTEIN's view, is DAVE SILL's handbook [?]. His free available guide "Life with qmail" is another valuable source [?].

postfix

The *postfix* project started in 1999 at IBM *research*, then called *VMailer* or IBM *Secure Mailer*. WIETSE VENEMA's program "attempts to be fast, easy to administer, and secure. The outside has a definite Sendmail-ish flavor, but the inside is completely different." [?]. In fact, *postfix* was mainly designed after qmail's architecture to gain security. But in contrast to *qmail* it aims much more on being fast and full-featured.

Today *postfix* is taken by many Unix systems and GNU/Linux distributions as default MTA.

The latest stable version is numbered 2.5.6 from December 2008. *postfix* is covered by the IBM *Public License 1.0* which is a Free Software license.

Additional information can be retrieved from the program's homepage [?]. DENT's *postfix* book [?] claims to be "the definitive guide", and it is.

3.3 Comparison of MTAs

This section does not try to provide a throughout MTA comparison, because this is already done by others. Remarkable comparisons are the one by DAN

Shearer [?] and a discussion on the mailing list *plug@lists.q-linux.com* [?]. Tabular overviews may be found at [?], [?], and [?, section 1.9].

Here provided is an overview on important properties of the four previously introduced MTAs. The data comes from the above stated sources and is collected in table 6.2[3].

[Table 2 about here.]

Architecture

Architecture is most important when comparing MTAs. Many other properties of a program depend on its architecture. Munawar Hafiz discusses in detail on MTA architecture, comparing *sendmail*, *qmail*, *postfix*, and *sendmail X* [?]. Jonathan de Boyne Pollard's MTA review [?] is a source, too.

Two different architecture types show off: monolithic and modular MTAs.

Monolithic MTAs are *sendmail*, *smail*, *exim*, and *masqmail*. They all consist of one single *setuid root*[4] binary which does all the work.

Modular MTAs are MMDF, *qmail*, *postfix*, and *MeTA1*. They consist of several programs, each doing only a part of the overall job. The different programs run with the least permissions they need, *setuid root* can be avoided completely.

The architecture does not directly define the program's security, but "[t]he goal of making a software secure can be better achieved by making the design simple and easier to understand and verify" [?, chapter 6]. *exim*, though being monolithic, has a fairly clean security record. But it is very hard to keep the security up as the program growth. Wietse Venema (the author of *postfix*) says, it was the architecture that enabled *postfix* to grow without running into security problems [?, page 13].

The modular design, with each sub-program doing one part of the overall job, conforms to the *Unix Philosophy*. The Unix Philosophy [?] demands "small is beautiful" and "make each program do one thing well". Monolithic MTAs fail here.

Today modular MTA architectures are the state-of-the-art.

Spam checking and content processing

Spam and malware increased during the last years. Today it is important for an MTA to be able to provide checking for bad mail. This can be done by implementing functionality into the MTA or by invoking external programs to do this job.

[3] The lines of code were measured with David A. Wheeler's *sloccount* [?].

[4] *setuid* lets a program run with the rights of its owner, here root. This is considered to be a security risk. Thus it it should be avoided if possible.

sendmail invented *milter*[5], which is used to interface external programs of various kind. *postfix* adopted the *milter* interface but is also able to easily include scanning modules into its modular structure. *qmail* is pretty old and did not evolve with the changing market situation. Anyhow, its modular structure enables external scanners to be included into *qmail*. *exim* has the advantage that it was designed with the goal to provide extensive scanning facilities; it is therefore very good suited to scan itself or invoke external scanners.

Future trends

In chapter 2, it was tried to figure out trends and future requirements for MTAs. The four programs are compared on these possible future requirements now.

Provider independence The first trend was provider independence, which requires easy configuration. *postfix* seems to do best here. It uses primary two configuration files (`master.cf` and `main.cf`) which are easy to manage. *sendmail* appears to have a bad position. Its configuration file `sendmail.cf` is cryptic and very complex (it has legendary Turing-completeness) thus it needs simplification wrappers around it to provide easier configuration. They exist in form of the *m4* macros that generate the `sendmail.cf` file. Unfortunately, adjusting the generated result by hand appears to be necessary for non-trivial configurations. *qmail*'s configuration files are simple but the whole system is complex to set up; it requires various system users and *qmail* is hardly usable without applying several patches that add functionality which is required nowadays. *netqmail* is the community's effort to help in the latter point. *exim* has only one single configuration file (`exim.conf`) which suffers most from its flexibility—like in *sendmail*'s case. Flexibility and easy configuration are almost always contrary goals.

Performance As second trend was the decreasing necessity for high performance identified. This goes along with the move of MTAs from service providers to home servers. *postfix* focuses much on performance, this might not be an important point in the future. Of course there will still be the need for high performance MTAs, but a growing share of the market will not require high performance. Energy and space efficiency is related to performance; it is a similar goal in a different direction. But optimization, be it for performance or other efficiencies, is often in contrast to simplicity and clarity; these two improve security. Optimizing does in most times decrease the simplicity and clarity. Simple MTAs that do not aim for high performance are what is needed in future. The simple design of *qmail*[6] is a good example.

[5] "milter" is a common abbreviation for "sendmail mail filter API".
[6] *qmail* is still fast

Security The third trend (even more security awareness) is addressed by each of the four programs. It seems as if all widely used MTAs provide good security nowadays. Even *sendmail* can be configured to be secure today. However, the modular architecture, used by *qmail* and *postfix*, is generally seen to be conceptually more secure. *sendmail*'s creators have started *MeTA1*, a modular MTA that merges the best of *qmail* and *postfix*, to replace the old *sendmail*. It will be interesting to watch *exim*'s future—will it become modular, too?

3.4 Summary

This chapter first took an overview over the field of MTAs. Three major types of MTAs were identified: Relay-only MTAs (also called forwarders), groupware, and the "real MTAs". *masqmail* belongs to the last group, it is additionally sendmail-compatible and Free Software.

Next a look at the market shares of MTAs was taken; It showed that four MTAs of *masqmail*'s group have high importance: *sendmail*, *postfix*, *qmail*, and *exim*. Their combined share is between one third and the half of the market. The other part splits into proprietary MTAs, unknown software behind mail security layers, and a reminder of really small market shares.

Each one of the four major Free Software MTAs was presented afterwards and finally these programs were compared on some selected properties.

Now, the reader should have a general knowledge about those four important MTAs. Further chapters will refer frequently to them.

Chapter 4

masqmail's present and future

This chapter identifies requirements for *masqmail*. They are compared against the current code to see what is already fulfilled and what is missing. Then the outstanding work is ordered by relevance and are presented in a list of pending work tasks. The end of this chapter is the evaluation of the best development strategy to get the work done in order to achieve the requirements.

4.1 The goal

Before requirements can be identified and further development can be discussed, it is important to clearly specify the goal to achieve. This means: What shall *masqmail* be like in, for instance, five years?

Should *masqmail* become more specific to a more narrow niche or rather become more general and move a bit out of its niche? Or should it even become a totally general MTA like *sendmail*, *exim*, *qmail*, and *postfix*?

Becoming completely general seems to be no choice because the competitors are too many and they are already too strong. It would require a strong base of developers and superior features to establish. There also seems to be no need for another general purpose MTA additional to those four programs. Thus the effort would most likely remain a try. Venema stated: "It is becoming less and less likely that someone will write another full-featured Postfix or Sendmail MTA *from scratch* (100 kloc)." [**?**]. At least *masqmail* is not going to try that.

masqmail was intended to be a small "real" MTA which covers the niche of managing the relay over several smart hosts. Small and resource friendly software is still important for workstations, home servers, and especially for embedded computers. Other software that focuses on the same niche is not known. Dial-up connections have become rare but mobile computers that move between different networks are popular. So, the niche is still present.

What has changed in general is the security that is needed for software. Graff and van Wyk describe the situation well: "[I]n today's world, your software is likely to have to operate in a very hostile security environment." [**?**, page 33]. Additionally they say: "By definition, mail software processes information from potentially untrusted sources. Therefore, mail software must be written with great care, even when it runs with user privileges and even when it does not talk directly to a network." [**?**, page 90]. As *masqmail* is mail software and trusted environments become rare, it is best for *masqmail* to become a secure MTA.

In summary, the goal for *masqmail* is to stay in the current niche with respect to modern usage scenarios and to become a secure MTA.

4.2 Requirements

This section identifies the requirements for *masqmail* to reach the above defined goal. Most of the requirements will apply to modern MTAs in general.

4.2.1 Functional requirements

Functional requirements are about the function of the software. They define what the program can do and in what way. The requirements are named "RF" for "requirement, functional".

RF 1: Incoming and outgoing channels *sendmail*-compatible MTAs must support at least two incoming channels: mail submitted using the `sendmail` command, and mail received on a TCP port. Thus it is common to split the incoming channels into local and remote. This is done by *qmail* and *postfix*. The same way is Hafiz's view [**?**].

SMTP is the primary mail transport protocol today, but with the increasing need for new protocols (see section 2.2.3) in mind, support for more than just SMTP is good to have. New protocols will show up; maybe multiple protocols need to be supported then. This would lead to multiple remote channels, one for each supported protocol as it was done in other MTAs. Best would be interfaces to add further protocols as modules.

Outgoing mail is commonly either sent using SMTP, piped into local commands (for example `uucp`), or delivered locally by appending to a mailbox. Outgoing channels are similar for *qmail*, *postfix*, and *sendmail X*: All of them have a module to send mail using SMTP, and one for writing into a local mailbox.

Local mail delivery is a job that uses root privilege to be able to switch to any user in order to write to his mailbox. It is possible to deliver without being root privilege, but delivery to user's home folders is not generally possible then. Thus even the modular MTAs *qmail* and *postfix* use root privilege for this job. As mail delivery to local users is *not* included in the basic job of an MTA and introduces a lot of new complexity, why should the MTA bother? In order to keep the system simple, reduce privilege, and to have programs that do one job well, the local delivery job should be handed over to a specialist: the MDA. MDAs know about the various mailbox formats and are aware of the problems of concurrent write access and the like. Hence passing the message, and the responsibility for it, over to an MDA seems to be best.

This means an outgoing connection that pipes mail into local commands is required. To other outgoing channels applies what was already said about incoming channels.

[Figure 7 about here.]

An overview on incoming and outgoing channels which are required for an MTA, gives figure 6.7. The reader may want to compare this diagram with *masqmail*'s incoming and outgoing channels, which are depicted in figure 6.3 on page 77.

RF 2: Mail queuing Mail queuing removes the need to deliver instantly as a message is received. The queue provides fail-safe storage of mails until they are delivered. Mail queues are probably used in all MTAs, even in some simple forwarders. The mail queue is essential for *masqmail*, as *masqmail* is intended for non-permanent online connections. This means, mail must be queued until a online connection is available to send the message. This may be after a reboot. Hence the mail queue must provide persistence.

The mail queue and the module(s) to manage it are the central part of the whole system. This demands especially for robustness and reliability, as a failure here can lead to mail loss. An MTA takes over responsibility for mail by accepting it, hence losing mail messages is absolutely to avoid. This covers any kind of crash situation, too. The worst thing acceptable to happen is an already sent mail to be sent again.

RF 3: Header sanitizing Mail coming into the system often lacks important header lines. At least the required ones must be added by the MTA. One example is the `Date:` header, another is the, not required but recommended, `Message-ID:` header. Apart from adding missing headers, rewriting headers is important, too. Changing the locally known domain part of email addresses to globally known ones is an example. *masqmail* needs to be able to rewrite the domain part dependent on the route used to send the message, to prevent messages to get classified as spam.

Generating the envelope is a related job. The envelope specifies the actual recipient of the mail, no matter what the `To:`, `Cc:`, and `Bcc:` headers contain. Multiple recipients lead to multiple different envelopes, all containing the same mail message.

RF 4: Aliasing Email addresses can have aliases, thus they need to be expanded. Aliases can be of different kind: another local user, a remote user, a list of local and remote users, or a command. Most important are the aliases in the `aliases` file, usually located at `/etc/aliases`. Addresses expanding to lists of users lead to more envelopes. Aliases changing the recipient's domain part may require a different route to be used.

RF 5: Route management One key feature of *masqmail* is its ability to send mail out over different routes. The online state defines the active route to be used. A specific route may not be suited for all messages, thus these messages are hold back until a suiting route is active. For more information on this concept see section 1.2.2.

RF 6: Authentication One thing to avoid is being an *open relay*. Open relays allow to relay mail from everywhere to everywhere. This is a source of spam. The solution is restricting relay[1] access. It may also be wanted to refuse all connections to the MTA except ones from a specific set of hosts.

Several ways to restrict access are available. The most simple one is restriction by the IP address. No extra complexity is added this way but the IP addresses need to be static or within known ranges. This approach is often used to allow relaying for local nets. The access check can be done by the MTA or by a guard (e.g. TCP *Wrapper* [**?**]) before. The main advantage here is the minimal setup and maintenance work needed. This kind of access restriction is important to be implemented.

This authentication based on IP addresses is impossible in situations where hosts with changing IP addresses, that are not part of a known sub net, need access. Then a authentication mechanism based on some *secret* is required. Three common approaches exist:

1. SMTP-after-POP: Uses authentication on the POP protocol to permit incoming SMTP connections for a limited time afterwards. The variant SMTP-after-IMAP exists, too.
2. SMTP authentication: An extension to SMTP. It allows to request authentication before mail is accepted. Here no helper protocols are needed.
3. Certificates: The identity of a user or a host is confirmed by certificates that are signed by trusted authorities. Certificates are closely related to encryption, they do normally satisfy both needs: encrypt the data transmission and identify the remote user/host.

Static authentication is the preferred type for authenticating clients. It should be chosen if possible. This means if the MTA resides within a trusted network

[1] Relaying is passing mail, that is not from and not for the own system, through it.

or it is possible to define trusted network segments on basis of IP addresses, then static authentication is the best choice.

If the MTA does its job in an untrusted network, if it can be expected that forged IP addresses will appear, or if mobile clients need access, then dynamic authentication should be used.

Any combination is possible, too. For example, it is preferred to allow relay access only to authenticated users. Either clients in local networks which are authenticated by their IP addresses or remote clients that authenticate by a secret-based method.

Static authentication is simpler and requires less administration work but it has limitations. Dynamic authentication should be used if static authentication reaches its limits. At least one of the secret-based mechanisms should be supported.

RF 7: Encryption Electronic mail is vulnerable to sniffing attacks, because in generic SMTP all data transfer is unencrypted. The message's body, the header, and the envelope are all unencrypted. But also some authentication dialogs transfer plain text passwords (e.g. PLAIN and LOGIN). Hence encryption is throughout important.

The common way to encrypt SMTP dialogs is using *Transport Layer Security* (short: TLS, the successor of SSL). TLS encrypts the datagrams of the *transport layer*. This means it works below the application protocols and can be used with any of them [**?**].

Using secure tunnels that are provided by external programs should be preferred over including encryption into the application, because the application needs not to bother with encryption then. Outgoing SMTP connections can get encrypted using a secure tunnel, created by an external application (like *openssl*). But incoming connections can not use external secure tunnels, because the remote IP address is hidden then; all connections would appear to come from the local host instead. Figure 6.8 depicts the situation of using an application like *stunnel* for incoming connections. The connection to port 25 comes from from local host and exactly this information is available to the MTA. Authentication based on IP addresses and many spam prevention methods are useless then.

[Figure 8 about here.]

To provide encrypted incoming channels, the MTA could implement encryption and listen on a port that is dedicated to encrypted SMTP (SMTPS). This approach would be possible, but it is deprecated in favor for STARTTLS. RFC 3207 "SMTP Service Extension for Secure SMTP over Transport Layer Security" shows this by not mentioning SMTPS on port 465. Also port 465 is not even reserved for SMTPS anymore [**?**].

STARTTLS—defined in RFC 2487—is what RFC 3207 recommends to use for secure SMTP. The connection then goes over port 25, but gets encrypted when the STARTTLS keyword is issued. Email depends on compatibility—only encryption methods that client and server support can be used. Hence it is best to act after the recommendations of the RFC documents. This means STARTTLS encryption should be supported for incoming and for outgoing connections.

RF 8: Spam handling Spam is a major threat nowadays, but it is a war that is hard to win. The goal is to provide state-of-the-art spam protection, but not more. (See section 2.2.1.)

As spam is, by increasing the amount of mail messages, not just a nuisance for end users but also for the infrastructure—the MTAs—they need to protect themselves.

Filtering spam can be done by either refusing it during the SMTP dialog or by checking for spam after the mail was accepted and queued. Both ways have advantages and disadvantages, so modern MTAs use them in combination.

Spam is usually identified by the results of a set of checks. Static rules, database querying (e.g. DNS blacklists [**?**] [**?**]), requesting special client behavior (e.g. *greylisting* [**?**], *hashcash* [**?**]), or statistical analysis (e.g. *bayesian filters* [**?**]) are checks that may be used. Running more checks leads to better results, but takes more system resources and more time.

Doing some basic checks during the SMTP dialog seems to be a must [**?**, page 25]. Including these checks into the MTA makes them fast to avoid SMTP dialog timeouts. For modularity and reusability reasons internal interfaces to specialized modules seem to be best. RAYMOND says: "Modularity (simple parts, clean interfaces) is a way to organize programs to make them simpler." [**?**, chapter 1].

More detailed checks after the message is queued should be done by external scanners. Interfaces to invoke them need to be defined. (See also the remarks about *amavis* in the next section.)

RF 9: Malware handling Related to spam is malicious content (short: *malware*) like viruses, worms, and trojan horses. They, in contrast to spam, do not affect the MTA itself, as they are in the mail's body. MTAs that search for malware are equal to post offices that open letters to check if they contain something that could harm the recipient. This is not a mail transport job. But by many people the MTA which is responsible for the recipient is seen to be at a good position to do this work, thus it is often done there. Though, it is nice to have interfaces to such scanners within the MTA.

In any way should malware checking be performed by external programs that may be invoked by the MTA. However, MDAs are better points to invoke content scanners.

A popular email filter framework is *amavis* which integrates various spam and malware scanners. The common setup includes a receiving MTA which sends mail to *amavis* using SMTP, *amavis* processes the mail and sends it then to a second MTA that does the outgoing transfer. (This setup with two MTA instances is discussed in more detail in section 5.1.3.)

RF 10: Archiving Mail archiving and auditability become more important as email establishes as technology for serious business communication. Archiving is a must for companies in many countries. In the United States, the *Sarbanes-Oxley Act* [**?**] covers this topic.

It is a goal to have the ability to archive verbatim copies of every mail coming into and every mail going out of the system, with relation between them.

postfix for example has a *always_bcc* feature, to send a copy of every outgoing mail to a definable recipient. At least this functionality should be given, although a more complete approach, like *qmail* provides, is preferable. *qmail* is able to save copies of all sent and received messages and additionally complete SMTP dialogs [**?**, page 12].

But if archiving is of high importance, a dedicated archiving solution is advisable, anyway.

4.2.2 Non-functional requirements

Now follows a list of non-functional requirements for *masqmail*. These requirements specify the quality properties of a software. The list is based on Hafiz [**?**, page 2], with inspiration from Spinellis [**?**, page 6] and Kan [**?**]. These non-functional requirements are named "RG" for "requirement, general".

RG 1: Security MTAs are critical points for computer security as they are accessible from external networks. They must be secured with high effort. Properties like the need for high privilege level, from outside influenced work load, work on unsafe data, and demand for reliability, increase the need for security. This is best done by modularization, also called *compartmentalization*, as described in section 4.2.3.

masqmail needs to be secure enough for its target field of operation. *masqmail* is targeted to workstations and private networks, with explicit warning to not use it on permanent online hosts [**?**]. But as non-permanent online connections and trustable environments become rare, *masqmail*'s security should be so good that it is usable with permanent online connections and in unsafe environments. For example should mails with bad content not be able to break *masqmail*.

RG 2: Reliability Reliability is the second essential quality property for an MTA. Mail for which the MTA took responsibility must never get lost while it is within the MTA's responsibility. The MTA must not be *the cause* of any mail loss, no matter what happens. Unreliable MTAs are of no value. However, as the mail transport infrastructure is a distributed system, one of the communication partners or the transport medium may crash at any time during mail transfer. Thus reliability is needed for mail transfer communication, too.

The goal is to transfer exactly one copy of the message. Tanenbaum evaluates the situation and comes to the conclusion that "in general, there is no way to arrange this." [?, pages 377–379]. Only strategies where no mail gets lost are acceptable; he identifies three of them, but one generates more duplicates than the others, so two strategies remain. (1) The client always reissues the transfer. The server first sends an acknowledgment and then handles the transfer. (2) The client reissues the transfer only if no acknowledgment was received. The server first handles the transfer and sends the acknowledgment afterwards. The first strategy does not need acknowledgments at all, however, it will lose mail if the second transfer fails, too.

Hence, mail transfer between two processes should use the strategy: The client reissues if it receives no acknowledgment. The server first handles the message and then sends the acknowledgment. This strategy only leads to duplicates if a crash happens in the time between the message is fully transferred to the server and the acknowledgment is received by the client. No mail will get lost.

RG 3: Robustness Being robust means handling errors properly. Small errors may get corrected, large errors may kill a process. Killed processes should get restarted automatically and lead to a clean state again. Log messages should be written in every case. Robust software does not need a special environment, it creates a friendly environment itself. RAYMOND's *Rule of Robustness* and his *Rule of Repair* are good descriptions [?, pages 18–21].

RG 4: Extendability *masqmail*'s architecture needs to be extendable to allow new features to be added afterwards. The reasons for this need are the changing requirements. New requirements will appear, like more efficient mail transfer of large messages or a final solution to the spam problem. Extendability is the ability of software to include new function with little work.

RG 5: Maintainability Maintaining software takes much time and effort. SPINELLIS guesses "40 % to 70 % of the effort that goes into a software system is expended after the system is written first time." [?, page 1]. This work is called *maintaining*. Hence making software good to maintain will ease all further work.

RG 6: Testability Good testability make maintenance easier too, because functionality is directly verifiable when changes are done, thus removing the uncertainty. Modularized software makes testing easier, because parts can be tested without external influences. SPINELLIS sees testability as a sub-quality of maintainability [?].

RG 7: Performance Also called "efficiency". Efficient software requires few time and few resources. The merge of communication hardware and its move from service providers to homes and to mobile devices demand smaller and more resource-friendly software. The amount of mail will be lower even if much more mail will be sent, thus time performance is less important. *masqmail* is not a program to be used on large servers, but on small devices. Thus more important for *masqmail* will be energy and heat saving, maybe also system resources.

As performance improvements are in contrast to many other quality properties (reliability, maintainability, usability, capability [**?**, page 5]), jeopardizing these to gain some more performance should not be done. Kernighan and Pike state clear: "[T]he first principle of optimization is *don't*." [**?**, page 165]. Simplicity and clearness are of higher value.

RG 8: Availability Availability is important for server programs. They must stay operational by blocking *denial of service* attacks and the like. Automated restarts into a clean state after fatal errors are also required.

RG 9: Portability Source code that compiles and runs on various operation systems is called portable. Portability can be achieved by using standard features of the programming language and common libraries. Basic rules to achieve portable code are defined by Kernighan and Pike [**?**]. Portable code lets software spread faster. Portability among the various flavors of Unix systems is a goal for *masqmail*, because these systems are the ones MTAs usually run on. No special care needs to be taken for non-Unix platforms.

RG 10: Usability Usability, not mentioned by Hafiz [**?**] (he focuses on architecture) but by Spinellis [**?**] and Kan [**?**], is a property which is very important from the user's point of view. Software with bad usability is rarely used, no matter how good it is. If substitutes with better usability exist, the user will switch to one of them. Here, usability includes setting up and configuring; the term "users" includes administrators. Having MTAs on home servers and workstations requires easy and standardized configuration. The common setups should be configurable with little action by the user. Complex configuration should be possible, but the focus should be on the most common form of configuration: choosing one of several common setups.

4.2.3 Architecture

masqmail's current architecture is monolithic like *sendmail*'s and *exim*'s. But more than the other two is it one block of interweaved code. *exim* has a highly structured code with many internal interfaces, a good example is the interface for authentication "modules". *sendmail* provides now, with its *milter* interface, standardized connection channels to external modules. *masqmail* has none of them—it is what *sendmail* was in the beginning: a single large block.

Figure 6.9 is a call graph generated from *masqmail*'s source code. It gives an impression of how interweaved the internals are. There are no compartments at all.

[Figure 9 about here.]

sendmail improved its old architecture by adding the milter interface, to include further functionality by invoking external programs. *exim* was designed, and is carefully maintained, with a modular-like code structure in mind. *qmail* started from scratch with a "security-first" approach, *postfix* improved on it, and *sendmail X/MeTA1* tries to adopt the best of *qmail* and *postfix* to completely replace the old *sendmail* architecture. Hafiz describes this evolution of MTA architecture very well [**?**].

Every one of these programs is more modular, or became more modular over time, than *masqmail* is. Modern requirements like spam protection and probable future requirements like the use of new mail transport protocols demand for modular designs in order to keep the software simple. Simplicity is a key property for security. "[T]he essence of security engineering is to build systems that are as simple as possible." [**?**, page 45].

Hafiz agrees: "The goal of making software secure can be better achieved by making the design simple and easier to understand and verify." [**?**, page 64]. He identifies the security of *qmail* to come from it's *compartmentalization*, which goes hand in hand with modularity:

> A perfect example is the contrast between the feature envy early *sendmail* architecture implemented as one process and the simple, modular architecture of *qmail*. The security of *qmail* comes from its compartmentalized simple processes that perform one task only and are therefore testable for security. [**?**, page 64]

Equal does Dent see the situation for *postfix*: "The modular architecture of Postfix forms the basis for much of its security." [**?**, page 7].

Modularity is also needed to satisfy modern MTA requirements in providing a clear interface to add functionality without increasing the overall complexity much.

Modularity is no direct requirement but a goal that has positive influence on important requirements like security, testability, extendability, maintainability, and not least simplicity. These quality properties then, on their part, make it easier to achieve the functional requirements.

Hence, aspiration for modularity, by compartmentalization, improves the overall quality and function of the software. It can be seen as an architectural requirement for a secure and modern MTA.

4.3 Fulfilled requirements

Here follows a description of how far the requirements are already fulfilled by *masqmail*.

RF 1: In/out channels The incoming and outgoing channels that *masqmail* already has (depicted in figure 6.3 on page 77) are the ones required for an MTAs at the moment. Currently, support for other protocols seems not to be necessary, although new protocols and mailing concepts are likely to appear (see section 2.2.2). As other protocols are not required today, *masqmail* is regarded to fulfill RF 1. Without any support in *masqmail* for adding further protocols, the best strategy is to delaying such work until the functionality is essential, anyway.

RF 2: Queuing One single mail queue is used in *masqmail*. It satisfies all current requirements.

RF 3: Header sanitizing The envelope and mail headers are generated when the mail is put into the queue. The requirements are fulfilled.

RF 4: Aliasing Alias expansion is done on delivery. All common kinds of aliases in the global aliases file are supported. So called *.forward* aliasing is not supported, but this is less common and seldom used.

RF 5: Route management Querying the name of the active route is done on delivery. Headers can get rewritten a second time then. This part does provide all the functionality required.

RF 6: Authentication Static authentication, based on IP addresses, can be achieved with VENEMA's TCP *Wrapper* [**?**], by editing the `hosts.allow` and `hosts.deny` files. This is only relevant to authenticate hosts that try to submit mail into the system. Dynamic (secret-based) SMTP authentication is already supported in form of SMTP-AUTH and SMTP-after-POP, but only for outgoing connections. For incoming connections only address-based authentication is supported.

RF 7: Encryption Similar is the situation for encryption which is also only available for outgoing channels; here a tunnel application, like *openssl*, is needed. A secure tunnel can be created to send mail trough. State-of-the-art, however, is using STARTTLS, but this is not supported. For incoming channels, no encryption is available. The only possible setup to provide encryption of incoming channels is using an application like *stunnel* to crypt between the secure connection to the remote host and the plain connection to the MTA. Unfortunately, this suffers from the problem explained on page 82 in figure 6.8. Anyway, it would still be no STARTTLS support.

RF 8: Spam handling *masqmail* does not provide special support for spam filtering. Spam prevention by not accepting spam during the SMTP dialog is not possible at all. Spam filtering is only possible by using two *masqmail* instances with an external spam filter in between. The mail flow is from the receiving MTA instance, which accepts mail, to the filter application that processes and possible modifies it, to the second MTA which is responsible for further delivery of the mail. This is a concept that works in general, and it is good to separate different work with clear interfaces. But the need of two instances of the same MTA, with doubled setup, makes it rather a work-around. Better is to have this data flow respected in the MTA design, like it was done in *postfix*. Anyway, the more important part of spam handling, for sure, is done during the SMTP dialog by completely refusing unwanted mail.

RF 9: Malware handling For malware handling applies nearly the same as for spam handling, except that all checks are done after mail is accepted. The possible setup is the same with the two MTA instances and the filter in between. *masqmail* does support such a setup, but not in a nice way.

RF 10: Archiving There is currently no way for archiving every message that does through *masqmail*.

RG 1: Security *masqmail*'s current security is bad. However, it seems acceptable for using *masqmail* on workstations and private networks, if the environment is trustable and *masqmail* is protected against remote attacks. In environments where untrusted components or persons have access to *masqmail*, its security is too low. Its author states that *masqmail* "is not designed to" such usage [**?**]. This is a clear indicator for being careful. Issues like high memory consumption, low performance, and denial of service attacks—things not regarded by design—may cause serious problems. In any way, a security report that confirms *masqmail*'s security level is missing.

masqmail uses conditional compilation to exclude unneeded functionality from the executable at compile time. Excluding code means excluding all bugs and weaknesses within this code, too. Excluding unused code is a good concept to improve security.

RG 2: Reliability Its reliability is also not good enough. Situations where only one part of a sent message was removed from the queue and the other part remained as garbage, showed off [**?**]. Problems with large mail messages in conjunction with small bandwidth were also reported [**?**]. Fortunately, lost email was no big problem yet, but KURTH warns:

> There may still be serious bugs in [masqmail], so mail might get lost. But in the nearly two years of its existence so far there was only one time a bug which caused mail retrieved via pop3 to be lost in rare circumstances. [**?**]

In summary: Current reliability needs to be improved. Implementing a state machine can help here.

RG 3: Robustness The logging behavior of *masqmail* is good, although it does not cover the whole code. For example, if the queue directory is world writeable by accident (or as action of an intruder), any user can remove messages from the queue or replace them with own ones. *masqmail* does not even write a debug message in this case. The origin of this problem, however, is *masqmail*'s trust in its environment.

RG 4: Extendability *masqmail*'s extendability is very poor. This is a general problem of monolithic software, but can though be provided with high effort. *exim* is an example for good extendability in a monolithic program.

RG 5: Maintainability The maintainability of *masqmail* is equivalent to other software of similar kind. Missing modularity and therefore more complexity makes the maintainer's work harder. Conditional compilation might be good for security, but *ifdef*s scattered throughout the source code is a pain for maintenance. In summary is *masqmail*'s maintainability bearable, like in average Free Software projects.

RG 6: Testability The testability suffers from missing modularity, too. Testing program parts is hard to do. Nevertheless, it is done by compiling parts of the source to two special test programs: One tests reading input from a socket, the other tests constructing messages and sending it directly. Neither is designed for automated testing of source parts, they are rather to help the programmer during development.

Two additional scripts exist to send a set of mails to different kinds of recipients. They can be used for automated testing, but both check only the function of the whole system, not its parts.

RG 7: Performance The performance—efficiency—of *masqmail* is good enough for its target field of operation, where this is a minor goal.

RG 8: Availability This applies equal to availability. Hence no further work needs to be done her.

RG 9: Portability The code's portability is good with view on Unix-like operation systems. At least *Debian*, *Red Hat*, SUSE, *Slackware*, *Free*BSD, *Open*BSD, and *Net*BSD are reported to be able to compile and run *masqmail* [**?**]. Special requirements for the underlying file system are not known. Thus, the portability is already good.

RG 10: Usability The usability is very good, from the administrator's point of view. *masqmail* was developed to suite a specific, limited job—its configuration does perfect match. The user's view does not reach to the MTA, as it is hidden behind the MUA. Configuration could be eased even more by providing configuration generators that enable *masqmail* to be used right "out of the box" after running one of several configuration scripts for common setups. This would improve *masqmail*'s usability for not technical educated people.

4.4 Work to do

After the requirements for modern MTAs were identified in section 4.2 and *masqmail*'s features were compared against them in section 4.3, here the pending work is identified. Table 6.3 lists all requirements with importance and the work that is needed to achieve them. The column "Focus" shows the attention a work task should get. The focus depends on the task's importance and the amount of work it includes.

[Table 3 about here.]

The importance is ranked from '--' (not important) to '++' (very important). The pending work is ranked from '--' (nothing) to '++' (very much). Large work tasks with high importance need to receive much attention, they need to be in focus. In contrast should small, low importance work tasks receive few attention. Here the focus for a task is calculated by summing up the importance and the pending work with equal weight. Normally, tasks with high focus are the ones of high priority and should be done first.

The functional requirements that receive highest attention are RF 6 (authentication), RF 7 (encryption), and RF 8 (spam handling). Of the non-functional requirements, RG 1 (security), RG 2 (reliability), and RG 4 (extendability), rank highest.

These tasks are presented in more detail in a to-do list, now. The list is sorted by focus and then by importance.

TODO 1: Encryption (RF 7)

Encryption is chosen for number one as it is essential to provide privacy. Using STARTTLS for encryption is definitely needed and should be added first; encrypted data transfer is hardly possible without support for it.

TODO 2: Authentication (RF 6)

Authentication of incoming SMTP connections is also highly needed and should be added second. It is important to restrict access and to prevent relaying. For workstations and local networks, this has only medium importance and address-based authentication is sufficient in most times. But secret-based authentication is mandatory to receive mail from the Internet. Additionally it is a guard against spam.

TODO 3: Security (RG 1)

masqmail's security is bad, thus the program is forced into a limited field of operation. This field of operation even shrinks as security becomes more important and networking and interaction increases. Secure and trusted environment become rare, thus improving security is an important thing to do. The focus should be on adding compartments to split *masqmail* into separate modules. (See section 4.2.3.) Furthermore, *masqmail*'s security should be tested throughout to get a definitive view how good it really is and where the weak spots are.

TODO 4: Reliability (RG 2)

Reliability is also to improve. It is a key quality property for an MTA, and not good enough in *masqmail*. Reliability is strong related to the queue, thus improvements there are favorable. Applying ideas of *crash-only software* [**?**] will be a good step. CANDEA and FOX see in killing the process the best way to stop a running program. Doing so inevitably demands for good reliability of the queue, and the start up process inevitably demands for good recovery. Those critical situations for reliability are nothing special anymore, they are common. Hence they are regularly tested and will definitely work.

TODO 5: Spam handling (RF 8)

As authentication can be a guard against spam, filter facilities have lower priority. But basic spam filtering and interfaces for external tools should be implemented in future. Configuration guides for a setup of two *masqmail* instances with a spam scanner in between should be written. And at least a basic kind of spam prevention during the SMTP dialog should be implemented.

TODO 6: Extendability (RG 4)

masqmail lacks an interface to plug in modules with additional functionality. There exists no add-on or module system. The code is only separated by function into various source files. Some functional parts can be included or excluded by conditional compilation. But the *ifdef*s are scattered through all the code. This situation needs to be improved by collecting related function into single places that interact through clear interfaces with other parts. Also should these interfaces allow efficient adding of further functionality.

4.5 Ways for further development

Knowing what needs to be done is only one part, the other is deciding *how* to do it by focusing on a global development strategy.

4.5.1 Possibilities

Further development of software can always go three different ways:

1. Improve the current code base. (S 1)
2. Add wrappers or interposition filters. (S 2)
3. Redesign the software from scratch and rebuild it. (S 3)

The first two strategies base on the available source code and can be applied in combination. The third strategy splits from the old code base and starts over again. Wrappers and interposition filters would be outright included into a new architecture; they are a subset of a new design. Of course, parts of existing code can be used in a new design if appropriate.

The requirements are now regarded, each on its own, and are linked to the development strategy that is preferred to reach each specific requirement. If some requirement is well achievable by using different strategies then it is linked to all of them. Implementing encryption (TODO 1) and authentication (TODO 2), for example, are limited to a narrow region in the code. Such features are addable to the current code base without much problem. In contrast can quality properties like reliability (TODO 4), extendability (TODO 6), and maintainability hardly be added to code afterwards—if at all. Security (TODO 3) is improvable in a new design, of course, but also with wrappers or interposition filters.

This linking of requirements to the strategies is shown in table 6.4. The requirements are ordered by their focus.

[Table 4 about here.]

Next, the best strategy for further development needs to be discovered. Therefore a score for each strategy is obtained by summing up the focus points of each requirement for which a strategy is preferred. Only positive focus points are regarded, with each plus symbol counting one. Requirements with negative focus are not regarded because they are already or nearly reached; the view here is on outstanding work.

Strategy 1 (Improve current code) has a score of 9 points. Strategy 2 (Wrappers and interposition filters) has a score of 7 points. Strategy 3 (A new design) scores on top with 17 points. S 1 and S 2 can be used in combination; the combined score is 13 points. Thus strategy 3 ranges first, followed by the combination of strategy 1 and 2.

This leads to the conclusion that S 3 (A new design) is probably the best strategy for further development.

But this result respects only the view on requirements and their relevance. Other factors like development effort and risks are important to think about, too. These issues are discussed in the following sections, comparing S 3 against the combination S 1+2.

4.5.2 Discussion

Quality improvements

Most quality properties can hardly be added afterwards. Hence, if reliability, extendability, or maintainability shall be improved, a redesign of *masqmail* is the best way to take. The wish to improve quality, inevitably point towards a modular architecture. Modularity with internal and external interfaces is highly preferred from the architectural point of view (see section 4.2.3). The need for further features, especially ones that require changes in *masqmail*'s structure, support the decision for a new design, too. Hence a rewrite is favored if *masqmail* should become a modern MTA with good quality properties.

Security

Similar is the situation for security. Security comes from good design, explain Graff and van Wyk:

> Good design is the sword and shield of the security-conscious developer. Sound design defends your application from subversion or misuse, protecting your network and the information on it from internal and external attacks alike. It also provides a safe foundation for future extensions and maintenance of the software. [**?**, page 55]

They also suggest to add wrappers and interposition filters *around* applications, but more as repair techniques if it is not possible to design security *into* a software the first way [**?**, pages 71–72].

Hafiz adds: "The major idea is that security cannot be retrofitted *into* an architecture." [**?**, page 64, (emphasis added)].

Effort estimation

Although a strategy might lead to the best result, one may choose another one if the required effort is too high. The effort for a redesign and rebuild is estimated now.

Wheeler's program *sloccount* calculates following estimations for *masqmail*'s code base as of version 0.2.21 (excluding library code):

```
Total Physical Source Lines of Code (SLOC)                = 9,041
Development Effort Estimate, Person-Years (Person-Months) = 2.02 (24.22)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.70 (8.39)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 2.89
Total Estimated Cost to Develop                           = $ 272,690
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
```

The development costs in money are not relevant for a Free Software project with volunteer developers, but the development time is. About 24 man-months are estimated. The current code base was written almost completely by Oliver Kurth within four years in his spare time. This means he needed around twice as much time. Of course, he programmed as a volunteer developer not as an employee with eight work-hours per day.

Given the assumptions that (1) an equal amount of code needs to be produced for a new designed *masqmail*, (2) a third of the existing code can be reused plus concepts and knowledge, and (3) development speed is like Kurth's, then it would take between two and three years for one programmer to produce a redesigned new *masqmail* with the same features that *masqmail* now has. Less time would be needed if a simpler architecture allows faster development, better testing, and less bugs. Of course, more developers would speed it up, too.

Risks

The gained result of a new design might still outweigh the development effort. But risks are something more to consider.

A redesign and rewrite of software from scratch is hard. It takes time to design a new architecture, which then must prove that it is as good as expected. As well is much time and work needed to implement the design, test it, fix bugs, and so on. If flaws in the design appear during prototype implementation, it is necessary to start again.

Such a redesign can fail at many points and it is long time unclear if the result is really better than the code that already exists. Even if the new code is working like expected, it is still not matured.

One thing is clear: Doing a redesign and rebuild *is* a risky decision.

Existing code is precious

If a new design needs much effort and additionally is a risk, what about the existing code base then?

Adding new functionality to an existing code base seems to be a secure and cheap strategy. The existing code is known to work and features can often be added in small increments. Risks like wasted effort if a new design fails are hardly existent, and the faults in the current design are already made and most probably fixed.

Functionality that is hard to add incrementally into the application, like support for new protocols, may be addable to the outside. *masqmail* can be secured to a huge amount by guarding it with wrappers that block attackers [**?**, page 71]. Spam and malware scanners can be included by running two instances of *masqmail*. All those methods base on the current code which they can indirectly improve.

The required effort is probably under one third of a new design and work directly shows results. These are strong arguments against a new design.

Repairing

Besides these advantages of existing code, one must not forget that further work on it is often repair work. Small bug fixes are not the problem, but adding something for which the software originally was not designed, will cause problems. Such work often destroys the clear concepts of the software, especially in interweaved monolithic code.

Doug McIlroy, a person with important influence on Unix especially by inventing the Unix pipe, demands: "To do a new job, build afresh rather than complicate old programs by adding new features." [**?**].

Repair strategies are useful, but only in the short-time view and in times of trouble. If the future is bright, however, one does best by investing into a software. As shown in section 2.3, the future for MTAs is bright. This means it is time to invest into a redesign with the intention to build up a more modern product.

In the author's view is *masqmail* already needing this redesign since about 2003 when the old design was still quite suitable ... it already delayed too long.

Anyway, further development on base of current code needs to improve the quality properties, too. Some quality requirements can be satisfied by adding wrappers or interposition filters to the outside. For those is the development effort approximately equal to a solution with a new design. But for adding quality requirements like extendability or maintainability which affect the source code throughout, the effort does increase with exponential rate as development proceeds. In case these properties get not improved, development will likely come to a dead end sooner or later.

A guard against dead ends

A new design does protect against such dead ends.

Changing requirements are one possible dead end if the software does not evolve with them. A famous example is *sendmail*; it had an almost monopoly for a long time. But when security became important, *sendmail* was only repaired instead of the problem sources—its insecure design—would have been removed. Thus security problems reappeared and over the years *sendmail*'s market share shrank as more secure MTAs became available. *sendmail*'s reaction to the new requirements, in form of *sendmail X* and *MeTA1*, came much to late—the users already switched to other MTAs.

Redesigning a software as requirements change helps keeping it alive.

The knowledge of Heraclitus, a Greek philosopher, shall be an inspiration: "Nothing endures but change."

Another danger is the dead end of complexity which is likely to appear by constant work on the same code base. It is even more likely if the code base has a monolithic architecture. A good example for simplicity is *qmail* which consists of small independent modules, each with only about one thousand lines of code. Such simple code makes it obvious to understand what it does. The *suckless* project [**?**] for example advertises such a philosophy of small and simple software by following the thoughts of the Unix inventors [**?**] [**?**]. Simple, small, and clear code avoids complexity and is thus also a strong prerequisite for security.

Modularity

The avoidance of dead ends is essential for further development on current code, too. Hence it is mandatory to refactor the existing code base sooner or later. Most important is the intention to modularize it, as modularity improves many quality properties, eases further development, and essentially improves security.

One example how modular structure makes it easy to add further functionality is described by SILL: He says that integrating the *amavis* filter framework into the *qmail* system can be done by simply renaming the `qmail-queue` module to `qmail-queue-real` and then renaming the `amavis` executable to `qmail-queue` [**?**, section 12.7.1]. Nothing more in the *qmail* system needs to be changed. This is a very admirable ability which is only possible in a modular system that consists of independent executables.

This thesis showed several times that modularity is a key property for good software design. Modularity can hardly be retrofitted into software, hence development on base of current code will need a throughout restructuring too, to modularize the source code. Thus a new design is similar to such a throughout refactoring, except the dependence on current code.

Function versus quality

Remarkable is the distribution of functional and non-functional requirements to the strategies. The strategies for current code (S 1+2) have a functional to non-functional ratio of 10 to 3. The new design strategy (S 3) has a ratio of 5 to 12.

This classifies current code to be better suited for adding functionality, and a new design to be better suited for quality improvements. Both strategies need to improve function as well as quality, however, the focus of the strategy is determined by this difference.

Easier work is likely to be done earlier in Free Software projects than hard work. Thus, by choosing S 1+2 volunteer developers tend to implement function first and delay quality improvements, no matter what the suggested order of the work tasks is. S 3, in contrast, would benefit early quality improvements and later function improvements. This is real-life experience from Free Software development.

Break Even

It is important to keep the time dimension in mind. This includes the separation into a short-time and a long-time view. The short-time view shall cover between two and four years, here. The long-time view is the following time.

In the short-time view, the effort for improving the existing code is much smaller than the effort for a new design plus improvements. But to have similar quality properties at the end of the short-time frame, a version that is based on current code will probably require nearly as much effort as a new designed version will take. For all further development afterwards, the new design will scale well while the old code will require exponential more work.

Break Even is the point in time when a new design is better than improvements of the old code for the first time. From this point on, the new design will be the better solution.

In the long-time view, a restructuring for modularity is necessary anyway to keep the maintaining effort bearable. The question is, when the restructuring should be done: Right at the start in a new design, or later as restructuring work.

The problem with "good enough"

The decision for later restructuring is problematic. Functionality is often more wanted than quality, thus more function is preferred over better quality, as quality is still "good enough". But it might be still "good enough" the next time, and the time after that one, and so on.

Quality improvement is no popular work, but it is required to avoid dead ends. As more code increases the work that needs to be done for quality and modularity improvements, it is better to do these improvements early. Afterwards, all further development will profit from it.

If some design is bad, it should get replaced by a sane solution.

Doug McIlroy gives valuable advice for these situations: "Don't hesitate to throw away the clumsy parts and rebuild them." [**?**].

Though, making such a cut is hard, especially if the bad design is still … "good enough".

Good software, good feelings

One last argument shall be added. This one is more common to Free Software but can also be found in non-free software.

Free Software "sells" if it has a good user base. For example: Although *qmail* is somehow outdated and its author has not released any new version since about ten years, *qmail* still has a very strong user base and community.

Good concepts, sound design, and a sane philosophy gives users good feelings for the software and faith in it. They become interested in using it and to contribute. In contrast do constant repair work and reappearance of weaknesses leave a bad feeling.

The motivation of most volunteer developers is their wish to do good work with the goal to create good software. Projects that follow admirable plans towards a good product will motivate volunteers to help. More helpers can get the 2,5 man-years for a new design in less absolute time done. Additionally is a good developers base the best start for a good user base, and users define a software's value.

4.6 Result

This chapter identified the requirements for a modern and secure *masqmail*, and the outstanding work to achieve them. Their importance and the required work for them lead to a focus ranking, which resulted in an ordered list of pending work tasks. Afterwards possible development strategies to control the work process were compared and discussed.

Strategy 3 (A new design) is slightly preferred over the combination of strategy 1 (Improve existing code) and 2 (Add wrappers and interposition filters), from the requirement's point of view.

The discussion afterwards did generally support the new design strategy. But some arguments stood against it. These were:

1. The development time and effort
2. The time delay until new features can be added
3. The risk of failure

The first two arguments are only relevant for the short-time view, because both will become *support arguments* for the new design, once the Break Even point is reached.

The third argument, the risk, remains. There are risks in every investment. Taking no risks means remaining the same, which eventually means, drifting towards a dead end in a world that does change.

With respect to the current situation, the suggested further development plan for *masqmail* is split into a short-time plan and a long-time plan:

1. The short-time plan: Add the most needed features, namely encryption, authentication, and security wrappers, to the current code base.
2. The long-time plan: Design a new architecture that satisfies the modern requirements, especially the quality requirements.

The background thought for this development plan is to first do the most needed stuff on the existing code to keep it usable. This satisfies the urgent needs and removes the time pressure from the development of the new design. After this is done, a new designed *masqmail* should be developed from scratch. This is the work for the future. It shall, after it is usable and throughout tested, supersede the old *masqmail*.

The basics of this development idea can be described as: Recurrent development of a new design from scratch, while the old version is still in use and gets repaired.

Hence a modern design will inherit an old one in periodic intervals. This is a very future-proof concept that combines the best of short-term and long-term planning. The price to pay is only the increased work, which gets covered by volunteers that *want* to do it.

Chapter 5

Improvement plans

The last chapter came to the result that further development is best done in a double-strategy: First the existing code base should be improved to satisfy the most important needs in order to make it usable for some more time. Then *masqmail* should get redesigned from scratch and rebuilt to gain a secure and modern MTA architecture for the future.

This chapter finally describes approaches and techniques for the work on the current code base, and it introduces ideas and plans for a new, modern MTA design which will become the next generation of *masqmail*.

The first part of the chapter covers the short-time goals that base on the current code. The second part deals with the long-time goal—the redesign.

5.1 Based on current code

The three most important work tasks are implementable by improving the current code or by adding wrappers or interposition filters. The following sections describe solution approaches to do that work.

5.1.1 Encryption

Encryption (TODO 1) should be the first functionality to be added to the current code. The requirement was already discussed on page 37. As explained there, STARTTLS encryption—defined in RFC 2487—should be added to *masqmail*.

This work requires changes mainly in three source files: `smtp_in.c`, `smtp_out.c`, and `conf.c`.

The first file includes the functionality for the SMTP server. It needs to offer STARTTLS support to clients and needs to initiate the encryption when the client requests it. Additionally, the server should be able to insist on encryption before it accepts any message

The second file includes the functionality for the SMTP client. It should start the encryption by issuing the STARTTLS keyword if the server supports it. It should be possible to send messages over encrypted channels only.

The third file controls the configuration files. New configuration options need to be added. The encryption policy for incoming connections needs to be defined. Three choices seem necessary: no encryption, offer encryption, insist on encryption. The encryption policy for outgoing connections should be part of each route setup. The options are the same: never encrypt, encrypt if possible, insist on encryption.

Dependencies

STARTTLS uses TLS encryption which is based on certificates. Thus the MTA needs its own certificate. This should be generated during installation. A third party application like *openssl* should be taken for this job. The encryption itself should also be done using an available library. *openssl* or a substitute like *gnutls* does then become a dependency for *masqmail*. *gnutls* seems to be the better choice because the *openssl* license is incompatible to the GPL, under which *masqmail* and *gnutls* are covered.

User definable paths to *masqmail*'s secret key, *masqmail*'s certificate, and the public certificates of trusted *Certificate Authorities* (short: CAs) are also nice to have.

Existing code

Frederik Vermeulen wrote an encryption patch for *qmail* which adds STARTTLS support [**?**]. This patch includes about 500 lines of code.

Adding this code in a similar form to *masqmail* will be fairly easy. It will save a lot of work as it is not necessary to write the code completely from scratch.

5.1.2 Authentication

Authentication (TODO 2) is the second function to be added. It is important to restrict the access to *masqmail*, especially for mail relay. The requirements for authentication where identified on page 36.

Static access restriction, based on the IP address is already possible by using TCP *Wrapper*. This makes it easy to refuse all connections from outside the local network for example, which is a good prevention against being an open relay. More detailed static restrictions, like splitting between mail for users on the system and mail for relay, should *not* be added to the current code. This is a concern for the new design.

One of the dynamic methods

Of the three dynamic, secret based, authentication methods (SMTP-after-POP, SMTP authentication, and certificates) the first one drops out as it requires a POP server running on the same or a trusted host. POP servers are rare on workstations and home servers do also not regularly include them. Thus it is no option for *masqmail*.

Authentication based on certificates does suffer from the certificate infrastructure that is required. Although certificates are already used for encryption, its management overhead prevented wide spread usage for authentication.

SMTP authentication (also referred to as SMTP-AUTH) support is easiest attained by using a *Simple Authentication and Security Layer* (short: SASL) implementation. DENT sees in SASL the best solution for dynamic authentication of users:

> None of these [authentication methods] is an ideal solution. They require additional code compiled into your existing daemons that may then require special write access to system files. They also require additional work for busy system administrators. If you cannot use any of the nonauthenticating alternatives mentioned earlier, or your business requirements demand that all of your users' mail pass through your system no matter where they are on the Internet, SASL is probably the solution that offers the most reliable and scalable method to authenticate users. [**?**, page 44]

These days SMTP-AUTH—defined in RFC 2554—is supported by almost all email clients. If encryption is used then even insecure authentication methods like PLAIN and LOGIN become secure.

Simple Authentication and Security Layer

masqmail best uses an available SASL library. *Cyrus* SASL is used by *postfix* and *sendmail*. It is a complete framework that makes use of existing authentication concepts like the `passwd` file or PAM. As advantage it can be included in existing user data bases. *gsasl* is an alternative. It comes as a library which helps with the decision for a method and with generating the appropriate dialog data; the actual transmission of the data and the authentication against some database is left open to the programmer. *gsasl* is used, for instance, by *msmtp*. It seems best to give both concepts a try and decide then which one to use.

Currently, outgoing connections already feature SMTP-AUTH but only in a hand-coded way. It is to decide whether this should remains as it is or should get replaced by the SASL approach that will be used for incoming connections. The decision should be influenced by the estimated time until the new design is usable.

Authentication needs code changes in the same places as encryption. The relevant code files are `smtp_in.c`, `smtp_out.c`, and `conf.c`.

The server code, to authenticate clients, must be added to `smtp_in.c` and the configuration options to `conf.c`. Several configuration options should be provided: the authentication policy (no authentication, offer authentication, insist on authentication), the authentication backend (if several are supported), an option to refuse plain text methods (PLAIN and LOGIN), and one to require encryption before authentication.

If the authentication code for outgoing connects shall be changed too, it must be done in `smtp_out.c`. The configuration options are already present.

Authentication backend

For a small MTA like *masqmail*, it seems preferable to store the login data in a text file under *masqmail*'s control. This is the most simple choice for many usage scenarios. But using a central authentication facility has advantages in larger setups, too. *Cyrus* SASL supports both, so there is no problem. If *gsasl* is chosen, it seems best to start with an authentication file under *masqmail*'s control.

5.1.3 Security

Improvements to *masqmail*'s security (TODO 3) are an important requirement and are the third task to be worked on. Retrofitting security *into masqmail* is not or hardly possible as it was explained in section 4.5.2. But adding wrappers and interposition filters can be a large step towards security.

Mail security layers

At first mail security layers like *smap* come to mind. The market share analysis in section 3.2.1 identified such software. Mail security layers are interposition filters that are located between the untrusted network and the MTA. They accept mail in replacement for the MTA in order to separate the MTA from the untrusted network. Thus they are *proxies*.

The work *smap* does is described in [**?**]: *smap* accepts messages as proxy for the MTA and puts it into a queue. *smapd* a brother program runs as daemon and watches for new messages in this queue which it submits into the MTA then.

Because the MTA does not listen for connections from outside now, it is not directly vulnerable. Unfortunately, the MTA can not react on relaying and spam by itself anymore because it has no direct connection to the mail sender. This job needs to be covered by the proxy now. Similar is the situation for encryption and authentication. However, care must be taken that the proxy stays small and simple as its own security will suffer otherwise.

The advantage of mail security layers is that the MTA itself needs not to bother much with untrusted environments. The proxy cares for this.

smap is non-free software and thus no general choice for *masqmail*. A way to achieve a similar setup is to copy *masqmail* and strip one copy to the bare minimum of what is needed for the proxy job. *setuid* could be removed, and root privilege too if *inetd* is used. This hardens the proxy instance.

Mail from outside would then come through the proxy into the system. Mail from the local host and from the local network could be directly accepted by the normal *masqmail*, if those locations are considered trusted. But it seems better to have them use the proxy, too, or maybe a second proxy instance with different policy.

The here described setup comes close to the structure of the incoming channels in the new design which is described in section 5.2. This shows the capabilities of the here chosen approach.

A concrete setup

A stripped down proxy needs to be created. It should only be able to receive mail via SMTP, encrypt the communication, authenticate clients, and send mail out via SMTP to an internal socket (named "X" in the figure). This is a straight forward task. The normal *masqmail* instance runs on the system, too. It takes input from *stdin* (when the `sendmail` command is invoked) and via SMTP where it listens on an internal socket (named "X" in the figure). Outgoing mail is handled without difference to a regular setup. Figure 6.10 depicts the setup.

[Figure 10 about here.]

Spam and malware handling

The presented setup is the same as the one with two MTA instances and a scanner application in between, which was suggested to add spam and malware scanner afterwards to an MTA. This is a fortunate coincidence, because a scanner like *amavis* can simply be put in replace for the internal socket "X".

5.2 A new design

In chapter 4 the requirements for a modern and secure *masqmail* were identified. Now modules that implement the various jobs of an MTA are defined and plugged together to create a new *masqmail*. The architecture is inspired by existing MTAs and driven by the identified requirements.

One wise experience was kept in mind during the design: "Many times in life, getting off to the right start makes all the difference." [**?**, page 32].

5.2.1 Design decisions

This section describes and discusses architectural decision that were made for the new design. The functional requirements are only referenced, as they were already discussed in chapter 4.

A number of major design ideas lead the development of the new architecture:

1. Throughout compartmentalization.
2. Free the internal system from the in and out channels. Provide interfaces to add arbitrary protocol handlers afterwards.
3. Have a single point for scanning where all mail goes through.
4. Concentrate on the mail transfer job. Use specialized external programs for other jobs.
5. Keep it simple, clear, and general.

Incoming channels

The functional requirements for incoming channels were already discussed as RF 1 on page 34. Two required incoming channels were identified: the `sendmail` command for local mail submission and the SMTP daemon for remote connections.

A bit different is the structure of *sendmail X* at that point: Locally submitted messages go also to the SMTP daemon, which is the only connection to the mail queue. Finch proposes a similar approach [**?**]: He wants the `sendmail` command to be a simple SMTP client that contacts the SMTP daemon of the MTA, like it is done by connections from remote. The advantage here is to have one single module where all SMTP dialog with submitters is done. Hence one single point to accept or refuse incoming mail. Additionally does the module which puts mail into the queue not need to be *setuid* or *setgid*, because it is only invoked from the SMTP daemon. The MTA's architecture would become simpler and common tasks are not duplicated in modules that do similar jobs.

But merging the input channels in the SMTP daemon makes the MTA heavily dependent on SMTP. To *qmail* and *postfix* new protocol handlers may be added without change in other parts of the system. The SMTP modules can even get removed if it is not needed. It is better to have a larger number of independent modules if each one is simpler then. The need to implement SMTP clients in every module for internal communication makes them more complicated.

With the increasing need for new protocols in mind, it seems better to have single modules for each incoming channel, although this leads to duplicated acceptance checks. Independent checks in different modules, however, have the advantage to be able to simply apply different policies. Thus it is possible to run two SMTP modules that listen on different ports: one accessible from the Internet which requires authentication, the other one only accessible from the local network without authentication.

The approach of simple independent modules, one for each incoming channel, should be taken.

A module which is a POP or IMAP client to import contents of other mailboxes into the system may be added afterwards as it is desired.

Outgoing channels

Outgoing mail is commonly either sent using SMTP, piped into local commands (for example `uucp`), or delivered locally by appending to a mailbox. The requirements were identified on page 34.

Outgoing channels are similar for *qmail*, *postfix*, and *sendmail X*: All of them have a module to send mail using SMTP and one for writing into a local mailbox. Local mail delivery is a job that should have root privilege to be able to switch to any user in order to write to his mailbox. Modular MTAs do not require *setuid root* but the local delivery process (or its parent) should run as root. root privilege is not a mandatory requirement but any other approach has some disadvantages thus commonly root privilege is used.

Local mail delivery should not be done by the MTA, but by an MDA instead. This decision was discussed in section 4.2.1. This means only an outgoing channel that pipes mail into a local command is required for local delivery.

Other outgoing channels, one for each supported protocol, should be designed like it was done in other MTAs.

Mail queuing

The mail queue is the central part of an MTA. This fact demands especially for robustness and reliability as a failure here can lead to mail loss. (See RF 2 on page 35.)

Common MTAs feature one or more mail queues, they sometimes have effectively several queues within one physical representation.

MTA setups that include content scanning tend to require two separate queues. To use *sendmail* in such setups requires two independent instances with one own queue each. *exim* can handle it with special *router* and *transport* rules but the data flow gets complicated. Hence an idea is to use two queues (*incoming* and *active* in *postfix*'s terminology) and have the content scanning within the move from the one to the other.

sendmail, *exim*, *qmail*, and *masqmail* all use at least two files to store one message in the queue: one file contains the message body, another the envelope and header information. The one containing the mail body is not modified at all. *postfix* takes a different approach in storing queued messages in an internal format within one file. FINCH suggest yet another approach: The whole queue should be stored in one single file with pointers to separating positions [?].

All of the presented MTAs use the file system to hold the queue; none uses a database to hold it. A database could improve the reliability of the queue through better persistence. This might be a choice for larger MTAs but is none for *masqmail* which should be kept small and simple. A running database system does likely require much more resources than *masqmail* itself does. And as the queue's job is more storing data, than running data selection queries, a database does not gain enough to outweigh its costs.

Hence the choice here is having a directory with simple text files in it. This is straight forward, simple, clear, and general ... and thus a good basis for reliability. It is additionally always an advantage if data is stored in the operating system's natural form, which is plain text in the Unix' case.

Robustness of the queue is covered in the next section.

Mail sanitizing

Mail coming into the system may be malformed, lacking headers, or can be an attempt to exploit the system. Care must be taken.

In *postfix*, mail is sanitized by the *cleanup* module, which invokes *rewrite*. The position in the message flow is after the message comes from one of the several incoming channels and before the message is stored into the *incoming* queue. *cleanup* does a complete check to make the mail header complete and valid.

qmail has the principle of "don't parse" which propagates the avoidance of parsing as much as possible. The reason is that parsing is a highly complex task which likely makes code exploitable.

In *masqmail*'s new design, mail should be stored into the queue without parsing. A scanning module should then parse the message with high care. SPINELLIS proposes reliable approaches to do this work [?, pages 17–18]; using a *parser generator*[1] is the best solution here. The parsed data should then get modified if needed and written into a second queue. This approach has several advantages. First, the receiving parts of the system are independent from content, they simply store it into the queue. Second, one single module does the parsing and generates new messages that contain only valid data. Third, the sending parts of the system will thus only work on messages that consist of valid data. Of course, it must be ensured that each message passes through the *scanning* module, but this is already required for spam and malware scanning.

The mail body will never get modified, except for removing and adding transfer protocol specific requirements like dot stuffing or special line ending characters. These translations are only done in receiving and sending modules.

[1] STEPHEN C. JOHNSON's paper about *yacc* is a good introduction into *parser generators* [?].

Jon Postel's robustness principle[2] should be respected in the *scanning* module. The module should parse the given input in a liberal way and generate clean output. Raymond's *Rule of Repair*[3] can be applied, too. But it is important to repair only obvious problems, because repairing functionality is likely a target for attacks.

Aliasing

The functional requirements were identified under RF 4 on page 36. From the architectural point of view, the main question about aliasing is: Where should aliases get expanded?

Two facts are important to consider: (1) Addresses that expand to a list of users lead to more envelopes. (2) Aliases that change the recipient's domain part may make the message unsuitable for a specific online route.

Aliasing is often handled by expanding the alias and re-injecting the mail into the system. Unfortunately, the mail is processed twice then; additionally does the system have to handle more mail this way. If it is wanted to check the new recipient address for acceptance and do all processing again, then re-injecting it is the best choice. But already accepted messages may get rejected in the second go, though the replacement address was set inside the system. This seems not to be wanted.

Doing the alias expansion in the *scanning* module appears to be the best solution. Unfortunately, a second alias expansion must be made on delivery, because only then is clear which route is used for the message. This compromise should get accepted.

Route management

The online state is only important for the sending modules of the system, thus it should be queried in the *queue-out* module which selects ready messages from the *outgoing* queue and transfers them to the appropriate sending module. Route-based aliasing, which was described in the last section, should be done in the same go.

Archiving

The best point to archive copies of every incoming mail is the *queue-in* module, respectively the *queue-out* module for copies of outgoing mail. But the changes that are made by the receiving modules (adding further headers) and sending modules (address rewrites) are not respected with this approach.

[2]"Be liberal in what you accept, and conservative in what you send.". In this wording in RFC 1122 and in different wordings in numerous RFCs

[3]"Repair what you can – but when you must fail, fail noisily and as soon as possible." [**?**, page 18]

qmail has the ability to log complete SMTP dialogs. Logging the complete data transaction into and out of the system is a great feature which should be implemented into each receiving and sending module. Though, as this will produce a huge amount of output, it should be disabled by default.

Archiving's functional requirements were described as RF 10 on page 39.

Authentication and Encryption

The topics were discussed as RF 6 and RF 7 on several places throughout this thesis remarkable ones are on page 36 and 37.

Authentication should be done within the receiving and sending modules. To encryption applies the same as to authentication here. Only receiving and sending modules should come in contact with it.

In order to avoid code duplicates, the actual implementation of both functions should be provided by a central source, for example a library, which is used in the various modules.

Spam and malware handling

The two approaches for spam handling were already presented to the reader in section 4.2.1 as RF 8 and RF 9. Here they are described in more detail:

1. Refusing spam during the SMTP dialog: This is the way it was meant by the designers of the SMTP protocol. They thought checking the sender's and recipient's mail addresses would be enough, but as they are forgeable, it is not. More and more complex checks are needed to be done. Checking needs time, but SMTP dialogs time out if it takes too long. Thus during the SMTP dialog, only limited time can be used for checking if a message seems to be spam. The advantage of this approach is that bad messages can simply get refused—no responsibility for them is taken and no further system load is added. See RFC 2505 (especially section 1.5) for detail.

2. Checking for spam after the mail was accepted and queued: Here it is possible to invest more processing time, thus more detailed checks can be done. But, as responsibility for messages was taken, it is no choice to simply delete spam mail. Checks for spam do not lead to sure results, they just indicate the possibility the message is unwanted mail. Eisentraut lists actions to take after a message is recognized as probably spam [**?**, pages 18–20]. For mail the MTA is responsible for, the only acceptable action is adding further or rewriting existing header lines. Thus all further work on the spam messages is the same as for non-spam messages.

Modern MTAs use both techniques in combination. Checks during the SMTP dialog tend to be implemented in the MTA to make them fast; checks after the message was queued are often done using external programs (*spamassassin* is a well known one). EISENTRAUT sees the checks during the SMTP dialog to be essential: "Ganz ohne Analyse während der SMTP-Phase kommt sowieso kein MTA aus, und es ist eine Frage der Einschätzung, wie weit man diese Phase belasten möchte." [**?**, page 25, (translated: "No MTA can go without analysis during the SMTP phase anyway, but the amount of stress one likes to put on this phase is left to his discretion.")]

Checks before a message is accepted, like DNS blacklists and *greylisting*, need to be invoked from within the receiving modules. Like for authentication and encryption, the implementation of this functionality should be provided by a central source.

All checks on queued messages should be done by pushing the message through external scanners like *spamassassin*. The *scanning* module is the best place to handle this. Hence this module needs interfaces to external scanners.

Malware scanning is similar to spam scanning of queued messages. The *amavis* framework is a popular mail scanning framework that includes all kinds of malware and also spam scanners; it communicates by using SMTP.

Providing SMTP in and out channels from the *scanning* module to external scanner applications is thus a desired goal. Using further instances of the already available *smtp* and *smtpd* modules appears to be the best solution.

The scanning module

A problem, which was probably noticed by the attentive reader, is the lot of work that was put onto the *scanning* module. This is not what is desired. Thus splitting this module into a set of single modules might be necessary.

The decision how to split shall not be discussed here. It is left up to the time of prototyping, because trying different approaches helps with the decision in such situations.

5.2.2 The resulting architecture

The result is a symmetric design, featuring the following modules:

1. Any number of receiver modules that handle incoming connections.
2. A module that stores the received mail into a first queue.
3. A central scanning module that takes mail from the first queue, processes it in various ways, and puts it afterwards into a second queue.
4. A module that takes mail out of the second queue and passes it to a matching transport module.

5. A set of transport modules that transfers the message to the destination.

In other words three main modules (*queue-in*, *scanning*, *queue-out*) are connected by two queues (*incoming*, *outgoing*). On each end is a set of modules to receive or send mail—one for each protocol. The queue includes also a message *pool* where the bodies of the queued messages are stored. Figure 6.11 depicts the new designed architecture.

[Figure 11 about here.]

This architecture is heavily influenced by the ones of *qmail* and *postfix*. Both have different incoming channels which merge in the module that puts mail into the queue; central is the queue (or more of them); and one module takes mail from the queue and passes it to one of the outgoing channels. But mail processing is built into the architecture in a more explicit way in this design than it was done in *qmail* and *postfix*.

Special regard was put on addable support for further mail transfer protocols. Here the design appears to be most similar to *qmail*, which was designed to handle multiple protocols.

The modules

Now follows a description of the modules of the new architecture. They are described in the same order in which a message passes through them.

Receiver modules They are the communication interface between external senders and the *queue-in* module. Each protocol needs a corresponding *receiver module* to be supported. Most popular is the *sendmail* module, which is a command to be called from the local host, and the *smtpd* module which usually listens on port 25. Other modules to support other protocols may be added as needed. Receiving modules that need to listen on ports should get invoked by *inetd*, or by Bernstein's more secure *ucspi-tcp*. This makes it possible to run them with least privilege.

The *queue-in* module Its job is to store new messages into the queue. When one of the receiving modules has a new message, it invokes the *queue-in* module which creates a spool file in the *incoming* queue and a data file in the *pool*. The receiver module then sends the envelope, the message header, and the message body. The *queue-in* modules writes the first two into the spool file, the latter one into the *pool*.

The *scanning* module It is the central part of the system. It reads spool files from the *incoming* queue, works on the data, and writes new spool files to the *outgoing* queue. Then the message is removed from the *incoming* queue. The main job of this module is the processing of the message. Headers are fixed and missing ones are added if necessary, aliasing is done, and external processing of any kind is triggered. The *scanning* module processes primary the spool files but may read the mail body from the *pool* if necessary.

The *queue-out* module This module takes messages from the *outgoing* queue, queries information about the online state, and passes the messages to the correct transport module. Successfully transferred messages are removed from the *outgoing* queue. The *masqmail* specific tasks of the route management are handled by this module, too.

Transport modules These modules send outgoing mail; they are the interface between *queue-out* and remote hosts or local commands. The most popular modules of this kind are the *smtp* module which acts as an SMTP client and the *pipe* module to interface gateways to other systems or networks like FAX and UUCP. A module for local delivery is not included; *masqmail* passes this job to an MDA which gets invoked through the *pipe* module. (See section 4.2.1 for reasons.)

The queue

The queuing system consists of two queues and a message pool. The queues store the spool files—in unprocessed form in *incoming* and in complete and valid form in *outgoing*. The *pool* is the storage of the data files. On disk, the three parts of the queuing system are represented by three directories within the queue path.

The representation of queued messages on disk is basically the same as in current *masqmail*: One file for the envelope and message header information (the "spool file") and a second file for the message body (the "data file").

The currently used internal structure of the spool files can remain. Following is a sample spool file from current *masqmail*. The first part is the envelope and meta information. The annotations in parenthesis are only added to ease the understanding. The second part, after the empty line, is the message header.

```
1LGtYh-0ut-00                ( backup copy of the file name )
MF:<meillo@dream>            ( envelope sender              )
RT: <user@example.org>       ( envelope recipient           )
PR:local                     ( receiving protocol           )
ID:meillo                    ( identity: user or IP address )
DS: 18                       ( data size                    )
TR: 1230462707               ( timestamp of recipience      )

HD:Received: from meillo by dream with local (masqmail 0.2.21) id
 1LGtYh-0ut-00 for <user@example.org>; Sun, 28 Dec 2008 12:11:47 +0100
```

```
HD:To: user@example.org
HD:Subject: test mail
HD:From: <meillo@dream>
HD:Date: Sun, 28 Dec 2008 12:11:47 +0100
HD:Message-ID: <1LGtYh-0ut-00@dream>
```

The spool file owner's executable bit shows if a file is ready for further processing: The module that writes the file into the queue sets the bit as last action. Modules that read from the queue can process messages that have the bit set. This approach is derived from *postfix*.

The data file is stored into the *pool* by *queue-in*; it never gets modified until it is deleted by *queue-out*. They consist of data in local default text format.

Inter-module communication

Communication between modules is required to exchange data and status information. This is also called "Inter-process communication" (short: IPC) because the modules are independent programs in this case and processes are programs in execution.

The connections between *queue-in* and *scanning*, as well as between *scanning* and *queue-out*, is provided by the queues, only signals might be useful to trigger runs. Communication between receiver and transport modules and the outside world is organized by their specific protocol (e.g. SMTP).

Left is only the communication between the receiver modules and *queue-in*, and between *queue-out* and the transport modules. Suggested for this communication is a simple protocol with data exchange through Unix pipes. Figure 6.12 shows a state diagram for the protocol.

The protocol is described in more detail now:

Timing One dialog consists of exactly three phases: (1) The connection attempt, (2) The envelope and header transfer, and (3) The transfer of the message body. The order is always the same. The three phases are all initiated by the client process. After each phase the server process sends a success or failure reply. Timeouts for each phase need to be implemented.

[Figure 12 about here.]

Semantics The connection attempt is simply opening the connection. This starts the dialog. A positive reply by the server leads to the transfer of the envelope and the message header. If the server again sends a positive reply, the message data is transferred. A last server reply ends the dialog.

The client indicates the end of each data transfer with a special terminator sequence. The appearance of this terminator sequence tells the server process that the data transfer is complete. The server then needs to send its reply. The server process takes responsibility for the data in sending a success reply. A failure reply immediately stops the dialog and resets both client and server to the state before the connection attempt.

Syntax Data transfer is done by sending plain text data. *Line Feed* ('`\n`')—the native line separator on Unix—is used as line separator. The terminator sequence used to indicate the end of the data transfer is the ASCII *null* character ('`\0`'). Replies are one-digit numbers with '`0`' meaning success and any other number ('`1`'–'`9`') indicating failure.

Rights and permissions

The set of system users that is required for *qmail* seems to be too complex for *masqmail*. One system user, like *postfix* uses, is more appropriate. *root* privilege and *setuid* permission should to be avoided if feasible.

The *queue-in* module is the part of the system that is most critical about permission. It either needs to run as daemon or be *setuid* or *setgid* in order to avoid a world-writable queue. Ian R. Justman recommends to use *setgid* in this situation:

> But if all you need to do is post a file into an area which does not have world writability but does have group writability, and you want accountability, the best, and probably easiest, way to accomplish this without the need for excess code for uid switching (which is tricky to deal with especially with setuid-to-root programs) is the setgid bit and a group-writable directory. **[?]**

Bernstein chose *setuid* for the *qmail-queue* module, Venema uses *setgid* in *postfix*, yet the differences are small. Better than running the module as a daemon is each of them. A daemon needs more resources and therefore becomes inefficient on systems with low mail amount, like the ones *masqmail* will probably run on. Short running processes are additionally higher obstacles for intruders, because a process will die soon if an intruder managed to take one over.

The modules *scanning* and *queue-out* are candidates for all-time running daemon processes. Alternatively they could be started by *cron* to do single runs.

Another possibility is to run a master process as daemon which starts and restarts the system parts. *postfix* has such a master process, *qmail* lacks it. The jobs of a master process can be done by other tools of the operating system too, thus making a master process abdicable. *masqmail* does probably better go without a master process, because it aims to save resources, not to get the best performance.

A sane permission management is very important for secure software in general. The *principle of least privilege* [**?**, section I.A.3.f], as it is often called, should be respected. If it is possible to use lower privilege then it should be done. An example for doing so is the *smtpd* module. It is a server module which listens on a port. One way is to start it as root and let it bind to the port and drop all privilege before it does any other work. But root privilege is avoidable completely if *inetd*, or one of its substitutes, listens on the port instead of the *smtpd* module. *inetd* will then launch the *smtpd* module to handle the connection whenever a connection attempt to the port is made. The *smtpd* module needs no privilege at all this way.

Chapter 6

Summary

This thesis is a comprehensive analysis of *masqmail*. It followed a clear structure from the present to the future, from the general to the special, and from problems to requirements to proposed solutions.

In the beginning, reasons why it is worth to revive the development of *masqmail* were given and the problems of the program were identified. Then the current and future market for electronic communication and email was analyzed. It was showed that email is future-proof and probable trends were spotted. Afterwards the different types of MTAs were classified and the most important alternatives to *masqmail* were presented and compared.

In the second half of the thesis, *masqmail* was in the focus. The goal to reach with further development was defined and the requirements were identified. The existing source code was compared against the requirements to see which ones are already fulfilled. The pending work tasks were ranked by their focus, which depends on the importance of the task and the amount of work it involves.

The possible strategies for controlling further development (improve existing code or redesign and rewrite) were compared against each other on basis of the required work. They were additionally discussed with regard to various other influences. The final decision was a twofold aim: First, improve the existing code to keep it usable for the next time. Second, design a new version of *masqmail* with respect for the modern goals for MTAs that were identified throughout the thesis.

In the end, more concrete plans for the improvements of the existing code were made and a suggestion for a new design for *masqmail* was presented. The description of this new design left quite a few questions open, however, it was intended as a discussion with suggested solutions. To cover such a topic throughout, much more information need to be collected and more detailed studies of the situations in other MTAs need to be made. This would take at least a second diploma thesis or a master's thesis.

Outlook

This diploma thesis is intended to be the begin of a long-time effort to revive *masqmail*. The next important step is creating a community of people that are interested in reviving *masqmail*'s development. Then comes implementing the identified tasks together with this group of volunteers, and afterwards, creating the next generation of *masqmail*.

Like expected for unmaintained software, there are known bugs in *masqmail*. Those need to be fixed.

Documentation and "marketing" are also important. Especially end user documentation is needed and people who help to distribute the knowledge of *masqmail*'s existence and its advantages.

masqmail is software with value. This thesis is a first effort to revive it—it shall not be the last.